1

Introduction

1.1. A little bit of history

The first version of Tcl sprang to life at UC Berkeley way back in 1988. Professor John Ousterhout's primary motivation[1] was to create a standardised, extensible language that could be easily embedded into applications to allow their functionality to be scripted. The accompanying graphical toolkit Tk, which used Tcl as its scripting language, came into being a couple of years later. The combination grew in popularity and was influential enough for Prof. Ousterhout to receive both the ACM Software System and USENIX STUG awards in 1998.

Since those early years, Tcl has grown from an "embeddable, scripting" language to a full fledged dynamic programming language versatile enough for programs ranging from one-line throwaways to enterprise scale distributed systems. Development of Tcl is now controlled through the Tcl Core Team[2] (TCT) which makes decisions on future enhancements through a formalized voting process. These enhancements may be proposed by any interested parties through a Tcl Improvement Proposal[3], or TIP.

1.2. What Tcl offers

Tcl's benefits permeate all aspects of the software development process. Programmers will appreciate

- a wide-ranging set of built-in commands as well as libraries and extensions for common tasks
- programming conveniences like seamless Unicode support and infinite precision arithmetic
- the malleability which enables metaprogramming, custom control structures and embedded mini-languages that are specialized for the target domain
- flexible and extensible support for object-oriented programming that lets you write code in a variety of object-oriented styles
- an advanced *channel* abstraction for I/O with support for defining new types of data streams and pluggable transforms. For example, automatically compress data while writing to a file. Or encrypt. Or both.
- a virtual file system facility that allows remote FTP sites, databases, in-memory structures etc. to be exposed and accessed as if they were local files.
- the ability to call out to other languages, such as C for a performance boost or to Java or .Net classes for integration
- an interactive mode which promotes rapid experimentation and iterative prototyping while facilitating test-driven development

From a software architect's perspective,

- non-blocking, asynchronous architectures are easily supported with Tcl's integrated event loop
- Tcl's coroutine and threading features allow a number of different concurrency models — traditional threads with shared data, message-passing actors or CSP

---

[1] http://www.tcl.tk/about/history.html
[2] http://www.tcl.tk/cgi-bin/tct/tip/0.html
[3] http://www.tcl.tk/cgi-bin/tct/tip/2.html

- data driven and reactive programming models are simplified by the availability of the tracing facility, in combination with the event loop
- applications can run multiple independent interpreters with sandboxing capabilities for executing untrusted code

Managers are people too and Tcl addresses their needs as well.

- Tcl's portability extends from Windows, Linux, OS X, Android and other mainstream operating systems to embedded systems like Cisco routers. Development for multiple platforms is simplified.
- Tcl's versatility means you can use it across multiple components in your product: command line tools, graphical interfaces, back end servers, and even test automation. Not only is code sharing facilitated, programming skills are also more easily transferred. Managers can deploy their minions where they are needed!
- Tcl's single file executable packaging makes distribution trivial in a corporate environment and facilities like online tracing and remotability simplify field support.
- Tcl's stability and backward compatibility means legacy code from the last century will continue to run with minimal or no changes.
- Last but not least, the open source BSD license means minimal dealing with lawyers. Yay!

No doubt by now you are chomping at the bit to get started on Tcl. That will have to wait for the next chapter though, as tradition demands we say a little bit about the book itself first.

1.3. Reading this book

I expect the book's audience to include those who are new to Tcl as well as those who already have a more than passing familiarity with the language.

For the newcomers...

The book requires no prior experience with Tcl but does assume some basic programming background on the reader's part. Knowledge of terms like *function, variable, for loops* etc. is about sufficient to start learning the basics of Tcl. More advanced constructs like asynchronous programming, threads and coroutines require a little more sophistication but you can get a lot of programming done without venturing into these areas. The book's attempt to be comprehensive does mean that it is easy to be distracted by the level of detail. My suggestion would be to not get bogged down by the minutia of every command but to focus on the high level conspectus of language features and idioms. You can then come back to refer to the details as and when needed. It may also be beneficial to go through one of the short online Tcl tutorials. The official [4] one has not been updated to Tcl 8.6 as of this writing but should suffice for introductory purposes.

The next chapter shows you how to install Tcl and run it in interactive mode. You are strongly encouraged to do so and try the illustrative code snippets from the book by entering them in the Tcl shell.

For the old hands...

For readers who have worked with Tcl before, the book can serve as a reference with a detailed table of contents and a comprehensive index. At the same time, browsing through the book may very well lead many to discover Tcl features and capabilities they might not have been aware of, or to gain a deeper understanding of specific topics. Advanced material, such as object-oriented programming, coroutines, reflected channels, virtual file systems etc. are treated in detail.

1.3.1. Typographic conventions

We now come to the obligatory section on formatting and typographic conventions even though they should be obvious to everyone but the publisher.

---

[4] https://www.tcl.tk/man/tcl8.5/tutorial/tcltutorial.html

Text formatting

Within the text, we use *italics* to define terms and **bold** for emphasis. File paths and program elements like commands and variables are shown in a `monospace` font. Additionally, we use capitalized italics in the same font for `PLACEHOLDERS` that stand for some variable part in a code fragment.

Highlighting

Certain notes and points of emphasis are highlighted in one of the following ways:

Important You must carry your driver's license and insurance papers at all times. *Stresses important points you must keep in mind.*

Caution Dangerous curves ahead. Reduce speed. *Actions you need to be careful about.*

Warning Do not drink and drive. *Stuff you would be foolish to do.*

Note Turning right on red is not permitted in New York. *Information that you might miss or overlook.*

Tip Use the exact change lanes for quicker service. *Tips for productivity.*

Sidebars

Material that is related to the discussion but not directly relevant is placed in a sidebar. For example,

History of driving regulations

Licenses were not required for driving in the United States until 1903 when Massachusetts and Missouri became the first states to make them mandatory.

Code samples

Code samples fall into three categories:

- syntax descriptions
- commands typed at the Tcl shell prompt
- scripts as they might be stored in files

All use the same font employed for code within descriptive text.

The first of these are intended to show syntax of commands and not expected to be executed as-is. Optional parts of the command are shown enclosed in ? characters.

```
set VARNAME ?VALUE?
```

The above syntax indicates that `VARNAME` and `VALUE` are only placeholders for the actual variable name and value respectively. Moreover `VALUE` is optional and need not be specified.

Commands that you might type at the Tcl shell prompt are shown as

```
% set x 1
→ 1
```

The % character is the Tcl shell prompt so the command itself is set x 1. Any output that the shell prints out is prefixed with the → character. Depending on the example, lines may be truncated, indicated by ellipsis ..., or wrapped, prefixed with a ↳ character.

Error messages printed by the shell are prefixed with Ø.

```
% set x $nosuchvariable
Ø can't read "nosuchvariable": no such variable
```

In the interest of saving space, short commands and the result may be shown on the same line without the prompt, particularly when the output from multiple commands is to be compared.

```
format %x 42 → 2a
format %b 42 → 101010
```

In this case, a result that is an empty string is shown as in the example below.

```
set x "" → (empty)
```

Finally scripts where the output of individual commands is not important or relevant are shown without the command prompt. Only the output of the last command is shown.

```
proc add {a b} {
    return [expr {$a+$b}] ❶
}
add 2 3
→ 5
```

❶ expr computes arithmetic expressions

The numbered callout shown in the above example is intended to either highlight or provide additional information about a line in the script.

1.3.2. Utility procedures used in the book

Throughout the book we use various simple utility procedures for convenience, for example to print a list. These procedures are shown in Appendix B.

1.4. Online resources

The primary website for Tcl is hosted at http://www.tcl.tk. Announcements of new releases, conferences etc. happen here. It also hosts the reference pages for Tcl and core libraries as well as the repository[5] for Tcl Improvement Proposals.

The Tcler's Wiki[6] is where you should go for all kinds of tips, code samples, and wide ranging discussions on a variety of Tcl-related topics.

The Usenet group comp.lang.tcl, also accessible through Google Groups[7], is dedicated to a discussion of Tcl-related topics and a good place to get any questions answered.

---

[5] http://tip.tcl.tk
[6] http://wiki.tcl.tk
[7] https://groups.google.com/forum/#!forum/comp.lang.tcl

Alternatively, the Tclers' Chat is a chat room accessible either via XMPP clients or through IRC gateways. Many knowledgeable Tcl programmers hang out here and you can learn a lot just by listening in. A specialized chat client tkchat[8] is also available.

The source code for Tcl and core extensions is hosted at http://core.tcl.tk in a Fossil[9] DVCS repository. This also hosts Tcl's bug reports[10] and a catalog[11] of Tcl libraries and extensions.

Tcl source distributions are available from the SourceForge download area[12]. Note however that SourceForge is no longer used for the source repository or for logging bug reports. Binary distributions are available from multiple sources as we list in the next chapter.

1.5. Chapter summary

We are now ready to actually begin our journey into the Tcl world. In the next chapter, you will learn how to install Tcl on your system, enter commands interactively and run Tcl programs.

---

[8] http://tkchat.tcl.tk/
[9] https://www.fossil-scm.org
[10] http://core.tcl.tk/tcl/ticket
[11] http://core.tcl.tk/jenglish/gutter/
[12] https://sourceforge.net/projects/tcl/files/Tcl/

2

Getting Started

Running Tcl requires you to first install the Tcl runtime and tools. We will start off by describing the installation procedure and then move on to the actual mechanics of running Tcl programs.

2.1. Installing Tcl

There are several options for installing Tcl on your system:

- Many operating systems provide bundled distributions of Tcl
- Several third parties distribute precompiled binaries
- You can build your own distribution from the Tcl sources

We describe each of these in the next few sections.

2.1.1. Installing bundled packages on Linux

Some operating systems include Tcl in their distribution. Note however that these are not always up to date and you may prefer to install the latest Tcl version separately as described later.

Bundled packages can be installed using the installation package manager for the system. For example, on a Debian Linux system, use apt-get to install Tcl.

```
apt-get install tcl
```

On Fedora and other rpm-based distributions, use yum for the same purpose.

```
yum install tcl
```

These will install Tcl in system-specific directories. In most cases, extensions to Tcl are distributed as separate packages and have to be individually installed.

2.1.2. Installing third party binary distributions

Alternatively, you may install one of the third party binary distributions of Tcl. Obviously this is necessary if your OS does not bundle Tcl but there are other reasons to do this. For example, the operating system bundled version of Tcl might not be the latest, or you may not have the system privileges required for its installation.

2.1.2.1. ActiveState multi-platform distributions

The company ActiveState maintains both free and commercial distributions of Tcl for multiple platforms including Windows, Linux, OS X, Solaris, AIX and HP-UX. In addition to Tcl itself, the distribution includes a wide variety of third party extensions and packages. There are however two caveats to keep in mind with respect to ActiveState distributions at the time of this writing:

- Some Tcl platforms and extensions are only available in the commercial edition.
- The free community edition has licensing restrictions pertaining to use on production systems and redistribution.

Nevertheless, these restrictions do not matter for your personal use and these distributions are the most popular Tcl binary distributions as of today.

To install ActiveTcl, download the distribution[1] for your platform and follow the detailed installation instructions[2].

For Linux and Unix based platforms, this involves extracting the downloaded archive into a temporary directory and running the `install.sh` shell script in the extracted directory. You may want to add the directory containing the installed binaries to your PATH environment variable.

On Windows platforms, the distribution is a self-extracting executable. To install Tcl, run this executable and follow the on-screen instructions. The installer will optionally modify the PATH environment variable and associate the `.tcl` file extension with the `wish` GUI application.

> Windows will sometimes prevent you from running downloaded executables. Therefore, after saving to disk, you **may** need to bring up the file properties dialog in Windows Explorer by right clicking on the file, selecting the `Properties` menu item and then clicking the `Unblock` button on the `General` tab in the properties dialog.

2.1.2.2. Installers for Windows

There are several Windows installers available for Tcl, two of which we detail here.

BAWT (Build Automation With Tcl) is a framework for building Tcl and extensions. Although its primary purpose is building the software, it also makes available a BAWT Tcl installer[3] for Windows which includes a very broad range of packages and extensions.

The Magicsplat distribution[4] for Windows is a Windows installer based MSI package. The distribution is maintained by this author and targets Windows 7 and later. It includes the most commonly used Tcl packages and extensions.

Unlike the ActiveTcl distribution, both the above are licensed under the same conditions as Tcl itself and do not prohibit commercial use and redistribution.

To install either distribution, download the appropriate setup program for BAWT, or the MSI package for Magicsplat, from their web sites. Then for BAWT, run the setup program from the command line or Windows Explorer. For the Magicsplat MSI package, either double click the file in Windows Explorer, or type

```
start FILENAME.msi
```

from the DOS prompt where `FILENAME` is the name of the downloaded package.

> As described earlier for ActiveTcl, you may need to unblock the program before you can run it in Windows.

Both installers will guide you through the install process permitting installation to different directories, optionally modifying the PATH environment variable and so on. For the Magicsplat package, you need to choose the Advanced option on the initial installation screen to see these options. Like the ActiveTcl distribution, the Magicsplat installer will also register file associations. However, instead of registering `.tcl` and `.tk` extensions, it registers `.tclapp` and `.tkapp` instead on the basis that plain `.tcl` files are likely to be library scripts and packages and not runnable applications.

---

[1] http://www.activestate.com/activetcl
[2] http://docs.activestate.com/activetcl/8.6/at.install.html
[3] http://www.bawt.tcl3d.org/download.html
[4] https://bintray.com/apnadkarni/tcl-binaries/tcl-binaries-windows

A third alternative for a Windows installer based Tcl distribution is the IronTcl[5] distribution. Its distinguishing feature is that it uses signed binaries and is available with a commercial support contract. However, 64-bit binaries are only available to commercial customers.

2.1.2.3. Perschak distributions for Linux and Windows

Thomas Perschak[6] maintains binary distributions for Linux and Windows. These distributions do not have an installer. You can just extract them to a directory and run them in place.

2.1.2.4. Tcl for Android

The AndroWish[7] distribution targets the Android platform and is available for both ARM and x86 architectures. While allowing many Tcl scripts to run unmodified on an Android device, it also includes a large number of commonly used packages and supports interfaces to much of the native Android API.

2.1.2.5. Single file Tcl executables

One final, and possibly simplest, option for installing a Tcl binary executable is to use *tclkits*. Also known by various other names such as *starpacks*, these are single file executables that contain Tcl and supporting core libraries. Because these are all self-contained, you can copy the file anywhere and run it without any installation step. Tclkits can greatly simplify deployment of Tcl applications and we will look at them in detail in Section 19.4. This might be the easiest way to try out Tcl. The libraries bundled with these kits depends on the specific download source although in Section 19.4.4 we will see how to add libraries that are missing.

There are several Web sites from where tclkits can be downloaded. One is Roy Keene's Tclkit site[8] which contains binaries for several platforms including Windows, OS X and Linux/Unix operating systems. The site allows you to customize which libraries are included in the binaries.

Another alternative is the kbskit distributions, which vary in terms of included libraries and extensions. These distributions can be downloaded from the kbskit download site[9].

The AndroWish[10] project also provides single file Tcl executables for Windows and Linux in addition to Android. This comes with a rather large number of extensions included in the executable.

2.1.3. Installing from source

You may at times want to build and install Tcl directly from sources. This may be because

- you cannot find a suitable binary distribution for your platform
- you need to integrate Tcl into a larger application build environment
- you want the cutting edge development release hot off the repository
- or whatever.

In this section we describe the simple steps to accomplish this.

2.1.3.1. Tcl source repository and releases

The official Tcl source repository resides at core.tcl.tk and uses the Fossil[11] distributed source code management system. However, we are not going to describe how to work with Fossil and build directly from the repository source. We will instead focus on the official Tcl source code releases. These are available from the SourceForge file distribution[12] site (8.6.6 is the latest Tcl version at the time of writing). The files of interest are:

---

[5] https://www.irontcl.com
[6] https://bitbucket.org/tombert/tcltk/downloads
[7] http://www.androwish.org
[8] http://tclkits.rkeene.org/fossil/wiki/Downloads
[9] https://sourceforge.net/projects/kbskit/files/kbs
[10] http://www.androwish.org/download/index.html
[11] https://www.fossil-scm.org
[12] https://sourceforge.net/projects/tcl/files/Tcl/8.6.6

- tcl8.6.6-src.tar.gz and tcl866-src.zip which contain the source code for Tcl and some core packages. The two only differ in the archive format.
- tk8.6.6-src.tar.gz and tk866-src.zip which contain the source for the Tk extension. This is strictly not part of Tcl itself but you will need it if you want to use the GUI version of the Tcl/Tk shell (wish).

2.1.3.2. Building on Unix-like platforms

Follow these steps to build and install Tcl and Tk on Unix-like systems.

- Extract tcl8.6.6-src.tar.gz into a directory, say tclsrc.
- Change to the tclsrc/unix directory.
- Run the commands in the shell

```
./configure --prefix=/usr/local/tcl --enable-threads
make
make install
```

The above builds the 32-bit version of Tcl and assumes you want to install Tcl in the /usr/local/tcl directory.

To build the 64-bit version, add the --enable-64bit option to the configure step.

```
./configure --prefix=/usr/local/tcl --enable-threads --enable-64bit
```

Next, build the Tk extension following similar steps.

- Extract tk8.6.6-src.tar.gz into a directory, say tksrc, residing at the same level as the tclsrc directory.
- Change to the tksrc/unix directory.
- Run the commands in the shell

```
./configure --prefix=/usr/local/tcl --enable-threads --with-tcl=../../tclsrc
make
make install
```

Note the --with-tcl option points to the location of the Tcl source directory. As before, if you are building the 64-bit version, you need to add the --enable-64bit switch to the configure step.

You will now have a Tcl installation along with the Tk extension in /usr/local/tcl.

2.1.3.3. Building on Windows

The following steps will build and install Tcl and Tk on Windows using Microsoft's compiler tool chain.

- Start the Visual Studio or Microsoft SDK command prompt for 32- or 64-bit release builds as appropriate.
- Extract tcl866-src.zip into a directory, say tclsrc.
- Change to the tclsrc\win directory.
- Run the commands

```
nmake /f makefile.vc INSTALLDIR=C:\Tcl
nmake /f makefile.vc INSTALLDIR=C:\Tcl install
```

This assumes you want Tcl installed under the C:\Tcl directory.

To build and install Tk,

- Extract tk866-src.zip into a directory, say tksrc, at the same level as the tclsrc directory.
- Change to the tksrc\win directory.
- Run the commands

```
nmake /f makefile.vc TCLDIR=../../tclsrc INSTALLDIR=C:\Tcl
nmake /f makefile.vc TCLDIR=../../tclsrc INSTALLDIR=C:\Tcl install
```

This will build and install Tk and the GUI shell wish.

2.1.3.4. Building on OS X

The process of building Tcl on OS X is similar to that for Unix. Full instructions are provided in the README file in the macosx directory in the Tcl source distribution.

2.1.3.5. Using BAWT for Tcl and extensions

Although Tcl itself is straightforward to build, it can be slightly more involved to build third party extensions due to additional dependencies, different build systems etc. The BAWT system we mentioned earlier specifically tackles this problem. It includes everything needed to build Tcl and a wide variety of extensions. Currently supporting Windows, Linux and OS X, BAWT requires the user to only run a single batch or shell script to build Tcl and the extensions of interest. See the BAWT documentation[13] for the procedure.

2.1.4. Files and directory structure

After installation, the target directory contains the three subdirectories shown in Table 2.1.

Table 2.1. Tcl directory structure

Directory	Description
bin	Contains main Tcl and Tk executables along with the core shared libraries. Most installations will add this directory to the PATH environment variable or link to the executables in this directory from a standard directory already included in PATH.
include	Contains C header files required for building Tcl extensions.
lib	Default location for all add-on packages and extensions. The C libraries required for Tcl extensions are also located here as are runtime support scripts and other files used by Tcl for locale and time zone information, character encodings etc.

The Tcl distributions bundled with the operating systems may differ from the above layout. Moreover, some distributions may also create additional directories for documentation, sample programs and such. As a special case, the single-file tclkit versions are all self contained and do not follow the above structure.

The main executables in the bin directory are tclsh and wish (tclsh.exe and wish.exe on Windows). These are the command line and GUI versions of the Tcl shell and we will be taking a closer look at them shortly.

Depending on the specific distribution, your Tcl shell may be named slightly differently such as tcl8.6, wish86t etc.

2.1.5. Reference documentation

The Tcl reference documentation is online at http://tcl.tk/man/tcl/contents.htm. This includes reference pages for Tcl as well as the core packages like Tk, TDBC etc.

On Unix systems, the Tcl reference documentation is also available in the form of man pages accessible via the standard Unix man program.

---

[13] http://www.bawt.tcl3d.org/documentation.html

On Windows systems, the ActiveTcl distribution comes with its own Windows Help file (.CHM format) for Tcl and extensions. Another alternative in the same format is available from the author's TWAPI [14] project.

2.2. Running a Tcl program

With all the preliminaries out of the way, let us now get around to actually running a Tcl program. Convention dictates that we must begin by greeting the world. Use your favourite editor and create a file called hello.tcl with the following line of text.

```
puts "Hello World!"
```

At your shell or DOS command prompt, run this program using tclsh as shown here.

```
C:\temp>tclsh hello.tcl
Hello World!

C:\temp>
```

You have now written your first Tcl program. Feel free to go add Tcl to your resumé.

2.2.1. The Tcl library and interpreter

We need to take a moment now to distinguish between Tcl, the Tcl interpreter, the Tcl library, Tcl programs or scripts, and Tcl applications.

- Tcl is the programming language. A Tcl program or script is a sequence of commands or program statements written in Tcl.
- The Tcl interpreter is kind of a virtual machine that provides the runtime environment for running Tcl programs. As we lay out in great detail in Chapter 20, an application may contain multiple such interpreters.
- The interpreter virtual machine is implemented as a library which may be statically linked or loaded as a shared library into any application to allow it to execute Tcl scripts. An application makes calls into the library to create Tcl interpreters and execute Tcl programs.
- A Tcl application is a program that is written in C, or some other language, that compiles to machine code and links to the Tcl library. In some cases, the application may do very little other than provide a means to execute Tcl. In such cases, the Tcl program or script itself implements the entire functionality of the application. In other cases, the application may natively implement much of the user visible functionality and the embedded Tcl interpreter acts as a means to allow end user scripting of the application.

If you are new to programming, you do not need to really worry about all these terms. It is just a prelude to introducing two applications that come as part of the Tcl distributions — the Tcl shells.

2.2.2. The Tcl shells

The Tcl shells are simple applications that provide a means of executing Tcl scripts, either interactively or stored in files. They provide practically no other application level functionality themselves. The Tcl distribution comes with two such shells — tclsh and wish. The former is for general purpose use including command-line and daemon or background applications while the latter is intended for applications having a graphical user interface. The shells can be used for interactive experimentation with Tcl or for full blown applications that are entirely coded in Tcl.

2.2.2.1. The tclsh command-line shell

In its essence, the tclsh application reads from the terminal (console on Windows, we use the terms interchangeably) or a file and executes the read input as Tcl code. We have already seen its use in our simple Hello World! program earlier. We now describe its functionality in more detail.

---

[14] https://sourceforge.net/projects/twapi/files/Combined%20Help%20Files

We remind you that `tclsh` may be named `tclsh86` or `tclsh86t` or a similar form depending on the specific Tcl distribution you have chosen to install.

2.2.2.1.1. Running `tclsh` interactively

When run with no arguments, `tclsh` runs in *interactive mode*. It displays a command prompt and any lines entered are treated as a Tcl script and executed. The result is then printed out. This is commonly known as the Read-Eval-Print-Loop (REPL). Typing the `exit` command will cause the program to terminate. A sample session is shown below.

```
C:\temp>tclsh
% puts "Hello World!"
Hello World!
% exit

C:\temp>
```

In interactive mode, `tclsh` has a few changes in behaviour as compared to running a script stored in a file. These differences are described here.

Startup scripts: `.tclshrc`, `tclshrc.tcl`

On starting up, `tclsh` checks for the existence of a file `.tclshrc` (`tclshrc.tcl` on Windows) in your home directory. If found, the contents of the file are evaluated as a Tcl script before `tclsh` displays its command prompt. Note this file is only automatically read in interactive mode.

Execution of external programs

Another feature of interactive mode is that if the line entered by the user does not correspond to a Tcl command, Tcl will execute a program of that name if one exists in a directory in the user's PATH environment variable. Again, we stress this action only happens in interactive mode. Morever, this behaviour can be disabled by setting the variable `auto_noexec` to any value. Here is a demonstration.

```
% uname -a
Linux vm2-debian7 3.2.0-4-686-pae #1 SMP Debian 3.2.81-2 i686 GNU/Linux
% set auto_noexec ""
% uname -a
invalid command name "uname"
```

The sample session above shows that as uname is not recognized as a command, Tcl runs an external program of that name. However, once we set the variable `auto_noexec`, an error is reported.

Command abbreviations

In interactive mode, `tclsh` will accept abbreviations for commands as long as there is no ambiguity. For example, the Tcl command `puts` can be abbreviated as `pu` as shown here.

```
% pu "Hello!"
→ Hello!
% proc print_hello {} {pu "Hello!"}
% print_hello
→ invalid command name "pu"
% p
→ ambiguous command name "p": package pid print_hello proc puts pwd
```

Notice how abbreviations are **not** accepted if used inside a procedure (`print_hello` in our example) or if the abbreviation does not uniquely identify a command.

Command history

In interactive mode, `tclsh` also maintains a list of previously executed commands each tagged with a *history event number*. These can be recalled at the interactive prompt using forms similar to those in the Unix C shell.

The `!!` form prints the previous command and executes it again.

```
% puts foo
→ foo
% !!
→ puts foo
  foo
```

The `^OLD^NEW` form replaces any occurences of OLD in the previous command with NEW and re-executes it.

```
% ^foo^bar
→ puts bar
  bar
```

The `!N` form re-executes the command tagged with the history event number N.

```
% history
     1   puts foo
     2   puts foo
     3   puts bar
     4   history
% !3
puts bar
bar
%
```

Also notice from the sample that you can print the list of commands executed with the `history` command. We will look at this command in more detail in Section 10.9.

> The command abbreviations, history and auto-execution of external programs are actually implemented by a handler run by default when a command name is not recognized. We will have more to say about this in Section 3.5.1.2.

Command-line editing

The `tclsh` shell does not itself include any facilities for command recall with cursor keys, line editing, tab completion for command and file names etc..

In a Windows environment, the DOS console already provides most of these features other than tab completion. On Unix platforms, you can avail of the same functionality through several alternatives:

- Use the `rlwrap`[15] program to start `tclsh`
- Load the `tclreadline`[16] extension to Tcl
- Source the pure Tcl `tclline`[17] script which implements most of `tclreadline` functionality
- Use `etclsh`[18] as your Tcl shell

The best option for interactive use may be to use one of the graphical shells, either the one built into `wish` or `tkcon`. In addition to line editing and tab completion the latter includes many very useful facilities for interactive use including hot errors, remote operation and ability to interact with multiple Tcl interpreters.

---

[15] http://wiki.tcl.tk/21599
[16] http://tclreadline.sourceforge.net/
[17] http://wiki.tcl.tk/20215
[18] http://homepages.laas.fr/mallet/soft/shell/eltclsh

Detecting interactive mode

Code that needs to behave differently depending on whether `tclsh` is running in interactive mode can check the `tcl_interactive` global variable. This is set to 1 when running in interactive mode and 0 otherwise. See Section 16.2.1 for how this variable affects `tclsh` behaviour.

2.2.2.1.2. Running scripts with `tclsh`

Running a Tcl application implemented as a Tcl script in a file simply involves passing the containing file path to `tclsh` as a command-line argument (or to `wish` if it is a GUI application). The general form of `tclsh` for running scripts in a file is

```
tclsh ?-encoding ENCODING? SCRIPTPATH ?ARG ...?
```

Here `SCRIPTPATH` is the path to the file containing the Tcl script. The `-encoding` option allows you specify the character encoding (see Section 4.14) for the file if it is different from the system encoding.

Every command in the script is executed in turn until the last at which point `tclsh` will exit. There are of course facilities for terminating the script early or to keep running as a server application would. The script may also pull in additional scripts stored in other files via the `source` command.

Any additional arguments to `tclsh` are treated as program arguments to the script. These are described in Section 2.3.1 along with a small example that also illustrates the use of `tclsh` to run scripts.

2.2.2.2. The `wish` graphical shell

The `wish` application is a "windowing shell". Like `tclsh`, it provides a wrapper for executing Tcl scripts. The difference is that `wish` is written as a GUI application and includes the Tk extension.

> Like `tclsh`, `wish` may also be named slightly differently, for example `wish86` or `wish86t`, depending on the specific Tcl distribution you have chosen to install.

As our book is about Tcl the language, and not GUI programming with Tk, we will only briefly describe `wish`. Our primary motivation is that `wish` provides an interactive environment for Tcl that has some benefits over `tclsh`.

2.2.2.2.1. Running `wish` interactively

Like `tclsh`, `wish` can be invoked without any arguments.

```
wish
```

When run in this manner, the special behaviours listed for `tclsh` in interactive mode also apply to `wish`.

Startup scripts: `.wishrc`, `wishrc.tcl`

On starting up, `wish` checks for the existence of a file `.wishrc` (`wishrc.tcl` on Windows) in your home directory. If found, the contents of the file are evaluated as a Tcl script when `wish` begins execution. Note this file is only automatically read in interactive mode.

The `wish` program differs slightly in its behaviour between Windows and other operating systems.

Running `wish` on Windows

On a Windows system, this will bring up the two windows as shown in Figure 2.1.

Figure 2.1. The wish windowing shell

The window titled wish is a toplevel window where you can add graphical elements using the Tk extension. The window titled Console is a Tcl command console where you can type in Tcl commands. For example, typing our usual

```
puts "Hello World!"
```

will output that line to the console window. Or typing the commands

```
ttk::label .l -text "It is easy to create interfaces in Tcl/Tk."
ttk::button .b -text Exit -command exit
grid .l .b -padx 5
```

will create a label and button arranging them as shown in Figure 2.2.

Figure 2.2. A sample Tk window

Clicking on the button will cause the program to exit. Tk makes it amazingly easy to create graphical user interfaces. Sadly, we do not have space in this book to cover it and refer you to one of the many books that do, such as TkDocs[19].

---

[19] http://www.tkdocs.com

Running wish on Unix systems

Running the wish shell on Linux and Unix systems has different behaviour as unlike Windows they do not distinguish between "console" mode and "GUI" mode. Invoking it will only create one new window, the wish toplevel. The Tcl console will continue to be displayed in the terminal window just as for tclsh. You can type commands in the terminal window in the same manner as you did above to display our sample Tk window.

2.2.2.2.2. Running scripts with wish

Like tclsh, wish can be passed the name of a file containing a Tcl script.

```
wish ?OPTIONS? SCRIPTFILE ?ARG …?
```

The contents of *SCRIPTFILE* are executed as a Tcl script with any additional arguments being passed to the script in exactly the same manner as for tclsh. There are more options that may be specified for wish but we will not cover them in this book.

There is however one important difference between tclsh and wish when it comes to execution of scripts. Unlike tclsh, wish does **not** exit when the last command in the script is executed. It starts running the event loop, discussed in Chapter 15, waiting for user interaction and other events.

2.2.2.3. The tkcon enhanced shell

Unless you are writing graphical user interfaces, there is only one reason to use wish for interactive development instead of plain old tclsh and that is the Tk Enhanced Console tkcon. This is an add-on that comes as a single file tkcon.tcl and is included in most Tcl binary distributions. It can also be downloaded from http://tkcon.sourceforge.net.

The tkcon console sports a number of very useful features not natively available in either tclsh or wish:

- Enhanced command-line editing, with additional keys for cursor movement, line editing and matching of parenthesis, brackets and braces.
- Tab completion for command, file names and object methods. For example, typing **o p Tab** will complete the command as open while **f o r Tab** will display for, foreach and format as alternatives.
- Additional command history features. In addition to the forms described for tclsh, the history can be searched in incremental fashion. For example, if you have entered exe, typing **Ctrl+r** will recall the last command containing those characters. Command history is also maintained across sessions.
- Ability to attach to multiple interpreters, namespaces and remote instances. These capabilities prove to be very convenient once you get into more advanced Tcl.
- Additional facilities to aid debugging and troubleshooting. These include new commands for interactive debugging, monitoring of state changes and checkpointing. "Hot error" links provide easy access to call stacks in case of errors.
- Package management options for loading libraries and extensions.

You can start a tkcon console by passing it as the argument to wish.

```
wish tkcon.tcl
```

This will bring up a command-line window where you can interactively execute Tcl. For details about usage, refer to its documentation[20].

---

[20] http://tkcon.sf.net

2.2.3. Exiting a Tcl application

Any Tcl application may be exited by invoking the exit command.

```
exit ?CODE?
```

The command causes the process to exit passing an integer exit code of *CODE* back to the operating system. If unspecified, *CODE* defaults to 0.

Applications may of course also have other means of exiting. For example, when running a script passed on the command-line, the tclsh Tcl shell exits after the last command in the script has been processed. Similarly, the wish graphical shell will exit when its main window is closed.

The exit code for a process has ramifications in terms of whether its termination is viewed from the outside as a normal exit or some error condition. Generally an exit code of 0 signifies a normal exit and any other value signifies an error. We will have more to say on exit codes in Chapter 16.

2.2.4. Error messages

Although we will delve into Tcl's sophisticated error handling facilities in Chapter 11, here it is worth mentioning that Tcl's error messages are generally very informative and useful when working in interactive mode. For example,

```
% open
0 wrong # args: should be "open fileName ?access? ?permissions?"
% string foo
0 unknown or ambiguous subcommand "foo": must be bytelength, cat, compare, equal, first, ...
```

This is convenient for reminding yourself of command arguments and their order.

2.2.5. Making Tcl scripts executable

As we have seen, any file may be executed as a Tcl script by passing it to a Tcl shell as an argument. However, it is convenient to be able to just type the script file name and have it executed. Thus we would rather execute a script by typing

```
myscript
```

as opposed to

```
tclsh myscript
```

The method for doing this differs between operating systems. We will describe Unix first as it is simpler.

2.2.5.1. Executable scripts on Unix

On Unix systems, any Tcl script that is intended to be an application or program (as opposed to a library) should be marked as executable (via chmod +x) and begin with the following line.

```
#!/usr/bin/env tclsh
```

Then assuming the tclsh executable lies in a directory somewhere on the path, just typing the name of the script file at the Unix shell prompt suffices to have it executed as a Tcl script.

Note that this line is treated as a comment by Tcl as it begins with a # character. Thus although this technique will not work on Windows, nor does it cause any harm if the same script is passed as an argument to tclsh on Windows.

2.2.5.2. Executable scripts on Windows

On Windows, making a script directly executable is more involved. Luckily, installers for binary distributions do these steps for you so you don't have to. If you do build and install from sources, follow the steps here.

The first difference from Unix is that on Windows file execution from the console is based on the file extension. It is not possible to mark individual files as executable. So we need to pick a file extension to associate with Tcl. Following the Magicsplat distribution, we will associate the extension `.tclapp` with Tcl applications that can be executed directly leaving `.tcl` to be used with secondary support files.

Moreover, the extension cannot be directly associated with the application (`tclsh.exe` in our case). It has to be first mapped to a file type. The file type can be any text that does not conflict with the file type set up by other applications. We will be inventive and call the type `TclApp`.

Because they modify the Windows registry, the commands below have to be run with elevated administrative privileges.

We first associate the extension with the file type through the the Windows `assoc` command and then use the Windows `ftype` command to register our `tclsh.exe` executable as the application to be invoked for that file type.

```
C:\temp>assoc .tclapp=TclApp
C:\temp>ftype TclApp=C:\Tcl\bin\tclsh.exe
```

(Assuming that is where our Tcl is installed.)

If you type "myscript.tclapp" at the DOS command-line, Windows will invoke `tclsh` to run your script. If you want to avoid having to type the `.tclapp` extension, there is one additional step. The `.tclapp` extension needs to be added to the list of extensions in the `PATHEXT` environment variable.

```
C:\temp>set PATHEXT=%PATHEXT%;.tclapp
```

Now just typing `myscript` is sufficient to invoke `tclsh` to execute the `myscript.tclapp` file.

As an alternative to the above, you can embed Tcl scripts into Windows `.BAT` batch files using a trick similar to that for Unix. Various variations of this are described in http://wiki.tcl.tk/2455. However, the author prefers the above method for a couple of reasons. First, a batch file involves execution of an intermediate Windows command shell and is therefore slower. Second, there are subtle scenarios where the suggested `.BAT` solutions do not work.

2.3. The application runtime environment

Tcl provides several commands that deal with application's runtime environment including

- arguments passed on the command-line
- the process environment such as working directory, environment variables etc.
- information about the Tcl interpreter itself such as version information
- platform information such as operating system, architecture and user context

2.3.1. Command-line arguments

Any additional arguments supplied on the command-line when invoking `tclsh` or `wish` are passed to the script in the global variables shown in Table 2.2.

Table 2.2. Command-line argument globals

Name	Description
argv0	Contains the path to the script file passed on the command line. If tclsh was invoked without any arguments, this will contain the name by which it was invoked (which is not necessarily tclsh in the presence of links etc.)
argv	List containing the command-line argument values
argc	Count of command-line arguments

Let us illustrate with a simple example. Create a file reverse.tcl with the following content which will simply reverse and print its arguments.

```
# reverse.tcl
if {$argc == 0} {
    puts "Need to provide at least one argument"
    puts "Usage: [info nameofexecutable] $argv0 arg ?arg ...?"
    exit 1
}

proc print_reversed {str} {
    puts [string reverse $str]
}

foreach arg $argv {
    print_reversed $arg
}
```

This example also introduces some very basic syntax:

- Variable values are referenced by prefixing the variable name with $.

- Procedures are defined using proc and invoked like any built-in command.

The script is executed by passing it to tclsh. In this initial run no arguments are passed and hence argc is 0 resulting in the script exiting with an error message.

```
C:\demo> tclsh reverse.tcl
Need to provide at least one argument
Usage: c:/tcl/866/x64/bin/tclsh.exe reverse.tcl arg ?arg ...?
```

When passed arguments, the script runs to completion and exits implicitly at the end of the file.

```
C:\demo> tclsh reverse.tcl abc def
cba
fed
```

2.3.2. The working directory: pwd, cd

The pwd command returns the current working directory for the process. The command below sets the variable dir to the current directory.

```
% set dir [pwd]
→ C:/temp/book
```

The cd command changes the current working directory to that specified.

```
cd ?DIRNAME?
```

If the optional *DIRNAME* argument is not present, the command changes the working directory to the home directory of the current user.

```
% cd ❶
% pwd
→ C:/Users/ashok/Documents
% cd $dir ❷
```

❶   Change to the home directory
❷   Change back to the directory we saved in dir

> ⚠️ The application may have multiple threads and multiple Tcl interpreters in each thread. The working directory is a process-wide setting and therefore the cd command affects **all** interpreters and threads in the process, even native code.

2.3.3. Environment variables: env

The environment variables for the current process are accessible through the env global array and can be accessed the same way as any other array variable.

```
% puts $env(PATH)
→ c:/msys/bin;C:\ProgramData\Oracle\Java\javapath;C:\Program Files (x86)\Intel\iCLS Clien...
% array names env
→ HOME COMSPEC LANG PROCESSOR_IDENTIFIER TERMCAP LOGONSERVER ProgramW6432 SHELL ProgramFi...
```

Arrays are fully described in Section 3.6.7.

There are however a few differences that distinguish env from normal Tcl arrays. First, any changes to the env array are automatically reflected back in the process environment. Addition and deletion of elements in the array is also reflected appropriately in the process environment. Any child processes will inherit the modified environment.

The second is that on **Windows platforms only** env keys are **not** case-sensitive. So

```
% puts $env(hOmE)
→ C:\Users\ashok\Documents
% puts $env(HOME)
→ C:\Users\ashok\Documents
```

would work as well unlike for normal arrays. Note however, array commands that accept wild card patterns are case-sensitive as illustrated by the following:

```
% array names env pat*
% array names env PAT*
→ PATHEXT PATH
```

> 💡 Because of the need to keep the env array synchronized with the process environment, access to elements of the array is almost two orders of magnitude slower than a normal array variable. Thus it is often beneficial to keep a "shadow" copy of the environment in a normal variable wherever possible.

2.3.4. The process identifier: pid

The process identifier, or PID, for the current process can be obtained with the pid command.

```
pid ?CHANNEL?
```

If no arguments are specified, the command returns the PID of the current process.

```
pid → 2572
```

If the *CHANNEL* argument is specified, it must be the channel associated with a process pipeline. In this case, the command returns the list of PID's for the processes in the pipeline. We cover process pipelines in Chapter 16.

2.3.5. Executable file path: info nameofexecutable

The info nameofexecutable command returns the path to the executable image for the current process that is hosting the Tcl interpreter.

```
info nameofexecutable → c:/tcl/866/x64/bin/tclsh.exe
```

Most commonly this is used to find other locations in the file system that are relative to the executable.

2.3.6. Tcl version information: info tclversion, info patchlevel

The tcl_version global variable contains the version of the Tcl library in use, and in effect the version of Tcl. The same information is also available with the info tclversion command.

```
puts $tcl_version → 8.6
info tclversion   → 8.6
```

More detailed version information that includes the *patch level* can be obtained from the tcl_patchlevel global variable and info patchlevel command.

```
puts $tcl_patchLevel → 8.6.6
info patchlevel       → 8.6.6
```

> Version numbers in Tcl have a specific syntax and associated semantics. We discuss these in detail in Section 13.3.2.

2.3.7. Platform information

The tcl_platform global array contains various bits of information about the hosting platform. The elements of this array are shown in Table 2.3.

Table 2.3. Platform information

Element	Description
byteOrder	Either littleEndian or bigEndian depending on whether the underlying CPU architecture is little-endian or big-endian.
engine	Identifies the interpreter implementation. This is normally Tcl but may hold other values if you are using other Tcl implementations or dialects such as jim, jtcl etc.

Element	Description
machine	The CPU architecture that this executable was built for. Note this is not necessarily the native architecture of the system. For example, running 32-bit binaries on a 64-bit Windows system will return intel and not amd64.
os	The operating system
osVersion	The version of the operating system
pathSeparator	The character used to separate directory entries in the PATH environment variable
platform	The operating system family
pointerSize	Either 4 or 8 depending on whether you are running a 32- or 64-bit Tcl interpreter
threaded	1 if threads are enabled in Tcl and 0 otherwise
user	The user account under which the process is running
wordSize	The number of bytes in the C type long for the current architecure

We can print the contents of the array with the parray command.

```
% parray tcl_platform
→ tcl_platform(byteOrder)    = littleEndian
  tcl_platform(engine)       = Tcl
  tcl_platform(machine)      = amd64
  tcl_platform(os)           = Windows NT
  tcl_platform(osVersion)    = 10.0
  tcl_platform(pathSeparator) = ;
  tcl_platform(platform)     = windows
  tcl_platform(pointerSize)  = 8
  tcl_platform(threaded)     = 1
  tcl_platform(user)         = ashok
  tcl_platform(wordSize)     = 4
```

Although the tcl_platform array provides some information about the underlying operating system and architecture, the information there is not complete and specific enough to distinguish hardware platforms and operating systems. For example, it cannot be used to load shared libraries from the appropriate location when multiple architectures are installed within the same directory hierarchy.

The platform package (see Section 13.7) addresses this requirement.

2.3.8. Tcl configuration: `tcl::pkgconfig`

The tcl::pkgconfig command returns additional information about the Tcl configuration and build environment, some of which is also available in the tcl_platform array.

The command has two subcommands. The first, list, returns the a list of keys each of which represents a piece of configuration information.

```
% print_list [tcl::pkgconfig list]
→ debug
  threaded
  profiled
  64bit
  optimized
...Additional lines omitted...
```

The key names printed from the above command should be self explanatory.

The second subcommand, get, is used to retrieve the value associated with a key. For example,

```
% tcl::pkgconfig get bindir,runtime
→ c:\tcl\866\x64\bin
```

returns the directory where the Tcl binaries are installed.

2.4. Chapter summary

In this chapter we described

- how to install and build Tcl
- how to run Tcl scripts as well as interactive shells
- the Tcl application runtime environment

We will now move on to the basics of the language. As we go along, you are encouraged to try out the described commands in an interactive Tcl shell.

3

Tcl Basics

This chapter lays the foundation for the rest of the book. It describes Tcl syntax, how Tcl parses and executes commands, and the use of variables and procedures that form the basis of all Tcl programming. Subsequent chapters will then focus on the details of individual commands.

Conceptually[1], execution of Tcl code occurs in two phases:

- The Tcl source code is parsed using some simple syntactic rules to break it up into a number of commands and their arguments.
- The commands are then invoked with the associated arguments.

The two phases may be intermixed in the sense that parsing a command may involve parsing and substitution of embedded commands and variables.

We will start off with the syntax of the language in the next section and then move on to the basic language commands.

3.1. Basic syntax

The formal syntax rules for Tcl are defined in the Tcl manual[2], often called the *dodekalogue* as it is made up of 12 rules. Here we informally describe the syntax.

A Tcl *program* or *script* is a sequence of commands separated by newline or semicolon characters that are not escaped or quoted. In the special case of command substitution, the trailing] character also terminates commands.

A command in turn is a sequence of *words*. Words are separated by space or tab characters. Spaces and tabs can be included as part of a word by escaping them with a \ or placing them within a quoted string. The line

```
puts -nonewline Hello ; puts " World!"
```

contains two commands separated by a semicolon. The first contains three words puts, -nonewline and Hello while the second contains two words: puts and a second word consisting of a space followed by World. The space preceding the W is not a word separator as it is within quotes. If you are coming from another language, note that **simple strings, like** Hello, **with no whitespace characters need not be placed in quotes**.

A word may also be a bracketed command whose result forms the value of the word. The command below has three words, set, time and the result of evaluating the command clock seconds.

```
set time [clock seconds]
```

The substitution of bracketed commands is described in Section 3.2.3.

A word may be spread over multiple lines when quoted with double quotes or braces, or when it comprises a bracketed command.

---

[1] In practice, Tcl scripts are converted to byte code form before execution
[2] http://tcl.tk/man/tcl/TclCmd/Tcl.htm

```
puts {
    Hello
    World
}
→
        Hello
        World
```

The above command consists of **two** words, puts which is the command name and its argument which is quoted, with braces in this case. The quoting allows spaces and newlines to be considered part of a single word.

> You can use info complete to check if a given string syntactically constitutes one or more complete commands.
>
> ```
> info complete {foo bar "x y z} → 0
> info complete {foo bar "x y z"} → 1
> ```
>
> Note the command does **not** check if the command names are valid, have the correct number of arguments and so on. It only checks whether the given argument can be parsed **syntactically** as a sequence of complete commands. Our first example above fails because of unmatched quotes.
>
> The info complete command is mostly used in applications imitating the interactive command loop as in tclsh and wish. The user may enter commands crossing multiple lines and info complete is used to check for unmatched braces, quotes, brackets etc.

3.2. Substitutions

Tcl performs a series of substitutions before a command is executed:

- Backslash substitutions
- Variable substitutions
- Command substitutions

> These substitutions are not done for strings enclosed in braces {} where different rules apply as we will describe later.

3.2.1. Backslash substitutions

Backslash substitution is a mechanism wherein a character sequence starting with a \ character is used to represent an arbitrary character. There are multiple uses for this such as:

- Representing non-printable ASCII control characters.
- Representing non-ASCII Unicode characters. Although Tcl itself will accept Unicode characters in various encodings in file or keyboard input, many text editors and terminal devices do not allow easy insertion of non-ASCII characters and this provides a way around those limitations.
- Forcing the Tcl parser to treat characters such as spaces or $ as ordinary characters, where they would otherwise be treated specially.

The full set of backslash substitutions, which we also sometimes refer to as backslash escape sequences is shown in Table 3.1.

Table 3.1. Backslash sequences

Sequence	Description
\a	Audible alert (ASCII 7)
\b	Backspace (ASCII 8)
\f	Form feed (ASCII 12)
\n	Newline / linefeed (ASCII 10)
| | ```
% puts a\nb
→ a
  b
``` |
\r	Carriage return (ASCII 13)
\t	Tab
\v	Vertical tab
\\OOO	One to three character octal sequence specifying an 8-bit Unicode code point in the range U+000000 - U+0000FF. For example, é may be represented by the sequence \351.
\xHH	The character x followed by one or two hexadecimal digits specifying a 8-bit Unicode code point in the range U+000000 - U+0000FF. Thus another representation for é would be the sequence \xe9.
\uHHHH	The character u followed by one to four hexadecimal digits specifying a 16-bit Unicode code point in the range U+000000 - U+00FFFF. In this form, é would be represented by either of the sequences \u0e9 and \u00e9.
\UHHHHHHHH	The character U followed by one to eight hexadecimal digits specifying a 21-bit Unicode code point in the range U+000000 - U+10FFFF. Thus é could be represented as \U000000e9 or \U000e9 etc.
\NEWLINE WHITESPACE	A \ followed by a newline character and any amount of whitespace is replaced by a single space character.
| | ```
% puts "abc\
    def"
→ abc def
```<br><br>This is often used to split a long command across multiple lines. Remember a newline character would normally terminate a command unless it was within quotes or braces.<br><br>```
% lsearch -nocase -inline -all \
    {abc def HIJ} hIj
HIJ
``` |

If the character sequence following a \ does not fall into one of the above categories, it is substituted as itself.

```
puts a\\nb → a\nb  ❶
puts \$foo → $foo  ❷
```

❶ \\ treated as a single ordinary \, not a \n sequence
❷ $ treated as itself, not as a variable substitution

Morever, if it happens to be a character, such as $, that has special meaning to the Tcl parser, it will be treated as an ordinary character instead.

```
puts \s    → s ❶
puts \xz   → xz ❷
```

❶   No backslash sequence corresponding to s
❷   \x not followed by hexadecimal digits

3.2.2. Variable substitutions

The second form of substitution that takes place is replacement of variable references by their values. Although variable references can take many shapes, here we only illustrate the simplest one of the form $*VARNAME*.

```
set greeting "Hello World!" → Hello World!
puts $greeting              → Hello World!
```

Variable substitution takes place **inside** words as well.

```
set greeting Hello      → Hello
set who World           → World
puts "$greeting $who!"  → Hello World!
```

Tcl will raise an error if the referenced variable does not exist.

```
% puts $nosuchvar
Ø can't read "nosuchvar": no such variable
```

You can prevent Tcl from treating $ as variable reference by prefixing it with a \ character.

```
puts $greeting  → Hello
puts \$greeting → $greeting
```

There are two important points to note about variable substitution. The first is that the value that is substituted is taken verbatim and **not** reparsed by Tcl. In other words, Tcl will not subject the contents of the variable to further backslash, variable or command substitution.

```
set avar "abc"     → abc
set bvar "\$avar" → $avar
puts $bvar         → $avar ❶
```

❶   Output is $avar, *not* abc

The second point, related to the first, is that substitution of variables does not change the word boundaries in a command. For example, in

```
set greeting "Hello World!" → Hello World!
puts $greeting              → Hello World!
```

the space character in the substituted value Hello World! does not act as a word separator. The second command still contains only two words, the entire contents of greeting including the space being the second word.

The other forms of variable references that we describe later are also subject to substitution as above. Note however, that a $ character by itself, or one that is followed by a character other than an alphanumeric, underscore (_), left parenthesis (() or left brace ({), is **not** a variable reference and will be treated as a literal $ character.

```
puts $$ → $$
puts $= → $=
```

3.2.3. Command substitutions

The final form of substitution is replacement of strings enclosed in [ ] pairs of brackets with the result of executing them as commands.

```
set i 0        → 0
puts [incr i] → 1 ❶
```

❶   The incr command increments a variable.

In the above example, the puts command gets a single argument — the result of executing the incr i command.

As for variables, command substitution can take place inside words as well.

```
puts a[incr i]b                        → a2b
puts "Incrementing $i gives [incr i]." → Incrementing 2 gives 3.
```

The string inside the [ ] pair is actually treated as a Tcl script and so may have multiple commands separated by semicolons or newlines as usual. In this case the substituted value is the return value from the last command.

```
% puts [incr i; incr i; incr i]
→ 6
% puts [
      set j 10
      incr i $j
    ]
→ 16
```

Moreover, because it is parsed as a fresh script, the bracketed string can itself contain quotes, substitutions etc. For example,

```
puts "The total is [expr "2+4"]" → The total is 6
```

The double quote following the expr **starts** a quoted string within the bracketed command, it does not terminate the double quotes that follow the puts.

As for variable substitution, command substitution does not reparse the returned value from the command string or change the word boundaries even when the substituted value contains whitespace or other special characters.

3.3. Quoting

Quoting is a means for telling the Tcl parser to treat a sequence of characters as a single word irrespective of whether it contains whitespace or other characters that would terminate a word or a command. We have already seen one mechanism to prevent special interpretation of characters — backslash sequences. Quoting provides an

alternative, and often more convenient, means for the same. Compare the following alternatives for assigning a string containing spaces to the variable var:

```
% set var This\ is\ a\ single\ word
→ This is a single word
% set var "This is a single word"
→ This is a single word
```

Tcl has two forms of quoting:

- enclosing the string in double quotes

- enclosing the string in braces

The two differ in how substitutions are handled.

3.3.1. Quoting using double quotes

When a string is enclosed in double quotes, word and command separators like spaces, tabs, newlines and semicolons are treated as ordinary characters.

```
% puts "This is line one;
This is line two"
→ This is line one;
  This is line two
```

Notice that spaces within the quoted string were ignored as word separators and neither the semicolon, nor the newline, terminated the command.

Some other points to note about quoting using double quotes:

- The double quote character only has effect if it appears at the beginning of a word. A double quote in the middle of a word is treated as an ordinary character. Moreover, the closing double quote must be followed by a word separator or command terminator. Thus the following result in errors.

```
% set var foo"b ar" ❶
∅ wrong # args: should be "set varName ?newValue?"
% set var "foo"bar  ❷
∅ extra characters after close-quote
```

  ❶ foo"b and ar" are treated as separate words since double quotes within a word carry no special meaning
  ❷ Closing quote not followed by word separator

- Backslash, variable and command substitutions are all enabled within a double-quoted string.

```
% puts "$i\n[incr i]"
→ 16
  17
```

- To have a double quote appear as an ordinary character, use the standard escaping mechanism of preceding it with a \ character.

```
% set var "foo\" bar"
→ foo" bar
```

3.3.2. Quoting using braces

The second form of quoting uses a pair of braces {} instead of double quotes to enclose the string. In this form, with the single exception noted below, all special treatment for characters and all types of substitutions are disabled within the enclosed string. Here is an example contrasting the two forms.

```
% puts "$i\n[incr i]" ❶
→ 17
  18
% puts {$i\n[incr i]} ❷
→ $i\n[incr i]
```

❶   Substitutions enabled inside double quotes
❷   Substitutions disabled inside braces

The one exception where substitution is still carried out is the backslash followed by newline sequence:

```
% puts {abc\
            def}
→ abc def
```

As always, the \, newline and any immediate whitespace is replaced by a single space character.

In other respects, quoting with braces follow similar rules to those for double quotes.

- The leading brace must be the first character of a word.
- The trailing brace must be followed immediately by a word separator or command terminator.

There is however an additional feature (or complication) with braces in that braced strings can nest so that the quoted string is terminated only when the number of closing braces matches the number of opening braces.

```
set nested {Outer {Inner Words} Words} → Outer {Inner Words} Words
```

As we shall see this nesting property is useful for defining lists or dictionaries and for creating "code blocks" to be executed by conditional or iterative commands like if or while.

There is one caveat with regarding to nesting and that has to with how you include a literal brace character within the braced string. When a brace is preceded by a \ it does *not* count towards the nesting depth. However, because backslash substitution rules are not in effect, the \ character is also included in the quoted string. For example,

```
puts {abc \}} → abc \}
```

So getting a literal brace character without a preceding \ character in a braced string is a little tricky. One alternative is to switch to double quotes instead (taking care to properly escape unwanted substitutions).

```
puts "abc \}" → abc }
```

The other option is to use an explicit string construction command such as format or subst or a backslash substitution such as \x7d. Luckily this situation rarely arises.

3.3.3. Choosing the quoting mechanism

The question then arises as how to pick between the two forms of quoting. In most cases, the choice is fairly clear. When interpolating strings with values in variable or computation results, the obvious choice is to use double quotes as braces will not give the intended result.

```
% puts "The current time is [clock format [clock seconds] -format %H:%M]"
```

```
→ The current time is 11:45
% puts {The current time is [clock format [clock seconds] -format %H:%M]}
→ The current time is [clock format [clock seconds] -format %H:%M]
```

Conversely, there are situations where the choice of braces is obvious. The most common is when script blocks are to be passed to Tcl commands such as while, proc etc. Other circumstances where braces are preferred include representation of nested data structures and special situations like file paths in Windows systems where \ is also a path separator. Using braces in this case is a lot more readable.

```
set path "C:\\Windows\\System32\\cmd.exe"
set path {C:\Windows\System32\cmd.exe}
```

Note that special treatment of braces as quoting characters is turned off when they occur within double-quoted strings. The converse is also true in that double quotes are not special inside braces.

```
set var VALUE → VALUE
puts "{$var}" → {VALUE} ❶
puts {"$var"} → "$var" ❷
```

❶   Braces inside double-quotes show up as literals and do not suppress variable substitution.
❷   Quotes inside braces show up as literal quotes.

Thus another basis for picking a quoting character is if the other one is present in the quoted string.

3.4. Argument expansion

The last action taken before a command is executed is argument expansion. Normally every word that is parsed is passed to the command as a single argument. However, when a word is prefixed with the character sequence {*}, the associated word is treated as a list of words and every element of the list is passed to the command as a separate argument.

This is slightly tricky to explain so we will just illustrate with an example. Because it is tied in with *list* structures which we will look at in detail in Chapter 5, we will first very briefly introduce the latter. A list is a ordered sequence of values, referred to as elements of the list. The simplest method of creating a list is as a "string literal" enclosed in braces. Thus the commands

```
set alist {one two three} → one two three
set blist {four five}     → four five
```

create two lists containing three and two elements respectively. Now suppose we wanted to append the elements of the second list to the first. We can use the lappend command for this. This command accepts any number of arguments and appends them to the list contained in a variable.

```
lappend alist $blist six → one two three {four five} six
```

The command has added four five as a single element (as shown by their being enclosed in braces) whereas we wanted four and five to be separate elements. This is where argument expansion using {*} comes in. It treats the following word as a list and "explodes" its value into its elements which are then passed as separate arguments.

```
set alist {one two three}   → one two three
lappend alist {*}$blist six → one two three four five six
```

Note how four and five are now separate elements. In effect this is as if the command were written as

```
lappend alist four five six
```

Note that argument expansion applies no matter in what form the following word is supplied. It could be a variable as in our example, or a quoted string or a bracketed command. For example,

```
% lappend alist {*}{7 8} ❶
→ one two three four five six 7 8
% proc cmd_returning_a_list {} { return {9 10} }
% lappend alist {*}[cmd_returning_a_list] ❷
→ one two three four five six 7 8 9 10
```

❶ String quoted with braces
❷ Bracketed command

In all cases, the value that would be substituted is treated as a list and its elements are passed as separate arguments to the command.

The above situation, where we need to pass the elements of a list to a command that expects them as separate arguments, is not uncommon in Tcl programming.

3.5. Commands

> He who wishes to be obeyed must know how to command.
>
> — Machiavelli

As described previously, the Tcl parser breaks up a Tcl program into a sequence of commands that are executed in turn. We will now go into a little more, but still basic, depth regarding commands: how they are invoked, the different types, their structure etc.

The first thing we will note is that the term *command* is used in the Tcl documentation (and in this book) in two distinct, though related, ways. In our earlier example, we referred to the code fragment

```
puts "Hello World!"
```

as a command. This is the first usage for the term. The other usage refers to puts itself as a command. In most cases, this distinction is clear from the context or immaterial. Where it matters, we will use the terms *command statement* for the first usage.

3.5.1. Command invocation

Once a command statement is parsed into its final form, including any substitutions, argument expansion etc., the first word is looked up in the interpreter's database of registered commands. If found, it is invoked with the remaining words passed as arguments.

Note that the interpretation of arguments is **completely up to the command**. Tcl itself does not care. For example, compare the following

```
puts "2 + 2"            → 2 + 2
expr "2 + 2"            → 4
regexp "2 + 2" "2 + 2 + 3" → 0
```

The puts command will treat its argument 2 + 2 as a string. The expr command on the other hand will treat it as an arithmetic expression. The regexp command will treat it as a regular expression to be matched against the second argument.

In other words, commands are completely free to treat their arguments as strings, numerics, even program code, or whatever, in any manner they see fit.

Indirect invocation of commands

From our earlier discussion remember that substitutions are applied to the first word as well before it is looked up as a command name. So we can write our ubiquitous example as

```
set str ts                    → ts
\x70\u0075$str "Hello world!" → Hello world!
```

That is not particularly useful. What is useful though it being able to invoke a command indirectly through a variable. This is commonly used in callbacks and such where a command name is passed as an argument and invoked as a callback.

```
set cmd puts          → puts
$cmd "Hello world!" → Hello world!
```

3.5.1.1. Counting command invocations: `info cmdcount`

The `info cmdcount` command returns the total number of commands invoked in a Tcl interpreter since it was started.

```
info cmdcount → 15261
info cmdcount → 15262
```

The count of command invocations is seen used in two scenarios. One involves the use of safe interpreters where a limit is set for the number of commands an interpreter is allowed to execute before it is terminated. We will examine this in Chapter 20.

The other use of `info cmdcount` is to generate identifiers at run time that are unique **within that interpreter**. Examples include naming of objects, handles for resources, coroutines and so on.

```
proc make_id {{prefix id}} { return ${prefix}[info cmdcount] }
```

We can use this to generate new unique identifiers.

```
make_id        → id15266
make_id coro → coro15269
```

⚠️ | The command invocation count is not incremented in certain cases due to optimizations in the Tcl byte code compiler. However, it is safe to use for the above purpose as the `info cmdcount` command itself will increment the count.

3.5.1.2. Unknown command handlers

If a command is not found, Tcl invokes a procedure called unknown passing it the name of the command and associated arguments. The result of this procedure is then returned as the result of the original command.

📋 | The handling of unknown commands is a little different in the presence of namespaces but since we have not discussed those yet, we will defer a full discussion to Section 12.5.3.4.

Tcl provides a default implementation of unknown which can be overridden by redefining the procedure. The default implementation takes the following steps to resolve an unknown command.

- It will attempt to load the command by searching Tcl's library paths via the auto_load command. This step is skipped if the auto_noload global variable is defined.

- If Tcl is not running in interactive mode, an invalid command error exception is raised. If running in interactive mode, the following additional steps are taken.

- It will use auto_execok to try and locate an external program of that name. If found, it will run it with the exec command returning the output of the program as the result of the original command.

- If the above steps fail, it will check if the command matches one of the patterns for recalling commands from the command history. If so, the corresponding entry from the command history is executed again and the result returned to the caller.

- As a last resort, Tcl checks if the command name is an abbreviated form of exactly one existing command (so there is no ambiguity) and if so, executes that command returning its result.

- If all the above fail, Tcl raises an exception.

```
% put "Hello World!"
→ Hello World!
```

The above command works because put is an unambigious prefix of a command — puts.

We can replace the unknown command to implement any behaviour that we choose.

```
rename unknown _old_unknown
proc unknown {args} {
    if {$::tcl_interactive && [info level] == 1} {
        if {![catch {expr $args} result]} {
            return $result
        }
    }
    error "Unknown command [lindex $args 0]"
}
```

Don't worry about the details of how that works as we need to go into several Tcl features first. In a nutshell, we treat any unknown command as an arithmetic expression but only if it was executed interactively from the command line.

We can now use the Tcl shell as an interactive calculator.

```
% 2 + 4*10
→ 42
```

Before we go on, let us restore the default implementation of unknown as we will need it later.

```
% rename unknown ""
% rename _old_unknown unknown
```

3.5.2. Comments

If the Tcl parser encounters a # character where it is expecting the first word of a command (the command name), the # character and all characters till the end of that line are treated as a comment and ignored.

```
# puts "This line will not print as it's commented out"
```

There are a number of subtleties involved in this seemingly simple description and we will go through them one at a time.

The # is not treated as a comment if it appears anywhere other than where the first (non-whitespace) character of a command is expected. So for example, you can print it, name a variable, or whatever.

```
puts #Hi              → #Hi
set # "Hello world!" → Hello world!
puts ${#}            → Hello world! ❶
```

❶   This uses the variable reference syntax described in Section 3.6.3

You can even define a command implemented as a procedure named #.

```
proc # {s} { puts $s }
```

However, the following invocation will not work because it will be treated as a comment.

```
# "Hello world!" → (empty)
```

Instead we have to call it using one of the following syntaxes.

```
\# "Hello world!"   → Hello world!
{#} "Hello world!" → Hello world!
set name #          → #
$name "Hello world" → Hello world
```

The above examples just illustrate the point that the check for the # character happens before any substitutions. They are not something you will run into in real-world Tcl code. But we now bring to your attention two mistakes that are commonly made at some point when you are learning Tcl.

Suppose we define a list of items as follows and add a comment to describe the list.

```
set fruit {
    # This is a list of fruit
    bananas
    oranges
}
```

Then we print out list.

```
% print_list $fruit
→ #
  This
  is
...Additional lines omitted...
```

What happened? Well, the # character was not in position where a command was expected and thus was not treated as a comment. It became part of the list!

A second mistake is having an unmatched brace character in a comment.

```
proc demo {n} {
    # This is a comment with an unmatched { character
    return $n
}
```

If you place this code in a file and try to source it, you will get the error

```
unmatched open brace in list
```

The check for comments happens *after* the parsing into words and *before* substitutions. When parsing the above script into words, Tcl encounters the left brace character. At that point, it will look for the matching closing brace, even across line boundaries. Not finding one will lead to the error.

The lesson in this? **Match your braces even within comments!** This is admittedly quirky coming from other languages but a small price to pay for Tcl's syntactic uniformity which is the root cause of this behaviour.

3.5.3. Command types

Commands may be implemented in Tcl by several means.

- Native commands implemented in C
- Procedures defined through proc or apply.
- Coroutines defined with the coroutine command
- Aliases defined with the interp alias command
- Namespace ensemble commands
- Objects and classes defined through Tcl's object-oriented facilities

Irrespective of how they are implemented, they are invoked the same way and can be manipulated in common manner. We will discuss all these in the book with the exception of native implementation of commands in C.

3.5.4. Renaming a command

Tcl is a completely dynamic language where programming constructs can be added, removed or otherwise manipulated at will. This applies to commands as well, even the ones built into Tcl. We can thus change the name of any command with the rename command.

```
rename OLDNAME NEWNAME
```

A common use for rename is to "wrap" a command to add some functionality or to modify its behaviour in some way. We saw this in Section 3.5.1.2. As another example, let us say we wanted all output to be in upper case without having to modify the application itself. We could then wrap the puts command as follows

```
% rename puts builtin_puts       ❶
% puts "Hello world!"            ❷
∅ invalid command name "puts"
% builtin_puts "Hello world!"  ❸
→ Hello world!
```

❶ Save the built-in puts under another name
❷ Fails because there is no longer a command called puts...
❸ ...but there is one called builtin_puts

Then we define a new puts command which makes use of the original command.

```
proc puts args {
    set str [string toupper [lindex $args end]]
    builtin_puts {*}[lreplace $args end end $str] ❶
}
puts "Hello world!"
```

```
→ HELLO WORLD!
```

❶ See Section 3.4 for explanation of the {*} sequence

We now get all output in upper case. The above code fragment uses commands and features we have not gotten to as yet but the main idea behind wrapping commands in this fashion should be clear. We transformed the original data and passed it on to the original command.

⚠️ Although the above method of "wrapping" commands works with puts, it is incomplete and will not work correctly with commands whose behaviour is dependent on the Tcl call stack. We will revisit this later in Section 10.5.7.

3.5.5. Deleting a command

We can also use the rename command to delete a command. If the second argument to rename is an empty string, the command is deleted instead. We can use this to put things back the way they were.

```
% rename puts "" ❶
% puts "Hello world!" ❷
Ø invalid command name "puts"
% rename builtin_puts puts ❸
% puts "Hello world!"
→ Hello world!
```

❶ Get rid of our version of puts
❷ Fails because the command has been deleted
❸ Restore the original built-in version of puts

3.5.6. Redefining commands

In our prior example, we renamed the command before creating our own version of it because we wanted to preserve the functionality of the original command. If that is not needed, we can just overwrite a command implementation simply by defining a command of the same name.

Suppose you had to write a procedure that needed some expensive one time initialization. You might write it as:

```
proc my_proc {} {
    global initialized
    if {![info exists initialized]} {
        set initialized 1
        puts "Pretend this is some expensive initialization"
    }
    puts "Now doing the real work"
}
```

We could write it like this instead redefining the procedure **within itself**.

```
proc my_proc {} {
    puts "Pretend this is some expensive initialization"
    proc my_proc {} {
        puts "Now doing the real work"
    }
    tailcall my_proc
}
```

The tailcall command is explained in Section 10.5.7.

Now let us call it twice and see it in action.

```
% my_proc
→ Pretend this is some expensive initialization
  Now doing the real work
% my_proc
→ Now doing the real work
```

Our procedure has gotten rid of both the `initialized` variable as well as an unnecessary check on every subsequent call. A generalized form of procedure self initialization is illustrated in Section 10.8.1.

Note that although we have used procedures in our examples above, the renaming, redefinition, deletion can be used for all command types. For example, renaming an object (see Chapter 14) to the empty string will destroy it. Also, the redefinition does not have to be of the same command type so you can define a procedure that will overwrite a C-implemented command of the same name.

Although Tcl allows redefinition of even the core commands like `set`, `proc` etc. you are strongly advised against doing so unless you **really** know what you are doing and can duplicate their **exact** behaviour.

You will find the renaming and redefining of commands commonly used in the Tk graphical toolkit where each GUI widget instance name is also a command. One technique for extending a widget is to rename its "owning" command and then define a new command of the same name that calls the renamed command while adding additional behaviours.

3.5.7. Enumerating commands: `info commands`

The `info commands` returns a list of names of commands visible in the current namespace context.

```
info commands ?PATTERN?
```

If `PATTERN` is not specified, the command returns the names of *all* visible commands. Otherwise, only those names matching `PATTERN` using the rules of `string match` are returned.

```
% info commands                      ❶
→ print_args tell socket subst write_file open eof ne pwd _SetupCawtPkgs print_file glob ...
% info commands co*                  ❷
→ coroutine concat continue
% info commands ::tcl::mathfunc::*  ❸
→ ::tcl::mathfunc::round ::tcl::mathfunc::wide ::tcl::mathfunc::sqrt ::tcl::mathfunc::sin...
% info commands ::tcl::*::*          ❹
```

❶ All commands visible in the current namespace
❷ All commands visible in the current namespace that start with `co`
❸ Commands in the namespace ::tcl::mathfunc
❹ Note that the namespace components in the pattern are *not* treated as wildcards so this returns an empty list

You can also use `info commands` to check for the existence of a command. For example, older versions of Tcl did not have an `lmap` command so you might see code of the form

```
if {[llength [info commands lmap]] == 0} {
    proc lmap {args} {
        # A fallback implementation of lmap
    }
}
```

If the lmap command existed, info commands would return a list containing lmap. If it returns a list of zero length instead, the command does not exist and the code defines a lmap command implemented in script.

3.5.8. Procedures

Procedures allow you to define new Tcl commands at the script level. We have already seen simple examples of procedures and we now delve into them in more detail.

Procedures may be *named* or *anonymous*. We will describe the former first before differentiating the latter.

3.5.8.1. Defining procedures: proc

Named procedures are defined with the proc command.

```
proc NAME PARAMS BODY
```

The command creates a new command called NAME replacing any command of that name if one exists.

The BODY argument is the Tcl script that implements the command defined by the procedure. The result of the command is the result of the last statement *executed* in BODY. This is not necessarily the last physical statement in BODY but may be a return or other control command.

> NAME may include *namespace qualifiers* in which case the procedure is defined within the corresponding namespace context. We postpone that discussion to Chapter 12.

Just another command

We will take a segue to reiterate that proc is itself just a command like any other, not a keyword or specially treated by the Tcl parser. Although by convention, the parameter definitions and the body arguments are braced, there is no such requirement imposed by Tcl. As far as it is concerned, the arguments to proc are evaluated with the same quoting and substitution rules as for any other command. For example, sometimes you may want define a procedure "on the fly" where you want Tcl's quoting and substitution rules to come into play. Let us define a procedure that will create *another* procedure that adds some fixed amount to a number.

```
proc make_adder {increment} { proc add$increment n "expr \$n + $increment" }
```

Then we can use it as follows.

```
make_adder 2   → (empty)
add2 3         → 5
make_adder 10  → (empty)
add10 20       → 30
```

Pay attention to the quoting and substitutions in the above and make sure you understand how it works. We will have much more to say about this type of code construction in Section 10.8 and Section 20.9.2.

3.5.8.2. Procedure parameters

PARAMS is a list of parameter definitions for the command, each element in the list corresponding to an argument that must be passed to the command when it is called (modulo some special cases described below). When invoked, each argument supplied by the caller is assigned to a variable named after the corresponding parameter.

A procedure definition may have any number of parameters, including zero.

```
proc likes {paramA paramB} {
    puts "I like $paramA and $paramB."
}
likes ham cheese
→ I like ham and cheese.
```

Passing a different number of arguments than the number of parameters in a procedure definition results in an error.

```
% likes ham
∅ wrong # args: should be "likes paramA paramB"
% likes ham cheese eggs
∅ wrong # args: should be "likes paramA paramB"
```

> Note the distinction we make between *parameters* and *arguments*. The former goes with the procedure definition. The latter refers to the values passed when the procedure is called. Each argument value is assigned to the corresponding parameter at the time of procedure invocation. Parameters are also referred to as *formal arguments*.

3.5.8.2.1. Default argument values

Each parameter definition in a parameter list is specified as a list of one or two elements. The first element in the list is the name of the parameter. The second element, if present, is the default argument value to assign to the parameter if the caller does not supply one. Thus we can rewrite our procedure as

```
proc likes {paramA {paramB jelly}} {
    puts "I like $paramA and $paramB."
}
```

Now if the command invocation does not supply a second argument, the default value of jelly is passed instead.

```
likes "peanut butter" → I like peanut butter and jelly.
likes ham cheese       → I like ham and cheese.
```

> Parameters without defaults must come before parameters that have a default specified; otherwise an error exception will be raised when the procedure is called.

3.5.8.2.2. Variable number of arguments

Some commands support an arbitrary number of operands, for example the list command which constructs a list whose elements are the arguments passed to it. In other cases, commands support various options that modify the behaviour of the command. In both cases, the command implementation has to be able to deal with an arbitrary number of arguments passed to it.

If the last parameter in a procedure definition is named args, any additional arguments in a call to the procedure beyond the number of parameters in the procedure definition are collected into a list. This list is then passed as the value of the args argument. Let us modify our example.

```
proc likes {paramA {paramB jelly} args} {
    puts "I like $paramA and $paramB."
    if {[llength $args] != 0} {
        puts "I also like [join $args {, }]"
    }
}
```

```
% likes "peanut butter"
→ I like peanut butter and jelly.
% likes ham cheese apples bananas broccoli
→ I like ham and cheese.
  I also like apples, bananas, broccoli
```

In the second call, ham and cheese are assigned to paramA and paramB respectively. The additional arguments apples, bananas and broccoli are collected into a list. If the list is not empty, we print out the second line.

For this variable argument list feature, the args parameter must be the *last* parameter. Otherwise it has not special significance.

3.5.8.2.3. Named parameters

Sometimes a procedure has a large number of parameters, many of which are optional. In such cases, it can be awkward to call the procedure, having to remember the position of the parameters and supplying values for any preceding parameters even when their defaults are adequate. For example, consider the following procedure definition:

```
proc fontify {text {family Arial} {style normal} {weight medium} {size 10}} {
    return "<span font-family='$family' font-style='$style' \
              font-weight='$weight' font-size=$size>$text</span>"
}
```

Calling this procedure for a non-default style and font size requires all arguments to be specified even though most use the default.

```
% fontify "Some text" Arial italic medium 12
→ <span font-family='Arial' font-style='italic'  font-weight='medium' font-size=12>Some t...
```

Some languages have *named parameters* to deal with this problem so the above procedure can be called as

```
fontify "Some text" size 12 style italic
```

Note that the order of optional arguments is immaterial as they are identified by parameter name.

Although Tcl does not have built-in named parameters, we can achieve similar results through the args facility as below.

```
proc fontify {text args} {
    lassign {Arial normal medium 10} family style weight size
    if {[llength $args] & 1} {
        error "No value specified for parameter [lindex $args end]"
    }
    foreach {param val} $args {
        set $param $val
    }
    return "<span font-family='$family' font-style='$style' \
              font-weight='$weight' font-size=$size>$text</span>"
}
fontify "Some text" size 12 style italic
→ <span font-family='Arial' font-style='italic'  font-weight='medium' font-size=12>Some t...
```

We initialize the local variables to defaults using `lassign` and then loop through the variable arguments overwriting the defaults. Other means of accomplishing the above include using arrays or dictionaries. For example, using dictionaries,

```
proc fontify {text args} {
    set opts [dict merge {
        family Arial style normal weight medium size 10
    } $args] ❶
    dict with opts {} ❷
    return "<span font-family='$family' font-style='$style' \
                font-weight='$weight' font-size=$size>$text</span>"
}
fontify "Some text" size 12 style italic
→ <span font-family='Arial' font-style='italic'  font-weight='medium' font-size=12>Some t...
```

❶ Merge default option value dictionary with provided arguments
❷ Copy dictionary elements into local variables

The above techniques have a couple of drawbacks. Errors like misspelt parameter names are not detected and introspecting parameters provides limited information. Some of the techniques in the next section may provide better solutions in this regard.

3.5.8.2.4. Option processing

In Tcl code, command options are far more prevalent than named parameters. They differ from named parameters in the following respects. By convention, they always start with a specific character, -. More importantly, options do not always have a value specified after them. The presence of the option itself acts as a boolean switch.

You can of course code up your own solution similar to what we did for named parameters in the previous section or use one of the scripted off-the-shelf implementations[3] available for the purpose. We go in a different direction for the following reason: though the scripted solutions are fine for parsing program options passed on the command line, they can be much slower than standard procedure calls for parsing arguments to procedures that are executed often. We will thus demonstrate options using the `parse_args`[4] extension which is implemented in C and much faster than Tcl-only solutions.

Let use redo our `fontify` procedure from the previous section except that we will change the style of the font using a boolean `-italic` option instead of a `-style` option-value pair. We should be able to call it as

```
 fontify "Some text" -italic -size 10
```

We will use the `parse_args::parse_args` command from the `parse_args` package to reimplement our procedure.

> A Tcl *package* is essentially a library of commands implementing functionality not in the Tcl core language. It can be implemented in many forms as we will detail in Chapter 13. We will make use of packages extensively throughout this book. For now it suffices to know that before the commands in the package can be used, it must be loaded with the package command as we do with the `parse_args` package in the code snippet below.

The command `parse_args` takes an *option descriptor* that describes the allowed options and their attributes. It then parses the arguments based on this descriptor setting local variables corresponding to the option names.

---

[3] http://wiki.tcl.tk/1730
[4] https://github.com/RubyLane/parse_args

```
package require parse_args
proc fontify {text args} {
    parse_args::parse_args $args {
        -family  {-default Arial}
        -italic  {-boolean}
        -size    {-default 10}
        -weight  {-default medium}
    }
    set style [expr {$italic ? "italic" : "normal"}]
    return "<span font-family='$family' font-style='$style' \
                font-weight='$weight' font-size=$size>$text</span>"
}
```

We will not describe the full option syntax here, see the documentation [5] of the extension for details. In our simple example, the -italic option is marked as boolean indicating it is boolean switch that takes the value 1 if specified and 0 if absent. The other options have defaults specified. We can then call our rewritten procedure as below.

```
% fontify "Some text" -italic -size 10
→ <span font-family='Arial' font-style='italic'  font-weight='medium' font-size=10>Some t...
% fontify "Some text" -slanted -size 10 ❶
Ø bad option "-slanted": must be -family, -italic, -size, or -weight
```

❶ Error - invalid option -slanted

Notice the automatically generated error message for the invalid option -slanted. The extension is flexible and has many other capabilities. See its documentation for details.

3.5.8.3. Returning from a procedure: return

The return command is more flexible and powerful than you might have seen in other languages. However, here we only describe its simplest forms. More advanced capabilities will be described in Section 11.3. Likewise, other mechanisms for terminating the execution of a procedure will also be described there.

The return command can be used within a procedure to stop its execution and return a result back to the caller. If an argument is supplied, it is used as the value to be passed back to the caller. If no argument is supplied, the return value defaults to an empty string.

```
proc signum {n} {
    if {$n < 0} {
        return -1
    } elseif {$n == 0} {
        return 0
    } else {
        return 1
    }
}
signum -5
→ -1
```

If the procedure does not have a return command, its execution is terminated after the last command in its body. The result of this command is returned as the result of the procedure. Thus the result returned by the double procedure below is the result of the expr command.

---

[5] https://www.tcl.tk/community/tcl2016/assets/talk33/parse_args-paper.pdf

```
proc double {n} { expr {2*$n} }
double 4
→ 8
```

3.5.8.4. Anonymous procedures: apply

There are many situations in programming, particularly the event driven style that is common in Tcl applications, where code is executed via a callback mechanism. Examples include callbacks when a timer goes off, the user clicks a mouse, a connection request arrives over the network and so on. In these cases where a "one-off" procedure is needed it is inconvenient to have to define a named procedure elsewhere to be used as a callback. In these situations, anonymous procedures provide a convenient alternative.

Why procedures are preferable to scripts as callbacks

There are a couple of advantages to passing a procedure as a callback as opposed to a script.

* The first is performance since procedures are compiled into a byte code form for faster execution.

* The second is that any non-trivial script will use variables whose names will "pollute" the callback context. For example, in the case of user interface callbacks from Tk which execute in the global context, they will result in extraneous global variables being created. This problem does not arise with procedures where any variables will be created in the procedure's own local context.

Since anonymous procedures have no name, there has to be some facility to actually invoke them. This facility is the apply command.

```
apply ANONPROC ?ARG ...?
```

The apply command takes a mandatory argument — the anonymous procedure — and invokes it passing it any additional arguments that are supplied. The anonymous procedure ANONPROC itself is a list containing two or three arguments:

* the parameter definitions in the same form as for a named procedure

* the body of the procedure, again as for a named procedure

* optionally, an identifier for the namespace (see Chapter 12) in which the anonymous procedure is to be defined.

We will illustrate with a simple example using the lsort command. This command is described in detail with its myriad options in Chapter 5 but here it suffices to know that it sorts lists and has an option, -command, which can be used to specify how elements in the list are to be compared. Let us assume we want to sort a list of integers based on their **absolute** values. The option -command takes a script which is invoked by lsort to compare pairs of elements. When this script is invoked, two additional arguments are appended to it — the elements to be compared. The script has to return -1, 0 or 1 depending on whether the first element is less than, equal or greater than the second element. We first create our list.

```
set list_of_ints {-1 5 -5 10 -5 -1000 100} → -1 5 -5 10 -5 -1000 100
```

We can then sort this with a custom anonymous procedure that does comparisons based on absolute values. Note the similarity of the argument to apply to the structure of a named procedure definition: a list containing the parameter definitions {a b} followed by the procedure body.

```
% lsort -command {apply {
    {a b}
    {
        if {[abs $a] < [abs $b]} { return -1 }
```

```
            if {[abs $a] > [abs $b]} { return 1 }
            if {$a < $b} { return -1 }
            if {$a > $b} { return 1 }
            return 0
        }
}} $list_of_ints
→ -1 -5 -5 5 10 100 -1000
```

Nevertheless, the above looks slightly clumsy so very often a helper procedure, lambda, is defined for constructing anonymous procedures.

```
proc lambda {params body args} {
    return [list ::apply [list $params $body] {*}$args]
}
```

Then the above sort can be written as

```
lsort -command [lambda {a b} {
    if {[abs $a] < [abs $b]} { return -1 }
    if {[abs $a] > [abs $b]} { return 1 }
    if {$a < $b} { return -1 }
    if {$a > $b} { return 1 }
    return 0
}] $list_of_ints
→ -1 -5 -5 5 10 100 -1000
```

which looks cleaner. This construct is used so often that Tcllib[6] contains a package lambda that defines this lambda command for you[7].

Although not relevant to our discussion of anonymous procedures, in case you are wondering, the third and fourth lines

```
if {$a < $b} { return -1 }
if {$a > $b} { return 1 }
```

are not superfluous. We want the comparison function to be consistent given the same pair of elements so 5 should always be deemed greater than -5 (not even equal) irrespective of whether the arguments are passed as 5 -5 or -5 5.

Could we have written the above as a named procedure? Of course. The choice is a personal preference. Defining a named proc means one more suitable name to think of[8], potentially less clarity compared with inline code and so on.

3.5.8.5. Introspecting procedures: info procs|args|default|body

While info commands, which we saw earlier, returns the list of commands visible in the current context, the info procs command only returns names of commands implemented as procedures.

```
info procs ?PATTERN?
```

If PATTERN is specified, it is matched using the rules of string match.

---

[6] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html
[7] The name lambda comes from Lambda Calculus and generically refers to anonymous functions in programming languages.
[8] This is not a trivial problem when there are a lot of callbacks, often with similar functionality, in an application!

```
info procs       → print_args write_file _SetupCawtPkgs print_file auto_load_index likes unknown my_proc
_...
info procs tcl* → tclPkgUnknown tclPkgSetup tclLog
```

A number of commands return detailed information about a specific procedure.

```
info args PROCNAME
info default PROCNAME PARAMNAME VARNAME
info body PROCNAME
```

The info args command returns a list containing the names of the parameters defined for a procedure. We will use our previously defined likes procedure as the procedure of interest.

```
info args likes → paramA paramB args
```

Note this only returns the *name* of each parameter, not the entire parameter definition. To get the defaults associated with a parameter, we have to use the info default command. If the parameter has a default, the command returns 1 and stores the default in the specified variable.

```
info default likes paramA paramA_default → 0
info default likes paramB paramB_default → 1
puts $paramB_default                      → jelly
```

Lastly, the info body command returns the entire body of the procedure.

```
% info body likes
→
     puts "I like $paramA and $paramB."
     if {[llength $args] != 0} {
         puts "I also like [join $args {, }]"
     }
```

We can use these commands to entirely reconstruct a procedure definition at run time without access to source.

```
proc reconstruct {proc_name} {
    set proc_name [uplevel 1 [list namespace which -command $proc_name]]
    set params [lmap param_name [info args $proc_name] {
        if {[info default $proc_name $param_name defval]} {
            list $param_name $defval
        } else {
            list $param_name
        }
    }]
    return [list proc $proc_name $params [info body $proc_name]]
}
```

We can then call it to reconstruct any procedure from the runtime environment.

```
% reconstruct likes
→ proc ::likes {paramA {paramB jelly} args} {
     puts "I like $paramA and $paramB."
     if {[llength $args] != 0} {
         puts "I also like [join $args {, }]"
     }
...Additional lines omitted...
```

There are a couple of commands in the `reconstruct` procedure that we will not look at until later so a short explanation is in order. The first line converts the supplied procedure name to a fully qualified name in case the procedure is defined in a namespace. The `lmap` command command, described in Chapter 5, loops through a list and constructs a new list containing the result from each iteration.

This kind of reconstruction is useful in Tcl tools like profilers and debuggers as well as in metaprogramming constructs like macros. Some Tcl packages, like `pipethread`, even use similar methods to ship code to other remote interpreters as part of an RPC-like mechanism.

3.6. Variables

We have already seen basic usage of variables. It is time to go into more details including name syntax, scopes, visibility and commands related to variable management.

3.6.1. Variable assignment: `set`

The basic command for assigning a value to a variable is the `set` command that we have seen before.

```
set VARNAME ?VALUE?
```

If the *VALUE* argument is not supplied, the command returns the value of the variable named *VARNAME* if it exists and raises an error if it does not. Otherwise, if the variable exists, its value is replaced and if it does not, it is created and initialized. In all cases, the command returns the value of the variable.

```
set avar "Some value" → Some value
set avar               → Some value
```

> Strange though it may seem, there is no guarantee that the return value of the command (which is also the new value of the variable) is the specified *VALUE*! This is true not only for `set` but other commands that modify variables as well. This can happen when there are *traces* on the variable that modify it. See Section 10.6.1.

3.6.2. Getting a variable's value

We have already seen how the value stored in a variable is retrieved either by prefixing its name with the $ character or using the single argument form of the `set` command.

```
puts $avar       → Some value
puts [set avar]  → Some value
```

We now look at some variations of these.

For starters, suppose we wanted to write a (very simple minded) procedure to return a phrase that pluralized a word by tacking an s on the end. Clearly the following does not work because the Tcl parser will treat $nouns as a reference to the variable nouns as opposed to a reference to noun followed by a literal s.

```
proc pluralize {count noun} {return "$count $nouns"} → (empty)
pluralize 10 car                                     Ø can't read "nouns": no such variable
```

To fix this, we can delimit the variable name with a pair of braces.

```
proc pluralize {count noun} {return "$count ${noun}s"} → (empty)
pluralize 10 car                                       → 10 cars
```

In addition to this use to delimit the name when it is not terminated by a word separator, this braced form of variable reference is also useful in a couple of other situations. The first is when the variable name itself is of an unusual form as we will illustrate in the next section. The second is in conjunction with namespace names stored in variables which we will describe in Chapter 12.

Variable indirection

There is a situation that sometimes arises, that involves accessing the value of a variable *indirectly* through another variable that holds the name of the first. Consider the following

```
set avar "Some value" → Some value
set bvar "avar"        → avar
```

Given bvar, how does one retrieve the value whose name is stored in bvar? The following attempts fail.

```
puts $$bvar   → $avar
puts ${$bvar} Ø can't read "$bvar": no such variable
```

The first failure illustrates an important characteristic of the Tcl parser. When a substitution is made, it will never go back and reparse the string. Thus the parser never sees the substituted string $avar.

The second attempt fails because variables are never substituted inside the {} in the first place.

This is where the single argument form of the set command can be put to good use.

```
puts [set $bvar] → Some value
```

An alternative is to use upvar to create an alias, which is more convenient if the variable is referenced multiple times.

```
upvar 0 $bvar ref → (empty)
puts $ref         → Some value
```

This is a special case of the more generally applicable upvar command (see Section 10.5.4).

3.6.3. Variable name syntax

We have been using variable names in our examples that are alphabetic. However, unlike most other languages, there are practically no restrictions on the characters that can be used in a variable name. It is convenient for many reasons to restrict names to the "standard" alphanumeric plus underscore (_) convention but this is not mandated. You can include spaces, newlines, and practically any character you wish in a variable name as long as you take care to appropriately quote or escape it as per the Tcl parser rules discussed previously.

The following are all valid variable names.

```
set a_traditional_variable "A variable name"
set {Funky + Var # Name} "can be anything"
set "" "you like." ❶
puts "$a_traditional_variable ${Funky + Var # Name} ${}"
→ A variable name can be anything you like.
```

❶   Even an empty string can be a variable name!

Notice in the above example, the ${} variable syntax used for accessing variable names that contain funky [9] characters. However, there are cases where even this syntax cannot be used and you have to resort to using the set command to retrieve the value:

---

[9] A technical term.

```
set "{namewithbraces}" avalue → avalue ❶
puts ${namewithbraces}        Ø can't read "namewithbraces": no such variable ❷
puts ${{namewithbraces}}      Ø can't read "{namewithbraces}: no such variable
puts [set "{namewithbraces}"] → avalue ❸
```

❶ The variable name is {namewithbraces}, i.e. the name itself contains braces
❷ Fails. Cannot use $ in either form
❸ Resort to the single argument form of the set command

Needless to say, there is no reason to use weird variable names in your programs. However, the ability to do so is useful in dynamic languages like Tcl where the variable can be indirectly referenced as we saw in the previous section. For example, a network server may choose to store information associated with a client in a variable of the same name as the client's DNS name [10].

However, there are two special cases to keep in mind regarding variable names. The first is with regard to use of parenthesis in array syntax. The other is that although a colon (:) character can be used in a variable name,

```
set var:with:colon "A variable with single colons" → A variable with single colons
```

two or more *consecutive* colons in the name signify a variable in a namespace. Namespaces are described later in the chapter Chapter 12.

3.6.4. Unsetting variables

The unset command deletes one or more variables.

```
unset ?-nocomplain? ?--? ?VARNAME ...?
```

The command destroys all specified variables. Unless the -nocomplain option is provided, the command will raise an error if the variable does not exist.

```
set avar "Some value"  → Some value
unset avar             → (empty)
unset avar             Ø can't unset "avar": no such variable
unset -nocomplain avar → (empty)
```

The -- character sequence is used to disambiguate the -nocomplain option from a variable of the same name.

```
% set -nocomplain "-nocomplain is a perfectly valid variable name"
→ -nocomplain is a perfectly valid variable name
% unset -nocomplain
% set -nocomplain ❶
→ -nocomplain is a perfectly valid variable name
% unset -- -nocomplain
% set -nocomplain ❷
Ø can't read "-nocomplain": no such variable
```

❶ The variable still exists because the command treats -nocomplain as an option with no variables specified!
❷ Now it is gone because the -- marked the end of options, resolving the ambiguity.

---

[10] Not to suggest that is necessarily a good idea. Using arrays or dictionaries would be better.

3.6.5. Variable scopes, lifetimes and visibility

A variable's *scope* is the region within which a variable is defined and can be referenced without special qualification. Tcl defines three scopes:

- *local* scope where a variable is defined within a procedure
- *namespace* scope where a variable is defined within a namespace. This is described in Chapter 12 and we will not say more about it here.
- *global* scope where the variable is defined outside any procedure or namespace.

3.6.5.1. Local variables

Local variables are defined within a procedure. There is no special declaration to declare them as local. They are automatically created as local when the variable name is assigned to and there is no global or variable command within the procedure that declares them to be global or within a namespace. The parameters defined for a procedure are also local variables that are automatically assigned from the arguments when the procedure is called.

Local variables can also be accessed from other procedures called by the procedure where they are defined. The mechanisms for this are described in Section 10.5.4.

Local variables live until the procedure returns or they are explicitly destroyed with the unset command.

```
proc demo {} {
    set localvar "I am local"
}
demo
puts $localvar ❶
∅ can't read "localvar": no such variable
```

❶ Generates error because localvar is only defined within the demo procedure and destroyed when it returns.

3.6.5.2. Global variables: global

Global variables are defined outside any procedure. Any reference to an unqualified variable outside a procedure refers to the global variable. (We are ignoring variables in namespaces which we discuss in Chapter 12). The variables in our interactive examples in the Tcl shell were all created as global variables. Within a procedure or namespace, global variables have to be either qualified or declared as global.

Qualifying a global variable is done by prefixing the variable name with the :: character sequence. This is a special case of namespace qualification where the :: prefix indicates that the variable resides in the global namespace.

```
set globalvar "I am global"
proc demo {} {
    puts $::globalvar
}
demo
→ I am global
```

Without the :: qualifier, an error would have been raised.

Alternatively, the variable can be declared to be global with the global command.

```
global ?VARNAME ...?
```

The command declares its arguments to be names of global variables and creates local variables of the same name linked to the corresponding global variables. The variable does not have to be explicitly qualified and access to it result in the corresponding global variable being accessed. Thus the above procedure could also be written as

```
proc demo {} {
    global globalvar
    puts $globalvar
}
demo
→ I am global
```

Choosing qualification versus declaration with global is a personal preference. Use of explicit qualification immediately makes it clear at the point of reference that a global variable is being used. On the other hand, if you are into microoptimizations, using global is a tad more efficient if you reference the variable multiple times as in a loop.

Global variables exist from the time of definition until they are explicitly destroyed with the unset command.

One more note regarding the global command. In common usage, the command links a local variable to a global variable of the same name. However, if the argument to the command contains namespace qualifiers, the local variable created is linked to that namespace variable, not a global one.

```
namespace eval ns {
    variable avar "This is the namespace variable"
}
set avar "This is the global variable"
proc demo {} {
    global ns::avar
    puts $avar
}
demo
→ This is the namespace variable
```

3.6.5.3. Creation is not definition

We have so far used the words "creation" and "definition" somewhat interchangeably. However, these are not the same so it is time to distinguish the two.

Following terminology from the Tcl reference pages, *creation* of a variable refers to creation of the variable **name** and associating it with a scope (local, global, namespace etc.). Commands such as global, or variable that we see in Chapter 12, perform this function. On the other hand, a variable is *defined* only when a value is assigned to it. The difference is illustrated in following short example where we use the info exists command to check if a variable has been defined.

```
proc demo {} {
    global created_but_undefined
    puts [info exists created_but_undefined]
    set created_but_undefined "a value"
    puts [info exists created_but_undefined]
}
demo
→ 0
  1
```

The global command only creates a local variable of that name linked to a global variable. However, since it has no value assigned to it, it is not defined as the output of info exists shows. Assigning a value results in the variable being defined. In the rare case that you really care about the variable creation rather than definition, you can use the namespace which command described in Section 12.5.4.

3.6.6. Variable introspection: `info exists|vars|locals|globals`

The `info exists` command can be used to check for existence of a variable within any scope.

```
info exists VARNAME
```

It returns 1 if a variable exists and 0 otherwise. Note *exists* means the variable is *defined*, i.e. has been created **and** has an associated value as noted in the previous section.

```
set globalvar "I am global"
proc demo {} {
    puts "localvar: [info exists localvar]"
    set localvar "I am local"
    puts "localvar: [info exists localvar]"
    puts "globalvar: [info exists globalvar]"
    puts "globalvar: [info exists ::globalvar]"
    global globalvar
    puts "globalvar: [info exists globalvar]"
}
demo
→ localvar: 0
  localvar: 1
  globalvar: 0
  globalvar: 1
  globalvar: 1
```

Notice from the above output that `info exists` follows the same rules regarding qualification and global declarations as any other variable reference.

The `info locals`, `info globals` and `info vars` commands can be used to enumerate local variables within a procedure, global variables, and all variables that are visible in the current scope respectively.

```
info locals ?PATTERN?
info globals ?PATTERN?
info vars ?PATTERN?
```

The script below illustrates their use.

```
proc demo {paramA} {
    set localvar "A local variable"
    puts "locals: [info locals]"
    puts "globals: [info globals]"
    puts "vars: [info vars]"
    global tcl_platform
    puts "vars: [info vars]"
}
demo "A parameter"
→ locals: paramA localvar
  globals: tcl_version tcl_interactive var globalvar fruit created_but_undefined nested b...
  vars: paramA localvar
  vars: paramA localvar tcl_platform
```

Some of the variables we see in the output are predefined in Tcl. Others were created as a result of commands executed in our earlier examples.

Notice that `info locals` includes the procedure parameters in its list. Also note the behaviour of `info vars`. Unlike `info globals`, it will only include global variables if they have been brought into the local scope with a

`global` declaration. Also, although not shown in our example, `info vars` will list namespace variables that have been brought into the local scope.

In all cases, if *PATTERN* is specified, only variables matching the pattern using `string match` rules are returned.

```
% info globals tcl_*
→ tcl_version tcl_interactive tcl_patchLevel tcl_platform tcl_library
```

If the pattern includes namespaces, only the last component of the namespace variable is treated as wildcard pattern. The namespace names are treated as literals. So for example,

```
info vars ns::* → ::ns::avar
info vars *::*  → (empty) ❶
```

❶   Returns empty list because the namespace component * is not treated as a wildcard but a literal namespace

3.6.7. Array variables

Many languages provide *array* constructs where values are stored as elements in a collection and referenced using a *key*. In languages like C, the elements have a specific sequence and the key must be an integer that specifies a position in this sequence. In other languages, including Tcl, the key is not restricted to integers. These arrays are also referred to as *maps* or *associative arrays* as they "map" or "associate" a value with a key.

In Tcl, arrays are actually not a collection of values but rather a *collection of variables*. They are denoted using the special variable syntax

```
ARRAYNAME(KEY)
```

where both the array name and key may be arbitrary strings.

Tcl also has *value* based keyed collections called *dictionaries*. We describe dictionaries, as well as contrast them with arrays, in Chapter 6.

3.6.7.1. Basic array operations

Because each element in an array is just a variable, they are used in the same fashion. We can access them with the $ prefix and use any commands like `set`, append, `incr` etc. that operate on variables to modify an element.

```
% set populations(Mumbai) 12500000
→ 12500000
% puts "The population of Mumbai is now $populations(Mumbai)."
→ The population of Mumbai is now 12500000.
% puts "Next year it will be [incr populations(Mumbai) 1000000]"
→ Next year it will be 13500000
```

The key (or index) need not be a literal string. We may use a variable or a bracketed command as well.

```
% set city "New York"
→ New York
% set populations($city) 8500000
→ 8500000
% parray populations
→ populations(Mumbai)   = 13500000
  populations(New York) = 8500000
```

3.6.7.2. Printing an array: `parray`

The `parray` command prints out the contents of an array.

```
parray ARRAYNAME ?PATTERN?
```

If *PATTERN* is not specified, the array prints on standard output all elements of *ARRAYNAME* based on the sort order of the keys. If *PATTERN* is specified, the commmand only outputs elements whose keys match *PATTERN* using the pattern matching rules of `string match`. The command is primarily intended for interactive use.

```
% parray populations
→ populations(Mumbai)   = 13500000
  populations(New York) = 8500000
% parray populations N*
→ populations(New York) = 8500000
```

3.6.7.3. Operating on multiple elements: `array set`, `array get`, `array unset`

Although individual array elements can be treated like any other variable, it is often convenient to operate on multiple elements at a time. Several subcommands of `array` are provided for this purpose.

The `array set` command assigns to multiple elements.

```
array set ARRAYNAME LIST
```

ARRAYNAME is the name of the variable which either does not exist or must be an array if it does. An error will be raised if *ARRAYNAME* is the name of an existing variable which is *not* an array.

LIST is a list of alternating key value pairs. Each value is assigned to the array element identified by the key, overwriting its value if it already exists and creating it otherwise.

```
array set populations {
    Moscow 12200000
    Lagos  17000000
    Mumbai 12500000
}
parray populations
→ populations(Lagos)    = 17000000
  populations(Moscow)   = 12200000
  populations(Mumbai)   = 12500000
  populations(New York) = 8500000
```

Correspondingly, the `array get` command returns multiple elements as a list of alternating keys and values.

```
array get ARRAYNAME ?PATTERN?
```

If *PATTERN* is not specified, the command returns all elements in the array. Otherwise, only those elements whose keys match *PATTERN* using `string match` rules are returned.

```
% array get populations
→ Moscow 12200000 Lagos 17000000 {New York} 8500000 Mumbai 12500000
% array get populations M*
→ Moscow 12200000 Mumbai 12500000
```

There is no guarantee regarding the order in which elements are returned. The order may not even be maintained across successive iterations. If a specific order is desired, you can use the `lsort` command.

```
% lsort -nocase -stride 2 [array get populations]              ❶
→ Lagos 17000000 Moscow 12200000 Mumbai 12500000 {New York} 8500000
% lsort -integer -index 1 -stride 2 [array get populations]    ❷
→ {New York} 8500000 Moscow 12200000 Mumbai 12500000 Lagos 17000000
```

❶   Sort by name
❷   Sort by population

See Section 5.7 for an explanation of the options.

Finally, while `unset` can be used with array elements as with other variables, `array unset` provides a means to unset multiple array elements.

```
array unset ARRAYNAME ?PATTERN?
```

If *PATTERN* is not specified, the entire array is unset. Otherwise, only those elements whose keys match *PATTERN* using `string match` rules are unset. If *ARRAYNAME* does not exist or is not an array, the command does not raise an error and has no effect.

```
array unset populations N*
parray populations
→ populations(Lagos)  = 17000000
  populations(Moscow) = 12200000
  populations(Mumbai) = 12500000
```

> Note the difference in the following commands
>
> ```
> array unset my_array *
> array unset my_array
> ```
>
> The first will remove all elements from the array but the array itself will continue to exist. In the second case, the array variable itself will be unset.

3.6.7.4. Checking for arrays: `array exists`

We can use the `array exists` command to check if a variable is an array. It returns 1 if a variable exists *and* is an array and 0 otherwise.

```
set scalar "some value"  → some value
array exists populations → 1
array exists scalar      → 0
array exists nosuchvar   → 0
```

3.6.7.5. Checking for element existence: `info exists`, `array names`

Since array elements are variables, the standard `info exists` command which can be used to check the existence of a variable can be also used to check for array elements. The command returns 1 if the element exists and 0 otherwise.

```
info exists populations(Mumbai) → 1
info exists populations(London) → 0
```

The `array names` command returns a list of keys for the elements in an array[11].

---

[11] The Tcl reference documents uses the nomenclature *names* for keys in an array and *keys* for a dictionary. We stick to using *keys* for both cases.

```
array names ARRAYNAME ?MODE? ?PATTERN?
```

If *PATTERN* is not specified, the command returns the keys for all elements in the array. Otherwise, only keys that match *PATTERN* are returned. The matching method depends on the *MODE* option. If it is unspecified or has the value -glob the matching is done using string match rules. If *MODE* is -regexp, the matching is done as for regular expressions. If *MODE* is -exact, only the key that exactly matches *PATTERN* is returned if it exists.

The command returns an empty list if no matching elements are found, or if the array variable does not exist or is not an array.

```
array names populations            → Moscow Lagos Mumbai
array names populations M*          → Moscow Mumbai
array names populations -regexp o..o → Moscow
```

3.6.7.6. Array statistics: `array size`, `array statistics`

The array size command returns the number of elements in an array. In case the specified variable does not exist or is not an array, the command returns 0.

```
array size populations → 3
array size nosuchvar   → 0
```

The array statistics command is rarely used in practice and we include it here only for completeness. It prints detailed internal statistics about the hash tables used to implement arrays.

```
% array statistics populations
→ 3 entries in table, 4 buckets
  number of buckets with 0 entries: 1
  number of buckets with 1 entries: 3
...Additional lines omitted...
```

Its primary use is development of the array implementation itself and possibly to diagnose pathological behaviour with very large arrays.

3.6.7.7. Iterating over arrays: `array startsearch|nextelement|anymore| donesearch`

Tcl has a very flexible iteration command for lists, foreach that can be used to iterate over arrays by retrieving their content in list form using array names or array get.

```
foreach city [array names populations] {
    puts "The population of $city is $populations($city)"
}
→ The population of Moscow is 12200000
  The population of Lagos is 17000000
  The population of Mumbai is 12500000
```

Or in an alterative form,

```
foreach {city population} [array get population] {
    puts "The population of $city is $population"
}
```

These commands provide the most efficient and convenient means of iteration and are fully described in Section 5.9. Here we describe an alternate means specific to iterating over arrays.

The primary benefit of this alternative method is that there is no conversion into list format and thus memory requirements are much smaller. **This is only an issue when very large arrays are involved.**

The first step is to retrieve a handle to an iterator with the array startsearch command.

```
set iter [array startsearch populations] → s-1-populations
```

The array anymore command is used in conjunction with array nextelement, which retrieves the next element from the iterator, to loop over all the elements. It returns 1 if there are more elements left and 0 otherwise.

```
while {[array anymore populations $iter]} {
    set city [array nextelement populations $iter]
    puts "The population of $city is $populations($city)"
}
→ The population of Moscow is 12200000
  The population of Lagos is 17000000
  The population of Mumbai is 12500000
```

Finally, when the iteration has ended, the handle has to be released with array donesearch.

```
array donesearch populations $iter → (empty)
```

There are a few special cases to be aware of, like multiple parallel searches and deletion of elements in the middle of the iteration. See the Tcl reference manual for details.

It is worth reiterating that this method is slow, almost never required and should not be used unless you are dealing with arrays that are so large as to cause memory allocation failures if converted to lists.

3.6.7.8. More on array keys

There are some additional points to be noted about keys.

Key equality

Keys are strings. Thus the keys 1 and 0x1 point to different array elements though in numeric calculations they represent the same values. Additionally, keys are case sensitive. Thus keys abc and Abc point to different array elements.

Multiple dimensions

There is no built-in notion of multidimensional arrays. They are sometimes simulated by concatenating the multiple "indices" using some separator string and using the result as the array key. For example, the results of tennis matches may be stored using keys like Federer,Nadal. However, you need to be careful that the separator string itself does not occur in the index values as it would lead to ambiguities. Also remember that Federer,Nadal and Federer, Nadal (with a space before the N) are different keys so even extraneous whitespace will lead to erroneous results if not used consistently. For these reasons, dictionaries are preferable for such structures.

Empty strings as keys

As a piece of trivia, note that empty strings are acceptable for both the array and the key. So for example,

```
set (key) value → value  ❶
set arr() value → value  ❷
set () value     → value  ❸
```

❶ Array name is the empty string
❷ The key is an empty string
❸ Both the array name and key are empty strings

Keys containing whitespace

Earlier we assigned an element for the key United States via a variable reference. What if we wanted to assign to an element directly instead when the key included a space character? The following attempts lead to either errors or unexpected results.

```
% set populations(Hong Kong) 7300000
Ø wrong # args: should be "set varName ?newValue?"
% set populations("Hong Kong") 7300000
Ø wrong # args: should be "set varName ?newValue?"
```

The correct way to set the variable is either of

```
set populations(Hong\ Kong) 7300000  → 7300000
set "populations(Hong Kong)" 7300000 → 7300000
```

On the other hand, when referencing the variable we can use the natural syntax or the braced form of variable references.

```
puts $populations(Hong Kong)    → 7300000
puts ${populations(Hong Kong)} → 7300000
```

Explaining treatment of keys with whitespace

To understand this seeming inconsistency in treatment of key literals containing white space, we have to go back to the Tcl parser we discussed earlier.

When parsing the statement

```
set populations(Hong Kong) 7300000
```

the Tcl parser breaks it up into four words, set, then populations(Hong, followed by Kong) and finally 7300000. In essence, it does not treat the parenthesis as a special character. The set command raises an error on receiving three arguments when it expects only one or two.

On the other hand, in the case of a statement like the one below, variable substitution comes into play.

```
puts $populations(Hong Kong)
```

When Tcl sees the $, the variable substitution rules are triggered. These do understand the variable syntax used for array elements and treat all characters until the terminating parenthesis as a single word doing backslash, command and variable substitution in the process.

In practice, most array accesses are through variables and rarely does this inconsistency matter.

3.6.8. Predefined variables

Tcl predefines a number of global variables such as tcl_platform, tcl_version etc. You can get a complete list from the info globals command in the Tcl shell. We will describe these variables elsewhere in the sections related to their use.

3.7. Getting error information

Tcl has powerful mechanisms for dealing with errors and exceptions that we describe in Chapter 11. Here we only mention a couple of points that are useful to know when starting with Tcl in interactive mode.

We have already seen that on invalid input most Tcl commands will print an informational message that identifies the cause of the error. For example,

```
% binary decode hex
Ø wrong # args: should be "binary decode hex ?options? data"
% string size "foo"
Ø unknown or ambiguous subcommand "size": must be bytelength, cat, compare, equal, first,
    ↳ index, is, last, length, map, match, range, repeat, replace, reverse, tolower, totitle,
    ↳ toupper, trim, trimleft, trimright, wordend, or wordstart
```

In addition, if an error occurs in a nested procedure call, you can examine the global variable errorInfo for the call stack at the point the error occured.

```
% proc demo2 {x y} {}
% proc demo args { demo2 {*}$args }
% demo A B C
Ø wrong # args: should be "demo2 x y"
% puts $errorInfo ❶
→ wrong # args: should be "demo2 x y"
      while executing
   "demo2 {*}$args "
      (procedure "demo" line 1)
      invoked from within
   "demo A B C"
```

❶   Print the error stack

This can be very useful in diagnosing the root cause of an error.

> If you are using an enhanced Tcl console like tkcon, error messages are highlighted and clicking on them with the mouse will display the error stack in a popup window.

3.8. Introspection

> There are three things extremely hard: steel, a diamond, and to know one's self.
>
> — Benjamin Franklin

Luckily for us, the last part does not hold for Tcl. Tcl offers deep and comprehensive introspection capabilities into almost every aspect of its runtime. Introspection is useful in all kinds of situations ranging from metaprogramming, runtime debugging and tracing, construction of dynamic object systems and more. It is even useful in interactive development. For example, what arguments does our demo2 procedure take?

```
info args demo2 → x y
```

In most cases this information is available through the info command. We have already seen a few examples such as info procs and info globals. You can see all the other categories of information available by passing a bogus argument to info.

```
% info bogus
Ø unknown or ambiguous subcommand "bogus": must be args, body, class, cmdcount, commands,...
```

We will describe these introspection capabilities in detail in the sections pertaining to their subject.

3.9. The EIAS principle

The universe is a symphony of vibrating strings.

— Michio Kaku

Having looked at the basics of the language, we will now touch upon a core philosophy on which Tcl is based — EIAS (*Everything Is A String*). You will hear this referenced from time to time in various language discussions and sometimes used to denigrate Tcl (the horror!).

Let us dispense with this last because it arises from a misunderstanding of what EIAS means:

- EIAS does not mean Tcl only operates on strings with no facilities for numerics, structured data etc.
- EIAS does not mean that all data is internally stored in string form.
- EIAS does not mean operations on numbers and structures entail conversion back and forth from string forms.

Having looked at what EIAS is **not**, let us look at what it **is**.

- Every **value** has a string representation. A "string" as we see in the next chapter, is a finite sequence of characters supporting operations that return its length, indexing and so on. This also means that every value is automatically serializable.
- Every value that produces the same string representation must be treated by every command in exactly the same way no matter how those values were constructed. For example, a value with the string representation 100 may arise as the concatenation of the strings 10 and 0 or as the result of squaring the numeric value 10. The result of both operations must be treated by all commands in the same manner. A command requiring numeric operands cannot accept the second value and reject the first.
- Arguments to procedures, values stored in variables, etc. are **conceptually** passed as strings though the **implementation** may not do so for reasons of efficiency.
- Because of the above, there is no need for mechanisms such as templates or generics because all values are treated uniformly. Your hash table can contain any value without needing "type-specific" versions.
- Although everything is a string to Tcl, commands are free to operate only on a subset of values in the string universe. The arithmetic operations will only operate on the subset of values that represent numbers.
- A program element **can** also be a string. That includes, for example, procedure bodies. You can dynamically construct procedure definitions as strings and invoke them. However, not all program elements are strings. Namespaces, interpreters are not themselves strings though they have names that are. This does not violate EIAS because they are not **values**. Thus EIAS is perhaps better named as EVIAS (Every Value Is A String).

Much of Tcl's malleability and ease of programming comes from this uniform treatment of values proscribed by EIAS.

3.10. Chapter summary

In this chapter we introduced the basic elements of Tcl — the syntax, command evaluation, procedures, and variables. In the next few chapters, we will focus on the Tcl commands for manipulating data in various forms.

4

Strings

For the most part, you can think of strings in Tcl as a sequence of characters, or specifically, Unicode characters [1]. However, they are actually a sequence of Unicode code points, not characters, in the range U+0000 to U+FFFF. The difference arises because a character may map to more than one sequence of Unicode code points. For example, the character é may be represented as either the single code point U+00E9 or the sequence U+0065 (letter e) followed by U+0301 (combining acute accent). Tcl considers the two representations as distinct characters. Tcl currently only supports Unicode code-points up to U+FFFF (the `Basic Multilingual Plane', or BMP). Support for Unicode characters beyond this range is a work-in-progress.

4.1. String indices

Commands that manipulate strings often take arguments that indicate the character positions, called *indices*, in the string. These indices are 0-based so 0 references the first character in the string, 1 references the second and so on. As a special case, end can be used to signify either the last character of a string or the position *after* the last character depending on the specific command.

In addition, string indices may be specified in one of the special forms

```
INTEGER[+|-]INTEGER
end[+|-]INTEGER
```

where *INTEGER* may be either an integer literal or a variable containing an integer. The resulting expression is used as the index into the string.

> There must be no whitespace between the operands and the operator in these forms.

We will see examples of these various forms throughout this chapter.

4.2. Constructing strings

At some level, since all values in Tcl have a string representation, every command can be thought of as constructing a string! In this section, we describe those commands whose main purpose is to construct a string, not produce a string as a side effect of some other computation.

4.2.1. String literals

We have already seen the most basic forms of string construction using quotes and braces. To refresh your memory,

```
% set interjection Hello   ❶
```

---

[1] If the term Unicode and code points are unfamiliar to you, please see one of the many tutorials on the Web, such as the one from Joel on Software [http://www.joelonsoftware.com/articles/Unicode.html] or unicode.org [http://unicode.org/standard/tutorial-info.html]

```
→ Hello
% set greeting {$interjection World!} ❷
→ $interjection World!
% set greeting "$interjection World!" ❸
→ Hello World!
```

❶   No quotes needed if no whitespace or special characters
❷   No string interpolation inside braces
❸   String interpolation inside double quotes

To enter certain non-printable or control characters such as newline, backslash sequences can be used.

```
% set text "First line\nSecond line"
→ First line
  Second line
```

In the general case, any Unicode character can be entered using one of the Unicode escape syntaxes \x, \u or \U.

```
% set text "\x55\x6e\x69\u0063\u006f\u0064\U00000065"
→ Unicode
```

The details regarding these various forms were discussed earlier in Section 3.2.1 and Section 3.3.

We now look at the additional commands provided in Tcl for conveniently and efficiently constructing strings from other strings.

4.2.2. Concatenating strings: `string cat`

An alternative to string interpolation using literals is the `string cat` command.

```
string cat ?STRING ...?
```

It takes an arbitrary number of arguments and returns the string formed by their concatenation.

```
% proc demo {} {return bar}
% set var foo
→ foo
% string cat $var {[this is a literal string, not a command]} [demo]
→ foo[this is a literal string, not a command]bar
```

This is more convenient than string interpolation in some cases. In the above example for instance, literal interpolation would be a little awkward due to the need to protect the braced string from substitutions while allowing it for $var and [demo].

The command is also useful when we need to return a result from a script that is the concatenation of one or more strings. Here is an example using the lmap command to construct a list.

```
set la {abc def}
set lb {123 456}
lmap a $la b $lb {
    string cat $a " " $b
}
→ {abc 123} {def 456}
```

The lmap command (see Section 5.5.1) constructs a list whose elements are the result of successive evaluation of a script. In this simple example, we want to construct a new list whose elements are formed from the corresponding

elements of la and lb separated by a space. Using `string cat` to construct the script result as above is more convenient and lucid than the alternatives such as append.

4.2.3. Constructing with substitutions: subst

The `subst` command offers yet another flexible form of string interpolation.

```
subst ?-nobackslashes? ?-nocommands? ?-novariables? STRING
```

The command performs backslash, variable and command substitution on the *STRING* argument in the same manner as the Tcl command parser and returns the result. Variable references of the form `$var`, command invocations enclosed in [] and backslash sequences are all replaced.

```
% set var 2
→ 2
% subst {The sum $var+$var\t=\t[expr {$var+$var}]}
→ The sum 2+2    =       4
```

> **!**
>
> Make a note that when the `subst` command is invoked two rounds of substitution take place, first by the Tcl parser, and then by the `subst` command.
>
> ```
> subst "(\\t)" → (       ) ❶
> subst {(\\t)} → (\t) ❷
> subst {(\t)}  → (       ) ❸
> ```
>
> ❶ `subst` sees (\t) as Tcl parser does one round of substitution
> ❷ `subst` sees (\\t) as the braces prevent substitution by the Tcl parser
> ❸ `subst` sees (\t)
>
> The examples below use {} to prevent the Tcl command parser from making substitutions so as to make the `subst` command behaviour clear.

The `-nobackslashes`, `-nocommands` and `-novariables` options provide additional control over what forms of substitutions are carried out by `subst`. These options selectively prevent substitution of backslash sequences, command invocations and variables respectively.

```
% subst {The sum $var+$var\t=\t[expr {2+2}]}
→ The sum 2+2    =       4
% subst -nobackslashes {The sum $var+$var\t=\t[expr {2+2}]}
→ The sum 2+2\t=\t4
% subst -nocommands {The sum $var+$var\t=\t[expr {2+2}]}
→ The sum 2+2    =       [expr {2+2}]
% subst -novariables {The sum $var+$var\t=\t[expr {2+2}]}
→ The sum $var+$var      =       4
```

Of course, multiple options may be combined as desired.

> **⚠**
>
> There are some subtleties in the interaction among the various options to `subst` and in cases where commands return with a result code other than ok. For example, consider
>
> ```
> % subst -novariables {The sum $var+$var\t=\t[expr {$var+$var}]}
> → The sum $var+$var      =       4
> ```

and notice how the variables inside the expr expression have been substituted despite the -novariables option. See the Tcl reference documentation for such special cases.

Many Tcl libraries for generating HTML via templates are based on the subst command. The HTML page is constructed from one or more fragments containing a mixture of HTML and Tcl variables whose values (for example) have been retrieved from a database. The page is generated by passing it through subst. See substify[2] in the Tcler's Wiki[3] for one example implemented in just a few lines.

4.2.4. Formatting strings: format

The commands discussed so far construct strings in a somewhat "free-form" fashion using either their natural representation or how they were initialized.

```
% set sixteen 0x10 ; puts "sixteen is $sixteen"
→ sixteen is 0x10
% incr sixteen 0 ; puts "sixteen is $sixteen"
→ sixteen is 16
```

Sometimes however, you need to construct strings where the values have precise representation and structure. For example, you may need to write floating point values to a CSV file to be imported by another application which requires exactly two decimal places. Or you may need to generate a report where values with different widths have to be adjusted to a specific column width.

The format command lets you do precisely that by letting you specify details of how values are represented, their location in the constructed string, maximum lengths, padding and so on.

```
format FORMATSTRING ?arg1 arg2...?
```

Here *FORMATSTRING* is a "template" for the string to be constructed and contains literal text as well as *field specifiers* that are placeholders for the values supplied as arguments to the command. The command returns *FORMATSTRING* with the field specifiers replaced by the argument values, appropriately formatted. For example,

```
% format "%d times %#x is %e" 10 10 100
→ 10 times 0xa is 1.000000e+002
```

Here %d, %#x and %e are field specifiers that control how the numbers are formatted.

A field specifier controls the representation, widths etc. of the corresponding argument value and consists of the parts or components listed below.

- A literal % character
- An optional XPG3 specifier
- An optional sequence of flag characters
- An optional minimal width
- An optional precision or bound
- An optional size modifier
- The conversion character

Note that all the parts above are optional except the starting % and conversion character. All parts that are present must be in the order listed.

We illustrate each of the above in turn with some examples.

---

[2] http://wiki.tcl.tk/18455
[3] http://wiki.tcl.tk

4.2.4.1. Conversion characters

The conversion character controls the type of conversion to be applied to the corresponding argument. In the simplest case, the format string includes only the conversion characters and no optional parts.

The conversion specifiers may be classified as string, integer and floating point depending on the type of value to be formatted. The string and character conversion characters are shown in Table 4.1.

Table 4.1. String specifiers for `format`

Character	Description	Example
s	Format as string (so effectively as-is). Useful with modifiers like width etc.	`format %s 0xffffffff → 0xffffffff`
c	Character corresponding to the Unicode code point given by the integer argument value.	`format %c 42      → *` `format %c 0x662D → 昭`
%	Inserts the percent character itself.	`% format "Tcl! %d%% pure fun!" 100` `→ Tcl! 100% pure fun!`

The format specifiers for integer values are shown in Table 4.2.

Table 4.2. Integer specifiers for `format`

Character	Description	Example
d	Signed decimal integer	`format %d 0xffffffff → -1` `format %d 42          → 42`
u	Unsigned decimal integer	`format %u 0xffffffff → 4294967295`
x	Lower case hexadecimal integer	`format %x 42 → 2a`
X	Upper case hexadecimal integer	`format %X 42 → 2A`
o	Octal integer	`format %o 42 → 52`
b	Binary integer	`format %b 42 → 101010`

Finally, the specifiers for formatting numbers in floating point representation are shown in Table 4.3.

Table 4.3. Floating point specifiers for `format`

Character	Description	Example
f	Signed decimal	`format %f 4.2e1 → 42.000000`
e	Scientific representation	`format %e 42 → 4.200000e+001`
E	Scientific representation	`format %E 42 → 4.200000E+001`

Character	Description	Example
g	Behaves as f or e depending on argument value except for trailing 0's and decimal point. See Tcl reference.	format %g 420e-1 → 42
G	Behaves as f or E depending on argument value except for trailing 0's and decimal point. See Tcl reference.	format %G 42e0 → 42

4.2.4.2. XPG3 format position specifiers

Normally, the format specifiers and supplied arguments are matched up in the order they occur. For example, in the command below %d and %s get matched up in order with 31 and January respectively.

```
% format "There are %d days in %s." 31 January
→ There are 31 days in January.
```

However, there are circumstances where you want to be able to change the order in which argument values are inserted without changing the order in which they are passed. An example of this is formatting of strings localized for different languages. The way localization is commonly done is by passing an identifier string into the message catalog facility (see Section 4.15), msgcat, which returns the appropriate string for that language. The location of the insertions then is dependent on the grammar for the language. When using the format command as above however, we do not know the order of the specifiers and therefore could very well pass the arguments in the wrong order. For example, assume this naive procedure to print a localized message for the days in a month.

```
set english "There are %d days in %s." ❶
set canadian "%s has %d days, eh!"
proc print_days {fmt month days} {puts [format $fmt $days $month]}
print_days $english January 31
→ There are 31 days in January.
```

❶   Assume returned from localized message catalogs

This worked fine for English because the order of arguments matches the order in the message catalog string. However, we have problems in Canada.

```
% print_days $canadian January 31
∅ expected integer but got "January"
```

Clearly that is not workable because the argument order no longer matches the specifiers in the catalog string.

The XPG3 position specifiers address this issue. A position specifer immediately follows the leading % and consists of a number followed by a $ character. This number indicates the position of the corresponding argument in the list of arguments.

The message catalog strings in the above example then should have been written as follows using XPG3 specifiers.

```
set english {There are %1$d days in %2$s.}
set canadian {%2$s has %1$d days, eh!}
```

Now the order of arguments that is passed to format is fixed while still allowing for the insertions to take place in a different order. The Canadians are now happy.

```
print_days $english January 31  → There are 31 days in January.
print_days $canadian January 31 → January has 31 days, eh!
```

Note that an argument index may be repeated if desired. For example,

```
% format {%1$d == 0x%1$x == 0o%1$o} 42
→ 42 == 0x2a == 0o52
```

repeats a single integer argument thrice with different formats.

 If a format string uses XPG3 position specifiers, **all** specifiers in the format string must include XPG3 position specifiers. Breaking this rule will generate an error exception.

4.2.4.3. Specifying minimum field widths

Although the flags component comes before the minimum field width in a field specifier, we describe the latter first as the flags act as modifiers for minimum widths.

The minimal width part of a specifier mandates a minimal number of characters in the inserted argument value and is particularly useful when formatting data in tabular form where the representation width has to match the desired width for a table column. The width can be specified as either a number or the * character which indicates the width is indirectly supplied as an additional argument.

```
format "(%d)" 10    → (10)
format "(%8d)" 10   → (      10)
format "(%*d)" 8 10 → (      10) ❶
```

❶   Field width 8 supplied as an additional argument

In the above examples, we have enclosed the format string in parentheses to make it clear how fields are formatted in the presence of whitespace padding.

4.2.4.4. Format flags

The flags component of the specifier controls a variety of attributes as illustrated in the following examples.

The first set of flags deal with justification and pad characters.

- The - flag forces left justification when padding to meet minimum width requirements.
- The 0 flag implies padding with 0's instead of spaces.

```
format (%8d) 10  → (      10)
format (%-8d) 10 → (10      )
format (%08d) 10 → (00000010)
```

The next set of flags affects representation of positive numbers.

- The + flag causes positive numbers to be preceded with the + sign.
- A single space character for the flag specifies a single space before a number unless a sign is present.

```
format (%+d) 10    → (+10)
format "(% d)" 10  → ( 10)
format "(% d)" -10 → (-10) ❶
```

❶   Note no space in output because a sign is present.

Finally, the flag # modifies the representation in various ways depending on the underlying conversion character. For binary, octal and hexadecimal fields, the flag specifies an appropriate prefix is to be output, for example 0x

for hexadecimal. For decimal and floating point fields, it specifies that a decimal point is output even for whole integers.

```
format %#x 10 → 0xa
format %#X 10 → 0XA
format %#o 10 → 012
format %#b 10 → 0b1010
format %#g 10 → 10.0000
```

4.2.4.5. Precision specifier

The fourth part, also optional, of a conversion specifier consists of a period (.) followed by a number or a * character. In the latter case, the number is supplied through an additional argument to the command.

The semantics depend on the conversion being applied. For string and integer conversion, it specifies the maximum and minimum number of characters to be printed respectively.

```
format %.2s abc  → ab ❶
format %.5d 10   → 00010 ❷
format %.*d 4 10 → 0010 ❸
```

❶ At **most** two characters
❷ At **least** five characters
❸ At **least** four characters as specified by additional argument

For the e, E and F conversions, it specifies the number of digits to output to the right of the decimal point.

```
format %f 1.12999   → 1.129990 ❶
format %.2f 1.12999 → 1.13
```

❶ Note by default 6 digits printed

For g and G conversions, it specifies the *total* number of digits output (with some caveats - see the Tcl reference documentation).

```
format %g 1.12999   → 1.12999
format %.2g 1.12999 → 1.1
```

4.2.4.6. The size modifier

The last optional component is the size modifier which specifies the range that an integer argument is to be truncated to. It consists of one of the character sequences h, l and ll whose effect is shown below.

```
set val 7777777777777777777777777777777777777777 → 7777777777777777777777777777777777777777
format %d $val                                    → 1908874353 ❶
format %hd $val                                   → 7281 ❷
format %ld $val                                   → 3296802724926397553 ❸
format %lld $val                                  → 7777777777777777777777777777777777777777 ❹
```

❶ Default: truncate to an int value, generally 32 bits
❷ h: truncate to an 16-bit value
❸ l: truncate to an wide value, generally 64 bits
❹ ll: no truncation is performed

4.2.5. Joining strings with separators: `join`

Done with the complexities of the `format` command, we move on to the simple `join` command which constructs a string by concatenating the elements of a list with a specified separator string placed between every pair of elements.

```
join LIST ?SEPARATOR?
```

We will look at lists in detail in the next chapter, but here is an example of using `join`.

```
% set quote [list "I came" "I saw" "I conquered"]
→ {I came} {I saw} {I conquered}
% join $quote ", "
→ I came, I saw, I conquered
```

The separator is optional, and if unspecified defaults to a single space character.

```
% join $quote
→ I came I saw I conquered
```

You can also specify the empty string as the separator when concatenating strings with `join`.

```
% join $quote ""
→ I cameI sawI conquered
```

4.2.6. Repeating strings: `string repeat`

One final form of string construction is repetition via the `string repeat` command.

```
string repeat STRING COUNT
```

The command returns the result of concatenating *COUNT* repetitions of *STRING*.

```
% set title "Underlined title"
→ Underlined title
% puts "$title\n[string repeat - [string length $title]]"
→ Underlined title
  ----------------
```

> Tcl commands often have subcommands. We've already seen an example in the `info` command. The `string` command is another example, that contains subcommands for manipulating strings. A command with subcommands is known as an *ensemble* command.

4.3. Modifying strings

A number of commands deal with modification of strings by adding or deleting characters. Some of these actually modify the contents of a variable while others take a string value as argument and return a new modified string.

4.3.1. Appending in place: append

The first of these is the append command which appends zero or more arguments to a *variable*.

```
append VAR ?STRING ...?
```

Unlike most other string related commands, note that this command alters the variable *VAR* **in place** in addition to returning the new string value. The command will create the variable if it does not already exist, effectively acting like the `set` command for that case.

```
% append newvar "Hello"      ❶
→ Hello
% set who "World"
→ World
% append newvar " " $who "!" ❷
→ Hello World!
```

❶   Creates the variable newvar if it does not exist
❷   append can take multiple arguments

Note that we do not use a dollar-sign when passing the variable newvar to this command. This is because the command expects the *name* of a variable, rather than the *value* contained in it.

> The above could also have been written making use of string interpolation as
>
> ```
> set newvar "$newvar $who!"
> ```
>
> which might even be clearer to read. The benefit of the append though is that it is significantly more efficient in both memory and CPU, particularly when long strings are involved.

4.3.2. Replacing substrings by position: `string replace`

The command `string replace` replaces a range of characters in a string with another string.

```
string replace STRING FIRST LAST ?REPLACEMENT?
```

The command returns a new string constructed by replacing the substring of *STRING* at indices *FIRST* to *LAST* with the string *REPLACEMENT*.

```
% string replace "Hello, World!" 0 4 Goodbye
→ Goodbye, World!
```

To replace substrings by content instead of by position, see the `string map` or `regsub` commands later.

4.3.3. Deleting substrings by position

There is no explicit command in Tcl to delete characters from a string. The `string replace` command can be used to delete a range of characters by not specifying a replacement string.

```
% string replace "Hello, World!" 5 end-1
→ Hello!
```

To delete occurrences of one or more substrings by content instead of by position, see the `string map` command later.

4.3.4. Deleting repeated characters at end: `string trim|trimleft|trimright`

A specialized form of deletion is provided by the `string trimleft`, `string trimright` or `string trim` commands.

```
string of STRING ?CHARS?
```

Here *OP* may be one of `trimleft`, `trimright` or `trim`. These trim any occurrences of a given set of characters from the start, end or both sides of a string respectively and return the result. The most common use of these commands is to trim leading and trailing whitespace from a string.

```
% set s "\t   Hello, World    \n"
→            Hello, World

% string trimleft $s
→ Hello, World

% string trimright $s
→            Hello, World
% string trim $s
→ Hello, World
%
```

However, any set of characters can be trimmed by providing an additional optional argument.

```
% string trimleft "Hello, World!" "lHe!"
→ o, World!
```

Note that the second argument is considered as a set of characters rather than a string.

The `textutil` package in Tcllib[4] offers more flexible versions of these built-in commands that permit specification of a regular expression that controls the trimmed characters.

4.4. Comparing strings

4.4.1. Comparing for equality: `string equal`

The `string equal` command compares two strings for equality.

```
string equal ?-nocase? ?-length COUNT? STRING1 STRING2
```

The command returns 1 if the two strings are identical and 0 otherwise. The comparison is case-sensitive by default. The `-nocase` option makes it case-insensitive instead.

```
set s Hello                   → Hello
string equal $s Hello         → 1
string equal hello $s         → 0
string equal -nocase hello $s → 1
```

You can use the `-length` option to indicate that only *COUNT* number of initial characters of the strings are to be compared.

```
string equal "Hello World!" "Hello Universe!"           → 0
string equal -length 5 "Hello World!" "Hello Universe!" → 1
```

4.4.2. Ordering strings: `string compare`

Instead of comparing for equality, you can also compare two strings for lexicographical ordering using the `string compare` command.

---

[4] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

```
string compare ?-nocase? ?-length COUNT? STRING1 STRING2
```

The command returns -1, 0 or 1 depending on whether the first argument is lexicographically less than, equal, or greater than the second.

```
string compare abcd BCDE           → 1
string compare -nocase abcd BCDE → -1
string compare 2 10                → 1 ❶
```

❶   Compared as strings, **not** numbers

The -nocase and -length options specify a case-insensitive comparison and a maximum character count just as for the string equal command.

4.5. Locating and extracting substrings

4.5.1. Locating substrings: `string first|last`

The string first and string last return the location of a substring within a string.

```
string first NEEDLE HAYSTACK START
string last NEEDLE HAYSTACK START
```

The former returns the location of the first occurence of NEEDLE in the HAYSTACK argument while the latter returns the last (effectively searching from the end). The commands return the string index of the first character of occurence if found and -1 otherwise.

```
string first "da" "Madam, I'm Adam" → 2
string last "da" "Madam, I'm Adam"  → 12
```

The commands accept an additional optional parameter START that controls where the search begins.

```
string first "da" "Madam, I'm Adam" 3     → 12
string last "da" "Madam, I'm Adam" end-5 → 2
```

Tcl has additional facilities for searching and locating substrings based on regular expressions. These are sufficiently powerful and flexible as to deserve their own sections and are described in Section 4.12.

> The string command has two additional subcommands, wordstart and wordend, that can be used to locate characters and substrings. However, these are deprecated and we do not describe them here.

4.5.2. Retrieving a character by position: `string index`

The command string index returns the character at a specified position in a string.

```
string index STRING INDEX
```

The INDEX argument specifies the position, starting with 0, in STRING. It can take any of the forms specified in Section 4.1.

```
set pos 4                        → 4
string index "Hello, World!" $pos   → o
string index "Hello, World!" end    → !
```

```
string index "Hello, World!" $pos+3 → W
string index "Hello, World!" end-5  → W
```

4.5.3. Retrieving substring ranges: `string range`

The related command `string range` returns a range of characters between two indices in a string.

```
string range STRING FIRST LAST
```

The returned string includes all characters between, and including, indices *FIRST* and *LAST* in *STRING*. Like `string index`, the command also accepts the special syntax for indexing from the end of the string.

```
string range "Hello, World!" 0 4        → Hello
string range "Hello, World!" $pos+2 end →  World!
```

The commands here extract substrings based on their position. For extracting substrings based on content, see Section 4.12.1.

4.6. Transforming strings

Several `string` subcommands described here return a string by applying some transform on a given string.

4.6.1. Replacing substrings: `string map`

Previously we described the `string replace` command that replaces a range of characters in a string based on position. Another form of replacement is provided by `string map`.

```
string map ?-nocase? MAPPING STRING
```

Rather than replacing by position, this command allows replacement of all occurences of one or more substrings within a string. The *STRING* argument is the string in which the replacement is to be done. The *MAPPING* argument is a list of alternating elements, the first being the substring to be replaced and the second being the corresponding replacement value. The command replaces all occurences of the former with the latter and returns the result.

```
% string map {ab Q cd XYZ} abacdabccd
→ QaXYZQcXYZ
```

Like the `string replace` command, `string map` can also be used for deleting characters. Setting the replacement value to an empty string will result in deletion of all occurences of the substring.

```
% string map {bc ""} abcdabcbdabc
→ adabda
% string map {rma {o} o {}} "Hello Norma!"
→ Hell No!
```

This last example illustrates another point about `string map` semantics. The target string is iterated over exactly once. At each position the mapping list is searched in sequence and the first match, if found, is replaced. This replaced substring is not matched again against the mapping list. So after rma is replaced with o, the o itself does not get replaced with an empty string.

A related point is that the order of strings in the mapping list is important since matches are checked in that order. Thus if one match string is a prefix of another, the latter should appear **first** else it will never match.

```
string map {bc XY bcd XYZ} abcdabcbdabc → aXYdaXYbdaXY
string map {bcd XYZ bc XY} abcdabcbdabc → aXYZaXYbdaXY
```

The string comparisons are case-sensitive unless the the -nocase option is specified.

```
string map {bC XY} abcdabcbdabc          → abcdabcbdabc
string map -nocase {bC XY} abcdabcbdabc → aXYdaXYbdaXY
```

A more flexible but less efficient command, regsub, that has similar functionality based on regular expressions is described in Section 4.12.2.

4.6.2. Changing character case: `string tolower, toupper, totitle`

A third set of string transform commands, string tolower, string toupper and string totitle pertain to character case.

```
string tolower STRING ?FIRST? ?LAST?
string toupper STRING ?FIRST? ?LAST?
string totitle STRING ?FIRST? ?LAST?
```

The meaning of the first two should be obvious. The last capitalizes the first letter in the string and changes all remaining letters to lower case.

```
string tolower "Hello, World!" → hello, world!
string toupper "Hello, World!" → HELLO, WORLD!
string totitle "hELLO, WORLD!" → Hello, world!
```

The optional *FIRST* and *LAST* arguments take the form of string indices and specify the indices of the substring to be modified within *STRING*.

```
string tolower "Hello, World!" 0 4       → hello, World!
string tolower "Hello, World!" 7         → Hello, world!
string toupper "Hello, World!" end-5 end → Hello, WORLD!
string totitle "Hello, World!" 1 end     → HEllo, world!
```

4.6.3. Reversing a string: `string reverse`

The string reverse command transforms a string by reversing the order of characters.

```
string reverse STRING
```

For example, Napolean's lament

```
% string reverse "able was I ere I saw elba"
→ able was I ere I saw elba
```

Hmm... probably not a good example!

4.6.4. Wrapping text: `textutil::adjust, textutil::indent`

The Tcl core does not have any built-in commands for wrapping and indenting multiple lines of text. The adjust and indent commands in the textutil::adjust module of Tcllib[5] may be used for this purpose instead.

```
package require textutil
set text [textutil::adjust {
    The adjust command has number of options that let you control
```

---

[5] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

```
      justification, line length and hyphenation.
} -length 40]
→ The adjust command has number of options
  that let you control justification, line
  length and hyphenation.
```

The indent command will indent each line in a string by prefixing it with the supplied argument.

```
% textutil::adjust::indent $text "...."
→ ....The adjust command has number of options
  ....that let you control justification, line
  ....length and hyphenation.
```

The undent command will undo the effect by removing the prefix that is common to all lines.

```
% textutil::adjust::undent $text
→ The adjust command has number of options
  that let you control justification, line
  length and hyphenation.
```

In combination, these three commands let you create wrapped text in various forms. See the Tcllib [6] reference documentation for details of working and options for these commands.

4.7. Parsing strings: scan

There are two Tcl commands that are commonly used in parsing. One of them, regexp, is based on regular expressions and we describe it in Section 4.12.1. The other one, scan, is similar to the sscanf library function in C and described here.

The format command we saw earlier generates a string composed from input values formatted as per a specification. Conversely, the scan command provides a means to parse strings that are known to be in a specific format, converting its substrings to values of a specific type.

The command takes one of two forms, one where the parsed values are returned as the result of the command and the other where they are stored in variables.

```
scan INPUTSTRING FORMATSTRING
scan INPUTSTRING FORMATSTRING VAR1 ?VAR2 …?
```

In both forms, *INPUTSTRING* is the string to be parsed while *FORMATSTRING* controls the parsing. The command works by iterating over each character in *FORMATSTRING* and matching it against *INPUTSTRING* as follows:

- If the format character is a space or a tab, the command skips over zero or more consecutive whitespace characters in the input string.
- If the format character is %, it is the start of a conversion specifier. The input string is parsed based on the specifier and the value extracted as per specifier type. This is detailed below.
- Any other format character must exactly match the character in the input string in which case the scan continues with the next character. Otherwise the scan is ended and any remaining characters in *INPUTSTRING* are ignored. Note that this is not treated as an error.

If the first form of the command is used where only two arguments are present, the extracted values are returned as the result of the command. We will refer to this as the inline form. In the second form, the additional arguments are treated as names of variables into which the extracted values are to be stored. In this case, the return value from the command is the number of conversions performed. For example,

---

[6] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

```
% scan "pi     =  3.14159" "%s = %f"
→ pi 3.14159
% scan "pi     =  3.14159" "%s = %f" name value
→ 2
% puts "The value of $name is $value."
→ The value of pi is 3.14159.
```

The parsing in the above example proceeds as follows:

- the %s format string is matched with pi in the input string
- the space character is matched against multiple spaces in the input string
- the = literal in the format string exactly matches the one in the input string
- spaces are skipped again
- finally the %f format results in the parsing of 3.14159 as a floating point value

The format string is composed of literal characters and conversion specifiers. A conversion specifier is a string of characters composed of the following parts or components **in order**.

- The character %
- An optional XPG3 specifier
- An optional maximum substring width
- An optional size modifier
- The conversion character

Note that only the starting % and the conversion character need be present.

4.7.1. Scan termination

There are several conditions under which the scan command will terminate further processing of the input string. The command results are different in each case as we illustrate below. Our examples use the %d specifier which attempts conversion of the input substring to an integer value.

In the first scenario, the end of the input string is reached before any conversions are **attempted** (although the reference manual says **performed**). In this case, scan returns an empty string in the inline version of the command. If variables are specified for storing the result, the command returns -1 and no variables are assigned.

```
scan abc abc%d      → (empty)
scan abc abc%d val → -1
info exists val    → 0
```

In the above example, the abc in the input string matches the abc in the format string. At that point, no further processing is done because no input remains.

In the second scenario shown below, the processing stops before the end of the input string is reached because a scan conversion fails. In this case, the inline version returns a list of the same length as the number of scan specifiers in the format string. The elements in the returned list corresponding to conversions that failed, or were not attempted due to scan termination, are set to the empty string.

```
scan abcX %d                    → {}
scan "abc10 def 20" "abc%d %d %d" → 10 {} {}
```

Compare the first result in this scenario with that in the first scenario above. There the command returned an empty string. Here it returns a list containing one element which is the empty string[7] corresponding to the single

---

[7] Tcl represents empty elements within a list as empty braces.

format specifier present. Similarly, in the second result, the last two elements in the returned list are empty as the second conversion failed thereby terminating the parse.

If variables are specified in this scenario, the return value of the command is the number of conversions performed. The variables corresponding to failed conversions will not be modified if they existed or created if they did not.

```
scan abcX %d vara                            → 0 ❶
info exists vara                             → 0
scan "abc10 def 20" "abc%d %d %d" vara varb varc → 1 ❷
set vara                                     → 10
info exists varb                             → 0
```

❶ Note the difference from the first scenario above where the return value was -1
❷ Only one conversion successful

Because the parsing was terminated by the failed match on the second %d specifier, variables varb and varc are not assigned.

The final scenario is when all conversions succeed. The scan is then terminated irrespective of whether there are any remaining characters in either the input or the format string. In the inline version, the returned value is a list each element of which is the value resulting from a successful conversion for the corresponding field specifier. In the non-inline version, the return value equals the number of field specifiers and each variable contains the corresponding value.

```
scan "abc10 15 20xyz" "abc%d %d %d"          → 10 15 20
scan "abc10 15 20" "abc%d %d %d" vara varb varc → 3
set varc                                     → 20
```

> When variables are specified, their number must match the number of successful conversions else the command will raise an error exception.

4.7.2. Conversion characters

The conversion character part of the specifier controls the type of conversion to be performed on the input string.

The string and character conversion characters are shown in Table 4.4.

Table 4.4. String specifiers for scan

Character	Description	Example
s	Parse as a string up to the next **white space** character.	scan "foo bar" %s → foo
c	Convert a character to its Unicode code point value	scan A %c → 65
[CHARS]	Matches any character in CHARS.	scan "A sentence. Or two." {%[^.?!] %[.?!]} → {A sentence} .
[^CHARS]	Matches any character **not** in CHARS.	See above.
%	Matches the percent character itself.	scan "10% off!" "%d%% off" → 10

The scan specifiers for integer values are shown in Table 4.5.

Table 4.5. Integer specifiers for scan

Character	Description	Example
d	Decimal integer.	scan 0100 %d → 100 scan -100 %d → -100
u	Unsigned decimal integer	scan 0100 %u → 100
x	Hexadecimal integer	scan 0100 %x → 256
X	Hexadecimal integer	scan 0100 %X → 256
o	Octal integer	scan 0100 %o → 64
b	Binary integer	scan 0100 %b → 4

For floating point conversion, any of f, e, E, g and G can be used. **Note these all have the same effect and are interchangeable.**

```
scan 100 %f       → 100.0
scan 12.34e-56 %g → 1.234e-55
```

The final conversion specifier is n which is a special case in that it does not parse the input string at all. Instead it returns the number of characters of the input string that have been parsed so far.

```
scan "100     200" "%d %n%d" → 100 8 200
```

The n conversion is useful when scanning incrementally through the input string. Its value can be used to determine where the next scan invocation should begin.

A conversion specifier keeps matching each successive character in the input string as long as the character is valid for the conversion For example, compare the following conversions:

```
scan 123abx %d%s → 123 abx
scan 123abx %x%s → 74667 x
```

The difference arises because a and b are valid hexadecimal characters but not valid decimal characters.

4.7.3. XPG3 scan position specifier

We now move on to the optional parts of a scan conversion specifier. By default, the extracted values are returned from the command, or stored in the passed variables, in the same order that they are encountered in the input string. The XPG3 position specifier allows this to be changed. This serves a purpose similar to that of the XPG3 position specifier described in Section 4.2.4.2 for the format command.

The position specifier, if present, must immediately follow the % at the start of a conversion specifier. It consists of either a number followed by a $ character or a single * character. In the former case, the number indicates the position that the extracted value should occupy in the returned values.

```
% scan "first second" "%s %s"
→ first second
% scan "first second" {%2$s %1$s} ❶
→ second first
% scan "first second" {%2$s %1$s} varA varB
→ 2
% puts "varA=$varA, varB=$varB"
→ varA=second, varB=first
```

❶   Note use of {} to protect the $ from interpretation by the Tcl command parser

A * character in a XPG3 position specifier indicates that the input string should be parsed as per the conversion specifier but the extracted value should not be returned or stored into the output variables.

```
% scan "100 200 300" {%d %*d %d}
→ 100 300
```

If a format string uses XPG3 position specifiers, **all** specifiers in the format string must have XPG3 position specifiers.

Position specifiers can be used in scan to help with localization similar to their use with format we described for format. However this is less common as parsing of localized strings is generally a much more complex process than their generation.

4.7.4. Specifying maximum widths

This optional specifier part is a number that limits number of characters consumed by a conversion.

```
% scan 12345 "%d%s"
→ 12345 {}
% scan 12345 "%2d%s" ❶
→ 12 345
```

❶   The d conversion can consume at most 2 input characters

Some file formats are based on fixed lengths for each field in a line representing a data record. This width modifer is useful in such cases.

4.7.5. The size modifier

The last optional component is the size modifier which defines the permitted range of an integer argument. It consists of one of the character sequences h, l, L and ll whose effect is shown in Table 4.6 below. When overflow occurs, the maximum value possible for that size is stored.

Table 4.6. Integer size modifiers for scan

Character	Description
h	An int value, generally 32 bits on most platforms. Overflows storing 0x7fffffff.
Not present	Defaults to h
l, L	A wide integer value, generally 64 bits. Overflows storing 0x7fffffffffffffff.
ll	Arbitrary precision with no overflow.

The examples below illustrate the difference between the various size modifiers.

```
% set val 777777777777777777777777777777777777777
→ 777777777777777777777777777777777777777
% scan $val %d
→ 2147483647
% scan $val %hd
→ 2147483647
% scan $val %ld
→ 9223372036854775807
% scan $val %Ld
→ 9223372036854775807
% scan $val %lld
→ 777777777777777777777777777777777777777
```

We have described the main features of the `scan` command with some rudimentary examples. Some examples of real-world use can be seen in the Tcl reference documentation.

4.8. Counting characters: `string length`

The command `string length` returns the length of a string.

```
string length STRING
```

Note that the length of the string is defined as the number of characters, Unicode code points to be precise, in the string. It should not be interpreted as the number of bytes required to represent the string in memory or in a transmitted message etc. unless STRING is a binary string produced by commands like `encoding convertto`.

```
string length "Hello, World!" → 13
```

4.9. String validation: `string is`

The `string is` command performs validation on a string to determine if it can be interpreted as belonging to a given class of values.

```
string is CLASS ?-strict? ?-failindex VAR? STRING
```

The command returns 1 if STRING belongs to the class specified by CLASS and 0 otherwise.

The empty string "" is treated as a valid value for any class unless the `-strict` option is specified in which case it is treated as invalid.

Validation of empty strings

This default treatment of the empty string as a valid value is an artifact of the fact that historically the command was meant primarily for validating user input, rather than for general type checking. In Tk GUIs for example, integer fields are permitted to be empty. This is really a misfeature and in most cases you will want to use the `-strict` option.

The `-failindex` option is used to retrieve the index in the string of the first character that does not belong to the specified class. If the command returns 0, the variable VAR is set to this index. It is not modified if the command returns 1.

The possible values of CLASS are shown in Table 4.7.

Table 4.7. String validation classes

Class	Description
alnum	Any alphanumeric Unicode characters
alpha	Any alphabetic Unicode characters
ascii	Ascii characters
false	Any string that is interpreted as a boolean false value. This includes 1, false, no and off or any upper case or abbreviated form of these.
true	Any string that is interpreted as a boolean true value. This includes 1, true, yes and on or any upper case or abbreviated form of these.
boolean	The boolean class includes any string that can be interpreted as a boolean value. Accepted values are the union of the ones listed above for false and true. Note the command returns 0 for integers other than 0 and 1 even though they are treated as valid booleans in numeric expressions.
control	Unicode control characters
digit	Unicode digits
double	Any representation of doubles
entier	Any representation of integers of arbitrary size
graph, print	Unicode printing characters. The print class includes whitespace while graph does not.
integer	Any Tcl representation of 32-bit integer values
list	Any string that can be interpreted as a valid Tcl list
lower	Lower case Unicode characters
upper	Upper case Unicode characters
punct	Unicode punctuation characters
space	Unicode whitespace characters
wideinteger	Any Tcl representation of 64-bit integer values
wordchar	Alphanumeric characters and connector punctuation such as underscore
xdigit	Lower or upper case hexadecimal characters

The following examples illustrate use of the command:

```
string is integer -10                    → 1
string is wideinteger 0x777777777777     → 1
string is xdigit -failindex charpos abcqdef  → 0 ❶
set charpos                              → 3
string is double 2.1828                  → 1
string is integer ""                     → 1 ❷
string is integer -strict ""             → 0 ❸
string is boolean 2                      → 0 ❹
```

❶ charpos will contain failing character index
❷ Empty strings are accepted by default...
❸ ...unless the -strict option is specified
❹ Integers other than 0/1 are **not** treated as boolean though they are accepted as booleans in numeric expressions.

The varying numeric classes as well as the list class accept surrounding whitespace in the string.

```
string is double "  -5e+10    "   → 1
string is list " a\ b c {d e}    " → 1
```

The commands for numeric classes also check for overflow and underflow, returning 0 in these cases and setting the -failindex variable, if specified, to -1.

```
% string is entier -failindex failpos 9999999999999999999999999999999999999999 ❶
→ 1
% string is wideinteger -failindex failpos 9999999999999999999999999999999999999
→ 0
% puts $failpos
→ -1
```

❶   Entiers have infinite precision

The alphanumeric character classes such as alnum, digit etc. all deal in Unicode. So for example the Unicode character U+096D (Devanagari 7) is a valid digit.

```
string is digit \u096d → 1
```

Note however that

```
string is integer \u096d → 0
```

because even though it is a Unicode digit, it is not a valid way to represent an integer in Tcl.

> Tcl's regular expression facilities provide an alternate means of validating strings using character classes.

4.10. String prefixes: ::tcl::prefix

The ::tcl::prefix command is used to check if a given string is a prefix of one or more strings. This operation is most commonly used for implementing new commands that accept unique prefixes of options and subcommands in similar fashion to Tcl itself. Note that the command lies in the tcl namespace, not at the global level.

The tcl::prefix command is actually an ensemble command with several subcommands. The first of these is tcl::prefix all.

```
::tcl::prefix all LIST PREFIX
```

The command returns a list of all strings from *LIST* that begin with *PREFIX* or an empty list if no such strings are found. For example, if *LIST* is a subset of the string classes used for the string is command,

```
::tcl::prefix all {alnum alpha digit integer} al → alnum alpha
```

One use case for this is to display valid choices for completion when the user types in the first few letters from a list of allowed options.

The tcl::prefix longest command returns the longest possible prefix common to all strings in a list that start with a given prefix.

```
::tcl::prefix longest LIST PREFIX
```

In scenarios like command completion, this provides an easy means to fill in as many characters as possible from the valid choices in *LIST* that begin with *PREFIX*.

```
% ::tcl::prefix longest {radix range repeat replace} re
→ rep
```

Here only repeat and replace are elements that begin with re and their longest common prefix is rep so that is what the command returns.

The last, and probably the most useful, command is tcl::prefix match.

```
::tcl::prefix match ?-exact? ?-message MESSAGE? ?-error OPTIONS? LIST PREFIX
```

If *PREFIX* is a prefix of exactly one string in *LIST*, that string is returned as the result of the command. If *PREFIX* matches a string **in its entirety**, it is returned even if it also happens to be a prefix of another.

```
% ::tcl::prefix match {radix range repeat replace} rad
→ radix
% ::tcl::prefix match {-ignore -ignorewarnings -ignoreerrors} -ignore
→ -ignore
```

By default, an error is generated if the number of matches is not exactly one.

```
% ::tcl::prefix match {radix range repeat replace} ra ❶
Ø ambiguous option "ra": must be radix, range, repeat, or replace
% ::tcl::prefix match {radix range repeat replace} rap ❷
Ø bad option "rap": must be radix, range, repeat, or replace
% puts $errorCode
→ TCL LOOKUP INDEX option rap
```

❶ Error because multiple matches
❷ Error because no matches

The -message option changes how the error message refers to the word being checked. For example,

```
% ::tcl::prefix match -message command {radix range repeat replace} ra
Ø ambiguous command "ra": must be radix, range, repeat, or replace
```

Note how the error message now says command instead of option.

The behaviour with respect to failed matches can be changed with the -error option. If specified as an empty string, the command will return an empty string on the above failures instead of raising an exception.

```
% ::tcl::prefix match -error "" {radix range repeat replace} ra
% ::tcl::prefix match -error "" {radix range repeat replace} rap
```

If not an empty string, the value passed for the -error option must be in the form of a return options dictionary. We will go into the details of the return options dictionary in Section 11.2.3. For now, here is an example that changes the global error code set on an exception.

```
% ::tcl::prefix match {radix range repeat replace} rap
Ø bad option "rap": must be radix, range, repeat, or replace
% set errorCode ❶
```

```
→ TCL LOOKUP INDEX option rap
% ::tcl::prefix match -error {-errorcode {BADOPT RTFM}} {radix range repeat replace} rap
Ø bad option "rap": must be radix, range, repeat, or replace
% set errorCode
→ BADOPT RTFM
```

❶ Default error code

One final option, -exact, specifies that the *PREFIX* must be an exact match, not just a prefix. Its effect can be seen through the following commands.

```
% ::tcl::prefix match {radix range repeat replace} ran
→ range
% ::tcl::prefix match -exact {radix range repeat replace} ran
Ø bad option "ran": must be radix, range, repeat, or replace
```

Most commonly, tcl::prefix is used for implementing subcommands and options in procedures. For example,

```
proc transform {s cmd} {
    switch -exact -- [tcl::prefix match {lower upper reverse} $cmd] {
        lower { string tolower $s }
        upper { string toupper $s }
        reverse { string reverse $s }
    }
}
```

Then we can call the command using abbreviations.

```
% transform foo rev
→ oof
% transform foo bogus ❶
Ø bad option "bogus": must be lower, upper, or reverse
```

❶ Error messages for free!

4.11. Glob pattern matching

Tcl provides a couple of ways of matching strings against patterns — the string match command based on wildcard patterns, and regexp based on regular expressions. We describe the former in this section.

A *glob* pattern is a sequence of characters which must be the same as the corresponding character in the string being matched except for the *wildcard* characters listed in Table 4.8 which are treated specially when present in the pattern.

Table 4.8. Pattern matching characters

Character	Description
*	Matches any number (including zero) of arbitrary characters.
?	Matches exactly one occurrence of an arbitrary character.
[...]	Matches one occurrence of any of the characters included within the brackets. A range of characters can also be specified. For example, [a-z] will match any lower-case letter.
\	The backslash escapes the following special character such as * or ? so that it is treated as an ordinary character. This allows you to write glob patterns that match literal glob-sensitive characters, which would otherwise be treated specially.

In addition to `string match`, glob pattern matching is also used by the `switch`, `lsearch` and `glob` commands.

The `string match` command takes a glob pattern and determines if a given string matches the pattern.

```
string match ?-nocase? PATTERN STRING
```

The command returns 1 if the specified glob pattern *PATTERN* matches *STRING* and 0 otherwise. Some examples using * to match arbitrary number of characters.

```
string match f*r fun      → 0
string match f*r fur      → 1
string match f*r* fury    → 1
string match f*r* furious → 1
```

The ? character on the other hand matches **exactly** one character.

```
string match f?r? fur     → 0
string match f?r? fury    → 1
string match f?r? furious → 0
```

Character sets can be used to match exactly one character as long they belong to that set.

```
string match {[a-f]*} boo   → 1
string match {[a-f]*} zoo   → 0
string match {[a-zA-Z]*} Zoo → 1
string match {[az]*} zoo    → 1
```

Backslash escaping is required to match literal characters that have special meaning in a glob pattern.

```
string match a*d abcd    → 1
string match {a\*d} abcd → 0
string match {a\*d} a*d  → 1
```

Notice from the examples above that we use braces to protect the pattern in cases where it might contain characters that are special to the Tcl parser.

Use of the `-nocase` option triggers case insensitive matching.

```
string match {[a-z]*} Boo        → 0
string match -nocase {[a-z]*} Boo → 1
```

If you have more complex pattern matching requirements, or need to simultaneously extract information as well as match it, then regular expressions provide a more powerful (but more complex) facility. We describe that next.

4.12. Regular expressions

Like the glob patterns we saw previously, a *regular expression* (RE) is a pattern used for matching against strings where certain characters in the pattern, termed *metacharacters*, have a special meaning. Compared to glob patterns though, regular expressions are both considerably more powerful and potentially more complex. For those new to regular expressions, here we will only provide a basic introduction in terms of their use in Tcl. For a full understanding of regular expressions, see the references cited at the end of this chapter.

For those who *do* understand regular expressions from other languages, such as the PCRE engine, note that their Tcl implementation differs slightly in their syntax, particularly when it comes to more advanced features.

In fact, Tcl itself supports three forms of regular expressions. Basic, Extended and Advanced. Only the last of these is described here as it is almost completely a superset of the others.

Table 4.9 gives the basic elements of RE syntax.

Table 4.9. Basic regular expression syntax

Character	Description
^	Matches the beginning of the string
$	Matches the end of the string
.	Matches any single character
[...]	Matches any character in the set between the brackets
\	Acts as an escape to assign special meaning to the next character or treat a metacharacter as a literal
[a-z]	Matches any character in the range a..z
[^...]	Matches any character *not* in the set given
(...)	Groups a pattern into a sub-pattern
$p \| q$	Matches pattern *p* or pattern *q*
*	Matches 0 or more occurences of the previous pattern
+	Matches 1 or more occurrences of the previous pattern
?	Matches 0 or 1 occurrences of the previous pattern
{n}	Matches exactly *n* occurrences of the previous pattern
{n,m}	Matches between *n* and *m* occurrences of the previous pattern

We will see examples of the above as well as more advanced constructs as we describe Tcl's RE support through regexp, used for searching, and regsub, which does substitutions.

4.12.1. Matching regular expressions: regexp

The regexp command has the syntax

```
regexp ?options? RE STRING ?MATCHVAR MATCHVAR . . .?
```

In its simplest form, with no optional parts specified, the command returns 1 if *STRING* matches the regular expression *RE* and 0 otherwise.

We will first describe regular expressions using this minimal form of the command.

4.12.1.1. Matching specific characters

A character that is not a metacharacter in *RE* will match that specific character in the *STRING*. Thus to look for the sequence XY in a string,

```
regexp XY aaXYbb → 1
regexp XY aaYXbb → 0
```

Notice that all of *STRING* does not have to match the expression, any substring will do.

Character escapes

Certain characters that do not have a printable representation or are otherwise difficult to include in text can be included via an escape sequence prefixed with a backslash (\). For example, the newline character is represented by the sequence \n and Unicode characters can be represented as \uhhhh or \Uhhhhhhhh sequences (where h is a hexadecimal digit). See the documentation for re_syntax in the Tcl command reference for a list of character escape sequences.

The processing of these backslash sequences is in addition to any backslash substitution that might be done by the Tcl parser. Thus the following two commands are equivalent.

```
regexp "\\t" abc\tdef  → 1
regexp {\t} abc\tdef   → 1
```

In the first case, the Tcl parser converts the \\ sequence to a single \ so the regexp command sees the argument as \t. In the second case, the enclosing braces prevent the Tcl parser from any backslash processing and again the regexp command sees \t.

> Backslashes are also used for purposes other than character escapes. We will see these as we go along.

4.12.1.2. Matching any character

The metacharacter period (.) in an *RE* matches *any* character in the string. For example, X.Y will match substrings containing an X and Y separated by exactly one character.

```
regexp X.Y aXcYb  → 1
regexp X.Y aXYb   → 0
regexp X.Y aXccYb → 0
```

4.12.1.3. Bracketed expressions and character classes

We have already seen that characters in *RE* that are not metacharacters are matched against themselves in the string. If instead of matching any character, we wanted to match any of a **set** of characters, we can specify them as a *character class* by enclosing them in brackets [ ].

```
regexp {ab[XYZ]cd} abYcd  → 1 ❶
regexp {ab[XYZ]cd} abQcd  → 0 ❷
regexp {ab[XYZ]cd} abXYcd → 0 ❸
```

❶ Match since Y is in the bracketed expression
❷ No match since Q is *not* in the bracketed expression
❸ No match since XY is not a single character

> The *RE* in the above example is enclosed in braces because the characters [ ] have special meaning to both the Tcl parser as well as RE syntax. Enclosing them in braces ensures they will not be treated as special characters by the Tcl parser. Because there are several other characters such as $ and \ that are treated specially by both the parser and RE, it is generally a good idea to enclose the RE in braces in all but the simplest cases.

A bracketed expression has its own set of special character sequences described below and most RE metacharacters like ., * and ? are treated as normal characters within the brackets. Notice in the next example how . loses its metacharacter status when placed within a bracketed character class.

```
regexp {a.c} abc    → 1
regexp {a[.]c} abc → 0
regexp {a[.]c} a.c → 1
```

When there are many characters to be included in the bracketed expression, several facilities are available for common cases.

An expression of the form *x-y* includes all characters between *x* and *y*. For example a-z includes all lower case English alphabetic characters, 0-9 includes all digits and so on.

```
regexp {[0-9]} abc   → 0
regexp {[0-9]} a5c   → 1
regexp {[-0-9]} a-c → 1 ❶
```

❶   To include -, specify it as the first character

We reiterate again that as illustrated above, regular expressions do not need to match the entire string, unless anchored (described later). In the above examples, we are matching a single character which may be present anywhere in the string.

Another way to specify characters in bracketed expressions involves *character classes* of the form `[:CLASSNAME:]` where *CLASSNAME* is a name denoting a predefined set of characters. Tcl defines several characters classes shown in Table 4.10.

Table 4.10. Regular expression character classes

Class	Description
[:alnum:]	Alphanumeric character
[:alpha:]	A letter
[:blank:]	Space or tab character
[:cntrl:]	Control characters (ASCII codes 0-31)
[:digit:]	Decimal digit
[:graph:]	A character with a graphical representation
[:lower:]	Lower case letter
[:print:]	A printable character (same as graph plus the space character)
[:punct:]	Punctuation character
[:space:]	White space character
[:upper:]	Upper case letter
[:xdigit:]	Hexadecimal digit

Our previous examples using character classes in bracketed expressions instead of character ranges would be

```
regexp {[[:digit:]]} abc → 0
regexp {[[:digit:]]} a5c → 1
```

Note the doubled [[]], the outermost set indicating a bracket expression and the inner set indicating character classes.

There are two additional features of bracket expressions:

- A bracketed expression can include multiple characters, character ranges and classes concatenated together to indicate a "inclusive-or" combination.

- If the bracketed expression starts with ^, it matches characters *not* in the rest of the expression.

The first of these is demonstrated by the following RE which will match a string beginning with a, followed by any of the characters x, y, any upper case letter or digit, and ending in a b.

```
regexp {a[xy[:upper:][:digit:]]b} axb → 1
regexp {a[xy[:upper:][:digit:]]b} a5b → 1
regexp {a[xy[:upper:][:digit:]]b} aQb → 1
regexp {a[xy[:upper:][:digit:]]b} aqb → 0
```

The second feature, the use of ^ to complement a character set is illustrated by the example below.

```
regexp {a[xy[:digit:]]b} a5b → 1
regexp {a[^xy[:digit:]]b} ayb → 0
regexp {a[^xy[:digit:]]b} a5b → 0
regexp {a[^xy[:digit:]]b} aQb → 1
```

Tcl regular expressions also support an additional \ prefixed shorthands for some commonly used classes. These are shown in Table 4.11.

Table 4.11. Character class shorthands

Shorthand	Equivalent bracket expression	Description
\d	[[:digit:]]	Digit
\D	[^[:digit:]]	Non-digit
\s	[[:space:]]	White space
\S	[^[:space:]]	Non-white space
\w	[[:alnum:]_]	Alphanumeric or underscore (_)
\W	[^[:alnum:]_]	Any character other than alphanumeric and underscore (_)

For example, using \d in lieu of [:digit:], expression.

```
regexp {a\db} a5b → 1
regexp {a\Db} a5b → 0
```

The \d, \s and \w shorthands can be used inside of bracketed expressions as well but the inverse versions of these, \D, \S and \W, cannot and you have to use the ^ prefix instead.

```
regexp {a[\d\s]b} a5b  → 1
regexp {a[\d\s]b} a\tb → 1
regexp {a[^\d\s]b} a5b → 0
```

4.12.1.4. Atoms and Quantifiers

An atom is a single character in any of the forms described earlier (literal character, character escape or character class) or a group that we will describe later. Thus in the RE

```
a[[:digit:]]\n
```

the components a, [[:digit:]] and \n are all atoms.

Quantifiers are appended to an atom to specify how many *consecutive* occurences of that atoms are permitted in a string. For example, the expression a+ would match one or more consecutive occurences of the character a.

The various forms of quantifiers are shown in Table 4.12.

Table 4.12. Regular expression quantifiers

Quantifier	Description	Example
*	Matches 0 or more occurences of the atom	regexp {aX*b} ab → 1 regexp {aX*b} aXb → 1 regexp {aX*b} aXXb → 1
+	Matches 1 or more occurences of the atom	regexp {aX+b} ab → 0 regexp {aX+b} aXb → 1 regexp {aX+b} aXXb → 1
?	Matches 0 or 1 occurences of the atom	regexp {aX?b} ab → 1 regexp {aX?b} aXb → 1 regexp {aX?b} aXXb → 0
{M}	Matches exactly M occurences	regexp aX{2}b aXb → 0 regexp aX{2}b aXXb → 1 regexp aX{2}b aXXXb → 0
{M,}	Matches M or more occurences	regexp aX{2,}b aXb → 0 regexp aX{2,}b aXXb → 1 regexp aX{2,}b aXXXb → 1
{M,N}	Matches M to N occurences (both inclusive)	regexp aX{2,4}b aXb → 0 regexp aX{2,4}b aXXXb → 1 regexp aX{2,4}b aXXXXXb → 0

4.12.1.5. Groups

Subexpressions within a RE can be grouped with parenthesis. This treats the contents within the parenthesis as a single atom to which quantifiers, alternation and such can be applied. In the first line in the example below, the + quantifier only applies to Y while in the second it applies to XY.

```
regexp {aXY+b} aXYXYb   → 0
regexp {a(XY)+b} aXYXYb → 1
```

Groups as used above use *capturing* parenthesis in that the string matching the subexpressions within parenthesis can be used in back references (see Section 4.12.1.8) and substring extraction.

An alternate form of grouping uses *non-capturing* parenthesis specified as (? : *RE*) where the leading left parenthesis is followed immediately by a ?. The equivalent non-capturing version of our example above would be

```
regexp {a(?:XY)+b} aXYXYb → 1
```

The difference from capturing parenthesis is that in this case the substring matching the *RE* expression is not accessible via back references and cannot be extracted.

We will see examples and use of these forms in later sections.

4.12.1.6. Alternation and branches

Regular expressions can be combined using the | metacharacter to form a RE that will match a string that matches any of the expressions being combined. Each subexpression is termed an *alternative* or a *branch* of the combined expression. For example, the expression apple|banana would match either apple or banana.

> The alternation metacharacter binds at a low precedence so apple|banana is equivalent to (apple)|(banana) and not appl(e|b)anana.

Any of the following would match day of the week.

```
% regexp {Sunday|Monday|Tuesday|Wednesday|Thursday|Friday|Saturday} Monday
→ 1
% regexp {(Sun|Mon|Tues|Wednes|Thurs|Fri|Sat)day} Friday
→ 1
% regexp {(Mon|Wednes|Fri|T(ues|hurs)|S(at|un))day} Tuesday
→ 1
```

4.12.1.7. Constraints

A regular expression *constraint* matches the empty string (i.e. it does not "consume" any characters in the string being matched) **but only when certain conditions are met**. An example of such a condition might be the beginning or a line or word. This section describes the available constraints in Tcl regular expressions.

4.12.1.7.1. Anchoring with ^ and $

As we saw above, the regular expression *RE* will match if it matches any substring of *STRING*. If instead we want to check that the *RE* matches *all* of *STRING*, we can "anchor" the RE with the metacharacters ^ and $. The former constrains the match to start at the beginning of the string.

```
regexp {^XY} aXY → 0
regexp {^XY} XYb → 1
```

Similarly, $ constrains the *RE* to match the end of the string.

```
regexp {XY$} aXY → 1
regexp {XY$} XYb → 0
```

They may of course be used in combination to force the entire string to match.

```
regexp XY aXYb      → 1
regexp {^XY$} aXYb → 0
regexp {^XY$} XY    → 1
```

> The options -line and -lineanchor impose different semantics on the ^ and $ anchors (see Section 4.12.1.14).

4.12.1.7.2. Constraint escapes

Tcl also defines a number of position based constraints via the escape sequences shown in Table 4.13.

Table 4.13. Constraint escape sequences

Escape	Description	Example
\A	Matches at the beginning of the string.	regexp {\AX} "aXb" → 0 regexp {\AX} "Xab" → 1
\Z	Matches at the end of the string.	regexp {X\Z} "aXb" → 0 regexp {X\Z} "abX" → 1
\m	Matches at the beginning of a word.	regexp {\mX} "aXb" → 0 regexp {\mX} "a Xb" → 1
\M	Matches at the end of a word.	regexp {X\M} "a Xb" → 0 regexp {X\M} "aX b" → 1
\y	Matches at the beginning or the end of a word.	regexp {\yX} "aXb" → 0 regexp {\yX} "a Xb" → 1 regexp {X\y} "aX b" → 1
\Y	Matches when *not* at the beginning or the end of a word.	regexp {\YX} "aXb" → 1 regexp {\YX} "a Xb" → 0 regexp {X\Y} "aX b" → 0

The sequences \A and \Z behave similarly to the ^ and $ constraints but do not change behaviour when "newline sensitive matching" (see Section 4.12.1.14) is in effect.

The word related constraints, \m, \M, \y and \Y treat alphanumeric characters and underscore (_) as word characters just like the \w character escape.

4.12.1.7.3. Lookahead constraints

Another form of constraint is based on matching a subexpression **without actually including the matched text as part of the match**. Lookahead comes in two forms:

- *Positive lookaheads* have the form (?=*LOOKAHEAD*) where *LOOKAHEAD* is the RE that should be matched at that point.
- *Negative lookaheads* have the form (?! *LOOKAHEAD*) and are similar except that the *LOOKAHEAD* must **not** be matched for the matching of the rest of the RE to proceed.

For example, suppose you wanted to match against part numbers whose format specifies a string of one or more uppercase alphabetic characters followed by one or more digits with the further constraint that the entire part number be at most 10 characters. Here is a regular expression that serves the purpose.

```
% set re {^(?=.{2,10}$)[[:upper:]]+[[:digit:]]+$}
→ ^(?=.{2,10}$)[[:upper:]]+[[:digit:]]+$
```

We can break this up into two parts. The first part of the RE is the lookahead

```
(?=.{2,10}$)
```

This ensures the length conditions are met (between 2 and 10 characters in the string) but does not say anything about the expected format. The second part

```
[[:upper:]]+[[:digit:]]+
```

then requires the part number to be a sequence of upper case letters followed by a sequence of digits.

We can then do syntactic checks for valid part numbers as

```
regexp $re A0            → 1
regexp $re ABC2345678    → 1
regexp $re 1234567890    → 0 ❶
regexp $re ABC12345678   → 0 ❷
```

❶ Only digits
❷ More than 10 characters

The crucial effect of using lookaheads as illustrated here is that the lookahead expression does not "eat up" characters in the target string; the following RE still matches from the same point as the lookahead expression. Try writing the expression without the constraint keeping in mind that both the alphabetic and the numeric parts may have more than one character.

4.12.1.8. Back references

There are times when it is useful to match a substring based on what was previously matched by the RE. The standard example of this is finding if a word is mistakenly repeated in a document, for example the word *has* in the sentence below. We can construct a RE to detect this.

```
% regexp {\mhas\s+has\M} "This sentence has has repeated words." ❶
→ 1
```

❶ Note use of \m and \M word constraints.

However this is not a general solution given that we do not know a priori which word might be repeated. Instead we have to match words using generic regular expressions. We then need a mechanism that lets us specify the next part of the expression to be the word that was just matched. Back references provide exactly that capability.

A back reference in a RE is specified in the form \N where N is a number which references a group enclosed by capturing parenthesis (see Section 4.12.1.5). When multiple groups are present, the corresponding "captures" are numbered in the order of the position of their opening parenthesis.

To solve our problem then, the matching RE should

1. begin only at a position that is the beginning of a word indicated by the \m constraint
2. followed by *any* word matched as \w+
3. followed by any amount of whitespace matched as \s+
4. followed by the **same string** that was just matched by the above \w+
5. followed by the end of word constraint \M.

So we need to "transport" the word matched in step 2 to the match required by step 4. To do this we enclose the word specifier in capturing parenthesis as (\w+) so that the result of its match can be referenced through a back reference. Since this is the only, and therefore the first, capturing parenthesis in the expression, it is referenced as \1 and we use that in step 4.

Thus the entire matching expression becomes that shown below:

```
% regexp {\m(\w+)\s+\1\M} "This sentence has has repeated words."
→ 1
% regexp {\m(\w+)\s+\1\M} "This sentence has no repeated words."
```

```
→ 0
% regexp {\m(\w+)\s+\1\M} "To be or not to be." ❶
→ 0
```

❶ Repeated but not consecutive

Back references are especially useful when substituting using regular expressions and we will see examples of their use when we describe the regsub command.

4.12.1.9. Counting number of matches

A regular expression may match multiple times in a string. If the -all option is specified, the command will return the number of matches found in the string.

```
regexp -all X+ aXXbXCXXX → 3
```

The -all option also has other uses as we will see in a bit.

4.12.1.10. Retrieving matches

Up to this point, we have only dealt with the simplest form of the regexp command — one that tells us whether a given string matches a RE or not. We now look at the various means of having regexp actually tell us **what** was matched.

4.12.1.10.1. Retrieving matched content

If additional arguments are specified for the regexp command, they are treated as names of variables in which the match is to be stored. For example,

```
% regexp X+ aXXXc xes
→ 1
% set xes
→ XXX
```

If the RE matches, the command returns 1 and stores the matched content in the passed variable (xes in this case). If no match occurs, the command returns 0 and the variable is unchanged.

If there is a need to retrieve the content of subexpressions, additional variables can be specified. Matches for subexpressions enclosed in capturing parenthesis are successively stored in any specified variables. Non-capturing subexpression matches are ignored for the purpose of storing.

```
% regexp {(X+)(?:Y+)(Z+)} aXXYYZZZb match xes zes
→ 1
% puts "$match, $xes, $zes"
→ XXYYZZZ, XX, ZZZ
```

4.12.1.10.2. Retrieving matched indices

In some parsing situations, it is more useful to retrieve the string indices of the matches than the actual content itself. Specifying the -indices option stores in each specified variable a pair consisting of the start and end indices of the corresponding match.

```
% regexp -indices {(X+)(?:Y+)(Z+)} aXXYYZZZb match xes zes
→ 1
% puts "$match, $xes, $zes"
→ 1 7, 1 2, 5 7
```

When parsing large amounts of text using regular expressions, storing indices is often more efficient in time and space than the matched content. The original text being parsed is maintained as the "master" copy and the consumer of the parse can use the indices to retrieve substrings as and when needed.

4.12.1.10.3. Retrieving matches with `-inline`

Instead of storing matches in variables, you can have `regexp` return the matches by specifying the `-inline` option. Additional variable name arguments must not be specified with the option.

The return value from `regexp` is a list containing the same values as would have been stored in any variable name arguments if `-inline` was not specified.

```
% regexp -inline {(X+)(?:Y+)(Z+)} aXXYYZZZb
→ XXYYZZZ XX ZZZ
% regexp -inline -indices {(X+)(?:Y+)(Z+)} aXXYYZZZb
→ {1 7} {1 2} {5 7}
```

If the RE does not match, the command will return an empty list.

```
regexp -inline {(X+)(?:Y+)(Z+)} aYYZZZb → (empty)
```

4.12.1.10.4. Retrieving all matches

As we saw earlier, the `-all` option can be specified to count the number of matches found. If variables are specified for the command, only the results corresponding to the last match found will be stored in them.

```
% regexp -all {(X+)(?:Y+)(Z+)} aXXYYZZZbXYZ match xes zes
→ 2
% puts "$match, $xes, $zes"
→ XYZ, X, Z
```

If you want information for all matches, not just the last one, use the inline version.

```
% regexp -inline -all {(X+)(?:Y+)(Z+)} aXXYYZZZbXYZ
→ XXYYZZZ XX ZZZ XYZ X Z
% regexp -inline -indices -all {(X+)(?:Y+)(Z+)} aXXYYZZZbXYZ
→ {1 7} {1 2} {5 7} {9 11} {9 9} {11 11}
```

The return value, as shown above, is a flat list containing all matches and submatches.

4.12.1.11. Option metasyntax

Some `regexp` command options can instead be embedded into the RE by beginning the expression with the metasyntax (`?OPTS`) where *OPTS* is a sequence of one or more characters, each corresponding to an option. Thus for example, `i` corresponds to the use of `-nocase` and `n` to newline sensitive matching, so the two statements

```
regexp -nocase -line {RE} STRING
regexp {(?ic)RE} STRING
```

are equivalent. **Embedded options can only appear at the beginning of the regular expression.**

We will discuss this embedded metasyntax alongside their option equivalents.

4.12.1.12. Case-independent matching

By default, regexp implements case sensitive matching.

```
regexp xy axyb → 1
regexp xy aXYb → 0
```

Specifying the -nocase option will result in case being ignored.

```
regexp -nocase xy aXYb → 1
```

Alternatively, the (?i) metasyntax can be used to specify case-insensitive matching. Conversely, (?c) specifies case-sensitive matching.

```
regexp {(?i)xy} aXYb → 1
regexp {(?c)xy} aXYb → 0
```

4.12.1.13. Matching literal strings

Because of its many options, regexp can be useful even for exact matching of literal strings. For example supposed we wanted to count the number of occurences of a literal string.

```
% set search_string "XY"
→ XY
% regexp -all $search_string aXYbXcXYd
→ 2
```

The above works fine when the search_string does not contain any literal characters that might be misinterpreted as metacharacters. But if it does, then we get unexpected results.

```
% set search_string "X."
→ X.
% regexp -all $search_string aX.bXcX.d
→ 3
```

The problem is with regexp treating . as a metacharacter when we want to actually treat it as a literal character. One solution is to preprocess the search string to escape any metacharacters with a \. An easier way is to prefix the search expression with ***= which indicates to the regexp command that the rest of the expression is to be treated as a literal string.

```
% regexp -all "***=$search_string" aX.bXcX.d
→ 2
```

The above construct ***= is not useful when the literal is part of a larger regular expression which is not a literal itself. In that case the metacharacters in the literal must be escaped, for example with the regsub command we will see later.

```
regsub -all {[][*+?{}()<>|.^$\\]} $literal_string {\\&}
```

The string map command, probably more efficient, may also be used for this.

4.12.1.14. Newline-sensitive matching

By default no special treatment is afforded to newline characters embedded in the string being matched. For some use cases, such as matching lines read from a file in a manner similar to egrep, this requires reading in the file line by line doing a regexp match on each line.

A more efficient option is to use newline-sensitive matching by specifying the -line option to regexp. When this option is specified, certain matching behaviour changes:

- The metacharacters ^ and $ are treated as matching the beginning of a line and end of a line respectively. Note that the \A and \Z constraints are unchanged and continue to match the beginning and end of the entire string.

- The . metacharacter is now treated as matching all characters **except** newlines. Similarly, bracket expressions of the form [^...] (ie. matching characters **not** in a set) never matches a newline.

Thus we can count the number of lines with extraneous trailing whitespace.

```
% set file_content "First line\nSecond line with trailing space    \nThird line with tab\t"
→ First line
  Second line with trailing space
  Third line with tab
% regexp -all {\s+$} $file_content  ❶
→ 1
% regexp -all -line {\s+$} $file_content ❷
→ 2
```

❶   Only sees trailing tab at end of content
❷   Sees all lines ending in whitespace

The two behavioural changes above can actually be controlled separately with the -lineanchor and -linestop options to regexp. The option -line is equivalent to the combination of these. Specifying -lineanchor changes the behaviour of ^ and $ as described above while -linestop controls the behaviour of . and [^...] matching.

Newline sensitive matching can also be enabled through embedded option metasyntax as an alternative to the above options. The correspondences are shown in the table below

Table 4.14. Table

(?n)	Equivalent to the -line option
(?w)	Equivalent to the -lineanchor option
(?p)	Equivalent to the -linestop option

So the following would be equivalent to the above example.

```
regexp -all {(?n)\s+$} $file_content → 2
```

4.12.1.15. Matching at an offset: -start

On occasion, for example incrementally parsing a grammar using regular expressions, you need to begin the matching from somewhere other than the start of the string. You can use the -start option for this purpose.

```
% regexp -inline -all a+ "aaabacaa"
→ aaa a aa
% regexp -start 4 -inline -all a+ "aaabacaa"
→ a aa
```

This feature is commonly useful in conjunction with the -indices option where the returned indices are used as the argument to -start for the next match attempt.

4.12.1.16. Controlling greediness

There are times when a RE may match a string in multiple ways. Consider the following match

```
% regexp -inline {^(x+)(.*y)$} xxyy
→ xxyy xx yy
```

The RE matches with the first subexpression matching xx and the second matching yy. The RE could also have matched with the first subexpression matching as x and the second as xyy.

The difference between the two matches is that by default a quantifier (like + above) will match as much as possible in a "greedy" manner. Hence the first subexpression matches the whole sequence of x characters. In some situations, examples of which we will see later, it is desirable to match the *fewest* number of characters possible. The greedy quantifier can be converted into a non-greedy one by appending a ?. It will then match the least number of characters required for the match to be successful.

```
% regexp -inline {^(x+?)(.*y)$} xxyy
→ xxyy x xyy
```

Make a note of the different subexpression matches with respect to the previous result.

The rules for greediness are detailed in the Tcl reference pages and we will not go into them here other than provide an example where the distinction is useful. Consider we want to extract content enclosed in an XML tag <Item>...</Item> . (Using regular expressions to parse XML is not recommended in general but is often adequate for quick throwaway scripts.) We might write an expression as follows

```
% regexp {<Item>(.*)</Item>} "<Item>Item 1</Item>" -> content
→ 1
% puts $content
→ Item 1
```

You will often see -> used in Tcl regexp commands to indicate that the full match (which lands up being stored in a variable of that name) is of no interest.

That seems to work except it doesn't. When you have multiple tags the result is not what is desired.

```
% regexp {<Item>(.*)</Item>} "<Item>Item 1</Item><Item>Item 2</Item>" -> content
→ 1
% puts $content
→ Item 1</Item><Item>Item 2
```

The problem is again one of greed where the (.*) expression matches as much as it can till the second </Item> while we would have wanted it to stop at the first. Appending a ? to the * quantifier to force non-greedy matching gives the desired behaviour.

```
% regexp {<Item>(.*?)</Item>} "<Item>Item 1</Item><Item>Item 2</Item>" -> content
→ 1
% puts $content
→ Item 1
```

4.12.1.17. Comments and expanded syntax

The power of regular expressions is accompanied by related complexity and it can be difficult to discern the purpose of various parts of even a moderately complex RE. Regular expressions in Tcl offer a solution to this problem in the form of an expanded syntax which is enabled by specifying the -expanded option to the regexp command.

Expanded syntax differs from normal RE syntax in the following ways:

- Whitespace in the RE is no longer significant unlike in the normal RE syntax. You can therefore use spaces and tabs to indent and spread a RE out over multiple lines.
- The # character starts a comment and all characters till the end of the line or the expression are ignored.

There are a some exceptions to the above.

- A whitespace or # character preceded by a \ is treated as a significant character and not ignored.
- A whitespace or # character within a bracketed expression is significant.
- Whitespace and # are illegal within multicharacter symbols. We don't discuss these at all here. See the Tcl reference page for more information.

As an example, here is a previous example for detecting repeated words rewritten in expanded syntax.

```
regexp -inline -all -expanded {
    \m       # Beginning of a word
    (\w+)    # followed by one or more word characters
    \s+      # then whitespace
    \1       # then the word that was matched
    \M       # then end of the word (a non-word char, end of string etc.)
} "This sentence has has repeated words."
→ {has has} has
```

Expanded syntax can also be enabled with the (?x) metasyntax instead of with the -expanded option.

> ⚠ The embedded metasyntax has to be right at the beginning of the regular expression since the expanded syntax begins after the closing parenthesis. Thus there must not be any character, including space or newline, preceding the (?x) at the start of the expression.

4.12.2. Substituting regular expressions: regsub

The regsub command allows substitutions to be performed on a string based on the matching of a RE pattern, either returning the modified string or saving it in a new variable. It has the syntax

```
regsub ?options? RE STRING SUBSPEC ?VARNAME?
```

where *RE* is the regular expression, *STRING* is the string in which substitutions are to be made and *SUBSPEC* is the specification of the substitution. If *VARNAME* is not specified, the command returns the result of the substitution. If *VARNAME* is specified, the result of the substitution is stored in the variable of that name and the number of substitutions is returned.

The substitution string *SUBSPEC* can itself refer to elements of the matched *RE* pattern, by using one or more back references of the form \N where N is a number between 0 and 9: \0 will be replaced with the string that matched the entire RE, \1 with the string that matched the first sub-pattern, and so on. You can also use the character & in place of \0.

```
% regsub {(\d+) (\d+)} "Example: 100 200" {\0 reversed is \2 \1}
→ Example: 100 200 reversed is 200 100
```

```
% regsub {(\d+) (\d+)} "Example: 100 200" {& reversed is \2 \1} var
→ 1
% puts $var
→ Example: 100 200 reversed is 200 100
```

Here \0 and & match 100 200 while the back references \1 and \2 refer to the capturing parenthesis content 100 and 200 respectively. **The part of the string that does not match the regular expression is preserved.** Thus the string `Example: ` is left untouched in the result.

By default, regsub only substitutes the first occurence of the RE. You can use the -all switch to substitute all occurences instead.

Going back to an example we saw earlier, detection of repeated words in text, we can instead use regsub to fix the errors instead of just detecting them.

```
% regsub -all {\m(\w+)(\s+)\1\M} {
    Words are often repeated when
    when a word appears at the end of a line
    line and is repeated on the next.
} {\1}
→
      Words are often repeated when a word appears at the end of a line and is repeated o...
```

Note again the parts of the text that do not match the regular expression are left as they are.

The regsub command accepts many, but not all, of the options of the regexp command, in particular -nocase, -start, -line, -linestop, -lineanchor and -expanded.

```
% regsub -all {(c)olor} "Colors colors" {\1olour}
→ Colors colours
% regsub -nocase -all {(c)olor} "Colors colors" {\1olour}
→ Colours colours
```

These options have the same effect as for the regexp command and we do not further describe them further here.

The following example from RosettaCode [8] illustrates the combined use of string map, regsub and subst to decode URL's. You will often find this combination of commands used in tasks involving decoding operations.

```
proc urlDecode {str} {
    set specialMap {"[" "%5B" "]" "%5D"}
    set seqRE {%([0-9a-fA-F]{2})}
    set replacement {[format "%c" [scan "\1" "%2x"]]}
    set modStr [regsub -all $seqRE [string map $specialMap $str] $replacement]
    return [encoding convertfrom utf-8 [subst -nobackslash -novariable $modStr]]
}
urlDecode "http%3A%2F%2Ffoo%20bar%2F"
→ http://foo bar/
```

Since we have covered the relevant commands (well, except encoding), grokking the code is left as an exercise for the reader.

4.12.3. Designing regular expressions

We have described the tools and features related to regular expressions that are available in Tcl. We have not delved into how to go about designing regular expressions for specific tasks. Regular expressions are useful and

---

[8] https://www.rosettacode.org/wiki/URL_decoding#Tcl

powerful but complex and often difficult to think about. You are encouraged to consult the references at the end of this chapter that go into the theory and practice of regular expressions.

There are also a number of tools are available to help experimenting with regular expressions such as Visual REGEXP[9]. These are very useful in both building and debugging regular expressions.

4.13. Binary strings

As we stated, strings in Tcl are sequences of Unicode code points or (roughly speaking) characters. Many applications, such as those dealing with network packets or compressed data, need to work with binary data where individual bytes, and even bits, are manipulated. Such data is handled in Tcl as *binary strings*, which are nothing but ordinary strings with all characters with Unicode code points in the range U+0000-U+00FF (thus each fitting in a byte). Most string commands such as string length, string index etc. work naturally with binary strings. For constructing and parsing binary strings, and conversion to common human readable encodings such as base64, Tcl provides the binary ensemble command.

For ease of displaying binary data, we will first define a simple (but not the most efficient) procedure, bin2hex, using the binary encode command that we will see later.

```
proc bin2hex {args} {
    regexp -inline -all .. [binary encode hex [join $args ""]]
}
```

This will dump each byte in a binary string in hexadecimal format.

> The above illustrates a quick and dirty method of using regexp for splitting strings into equal size chunks.

4.13.1. Binary literals

The easiest way to create simple binary string constants with known content is to use the \x syntax.

```
set lit "\x01\x80\xff"
bin2hex $lit
→ 01 80 ff
```

This creates a binary string that is a sequence of three bytes. Creating a binary literal is even easier if it contains only 7-bit values, i.e. a pure ASCII string. In that that case the ASCII value of a character is used as the value of the byte.

```
bin2hex "XYZ" → 58 59 5a
```

4.13.2. Encoding binary strings as ASCII

There are many times when binary data has to be encoded into 7- or 8-bit ASCII form. This might be required for transporting through binary data through email, for human readability, storage of binary data in files based on ASCII encodings and so on.

There are three commonly used ASCII based formats used for encoding binary data — plain hexadecimal encoding, base64 and uuencode. Tcl supports encoding and decoding from all three formats with the binary encode and binary decode commands.

---

[9] http://laurent.riesterer.free.fr/regexp/

4.13.2.1. Hexadecimal encoding of binaries: `binary encode|decode hex`

The `binary encode hex` and `binary decode hex` commands convert binary strings to and from hexadecimal ASCII strings.

```
binary encode hex BINDATA
binary decode hex ?-strict? ENCODED
```

Each byte of the `BINDATA` is encoded as a pair of hexadecimal digits, most significant nibble first.

```
binary encode hex XYZ            → 58595a
binary encode hex "\xfe\xf0\x0f" → fef00f
binary decode hex "58595a"       → XYZ
```

The decoding will raise an error if the argument contains anything other than hexadecimal characters and whitespace (which are ignored). If given the `-strict` option even whitespace is not allowed.

```
% binary decode hex "58 595a"
→ XYZ
% binary decode hex -strict "58 595a"
Ø invalid hexadecimal digit " " at position 2
```

4.13.2.2. Base64 format: `binary encode|decode base64`

The `binary encode base64` and `binary decode base64` commands convert binary strings to and from base64 encoded ASCII strings.

```
binary encode base64 ?-maxlen COUNT? ?-wrapchar CHAR? BINDATA
binary decode base64 ?-strict? ENCODED
```

The `binary encode base64` command can take two options. The `-maxlen` option specifies the maximum line length of the encoded string beyond which a line should be split into multiple lines. By default lines are not split. The `-wrapchar` option, which is the newline character by default, specifies the character to use to separate the split lines.

```
% binary encode base64 "\xfe\xf0\x0f"
→ /vAP
% set enc [binary encode base64 -maxlen 30 [string repeat XYZ 20]]
→ WFlaWFlaWFlaWFlaWFlaWFlaWFlaWF
  laWFlaWFlaWFlaWFlaWFlaWFlaWFla
  WFlaWFlaWFlaWFlaWFla
% binary decode base64 $enc
→ XYZXYZXYZXYZXYZXYZXYZXYZXYZXYZXYZXYZXYZXYZXYZXYZXYZXYZXYZXYZ
```

Like for hexadecimal decoding, the base64 decoding form takes a `-strict` option that will raise an error if any whitespace is present. By default whitespace is ignored. Thus the following will raise an error since we wrapped the encoded output with newlines above.

```
% binary decode base64 -strict $enc
Ø invalid base64 character "
  " at position 30
```

4.13.2.3. Uuencode format: `binary encode|decode uuencode`

The final form of binary to ASCII encoding, the *uuencode* format, is implemented by the `binary encode uuencode` and `binary decode uuencode` commands.

```
binary encode uuencode ?-maxlen COUNT? ?-wrapchar CHAR? BINDATA
binary decode uuencode ?-strict? ENCODED
```

This supports the same `-maxlen` and `-wrapchar` options as the base64 encoding above.

```
% binary encode uuencode "\xfe\xf0\x0f"
→ #_0`/
% set enc [binary encode uuencode -maxlen 30 [string repeat XYZ 20]]
→ 56%E:6%E:6%E:6%E:6%E:6%E:6%E:
  56%E:6%E:6%E:6%E:6%E:6%E:6%E:
  26%E:6%E:6%E:6%E:6%E:6%E:
% binary decode uuencode $enc
→ XYZXYZXYZXYZXYZXYZXYZXYZXYZXYZXYZXYZXYZXYZXYZXYZXYZXYZXYZXYZ
```

Note however, that the `-strict` option for the uuencode version only throws an error if whitespace appears in unexpected places as the format itself allows for it in some locations.

4.13.3. Constructing binary strings: `binary format`

The `binary format` command is used to construct a binary string in a similar fashion to how the `format` command is used for constructing character strings.

```
binary format FORMATSTRING ?ARG ARG ...?
```

The *FORMATSTRING* argument specifies the structure or layout of the binary string as a sequence of fields of various types and sizes. The command returns the binary string constructed by filling each field with the value of the corresponding argument formatted appropriately.

As an example, consider the initial part of a TCP header for an HTTP connection, which consists of

- a 16-bit source port numbers, say 5000 or 0x1388 in hex
- a 16-bit destination port number 80, or 0x0050 hex
- a 32-bit sequence number, say 1000000, or 0x000F4240 hex
- a 32-bit acknowledgement number, say 100 or 0x00000064 hex

All fields are sent in network byte order (big endian, most significant byte first) so the stream of bytes appear in hexadecimal as

```
13 88 00 50 00 0F 42 40 00 00 00 64
```

Within a Tcl script the above header fields might be stored in variables and the `binary format` command used to construct the packet header.

```
set srcport 5000
set dstport 80
set seqnum 1000000
set acknum 100
set header [binary format SSII $srcport $dstport $seqnum $acknum]
bin2hex $header
→ 13 88 00 50 00 0f 42 40 00 00 00 64
```

The format types S and I specify 16-bit big endian and 32-bit big endian fields as per the desired layout. Because the constructed header contains non-printable data, we use our bin2hex wrapper around the `binary encode hex` command to display it in hexadecimal form.

The format string may include spaces for readability purposes. In the above example the format string SSII may have been specified as S S I I or SS I I etc. with no difference in the generated binary string.

In general, *FORMATSTRING* should be a sequence of field specifiers each of which is

- a single character that either specifies a type or a cursor movement
- optionally followed by an flag character
- optionally followed by a numeric count field

The type and cursor specifiers are detailed later.

The flag character, for which u is the only valid value, is ignored and not discussed here. It is accepted by the binary format command only for compatibility with the binary scan command allowing the same format string to be used for both.

The count field may be either a positive integer value or the character *. An integer value specifies the number of fields of that type to be placed at that position. The values are picked up from the corresponding argument which may be a string or a list depending on the type specifier. The * character works similarly except that it indicates that all the values in the corresponding argument are to be used.

Thus our previous example could also have been written (among many other possibilities) as

```
set header [binary format "S2 I*" [list $srcport $dstport] [list $seqnum $acknum]]
bin2hex $header
→ 13 88 00 50 00 0f 42 40 00 00 00 64
```

4.13.3.1. Type specifiers for binary format

The type character, such a S or I in our example, indicates both the type (integer, real etc.) of a field as well as its layout (width, endianness). Table 4.15 summarizes the various type specifiers.

Table 4.15. Type specifiers for binary format

Specifiers	Description
a, A	Byte string padded with null bytes or binary value 32/0x20 (ASCII space) respectively.
b, B	Bit string. Arguments must be a string of binary digits 0 and 1. Packed within each output byte in low to high or high to low order respectively.
h, H	String of hexadecimal digits packed in each byte in low to high or high to low order respectively.
c	List of integers if a count is specified. Only the low order 8 bits are stored in the output byte.
s, S, t	List of integers. Only the low order 16 bits are stored in the output in little endian, big endian and native order respectively.
i, I, n	List of integers. Only the low order 32 bits are stored in the output in little endian, big endian and native order respectively.
w, W, m	List of integers. Only the low order 64 bits are stored in the output in little endian, big endian and native order respectively.
r, R, f	List of single precision floating point numbers. Stored in little endian, big endian and native order respectively.
q, Q, d	List of double precision floating point numbers. Stored in little endian, big endian and native order respectively.
x	Stores zeroes in the output.

The details of each format with examples are given below.

Binary formats: a, A

The character a specifies a single byte field. The argument is a character string and the value stored in the field is taken from the **low 8 bits** of the Unicode code point for the corresponding character. Thus this command should not be used to generate binary representations of general Unicode strings. Use the encoding command instead.

```
bin2hex [binary format a z]      → 7a
bin2hex [binary format a \u0102] → 02 ❶
```

❶ Note truncation to low 8 bits

If a count specifier is present, the appropriate number of characters from the argument string are used. Any extra characters in the argument are ignored. If the argument has fewer characters than the specified count, the remaining bytes are filled with null bytes.

```
bin2hex [binary format a3 wxyz]  → 77 78 79 ❶
bin2hex [binary format a3a yz x] → 79 7a 00 78 ❷
```

❶ Only 3 characters used
❷ Note padding first argument with nulls

> If the string has only ASCII characters, calling `binary format` is essentially a no-op.
>
> ```
> bin2hex [binary format a* wxyz] → 77 78 79 7a
> bin2hex wxyz                    → 77 78 79 7a
> ```

The specifier A is similar except that if the string argument has fewer characters than the specified count, the remaining bytes are filled with the binary value 32/0x20 (corresponding to an ASCII space) instead of null bytes.

```
bin2hex [binary format A*A3 wxyz yz] → 77 78 79 7a 79 7a 20 ❶
```

❶ Note padding with spaces

Binary format: b, B

Arguments must be a string of binary digits 0 and 1. For b, these are packed into output bytes in low to high order within each byte. Zeroes are used if the argument string is shorter than the count for a field or if the number of bits is not a multiple of 8. B is similar except that bits are stored in high to low order within a byte.

```
bin2hex [binary format b8 10101010]         → 55
bin2hex [binary format B8 10101010]         → aa ❶
bin2hex [binary format "b8 b5" 101 11111]   → 05 1f ❷
bin2hex [binary format "B8 B5" 101 11111]   → a0 f8 ❸
bin2hex [binary format b* 1011001110001110] → cd 71 ❹
```

❶ Note different output bit order from above
❷ Zero fill high bits
❸ Zero fill low bits
❹ Output as many bytes as needed

Binary format: h, H

The argument is a string of hexadecimal digits. Both lower and upper case characters are accepted. In the case of h (almost never used), the hex digits are packed in the output bytes in low to high order whereas for H they are

packed in the high to low order which is what is normally desired. Zeroes are used to fill if the argument string is shorter than the count for a field or if the number of hexadecimal characters is not even.

```
bin2hex [binary format h* 0aB] → a0 0b
bin2hex [binary format H* 0aB] → 0a b0
```

Binary format: c

If count is not specified, the argument must be an integer the low 8 bits of which are stored in the byte. If count is specified, the argument must be a list of *at least* that many integers. The generated output is then a sequence of bytes each containing the low 8 bits of the corresponding integer element. Extra elements in the list are ignored.

```
bin2hex [binary format cc2 10 {-1 1}]          → 0a ff 01
bin2hex [binary format c* {254 255 256 257}] → fe ff 00 01 ❶
```

❶ Note truncation to low 8 bits

Binary format: s, S, t

If count is not specified, the argument must be an integer the low 16 bits of which are stored in two bytes in little endian, big endian and native order for s, S and t respectively.

If count is specified, the argument must be a list of *at least* that many integers. The generated output is then a sequence of bytes each containing the low 16 bits of the corresponding integer element. Extra elements in the list are ignored.

```
bin2hex [binary format ss* 33825 {-2 65537}] → 21 84 fe ff 01 00
bin2hex [binary format SS* 33825 {-2 65537}] → 84 21 ff fe 00 01
bin2hex [binary format tt* 33825 {-2 65537}] → 21 84 fe ff 01 00
```

Binary format: i, I, n

Similar to s except that i, I and n store 32-bit integers in 4 byte output sequences in little endian, big endian and native order respectively.

```
bin2hex [binary format ii* 2151678465 {-2 65537}] → 01 02 40 80 fe ff ff ff 01 00 01 00
bin2hex [binary format II* 2151678465 {-2 65537}] → 80 40 02 01 ff ff ff fe 00 01 00 01
```

Binary format: w, W, m

Similar to s except that w, W and m store 64-bit integers in 8 byte output sequences in little endian, big endian and native order respectively.

```
bin2hex [binary format w 18049651735527937] → 01 02 04 08 10 20 40 00
bin2hex [binary format W 18049651735527937] → 00 40 20 10 08 04 02 01
```

Binary format: r, R, f

Stores single precision floating point number in little endian, big endian and native order respectively. The number of bytes produced is dependent on the machine architecture.

```
bin2hex [binary format r 2.71828] → 4d f8 2d 40
bin2hex [binary format R 2.71828] → 40 2d f8 4d
```

Binary format: q, Q, d

Stores double precision floating point number in little endian, big endian and native order respectively. The number of bytes produced is dependent on the machine architecture.

```
bin2hex [binary format q 2.71828] → 90 f7 aa 95 09 bf 05 40
bin2hex [binary format Q 2.71828] → 40 05 bf 09 95 aa f7 90
```

Binary format: x

Stores zeroes in the output. This differs from the other types in that it does not consume an argument and does not permit the count to be specified as *.

```
bin2hex [binary format cxcx2c 255 254 253] → ff 00 fe 00 00 fd
```

4.13.3.2. Cursor movement for formatting

In addition to the types, the format specification can include cursor movement characters. The binary format command writes output bytes at a position in the string indicated by a *cursor*. Normally the cursor is positioned right after the last position that was written in the output string. Cursor movement characters change the position of this cursor and unlike type specifiers do not consume any arguments.

Table 4.16. Binary format cursor movement characters

Specifier	Description

X	Moves the cursor backward in the output string by the specified count or by one character if not count is specified. If the count is is * or greater than the current position, the cursor is placed at the first position.
	
	 ```bin2hex [binary format c3c2 {0 1 2} {3 4}] → 00 01 02 03 04```
	```bin2hex [binary format c3X2c2 {0 1 2} {3 4}] → 00 03 04```

@	Moves the cursor to the absolute position given by count which must be specified. If the count is greater than the current output string length, the output is padded with the appropriate number of zeroes. If the count is *, the cursor is placed at the end of the string. ```bin2hex [binary format c5@2c2@*c {0 1 2 3 4} {5 6} 7] → 00 01 05 06 04 07```

4.13.4. Parsing binary strings: binary scan

The binary scan command is used to parse a binary string in a similar fashion to how the scan command is used for parsing character strings. It is conceptually the inverse of the binary format command.

```
binary scan BINSTRING SCANFORMAT ?VARNAME VARNAME …?
```

It parses the binary string *BINSTRING* driven by a format string *SCANFORMAT* that specifies the expected structure or layout of *BINSTRING* as a sequence of fields of various types and sizes. The values are extracted and stored in the variables passed as additional arguments. The command returns the number of variables that were set.

As an example, the following code parses the binary TCP header we generated in the previous section.

```
% bin2hex $header
→ 13 88 00 50 00 0f 42 40 00 00 00 64
% binary scan $header SSII scan_srcport scan_dstport scan_seq scan_ack
→ 4
% puts "$scan_srcport, $scan_dstport, $scan_seq, $scan_ack"
→ 5000, 80, 1000000, 100
```

The syntax of the *SCANFORMAT* argument is the same as the format specifiers used for the `binary format` command. It is a sequence of field specifiers each of which is

- a single character that either specifies a type or a cursor movement.
- optionally followed by an flag character
- optionally followed by a numeric count field.

The field specifiers may be optionally separated by spaces.

The scan begins at the start of the input binary string and maintains a cursor position within the string that is updated after each field specifier. If the field specifier denotes a type, the bytes following the cursor position are scanned as binary data of that type and the cursor is moved to point to the following byte. If the field specifier denotes cursor movement, the cursor is moved without any bytes being scanned.

The flag character, for which u is the only valid value, may be specified with any type but only has effect for certain integer types where it marks the field to be interpreted as an unsigned value. For example,

```
% bin2hex [set bin [binary format i 0xffffffff]]
→ ff ff ff ff
% binary scan $bin i value; puts $value ❶
→ -1
% binary scan $bin iu value; puts $value ❷
→ 4294967295
```

❶   i specifies 32-bit little endian integer
❷   iu specified *unsigned* 32-bit little endian integer

The count field may be

- a positive integer value in which case it specifies the number of fields of that type to be parsed and stored in the corresponding variable
- the character * which indicates that all the remaining bytes are to be parsed as that type

The binary string being parsed may not have sufficient bytes to satisfy the scan string specification. This is not treated as an error. Instead as many field specifiers as can be *fully* satisfied are parsed and stored in the corresponding variables. Remaining variables are not affected.

```
% binary scan $header SSIII scan_srcport scan_dstport scan_seq scan_ack extra_var
→ 4
% puts "$scan_srcport, $scan_dstport, $scan_seq, $scan_ack"
→ 5000, 80, 1000000, 100
% puts [info exists extra_var]
→ 0
```

4.13.4.1. Type specifiers for `binary scan`

The type character, such a S or I in our example, indicates both the type (integer, real etc.) of a field as well as its layout (width, endianness). Table 4.17 shows the various type specifiers available.

Table 4.17. Type specifiers for binary scan

Specifier	Description
a, A	Extract a single byte differing in their treatment of trailing spaces and zero bytes. The byte is treated as a Unicode character in the range U+0000-U+00FF.
b, B	Extract bits in a byte in low to high and high to low order respectively.
h, H	Extract the nibbles of a byte as a pair of hexadecimal digits in low to high or high to low order respectively.
c	Extracts bytes as signed 8-bit integers or unsigned if the u flag is specified.
s, S, t	Extract pairs of bytes as 16-bit signed, or unsigned if the u flag is specified, integers in little endian, big endian and native order respectively.
i, I, n	Extract pairs of bytes as 32-bit signed, or unsigned if the u flag is specified, integers in little endian, big endian and native order respectively.
w, W, m	Extract pairs of bytes as 64-bit signed, or unsigned if the u flag is specified, integers in little endian, big endian and native order respectively.
r, R, f	Extract single precision floating point numbers stored in little endian, big endian and native order respectively.
q, Q, d	Extract double precision floating point numbers stored in little endian, big endian and native order respectively.

Details and examples of each format are below.

Binary scan: a, A

The a specifier denotes a single byte field. The value is stored as a Unicode character in the range U+0000-U+00FF. The A specifier is similar with the solitary difference that **trailing** spaces and zero bytes are stripped from **each** value stored.

```
% set bin "abc   def  " ❶
→ abc   def
% binary scan $bin a5a* val1 val2
→ 2
% puts "<$val1>, <$val2>"
→ <abc  >, < def  >
% binary scan $bin A5A* val1 val2
→ 2
% puts "<$val1>, <$val2>"
→ <abc>, < def>
```

❶ Remember for pure ASCII this is the same as [`binary format a* "abc def "`]

Binary scan: b, B

The b specifier parses bits in a byte in low to high order storing them in the variable as a string of 0 and 1 characters. The B specifier is similar except that the bits are processed in high to low order within a byte.

```
% binary scan "\x00\x5f\xaa" b13b* val1 val2
→ 2
% puts "$val1, $val2"
→ 0000000011111, 01010101
% binary scan "\x00\x5f\xaa" B13B* val1 val2
→ 2
% puts "$val1, $val2"
→ 0000000001011, 10101010
```

Note how each field specifier always begins at a byte boundary. The first specifier maps 13 bits. The remaining 3 bits to the next byte are skipped since the next specifier will only start at the next byte boundary.

Binary scan: h,H

Parses the binary data into a string of hexadecimal digits. The digits are taken from low to high order for each byte for h and the (natural) high to low order for H.

```
% binary scan "\xab\xcd\xef" H3H* val1 val2
→ 2
% puts "$val1, $val2"
→ abc, ef
% binary scan "\xab\xcd\xef" h3h* val1 val2
→ 2
% puts "$val1, $val2"
→ bad, fe
```

Again, note how each field specifier always begins at a byte boundary.

Binary scan: c

The byte(s) in the binary string are converted to signed 8-bit integers and stored in the corresponding variable as a list. Adding the u flag stores treates the bytes as unsigned 8-bit integers.

```
% binary scan \xff\x00\x01\xfe\x0f\x80 cc2c* var1 var2 var3
→ 3
% puts "$var1, $var2, $var3"
→ -1, 0 1, -2 15 -128
% binary scan \xff\x00\x01\xfe\x0f\x80 cuc2cu* var1 var2 var3
→ 3
% puts "$var1, $var2, $var3"
→ 255, 0 1, 254 15 128
```

Binary scan: s, S, t

The data is interpreted as 16-bit signed integers stored in little endian, big endian and native order respectively. As for the c specifier, adding the u flag results in a field being treated as unsigned.

```
% binary scan \xff\x00\x00\xff\xff\x00\x00\xff s2su* val1 val2
→ 2
% puts "$val1, $val2"
→ 255 -256, 255 65280
% binary scan \xff\x00\x00\xff\xff\x00\x00\xff S2Su* val1 val2
→ 2
% puts "$val1, $val2"
→ -256 255, 65280 255
```

Binary scan: i, I, n

The data is interpreted as 32-bit signed integers stored in little endian, big endian and native order respectively. Adding the u flag results in a field being treated as unsigned.

```
% binary scan \x00\x00\x00\xff\x00\x00\x00\xff iiu val1 val2
→ 2
% puts "$val1, $val2"
→ -16777216, 4278190080
```

Binary scan: w, W, m

The data is interpreted as 64-bit signed integers stored in little endian, big endian and native order respectively. Adding the u flag results in a field being treated as unsigned.

```
% binary scan \xff\x00\x00\x00\x00\x00\x00\x00 wu val1
→ 1
% puts "$val1"
→ 255
% binary scan \xff\x00\x00\x00\x00\x00\x00\x00 W val1
→ 1
% puts "$val1"
→ -72057594037927936
```

Binary scan: r, R, f

The data is interpreted as single precision floating point numbers stored in little endian, big endian and native order respectively.

```
% bin2hex [set bin [binary format r 2.71828]]
→ 4d f8 2d 40
% binary scan $bin r e
→ 1
% puts "$e"
→ 2.718280076980591
```

The difference of course stems from floating point representation rounding errors.

Binary scan: q, Q, d

The data is interpreted as double precision floating point numbers stored in little endian, big endian and native order respectively.

```
% bin2hex [set bin [binary format q 3.14159]]
→ 6e 86 1b f0 f9 21 09 40
% binary scan $bin q pi
→ 1
% puts "$pi"
→ 3.14159
```

4.13.4.2. Cursor movement for scanning

In addition to the field types, the scan specification can include the cursor movement characters shown in Table 4.18 that control the scan position for the next specifier.

Table 4.18. Binary scan cursor movement characters

Specifiers	Description
x	Moves the cursor forward.
X	Moves the cursor backward.
@	Moves the cursor to an absolute position.

Binary scan: x

Moves the cursor forward by one byte or the specified count. If count is specified as * or is larger than the remaining byte count, the cursor is placed at the end of the input binary string.

```
binary scan \x01\x02\x03\x04\x05\06 cxcx2c val1 val2 val3 → 3
puts "$val1, $val2, $val3"                              → 1, 3, 6
```

Binary scan: X

| Moves the cursor backward by the specified count or by one if no count is specified. If the count is * or greater than the current position, the cursor is placed at the start.

```
binary scan \x01\x02\x03\x04\x05\06 c2Xc3X2c val1 val2 val3 → 3
puts "$val1, $val2, $val3"                               → 1 2, 2 3 4, 3
```

Binary scan: @

Moves the cursor to the absolute position given by count which must be specified. If the count is greater than the current output string length, the cursor is placed at the end of the string.

```
binary scan \x01\x02\x03\x04\x05\06 c2@0c3@5c val1 val2 val3 → 3
puts "$val1, $val2, $val3"                               → 1 2, 1 2 3, 6
```

4.14. Character encoding

Although at the script level, Tcl strings are best thought of as an abstract sequence of Unicode characters, when it comes to storing on disk or passing data to other programs, these strings need to be converted to a specific physical format as a sequence of bytes.

The method by which a character sequence is transformed into a sequence of bytes is defined by an *encoding* and naturally there are multiple ways this might be done. Standards for this purpose are defined by various international standards bodies or by system vendors for their specific platforms. For example, consider the encoding of Unicode code point sequence U+004f U+006c U+00e1 which is the Portuguese word Olá. As a physical sequence of bytes in a file it may be stored as

- 4f 6c e1 (ISO8859-9)
- 4f 6c c3 a1 (UTF-8)
- 4f 00 6c 00 e1 00 (UCS-2)

amongst many other possible encodings.

> An encoding may be variable length with different characters encoded to byte sequences of differing length. Moreover, not all characters can be represented in every encoding. In modern times, the UTF-8 encoding which is capable of representing all Unicode characters, is generally used for sharing data between applications.

Some commonly used encodings are UTF-8 which is almost universally the encoding of choice in modern protocols, ISO8859-1 intended for Western European languages, ShiftJIS for Japanese and Big5 for Chinese.

Tcl provides built-in facilities for conversion to and from a wide variety of encodings with the `encoding` command.

4.14.1. Retrieving supported encodings with `encoding names`

The list of encodings supported by the Tcl application can be obtained with the `encoding names` command.

```
% encoding names
→ cp860 cp861 cp862 cp863 tis-620 cp864 cp865 cp866 gb12345 gb2312-raw cp949 cp950 cp869 ...
```

Note the encoding names are all lower case and can differ slightly from their common usage forms.

The supported encodings can differ even within a single Tcl version as the list of included encodings can be changed at compile time though some like ascii, utf-8, unicode and iso8859-1 will always be present.

> The encoding named unicode is really a misnomer. It is actually little endian UCS-2. In any case, it is very rarely used in general data exchange though the Windows API uses it for character strings.

4.14.2. Encoding characters: encode convertto

A string can be converted to a specific encoding with the encoding convertto command.

```
encoding convertto ENCODING STRING
```

This returns a binary string containing the sequence of bytes in the specified encoding *ENCODING*. Thus assuming the variable hello contained the word Olá,

```
% bin2hex [encoding convertto iso8859-9 $hello]
→ 4f 6c e1
% bin2hex [encoding convertto utf-8 $hello]
→ 4f 6c c3 a1
% bin2hex [encoding convertto unicode $hello]
→ 4f 00 6c 00 e1 00
```

4.14.3. Decoding characters: encode convertfrom

The encoding convertfrom command performs the inverse operation, converting encoded data to a string of characters.

```
encoding convertfrom ENCODING BINSTRING
```

Here *ENCODING* is the name of the encoding that the binary string *BINSTRING* was encoded in. Thus the inverse of our previous encoding would be of the form

```
% encoding convertfrom utf-8 "\x4f\x6c\xc3\xa1"
→ Olá
```

4.14.4. Adding new encodings: encoding dirs

The encodings supported within a Tcl executable can be extended by adding new ones. The encoding dirs command returns a list of directories containing files with extension .enc from which encoding definitions are loaded.

```
% encoding dirs
→ c:/tcl/866/x64/lib/tcl8.6/encoding
```

New encodings can be placed in one of the directories returned by the command.

You can also change the list of directories that are searched by supplying an additional argument to the command. For example, to add a new directory to the search path for encodings,

```
% encoding dirs [linsert [encoding dirs] end C:/my/extra/encodings]
→ c:/tcl/866/x64/lib/tcl8.6/encoding C:/my/extra/encodings
```

The linsert command that we will see in Section 5.4.3 inserts elements into a list.

4.14.5. The system encoding

The system encoding is the encoding used when Tcl makes system calls that take string arguments. The `encoding system` command returns the encoding in use for this purpose.

```
% encoding system
→ cp1252
```

Because the ramification of doing so can be both unexpected and severe, we will not mention that the encoding used for system interaction can be changed by supplying the name of another encoding as an argument to this command. Do not do that unless you *really* know what you are doing.

4.14.6. Reading and writing encoded data

Tcl provides the ability to configure input and output streams to automatically convert from and to a specific encoding without having to explicitly invoke the encode command. We will discuss this when we talk about I/O and channel encodings in Section 9.3.8.

There are times though when you need to explicitly use the encoding command, for example when you need to calculate a checksum over the encoded data. In this case, you need to remember that the channel must be placed in binary mode so that it does not do any encoding itself. Otherwise conversions will effectively happen twice which is probably not what you want.

4.15. Localization and message catalogs

> To speak another language is to possess a second soul.
>
> — Charlemagne

Tcl's message catalog facility provides a means for applications that support multiple languages to easily display text in the user's preferred language. It separates the application code from the language specific text allowing for easy addition of new languages, modification of existing text etc. without having to change the application.

As an introductory example, consider localizing our famous `Hello world!` greeting.

```
% puts "Hello world!"
→ Hello world!
```

The message catalog commands are implemented by the `msgcat` package so we need to load that first.

```
% package require msgcat
→ 1.6.0
```

The translation for the greeting has to be defined. Normally this is done in message catalog files that are loaded by the application as we will see. But for our example we will just define it interactively.

```
% msgcat::mcset fr "Hello world!" "Bonjour le monde!"
→ Bonjour le monde!
```

The command to output our greeting now becomes

```
puts [msgcat::mc "Hello world!"]
→ Hello world!
```

Now, to switch to French at the user's request, we would simply change the locale to `fr`.

```
% msgcat::mclocale fr
→ fr
```

Our greeting would then show up in French.

```
% puts [msgcat::mc "Hello world!"]
→ Bonjour le monde!
```

Notice that we only needed to add appropriate entries in the message catalog; **the actual** puts **call itself did not have to change after switching to French**.

4.15.1. Locales

A *locale* is a container for a collection of settings such as time and date formats and string translations. Locales are identified in Tcl by a locale string consisting of

- a language code as defined in international standard ISO 639
- optionally followed by an _ and a country code as defined in standard ISO 3166,
- optionally followed by an _ and a system specific code.

For example, en identifies the generic English locale while en_US and en_GB identify the variations for USA and Great Britain respectively.

When an application starts, the initial locale is set based on the values of the LC_ALL, LC_MESSAGES and LANG environment variables in that order. On Windows, if none of these are defined, the locale is retrieved from registry settings. If none of these are available, the initial locale is set to C.

4.15.1.1. Retrieving and setting the locale: mclocale

The msgcat::mclocale is used to set and retrieve the current locale for the application.

```
msgcat::mclocale ?LOCALE?
```

If no argument is specified, the command returns the current locale. If *LOCALE* is specified, the current locale is changed to the one specified.

```
% msgcat::mclocale
→ fr
% msgcat::mclocale en_gb
→ en_gb
```

In the case of an application running multiple Tcl interpreters, the mclocale only changes the locale for the interpreter in which the command is invoked. We will discuss multiple interpreters in Chapter 20.

4.15.1.2. Locale inheritance: mcpreferences

Locales are structured in a hierarchy so for example en_gb inherits from en. If a setting is not found in en_gb, it will be looked up in en. The `msgcat::mcpreferences` returns this list of locales.

```
msgcat::mcpreferences → en_gb en {}
```

The top of this inheritance is always the ROOT locale identified by the empty string.

4.15.2. Creating message catalogs: `mcset`, `mcmset`, `mcflset`, `mcflmset`

The translations for each locale are stored in separate files within a single directory. The files have the form *LOCALE*.msg where *LOCALE* is a lower case string identifying the locale. Thus the file es.msg will store the Spanish translations. As a special case the translations for the ROOT locale are stored in a file called ROOT.msg (note the upper case name).

Each application or package will normally store all its localization files within a single application or package specific directory. The Tcl core localization files are stored in the *TCLINSTALLDIR*/lib/Tcl*VERSION*/msgs directory.

Within each file, the localization strings for that locale are defined using one of the four commands msgcat::mcset, msgcat::mcmset, msgcat::mcflset and msgcat::mcmflset.

```
mcset    LOCALE KEY ?LOCALIZEDSTRING?
mcmset   LOCALE LOCALIZATIONLIST
mcflset  KEY ?LOCALIZEDSTRING?
mcflmset LOCALIZATIONLIST
```

The commands are all similar and provide different syntactic conveniences.

We have already seen an example of mcset which defines mapping of a single key for a specific locale. If *LOCALIZEDSTRING* is not specified, it defaults to *KEY* itself. Thus the following two are equivalent.

```
msgcat::mcset en_us "Hello world!" "Hello world!"
msgcat::mcset en_us "Hello world!"
```

The msgcat::mcmset is both more convenient and more efficient when multiple strings are being defined. It takes an argument *LOCALIZATIONLIST* which is a list of alternating keys and localized strings. Thus the following

```
msgcat::mcset fr "Hello world!" "Bonjour le monde!"
msgcat::mcset fr "Goodbye cruel world!" "Adieu monde cruel!"
```

may be more conveniently written as

```
msgcat::mcmset fr {
    "Hello world!"          "Bonjour le monde!"
    "Goodbye cruel world!" "Adieu monde cruel!"
}
```

when many strings are being defined.

The mcflset and mcflmset are analogous to mcset and mcmset respectively except they do not even require specification of the locale. They can only be used inside of message catalog files loaded with the mcload command and default to the locale based on the file name being loaded. For example, the following inside a message catalog file de.msg will add the strings to the de locale.

```
msgcat::mcflmset {
    "Hello world!"          "Hallo Welt!"
    "Goodbye cruel world!" "auf Wiedersehen, grausame Welt!"
}
```

Both mcflset and mcflmset will raise exceptions unless called via a mcload command.

4.15.3. Loading message catalogs: mcload

Before the translations defined by an application can take effect, they must be loaded with the msgcat::mcload command.

```
msgcat::mcload MSGCATDIR
```

The message catalog files may be stored in any directory but it is common to store them in a subdirectory under the package's script directory. Thus a common method for loading message files is invoking

```
msgcat::mcload [file join [file dirname [info script]] msgs]
```

from the main package script at the time it is loaded.

4.15.4. Retrieving localized strings: mc

The msgcat::mc command returns localized strings based on the current locale as returned by mclocale.

```
msgcat::mc KEY ?ARG ...?
```

We have already seen this command and examples of use at the beginning of this section. We now expand on some of its other features.

The first argument KEY passed to the mc command is used as the key for looking up the localized strings. If no entry is found, by default the key itself is returned from the command unless this behaviour is changed. In our earlier example, we used the English localization itself as the key but this is not necessary. We could have used any token as the translation lookup key, say greet001.

```
% puts [msgcat::mc greet001] ❶
→ greet001
% msgcat::mcset en greet001 "Hello world!"
→ Hello world!
% puts [msgcat::mc greet001]
→ Hello world!
```

❶   We have not assigned a value for greet001 for the current locale

Although it is convenient to use the English (or any language for that matter) localization as the key so as to not have to explicitly define an entry in the message catalog for it, this relies on the default behaviour of the mcunknown command. It is therefore sometimes recommended to explicitly define localizations for every string and language pair as above.

If any additional arguments are passed to the mc command, it passes them to the format command along with the localized string and returns the result. For example, assume the fr and en message catalogs contain the following lines respectively:

```
msgcat::mcset fr TIME "L'heure actuelle est %s"
msgcat::mcset en TIME "The current time is %s"
```

We can print the current time as

```
% set now [clock format [clock seconds] -format %T]
→ 11:45:41
% puts [msgcat::mc TIME $now]
→ The current time is 11:45:41
```

The above is roughly equivalent to

```
% set fmt [msgcat::mc TIME]
→ The current time is %s
% puts [format $fmt $now]
→ The current time is 11:45:41
```

If we were to switch to the fr locale,

```
% msgcat::mclocale fr
→ fr
% puts [msgcat::mc TIME $now]
→ L'heure actuelle est 11:45:41
```

we get the French version of the message.

4.15.5. Partitioning catalogs with namespaces

One issue that can arise when multiple independent packages have their own message catalogs is the potential for conflict between the key strings used by each package. The message catalog system solves this through the use of namespaces, a topic we cover in Chapter 12.

Packages that make use of the message catalogs should invoke the mcset family of definition commands from within the package's namespace. For example, the de.msg file example in the previous section should contain the following content instead.

```
namespace eval greetings {
    msgcat::mcflmset {
        "Hello world!"         "Hallo Welt!"
        "Goodbye cruel world!" "auf Wiedersehen, grausame Welt!"
    }
}
```

The localized strings are then loaded within the greetings namespace and will not conflict with localizations defined in the global or other namespaces.

Conversely, when localized strings are retrieved with the `mc` command, it looks up the message catalog within the context of the namespace from which it is called.

Here is an illustrative example. The English localization file for our anniversary package may contain

```
namespace eval anniversary {
    msgcat::mcset en greeting "Happy anniversary!"
}
→ Happy anniversary!
```

Similarly, the Christmas greetings package localization file contains

```
namespace eval xmas {
    msgcat::mcset en greeting "Merry Christmas!"
}
→ Merry Christmas!
```

These files define the greeting message based on the occasion as reflected by the containing namespace. The printed greeting as shown below will then depend on the namespace context in which the message is retrieved with the mc command.

```
% msgcat::mclocale en_us
→ en_us
% puts [msgcat::mc greeting]                          ❶
→ greeting
% namespace eval anniversary {puts [msgcat::mc greeting]}
→ Happy anniversary!
% namespace eval xmas {puts [msgcat::mc greeting]}
→ Merry Christmas!
```

❶   Will output greeting because no message catalog entry for greeting in global namespace

One final point to be related to the use of namespaces with msgcat is that if a namespace does not define a message catalog entry that matches the locale, all ancestor namespaces are searched in order. So if the anniversary namespace had a child namespace golden, the following would work.

```
namespace eval anniversary::golden {puts [msgcat::mc greeting]} → Happy anniversary!
```

On failing to find a greeting entry in any suitable English locale in the anniversary::golden namespace, the mc command would check anniversary and the global namespaces in turn.

4.15.6. Handling unknown message keys

When the mc command does not find a localization defined in the current locale, it invokes the msgcat::mcunknown procedure and returns its value. The default definition of mcunknown simply returns the passed lookup key. An application can redefine this to take some other action it wishes, like logging or raising an error, or using an online automatic translation API etc.

The redefined command is called using the following syntax and should be defined accordingly.

```
msgcat::mcunknown LOCALE KEY ?ARG ...?
```

where LOCALE is the locale to be looked up. KEY and the remaining arguments are as passed to the mc command.

The return value from the command is passed back to the original caller. Therefore any redefinition should take care to handle additional arguments in the same manner as mc.

4.16. Data compression

The zlib family of data compression algorithms and data formats are widely used in the computing world. The most common applications include compression in the HTTP protocol used for Web access and the *zip* and *gzip* file compression formats. Because of their ubiquity, Tcl provides built-in commands for compressing and decompressing using these formats.

The zlib family really consists of three different specifications:

- The raw compression algorithm, DEFLATE, defined in RFC 1951 [10]. We refer to data compressed using this algorithm as *deflated* data.
- A data format, the *ZLIB compressed data format*, defined in RFC 1950 [11] that wraps the raw deflated data to include additional metadata such as checksums. We refer to this as *zlib compressed* data.
- Another data format, the *GZIP file format*, defined in RFC 1952 [12] that also wraps the raw compressed data to include additional metadata. We refer to this as *gzip compressed* data.

Tcl provides commands related to all three of these. Moreover, these commands fall into four categories:

---

[10] https://tools.ietf.org/html/rfc1951
[11] https://tools.ietf.org/html/rfc1950
[12] https://tools.ietf.org/html/rfc1952

- Commands that operate on the entire data to be compressed or decompressed. These are discussed below in Section 4.16.1.
- Commands that operate in *stream* mode where the data is incrementally compressed or decompressed These are described below in Section 4.16.2.
- Channel transforms where data is **transparently** compressed or decompressed during input-output operations. We will postpone a discussion of these to Section 17.2.5 in the Chapter 17 where we introduce channel transforms.
- Utility commands for calculating checksums. These are discussed in Section 4.16.3.

All commands related to zlib compression are subcommands of the zlib command.

> **!** Since the compression algorithms all operate on binary data, they must be passed binary strings, for example those directly constructed with the binary format command or by encoding text strings with encoding convertto command.

4.16.1. Compressing strings

The commands discussed in this section expect the binary string that is to be operated on to be provided in a single argument. Let us create such a string to use for our examples.

```
% set bin [encoding convertto utf-8 [string repeat abcd 200]]
```

4.16.1.1. Raw DEFLATE compression: zlib deflate|inflate

The first pair, zlib deflate and zlib inflate, implement compression and expansion respectively using the raw DEFLATE algorithm of RFC 1951. No headers or metadata are attached to the compressed data.

```
zlib deflate BINSTRING ?LEVEL?
zlib inflate COMPRESSED ?BUFFERSIZE?
```

The *LEVEL* argument should be a number between 0 and 9 with a level of 0 indicating no compression and 9 indicating maximal compression, at the cost of performance. The default value is 1.

```
% bin2hex [set zbin [zlib deflate $bin]]
→ 4b 4c 4a 4e 49 1c c5 a3 78 14 63 c5 00
```

(Our repeated input string results in excellent compression!)

The inverse command is zlib inflate.

```
% encoding convertfrom utf-8 [zlib inflate $zbin]
→ abcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabc...
```

When uncompressing, the inflate command will grow the buffer required for the uncompressed data as required. As a performance optimization, you can specify *BUFFERSIZE* as the expected length of data so that memory reallocations are avoided.

```
% encoding convertfrom utf-8 [zlib inflate $zbin 1000]
→ abcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabc...
```

4.16.1.2. Zlib compression: `zlib compress|decompress`

A second set of commands, `zlib compress` and `zlib decompress`, implement compression and expansion respectively using the Zlib compressed data format defined in RFC 1950. This format uses the same DEFLATE compression algorithm used by the `zlib deflate` but includes additional meta information, in particular an Adler-32 checksum. The well known `zip` compressed files primarily use this format.

The syntax is similar to that of `zlib deflate` and `zlib inflate`.

```
zlib compress BINSTRING ?LEVEL?
zlib decompress COMPRESSED ?BUFFERSIZE?
```

The optional *LEVEL* and *BUFFERSIZE* parameters have the same meaning as described above for the `zlib deflate` and `zlib inflate` commands respectively.

```
% bin2hex [set zbin [zlib compress $bin]]
→ 78 9c 4b 4c 4a 4e 49 1c c5 a3 78 14 63 c5 00 aa 4f 33 e0
% encoding convertfrom utf-8 [zlib decompress $zbin]
→ abcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabc...
```

Notice from the output of the `bin2hex` command that the Zlib compression format contains within it the output of the raw DEFLATE data we printed in the previous section.

4.16.1.3. Gzip compression: `zlib gzip|gunzip`

The last set of commands in this category `zlib gzip` and `zlib gunzip` are also based on the DEFLATE compression algorithm but this time using the format defined in RFC 1952. The popular `gzip` and `gunzip` command line utilities that produce `.gz` files use this format.

The syntax of these commands differs slightly from their brethren because the format supports more metadata values.

```
zlib gzip BINSTRING ?-level LEVEL? ?-header HDRDICT?
zlib gunzip COMPRESSED ?-headerVar VARNAME?
```

The `-level` option serves the same purpose as the *LEVEL* argument in `zlib deflate` except it is supplied as an option switch instead of a plain argument.

The `-header` option allows the caller to supply the associated metadata. *HDRDICT* should be a dictionary containing any of the keys shown in Table 4.19.

Table 4.19. Gzip header keys

Key	Value
`comment`	A comment to be included in the Gzip metadata
`crc`	A boolean value. If `true`, the GZIP header CRC is computed. This should be `false` if interoperability with the `gzip` program is desired.
`filename`	The name of the file that was the source of the data.
`os`	The operating or file system type code as defined in RFC 1952. Common ones are 0 for FAT, 3 for Unix, 11 for NTFS.
`time`	The last modified time of the file as returned by `clock seconds` or `file mtime`.
`type`	One of the values `binary` or `text` indicating the type of data being compressed. Programs handling Gzip format files may or may not pay heed to this flag.

All keys above are optional.

Correspondingly, if the -headerVar option is used with the zlib gunzip command, the metadata values retrieved from the compressed data are stored in the variable *VARNAME* in the caller's context. The data is stored as a dictionary which may contain the same keys shown in Table 4.19 and an additional key, size, that contains the size of the compressed data.

```
% set hdr [list time [clock seconds] comment "A demo file"]
→ time 1499148941 comment {A demo file}
% bin2hex [set zbin [zlib gzip $bin -header $hdr]]
→ 1f 8b 08 10 8d 32 5b 59 00 00 41 20 64 65 6d 6f 20 66 69 6c 65 00 4b 4c 4a 4e 49 1c c5 ...
% encoding convertfrom utf-8 [zlib gunzip $zbin -headerVar hdr2]
→ abcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabc...
% print_dict $hdr2
→ comment   = A demo file
  crc       = 0
  os        = 0
  size      = 800
  time      = 1499148941
...Additional lines omitted...
```

4.16.2. Compressing streams

The commands discussed in the previous section all work with in a "single-shot" manner where all the data that is to be operated on is provided in one call. This is neither convenient nor performant in terms of memory usage when the data becomes available in a discrete or piecemeal fashion. For such cases, Tcl provides the zlib stream command where data can be fed into the compression engine in incremental fashion.

Before going into the details, a short example that mirrors our previous ones:

```
set strm [zlib stream deflate]
for {set i 0} {$i < 200} {incr i} {
    $strm put [encoding convertto utf-8 "abcd"]
}
$strm finalize
set zbin [$strm get]
$strm close
bin2hex $zbin
```

The script creates a new zlib stream command that will compress any data passed to it via the put subcommand. When we are done, finalize completes the compression process. The compressed data can then be retrieved with get. Finally, we release resources associated with the stream by calling close.

The sequence of commands for decompression would be very similar except that we call zlib stream inflate to create the stream. Likewise, to compress using the Zlib or Gzip formats, we would call zlib stream compress and zlib stream gzip respectively.

4.16.2.1. Creating a compression stream

A compression stream is created with a command of the form

```
zlib stream ENGINE ?OPTIONS?
```

The *ENGINE* parameter is one of deflate, inflate, compress, decompress, gzip or gunzip and corresponds to the various compression and decompression commands described in the preceding sections. The command returns a new command representing a streaming compression instance to which data can be written and read.

The options that can be used with the various engines is shown in Table 4.20.

Table 4.20. Compression stream options

Option	Description
-dictionary *BINDATA*	Specifies a compression dictionary to be used for compressing or decompressing. *BINDATA* is a binary string and is not to be confused with a Tcl dictionary. See the explanation of the preset dictionary in RFC 1950[13]. This option can be used with the deflate, inflate, compress and decompress engines.
-header *HEADER*	Specifies the Gzip format metadata header. This option can only be used with the gzip engine.
-level *LEVEL*	Specifies the compression level. This option can be used with the deflate, compress and gzip engines.

Let us open streams to do Gzip compression and decompression.

```
% set compressor [zlib stream gzip -header {comment "A zlib demo"}]
→ ::tcl::zlib::streamcmd_2
% set decompressor [zlib stream gunzip]
→ ::tcl::zlib::streamcmd_3
```

The commands returned are then invoked for various read and write operations on the stream.

4.16.2.2. Writing to a compression stream

A stream is written to with the put command.

```
STREAM put ?OPTIONS? BINDATA
```

Multiple put commands may be invoked to add data in incremental fashion. For example,

```
% $compressor put [encoding convertto utf-8 "abcd"]
% $compressor put [encoding convertto utf-8 "efgh"]
```

The command supports the options shown in Table 4.21.

Table 4.21. Compression stream put options

Option	Description
-dictionary *BINDICT*	Sets *BINDICT* as the compression dictionary as described in Table 4.20.
-finalize	The use of this option is described in Section 4.16.2.3.
-flush	The use of this option is described in Section 4.16.2.9.
-fullflush	The use of this option is described in Section 4.16.2.9.

Note that only one of -finalize, -flush or -fullflush may be specified.

4.16.2.3. Finalizing a compression stream

Once all data has been written to a stream, it needs to be told accordingly so that it can complete the compression process, write out meta data and so on. This can be done in one of two ways:

If you know the data you are writing is the last piece, you can specify the -finalize option to the put command.

---

[13] https://tools.ietf.org/html/rfc1950

```
% $compressor put -finalize [encoding convertto utf-8 "ijkl"]
```

Alternatively, in cases where you do not know a priori that the data being written is the final bit, you can call the `finalize` command once you know that no more data will be forthcoming.

```
$compressor finalize
```

4.16.2.4. Getting the stream checksum

The compression stream keeps track of the checksum of the uncompressed data written to it. This checksum can be retrieved at any point with the `checksum` command.

```
% $compressor checksum
→ 4135066404
```

4.16.2.5. Reading from a compression stream

The compressed data is read back from a compression stream with the `get` command.

```
STREAM get ?COUNT?
```

If the `COUNT` argument is specified the command returns that many compressed bytes from the stream. If it is not specified, all remaining data in the stream is returned.

```
set compressed_1 [$compressor get 2]
set compressed_remaining [$compressor get]
bin2hex $compressed_1 $compressed_remaining
→ 1f 8b 08 10 00 00 00 00 00 00 41 20 7a 6c 69 62 20 64 65 6d 6f 00 4b 4c 4a 4e 49 4d 4b ...
```

4.16.2.6. Reusing a compression stream

To reuse an existing stream for new data, call the `reset` command. The stream can then be used to compress exactly as if it were a new stream opened with `zlib stream`. This command is a little more efficient than closing the stream and opening a new one.

```
$compressor reset → (empty)
```

4.16.2.7. Closing a compression stream

Once a compression stream is no longer required, it must be released by calling the `close` command.

```
$compressor close → (empty)
```

4.16.2.8. Decompression streams

Decompression streams work exactly like compression streams except for the engine used. We can incrementally decompress by writing the compressed data to the stream with `put`.

```
% set decompressor [zlib stream gunzip]
→ ::tcl::zlib::streamcmd_4
% $decompressor put $compressed_1
% $decompressor put -finalize $compressed_remaining
```

In the case of Gzip format, we can retrieve the metadata header for the compressed data with `header` command.

```
% print_dict [$decompressor header]
→ comment    =
  crc        = 0
  filename   =
  os         = 0
  type       = binary
```

As before the decompressed data itself is obtained with `get` command.

```
set decompressed [$decompressor get 2]
append decompressed [$decompressor get]
encoding convertfrom utf-8 $decompressed
→ abcdefghijkl
```

4.16.2.9. Flushing of compression streams

Because an explanation of flushing requires an understanding of how the DEFLATE algorithm works, we do not go into details but refer you to the article by Thomas Pornin[14]. From a practical point of view, flushing a compression stream allows the decompressing end to correctly decompress the data under certain conditions.

- In the case of a "sync flush", a decompressor can decompress data up to the point at which the compressor invoked the flush even in case of errors in writing subsequent data.
- In the case of a "full flush", it additionally allows a decompressor to decompress data beyond the point of the flush even in case of errors in transmission of earlier bytes.

For a Tcl Zlib compression stream, a sync flush or full flush is effected by calling the `flush` and `fullflush` commands respectively.

```
STREAM flush
STREAM fullflush
```

Alternatively, you can use the `-flush` or `-fullflush` options when invoking the `put` command.

Flushing incurs a cost in compression efficiency. Generally, its use is dictated by the upper layer protocols that make use of compression. For example, compression in HTTP entails no flushing as the content is sent as a single blob. Packet oriented protocols on the other hand may mandate flushing at packet boundaries.

4.16.3. Checksum computation

The `zlib` command has two subcommands for computing Adler-32 and CRC-32 checksums on a binary string.

```
zlib adler32 BINDATA ?INITDATA?
zlib crc32 BINDATA ?INITDATA?
```

`BINDATA` is the binary string whose checksum is to be computed. `INITDATA` is used to initialize the computation.

```
zlib adler32 [encoding convertto utf-8 abcd] → 64487819
zlib crc32 [encoding convertto utf-8 abcd]    → 3984772369
```

---

[14] http://www.bolet.org/~pornin/deflate-flush.html

4.16.4. Channel compression transforms

In this chapter we have not discussed one additional mechanism for compressing data — through I/O channel transforms. We will postpone that discussion to Section 17.2.5.

4.17. Chapter summary

In this chapter we covered Tcl's facilities for dealing both text and binary data. Along the way we covered the more advanced topics of pattern matching, compression and internationalization. In succeeding chapters, we will look at more structured forms of data in Tcl such as lists and dictionaries.

4.18. References

WWWUNICODE
Unicode Tutorials and Overviews[15], The Unicode Consortium. Unicode tutorials and resources from the Unicode consortium.

FRIEDL
Mastering Regular Expressions, Friedl, O'Reilly, 2007. Describes regular expressions and their use across multiple programming languages and applications including Tcl.

TUTREGEXP
Online Tcl tutorial[16], Provides examples and hints for how to go about designing a regular expression.

WWWREXEGG
www.rexegg.com[17], A website dedicated to regular expressions in general (not Tcl specific). It offers tutorials and detailed explanations along with examples of their use.

WWWREXINFO
www.regular-expressions.info[18], Another useful website dedicated to regular expressions similar to the above.

---

[15] http://unicode.org/standard/tutorial-info.html
[16] https://www.tcl.tk/man/tcl8.5/tutorial/Tcl20a.html
[17] http://www.rexegg.com
[18] http://www.regular-expressions.info

<div style="text-align: right; font-size: 3em; font-weight: bold;">5</div>

Lists

> The human animal differs from the lesser primates in his passion for lists.
>
> — H. Allen Smith

A *list* in Tcl is an ordered collection, or sequence, of values and serves a purpose similar to that of integer indexed arrays in other languages.

Because lists are pervasive in Tcl programming, there are a number of commands for their manipulation. Further, lists can be nested, allowing their use for construction of more elaborate data structures like trees. Many commands therefore also support operations that facilitate such use.

5.1. Constructing lists

We will begin our discussion of lists with the basic means by which they can be created.

5.1.1. List literals

As with all values, Tcl lists have a string representation and they can be constructed in literal form through that string format. For example, the command

```
set mylist {One Two Three} → One Two Three
```

assigns the string One Two Three to the variable mylist. A command that operates on lists will interpret this string as a list with three elements One, Two and Three.

```
llength $mylist    → 3
lindex $mylist end → Three
```

It is common practice to define literal lists using braces as opposed to quotes. **However, the use of braces does not in any way or form imply that the value is a list.** Both values are simply strings. As described in Section 3.3, both braces and quotes denote string literals with the only difference being the treatment of special characters. Thus the following two assignments result in the same list as demonstrated below.

```
% set la {{Item One} Item_with_no_spaces "Item Three"}
→ {Item One} Item_with_no_spaces "Item Three"
% set lb "\"Item One\" Item_with_no_spaces {Item Three}"
→ "Item One" Item_with_no_spaces {Item Three}
% foreach a $la b $lb { puts "$a == $b" }
→ Item One == Item One
  Item_with_no_spaces == Item_with_no_spaces
  Item Three == Item Three
```

Multiple string representations

One point to be noted about lists is that they do not have a "standard" string representation. Just as 0xa and 10 are different strings representing the same number in hexadecimal and decimal format, strings that do not compare

equal may represent the same list. For example, the two strings below are valid representations of the same 4-element list though they do not compare equal as strings.

```
"    this    is    a    list    "
"this is a list"
```

Although a list may have many string representations, Tcl always ensures that the **individual** elements of lists are retrieved in exactly the format that they were created. In other words, adding an element to a list does not change the element's string representation and the following invariant holds for all list operations:

```
[lindex [linsert LIST INDEX ELEM] INDEX] eq ELEM
```

where eq is the string equality operator. Essentially what that means is that inserting and then retrieving an element will not change its string representation.

Parsing string representations of lists

The manner in which the string is parsed into elements is the same as what was described in Section 3.1 for parsing commands into words except that no variable and command substitution is performed. In particular, note that the list commands will still perform **backslash substitution** when converting a string value to a list.

Because the parsing and interpretation of a string as a list is often a source of confusion, let us walk through another example. Consider the following two commands to retrieve the first element of a list using lindex.

```
lindex "a\ b c" 0 → a
lindex {a\ b c} 0 → a b
```

Note that the characters within the quotes and braces are identical but the results are different. If you understand why, skip to the next section. Else read on.

In the first case, following the rules described in Chapter 3, the Tcl parser treats backslashes inside the quoted string as escapes for the character that follows. It replaces the backslash and following space with a single space character. Consequently, the lindex command receives the string value

```
a b c
```

as its first argument. As we said earlier, the list based commands apply the same rules as the Tcl parser for breaking up a string into words. Thus when lindex parses the argument following those rules, the string gets parsed into three words (list elements), a, b and c. These form the elements of the constructed list whose first element is returned.

On the other hand, since backslash replacement is not done inside braces, in the second case lindex receives the string value

```
a\ b c
```

Now when lindex parses the value, it uses the same backslash substitution rules whereby the backslash protects the following space from being treated as a word separator. Consequently the string is parsed as just two words — a b, and c — which then make up the constructed list.

In essence, when enclosed in double quotes, the argument undergoes one round of backslash substitution by the Tcl parser and then a second round when the lindex command interprets the contents of the variable as a list. When the string is enclosed in braces on the other hand, it is protected from substitution by the Tcl parser and only undergoes the backslash substitution by lindex.

> For those concerned with the performance impact of the parsing and interpretation of the string: Tcl maintains an internal representation of the list just as it does for numeric values. As long as the values are further manipulated using list commands, there is never an impact due to conversions to and from string representations.

In most cases, the limited substitution inside braces makes their use more convenient that the use of quotes when defining lists. In more complex cases, an even better option for list construction is the `list` command described next.

5.1.2. Basic list construction: `list`

Lists can also be constructed explicitly with the `list` command which takes an arbitrary number of arguments and returns a list containing those elements.

```
list ?VALUE VALUE ...?
```

Thus the example in the previous section could also have been written as

```
% set items [list "Item One" Item_with_no_spaces "Item Three"]
→ {Item One} Item_with_no_spaces {Item Three}
```

The `list` command is most useful when the values used to construct the list reside in variables or are the result of a command.

```
set i 0                                    → 0
set numbers [list $i [incr i] [incr i]] → 0 1 2
```

> Do **not** use string interpolation in lieu of the `list` command to construct lists whose elements come from variables and command evaluation. Although string interpolation may seem to work in some cases, it will not give the desired result when the values contain whitespace or other special characters as in the example below.
>
> ```
> set a "First elem"        → First elem
> set b "Second elem"       → Second elem
> lindex [list $a $b] end → Second elem ❶
> lindex "$a $b" end        → elem ❷
> ```
>
> ❶   Correctly retrieves last element
> ❷   Incorrect as a result of whitespace in elements
>
> In general, you should always stick to commands that operate on lists to work with data that is structured as lists. Do not use string interpolation or string targeted commands for the purpose.

5.1.3. Splitting strings into lists: `split`

The `split` command constructs a list by breaking apart a string into a list of substrings based on a set of delimiter or separator characters.

```
split STRING ?SEPARATORS?
```

The *SEPARATORS* argument to `split` is a string that is treated as a **set** of separator characters, not as a single string to be treated as a separator. If unspecified, it defaults to the set of whitespace characters. Thus to break up a paragraph (simplistically) into words based on whitespace,

```
% set paragraph "A sentence.\tAn exclamation! Any questions?"
→ A sentence.     An exclamation! Any questions?
% print_list [split $paragraph]
→ A
  sentence.
  An
  exclamation!
...Additional lines omitted...
```

Or to break it up into sentences based on sentence terminators,

```
% print_list [split $paragraph ".!?"]
→ A sentence
          An exclamation
   Any questions
```

Notice that the leading spaces and tabs are preserved in this last example as they are not treated as separators.

Leading and trailing as well as consecutive separators will result in empty elements in the resulting list. For example,

```
% split "  one  two  "
→ {} {} one {} two {} {}
```

The `regexp` command we saw in Chapter 4 provides an alternative to `split` for more generalized separator patterns and flexibility. The `textutil::split` package in Tcllib[1] offers `splitx` as a convenient wrapper based on this.

```
% package require textutil::split
→ 0.7
% print_list [textutil::split::splitx $paragraph {\s*[.!?]\s*}]
→ A sentence
  An exclamation
  Any questions
```

Notice the leading tabs and spaces are removed from the elements.

A commonly seen idiom for processing file content line by line is to read the entire file and then use `split` to transform it into a list of lines using the newline character as the separator. The `foreach` command can then be used to iterate over list. The pseudocode below outlines this method which is generally faster than reading the file in a line at a time.

```
set fd [open datafile.txt]
foreach line [split [read $fd] \n] {
    ...Do something with $line...
}
close $fd
```

Similarly, `split` is often used for processing a string a character at a time by passing an empty string as the separator argument.

```
% foreach char [split "string" ""] { puts $char }
→ s
  t
```

---

[1] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

```
...Additional lines omitted...
```

The `split` command and the `join` command we saw in Section 4.2.5 can be roughly thought of as inverses of each other. However, they are not exact inverses. In particular, it is not guaranteed that joining a list using a separator and then splitting that list on the same separator will result in the original list.

```
set l [list a b/c d/e f] → a b/c d/e f
set s [join $l /]         → a/b/c/d/e/f
split $s /                → a b c d e f
```

The presence of the separator character within the list elements will break the inversion.

5.1.4. Concatenating lists: `concat`

The `concat` command returns a new list formed by concatenating zero or more lists. It has the syntax

```
concat ?LIST LIST …?
```

For example,

```
% concat {a b c} {d {e f} g} {} h
→ a b c d {e f} g h
```

Note that concat preserves any nested list structure; only the outermost lists are merged.

Although `concat` is defined as operating on lists, it does not actually validate that the operands are well-formed lists. In that case, the result may not be a well formed list either. For that reason, many people think of `concat` as a command that operates on strings. However, the Tcl reference describes it as operating on lists so we will stick with that description.

One caveat to be aware of with regard to the use of `concat` with strings is that it will trim any leading or trailing whitespace from each operand. This does not affect list semantics as leading spaces are anyway ignored in the interpretation of strings as lists.

5.1.5. Repeating elements: `lrepeat`

The `lrepeat` command returns a list constructed by repeating its arguments a specified number of times.

```
lrepeat COUNT ?ELEMENT ELEMENT …?
```

It is often useful in initialization, for example

```
set download_counters [lrepeat 12 0] → 0 0 0 0 0 0 0 0 0 0 0 0
```

More than one argument may be supplied for repetition.

```
lrepeat 3 a [lrepeat 2 b] → a {b b} a {b b} a {b b}
```

5.2. List indices

Elements within a list are referenced by their position in the list using *list indices* in the same manner as described for strings in Section 4.1. These indices are 0-based so 0 references the first element in the list, 1 references the second and so on. As a special case, end can be used as an index. Unless mentioned otherwise, it refers to the last element in a list. We will explicitly note the commands where it refers to the position **after** the last element.

As for string indices, list indices may be specified in the special arithmetic form

```
INTEGER[+|-] INTEGER
end[+|-] INTEGER
```

5.2.1. Nested list indices

Elements in a list may themselves be interpreted as lists. For instance, we might represent a person as a list of three elements: a name, an address, and a date of birth where the address field is itself a list of strings.

```
set address [list "Buckingham Palace" Westminister England]
set her_majesty [list "Elizabeth Alexandra Mary" $address "21-Apr-1926"]
```

To facilitate operations on these kind of nested lists, many list commands accept a sequence of indices in place of a single index. The indices act like a "path" through the nested list where the elements of the index list identify an element at each nesting level. Thus the index 1 would refer to the second element of the outermost list, ie. the entire address. The index sequence 1 0 would in turn reference the second element of the **outer** list as before and then the first element of the **inner** (address) list, ie. Buckingham Palace. The lists and the indices may be nested to any depth.

We will see examples of nested lists as we proceed through this chapter.

5.3. Retrieving elements

5.3.1. Retrieving elements by position: lindex

The lindex command returns the element at a specific position in the list.

```
lindex LIST ?INDEX ...?
```

In the simple case where only one index is specified, the command returns the element at that position in the list.

```
% puts "[lindex $her_majesty 0] was born on [lindex $her_majesty 2]"
→ Elizabeth Alexandra Mary was born on 21-Apr-1926
```

The number of indices determines the nesting depth of *LIST*. For each provided index, the command will burrow an additional level into the element at that index effectively treating the indices as a path through the nested list.

```
% lindex $her_majesty ❶
→ {Elizabeth Alexandra Mary} {{Buckingham Palace} Westminister England} 21-Apr-1926
% lindex $her_majesty 1
→ {Buckingham Palace} Westminister England
% lindex $her_majesty 1 end
→ England
```

❶   Nesting depth of 0 (no indices specified), entire list argument is returned.

The multiple indices may be provided as a single list or as independent arguments. Thus the following are equivalent.

```
lindex $her_majesty 1 end    → England
lindex $her_majesty {1 end} → England
```

Note that lindex will return an empty string, not raise an error, if passed an index that is negative or greater than the list size.

5.3.2. Retrieving a list subrange: lrange

While lindex returns a single element, the lrange command returns a list containing all elements between the two specified indices.

```
lrange LIST FIRST LAST
```

The returned list contains all elements between indices *FIRST* and *LAST* (both inclusive) in *LIST*.

```
% set downloads_by_month {120 110 130 100 90 85 92 105 114 140 156 190}
→ 120 110 130 100 90 85 92 105 114 140 156 190
% lrange $downloads_by_month 0 2 ❶
→ 120 110 130
% + {*}[lrange $downloads_by_month end-2 end] ❷
→ 486
```

❶   Downloads by month in the first quarter
❷   Total downloads in the last quarter

Note that lrange does not support nested lists.

5.3.3. Retrieving leading elements: lassign

The lassign command sequentially assigns the leading elements in a list to the specified variables and returns a new list (possibly empty) containing the remaining elements.

```
lassign LIST ?VARNAME VARNAME …?
```

If the number of elements in the list is less than the number of variables specified, the additional variables are set to the empty list.

```
% set remaining [lassign {A B C D} x y]
→ C D
% puts "x=$x, y=$y, remaining=$remaining"
→ x=A, y=B, remaining=C D
% unset -nocomplain x y z
% lassign {A B} x y z
% puts "x=<$x>, y=<$y>, z=<$z>" ❶
→ x=<A>, y=<B>, z=<>
```

❶   Notice z is assigned an empty string.

5.4. Modifying lists

5.4.1. Appending elements: `lappend`

The `lappend` command is useful for incrementally constructing a list by adding trailing elements.

```
lappend VARNAME ?VALUE ...?
```

This command takes the name of a variable as its first argument interpreting its content as a list. Any additional arguments are treated as values to be added to the end of the list stored in the variable.

```
set greek [list alpha \u03b2]   → alpha β ❶
lappend greek \u03b3 delta       → alpha β γ delta
set greek                        → alpha β γ delta
```

❶   Beta is Unicode character U+03b2

Note that if the variable does not exist, `lappend` will create it as an empty list and then append the remaining arguments. Thus the statements

```
set greek [list alpha \u03b2]
lappend greek alpha \u03b2
```

are equivalent if the variable `greek` did not already exist.

> Because the `lappend` command modifies a **variable** "in-place" it is one of the more efficient means of constructing or modifying a list as opposed to commands like `linsert` which operate on list **values** and hence have to make a second copy of the list. For similar reasons, other list commands, such as `lset`, which operate on variables are to be preferred wherever possible to similar commands, such as `lreplace`, that operate on values.

5.4.2. Setting element values: `lset`

The `lset` command replaces any element of a list stored in a variable with a new value.

```
lset VARNAME ?INDEX ...? VALUE
```

Like `lappend`, `lset` operates on variable *VARNAME* that is presumed to contain a list. However, unlike `lappend`, which only adds new elements to the end of the list, `lset` permits assignment to any element of the list at the position specified by *INDEX*. The new list is stored in the variable and also returned as the result of the command.

```
set items {a {B C} d} → a {B C} d
lset items 2 e          → a {B C} e
```

The command supports nested lists. The multiple indices comprising the path to the nested element may be specified as a single argument or separately. Both variations are shown below.

```
lset items {1 0} X; → a {X C} e
lset items 1 end Y  → a {X Y} e
```

Indices must lie between 0 and the length of the list. In the latter case, the value is appended to the list. Note that end when used with `lset` refers to the last element in the list and end+1 is used to extend the list.

```
lset items end f   → a {X Y} f
```

```
lset items end+1 g → a {X Y} f g
```

> Because `lset` works with nested lists, it can be used to append elements to inner lists, something you cannot do with `lappend`.

5.4.3. Inserting elements: `linsert`

The `linsert` command inserts one or more elements into a list at a specified index and returns the resulting list.

```
linsert LIST INDEX ?VALUE VALUE ...?
```

The `INDEX` argument must be a single index as the command does not support nested lists. Unlike `lappend` and `lset`, `linsert` operates on a list value `LIST`, not on a variable.

```
set items {a b c d}    → a b c d
linsert $items 1 X Y Z → a X Y Z b c d
```

Unlike most list commands such as `lindex`, `lset` and `lreplace`, the `linsert` command treats the index value end as **beyond** the last element, not the last element itself. To insert before the last element in the list, specify the index as end-1.

```
lindex $items 1        → b
linsert $items 1 X     → a X b c d ❶
lindex $items end      → d
linsert $items end X   → a b c d X ❷
linsert $items end-1 X → a b c X d
```

❶　Item inserted **before** b
❷　Item inserted **after** d

5.4.4. Replacing elements: `lreplace`

The `lreplace` command returns a new list formed by replacing one or more elements of a list with zero or more new values.

```
lreplace LIST FIRST LAST ?VALUE ...?
```

Like `linsert`, it operates on list values, not variables, and does not support nested lists. All elements of `LIST` at positions between `FIRST` and `LAST` (inclusive) are removed and replaced with the new values.

```
lreplace {a b c d} 1 2 X Y → a X Y d
```

The count of replacement values does not have to equal the count of replaced elements.

```
lreplace {a b c d} 0 2 X Y   → X Y d
lreplace {a b c d} 1 1 X Y Z → a X Y Z c d
```

Like most list commands, `lreplace` interprets the index end as the **last** element in the list.

```
lreplace {a b c d} end-1 end X Y → a b X Y
```

5.4.5. Deleting elements

Just as is the case with strings, there is no separate command for removing elements from a list. Instead, the lreplace command with no replacement values specified is used for the purpose.

```
lreplace {a b c d e} 1 1 → a c d e ❶
lreplace {a b c d e} 1 3 → a e ❷
```

❶ Delete one element
❷ Delete range of elements

5.5. Transforming lists

5.5.1. Mapping list elements: lmap

The lmap command provides a generalised way to create a new list by mapping elements of a list to new values. It takes one of the following forms.

```
lmap VARNAME LIST SCRIPT
lmap VARLIST1 LIST1 ?VARLIST2 LIST2 …? SCRIPT
```

In the simpler form, the command executes *SCRIPT* once for each element in the list *LIST*. At the beginning of each iteration of the script, the variable *VARNAME* is assigned the value of the element. The result of the iteration is appended to a result list which is returned as the result of the command.

```
% lmap n {1 2 3 4 5 6 7 8} {expr {$n*2}}
→ 2 4 6 8 10 12 14 16
```

We can terminate the iterations early with the break command. In this case, the returned list will only include the results of the iterations up to that point.

We can further control which elements are mapped in the returned list. If the continue command is invoked within the script, the loop continues with the next element of the source list without adding the result of the current iteration to the command result.

For example, to generate a new list only for all even numbers less than 7:

```
% lmap n {1 2 3 4 5 6 7 8 9} {
    if {$n > 6} break
    if {$n % 2 } continue
    expr {$n*2}
}
→ 4 8 12
```

The more complex form of lmap accepts multiple variables and lists.

```
lmap VARLIST1 LIST1 ?VARLIST2 LIST2 …? SCRIPT
```

In this form, *VARLIST1* etc. each is a **list** of variables. In each iteration of *SCRIPT*, consecutive elements of the corresponding lists *LIST1*, *LIST2* etc. are assigned to the variables. As before, the result of the command is the list of values generated by evaluating *SCRIPT*.

```
% lmap {x y} {A B C D} {m n} {1 2 3 4} {
    string cat "$x$m,$y$n"
}
```

```
→ A1,B2 C3,D4
```

There need not be the same number of variables in each variable list or the same number of values in each value list. If some value list has fewer values that required, the empty string is assigned to the corresponding variables.

```
% lmap {x y} {A B C D} {n} {1 2 3 4} {
    list $n $x $y
}
→ {1 A B} {2 C D} {3 {} {}} {4 {} {}}
```

5.5.2. Reversing a list: lreverse

The lreverse command returns the elements of the passed list in reverse order.

```
lreverse LIST
```

Use of the command is straightforward.

```
lreverse {a b c d e} → e d c b a
```

The command is often useful in some algorithms where it is more efficient to generate an intermediate result in an order opposite to that desired, and then reverse it to get the final result.

5.6. Counting elements: llength

The llength command returns a count of the number of elements in a list.

```
llength {a b c d e} → 5
```

5.7. Sorting lists: lsort

The lsort command offers a number of ways to sort a list based on some ordering relation.

```
lsort ?options ...? LIST
```

The default ordering relation compares elements as string values as though string compare were used as the comparison function.

```
lsort {b c E g f a d} → E a b c d f g
```

Notice the default sort is case-sensitive and sorts the elements in increasing order.

5.7.1. Comparing elements

The default ordering may not always be suitable for the desired sort. For example, you may wish to sort in a case-insensitive manner. The command therefore allows you to control the ordering function used to compare elements through the options shown in Table 5.1.

Table 5.1. Lsort comparison options

Option	Description
-ascii	Compares elements using Unicode code-point collation order. This is the default.

Option	Description
-dictionary	Compares elements using "dictionary" order. This is the same as -ascii, except for two differences. First, embedded numbers within the strings are compared as integers rather than as character strings. For example, a100b will sort after a9b. However, a - character preceding a number is **not** considered part of the number so in effect all numbers are considered positive. Thus a-2b will be considered greater than a-1b.
	The other difference from -ascii is that case is ignored when comparing strings **except** that if two strings compare as equal when ignoring case, they are then compared in case-sensitive fashion. For example, abc compares as less than Bbc but greater, **not equal**, to Abc.
	The -nocase option is ignored for dictionary comparison.
-integer	Elements are treated as integers and sorted using integer comparisons.
-real	Elements are treated as floating point numbers and sorted using floating point comparisons.
-command *COMMAND*	This option allows the element ordering to be based on any arbitrary caller-defined command. This command is passed two arguments and should return a negative integer to indicate the first argument is less than the second, 0 if they are equal, and a positive number if the first is greater than the second.
-nocase	Specifies that comparisons should be done in a case-insensitive manner. The option is ignored except if the -ascii sort mode is in effect.

The following examples illustrate the difference between the various options that affect comparisons. First, numeric comparisons:

```
set integers {5 10 30}      → 5 10 30
lsort $integers             → 10 30 5
lsort -integer $integers    → 5 10 30
set reals {1.0 0.1e2 5e-2}  → 1.0 0.1e2 5e-2
lsort $reals                → 0.1e2 1.0 5e-2
lsort -real $reals          → 5e-2 1.0 0.1e2
```

Similarly, string comparisons:

```
set part_numbers {p_100_b P_100_C P_20_B} → p_100_b P_100_C P_20_B
lsort $part_numbers                        → P_100_C P_20_B p_100_b
lsort -ascii $part_numbers                 → P_100_C P_20_B p_100_b ❶
lsort -nocase $part_numbers                → p_100_b P_100_C P_20_B ❷
lsort -dictionary $part_numbers            → P_20_B p_100_b P_100_C ❸
```

❶ Same as default
❷ Case-insensitive
❸ Case-insensitive and embedded numerics compared as numbers

If none of the built-in comparisons are suitable for your purpose, you can use the -command option to specify a custom sort ordering. In the example below, we sort the part numbers based on the number of items in stock.

```
proc nstock {part} { return [string length $part] }  ❶
proc compare_stock {s1 s2} { return [expr {[nstock $s1] - [nstock $s2]}] }
lsort -command compare_stock $part_numbers
→ P_20_B p_100_b P_100_C
```

❶ Number in stock happens to match length of part number!

> In many cases, instead of sorting using the -command option, it is faster to transform the list to a format suitable for sorting using the built-in ordering functions. This is discussed in the Custom sorting[2] page on the Tcler's Wiki[3].

5.7.2. Sort ordering

The order in which list elements are sorted can be controlled with the -increasing and -decreasing options. The former, which is the default, sorts the elements from smallest to largest while the latter sorts from largest to smallest.

```
set L [list John Paul Ringo George]  → John Paul Ringo George
lsort $L                             → George John Paul Ringo
lsort -increasing $L                 → George John Paul Ringo
lsort -decreasing $L                 → Ringo Paul John George
```

The sort is *stable*, meaning that the ordering of elements that compare as equal will be preserved after the sort. For example,

```
lsort -nocase {b a B} → a b B
lsort -nocase {B a b} → a B b
lsort -real {1 1.0 0} → 0 1 1.0
lsort -real {1.0 1 0} → 0 1.0 1
```

Notice that the order in which equal elements are returned is the same as their order in the original list. For instance, b and B are equal when sorting in case-insensitive mode and their order in the sorted list is the same as the order in the original list.

5.7.3. Sorting structured lists

Lists are often used in Tcl for storing structured data similar to records in a database. For example, suppose you need to store records containing students' name and test scores. There are multiple ways this might be done in Tcl, the choice depending on the access patterns and relation with other data.

5.7.3.1. Sorting nested lists with -index

The most obvious way would be to use a nested list where each inner list contains the person's name and score. This format is often used for results returned from databases.

```
% set students {{Mike 90} {John 85} {Michelle 90} {Ann 92}}
→ {Mike 90} {John 85} {Michelle 90} {Ann 92}
```

Sorting records stored in this manner requires comparisons based on the value of elements in the inner lists. lsort provides the -index option for this purpose. The value of the index option indicates which element of the inner list is to be used in the sort comparisons. This allows sorting of our student database by either name or test score.

```
% lmap record [lsort -index 0 $students] {lindex $record 0} ❶
→ Ann John Michelle Mike
% lmap record [lsort -index 1 -integer $students] {lindex $record 0} ❷
→ John Mike Michelle Ann
```

---

[2] http://wiki.tcl.tk/4021
[3] http://wiki.tcl.tk

❶   Sort by name
❷   Sort by score

If the list is nested more than one level deep, you can even pass multiple indices, in which case they will be treated as a path through each nested sub-list, exactly as for `lindex`.

5.7.3.2. Sorting dictionaries with `-stride`

Another method of storing records uses a dictionary format which alternates the names and scores.

```
% set student_dict {Mike 90 John 85 Michelle 90 Ann 92}
→ Mike 90 John 85 Michelle 90 Ann 92
```

When structured in this manner the `-stride` option of `lsort` can be used to sort the records. The list is then treated as implicitly consisting of groups of the size specified by the option.

```
% lsort -stride 2 $student_dict
→ Ann 92 John 85 Michelle 90 Mike 90
```

By default the sort comparison element is the first element of each group. So the above fragment will sort based on names. The `-index` option can be used with `-stride` to change this.

```
% lsort -stride 2 -index 1 $student_dict
→ John 85 Mike 90 Michelle 90 Ann 92
```

This now sorts based on the **second** field of the grouping, the score.

Note that `-stride` works equally well for flat lists containing records with more than two fields.

```
% set math_english_scores {Mike 90 85 John 85 90 Michelle 90 92 Ann 92 86}
→ Mike 90 85 John 85 90 Michelle 90 92 Ann 92 86
% lmap {name math english} [lsort -stride 3 -index 1 $math_english_scores] {
    set name ❶
}
→ John Mike Michelle Ann
% lmap {name math english} [lsort -stride 3 -index 2 $math_english_scores] {
    set name ❷
}
→ Mike Ann John Michelle
```

❶   Sort by Math scores
❷   Sort by English scores

5.7.3.3. Retrieving sorted indices with `-indices`

One common method of storing data is to maintain a single master list of records which is then sorted multiple ways using different keys (e.g. for display purposes). The `-indices` option instructs the `lsort` command to return the *indices* of the sorted elements instead of the sorted values themselves. We can use this to display our sample data sorted by name or test score without having to maintain multiple lists holding the same data.

```
% lmap recnum [lsort -indices -index 0 $students] {
    lindex $students $recnum 0
}
→ Ann John Michelle Mike
```

```
% lmap recnum [lsort -indices -index 1 -integer $students] {
    lindex $students $recnum 0
}
→ John Mike Michelle Ann
```

Another use case for -indices arises when data is stored as parallel lists. For example

```
set scores(names) {Mike John Michelle Ann} → Mike John Michelle Ann
set scores(math)  {90 85 90 92}             → 90 85 90 92
```

Sorting in name order is straightforward but what if we wanted names in order of test scores as we did above? The -indices option of lsort is useful in this kind of situation where we want to retrieve elements in one list based on a sort order on a different list.

```
lsort -indices -integer $scores(math) → 1 0 2 3
```

We can thus print names in order of test score as follows

```
% lmap recnum [lsort -indices -integer $scores(math)] {
    lindex $scores(names) $recnum
}
→ John Mike Michelle Ann
```

5.7.4. Removing duplicate elements

One final option for lsort is -unique which removes duplicate elements from the returned sorted list.

```
lsort -unique {b a b d a c} → a b c d
```

A common use of the -unique option is in the implementation of sets to remove duplicate elements in operations like union.

Note that duplicate elements are those which compare as equal as per the sort options, not just those that identical. Moreover, in case of duplicates, it is the **last** duplicate element from the input list that is preserved. The following example should clarify both these points.

```
lsort -unique {b a B d A c}        → A B a b c d
lsort -nocase -unique {b a B d A c} → A B c d
```

Note how the "last" duplicate is preserved and how the use of -nocase impacts the result.

In similar fashion, when the -indices option is specified alongside -unique, it is the index of the last duplicated element is included in the returned list.

```
lsort -indices -unique {a c b e d b d} → 0 5 1 6 3
```

In the case of nested lists with the -unique option, when the inner elements used for comparison are deemed equal, only the last of the outer elements whose inner elements are equal will be included in the result.

```
lsort -unique -index 0 {{1 a} {3 b} {1 c} {2 d}} → {1 c} {2 d} {3 b}
```

The element 1 a is not included in the result as the comparison key 1 recurs later in the list.

5.8. Searching lists: lsearch

The lsearch command searches a list for elements matching specified criteria.

```
lsearch ?options …? LIST PATTERN
```

In its simplest form with no options specified, the command returns the index of the first element in *LIST* that matches *PATTERN*. By default, this matching is done by treating *PATTERN* as a glob-pattern as described in Section 4.11.

```
lsearch {foo bar jim} b* → 1
```

5.8.1. Search match operators

The type of matching used can be controlled by specifying the options shown in Table 5.2.

Table 5.2. Lsearch matching options

Option	Description
-exact	The pattern string is treated as a literal string with no special characters and compared against list elements for equality. Note that equality does not mean the two strings are identical as the meaning of equality depends on other options like -integer which control the comparison.
-glob	Use glob-style matching (the default) as described in Section 4.11.
-regexp	Use regular-expression matching as described in Section 4.12.
-nocase	Specifies that differences in character case are to be ignored when comparing the pattern with the list elements.
-not	Negates the sense of the match, thereby including only those elements do **not** match the pattern.

The options -exact, -glob and -regexp are mutually exclusive. If more than one is specified, the last one takes effect. Somewhat counterintuitively, the default matching option for lsearch is -glob, **not** -exact.

Here are a few examples to clarify the various matching types.

```
set l {a a..* a.* ab}  → a a..* a.* ab
lsearch         $l a.* → 1 ❶
lsearch -glob   $l a.* → 1
lsearch -exact  $l a.* → 2
lsearch -regexp $l a.* → 0
lsearch -exact  $l b   → -1
```

❶ Defaults to -glob

Notice the differing results with the same search pattern.

> One thing to keep in mind when using the -regexp option is that regular expression matches succeed even if just a substring of the element being compared matches the expression. If you want to match the entire element, constraints like ^ and $ must be specified.
>
> ```
> lsearch -regexp {abcde abcd bcd} b.d     → 0
> ```

```
lsearch -regexp {abcde abcd bcd} {^b.d$} → 2
```

The -nocase and -not options can be used with any of the above to modify the matching mode.

```
lsearch                 {abc BCD bcd} b*      → 2
lsearch -nocase         {abc BCD bcd} b*      → 1
lsearch -exact          {abc BCD bcd} bcd     → 2
lsearch -nocase -exact {abc BCD bcd} bcd     → 1
lsearch -regexp        {100 abc a10 xyz} {^\d+} → 0
lsearch -not -regexp {100 abc a10 xyz} {^\d+} → 1
```

5.8.2. Search operand types

When exact matching is in effect, the options shown in Table 5.3 may be used for specifying the data type to be assumed in the comparisons in a similar fashion to what we described earlier for lsort.

Table 5.3. Lsearch data type options

Option	Description
-ascii	Use string comparison with Unicode code-point collation. This is the default.
-dictionary	Use "dictionary" comparison. See the description of the option in Table 5.1. However, as detailed in the Tcl lsearch reference page, this only differs from the -ascii option if the -sorted option is also present.
-integer	Treats the elements of the list as integers, and compares the pattern using integer comparisons. Ignored if -glob or -regexp are in effect.
-real	Treats the elements of the list as floating-point numbers and compares the pattern accordingly. Ignored if -glob or -regexp are in effect.

So, for example, to search for a value in a list of integers irrespective of the integer representation, the -integer option is required.

```
lsearch -exact          {0x10 10 16} 16 → 2
lsearch -exact -integer {0x10 10 16} 16 → 0
```

Without the -integer option, the first example treats the values 0x10 and 16 as different.

Always include the -exact option with the -integer or -real options. For example,

```
lsearch -integer {0x10 10 16} 16 → 2
```

does not give the expected result because without the -exact option the command defaults to -glob pattern matching wherein -integer has no effect.

5.8.3. Searching nested lists

To locate elements based on values in nested lists, use the -index option.

```
% set students {{Martin 90} {John 85} {Mike 90} {Ann 92}}
→ {Martin 90} {John 85} {Mike 90} {Ann 92}
% lsearch -exact -index 0 $students Mike
→ 2
```

This returns the index in the outermost list that contains the matching element. You can add the `-subindices` option to get the complete index based path to the matched element. For example, if Mike prefers to be called Michael,

```
% set pos [lsearch -exact -index 0 -subindices $students Mike]
→ 2 0
% lset students $pos Michael
→ {Martin 90} {John 85} {Michael 90} {Ann 92}
```

Note the return value when `-subindices` is specified if no match is found as below.

```
% lsearch -exact -index 0 -subindices $students Albert
→ -1 0
```

5.8.4. Retrieving all matches

In cases where you want to retrieve *all* matching elements and not just the first, you can specify the `-all` option, either by itself or in combination with other options.

```
lsearch -all -index 0      $students M* → 0 2
lsearch -all -index 0 -not $students M* → 1 3
```

5.8.5. Retrieving element values

By default, the `lsearch` command retrieves the indices of the matched elements. If the `-inline`` option is specified, it will return the element values themselves.

```
lsearch -index 0              $students M* → 0
lsearch -inline -index 0      $students M* → Martin 90
lsearch -inline -index 0 -all $students M* → {Martin 90} {Michael 90}
```

When used with `-subindices`, `-inline` will only return the matching subelement values, not the whole outer element. So we can get a list of matching names with

```
% lsearch -inline -all -index 0 -subindices $students M*
→ Martin Michael
```

5.8.6. Searching sorted lists

When a list is sorted, `lsearch` can use a more efficient algorithm to locate exact matches. You can indicate that the list is sorted by passing the command the `-sorted` option. This option also implies `-exact` and cannot be used with either the `-regexp` or `-glob` option. The options `-increasing` or `-decreasing` may be specified to indicate the order in which list is sorted.

```
% lsearch -sorted {ab bc cd} bc
→ 1
% lsearch -all -sorted -integer -decreasing {20 16 0x10 10} 16
→ 1 2
```

Note that the `-sorted` option is primarily a performance feature and does not add any new capabilities to `lsearch`.

❌ Using the -sorted option with a list that is not sorted in the expected manner will give erroneous results without raising an error. An example is when the -nocase option is used with lsearch on a list that was sorted with lsort without the -nocase option.

The lsearch command also provides another very useful option, -bisect, when working in conjunction with sorted lists. When specified, lsearch returns the index at which the value is found if present (just as if -bisect is not specified). If the value is not present, instead of returning -1, it will return the position after which the value should be inserted into the list. In the case of lists in increasing order, the returned value is last index where the element is less than or equal to the searched value. For lists in descending order, it is the last index for which the element is greater than or equal to the search value. As a special case, if the search value would be placed before the first element in the list or if the list is empty, the command returns -1.

```
% lsearch -sorted -integer -bisect -decreasing {20 0x10 16 10} 16
→ 2
% lsearch -sorted -integer -bisect {10 0x10 16} 12
→ 0
```

Note that -bisect implies -sorted and that it cannot be used in conjunction with the -all and -not options.

The option is useful in inserting values into a sorted list while maintaining the sort order and not having to resort the list.

```
proc sorted_insert {l val} {
    set pos [lsearch -integer -bisect $l $val]
    if {$pos == -1 || [lindex $l $pos] != $val} {
        return [linsert $l [incr pos] $val]
    } else {
        return $l
    }
}
```

We try it with a value already present and one which is not.

```
% sorted_insert {10 20 30 40} 20
→ 10 20 30 40
% sorted_insert {10 20 30 40} 25
→ 10 20 25 30 40
```

5.8.7. Specifying a start offset

There is one final option to lsearch that is useful when conducting multiple searches through a list, each starting where the previous left off. The -start option takes a value that specifies that lsearch should begin search at that index position instead of at 0.

```
lsearch          {abx aby def abz} ab* → 0
lsearch -start 2 {abx aby def abz} ab* → 3
```

5.9. Iterating over a list: foreach

The foreach command provides a general purpose means of iterating over a list.

```
foreach VARLIST LIST ?VARLIST LIST …? SCRIPT
```

The *VARLIST* arguments are lists of one or more variable names and the *LIST* arguments are the lists to be iterated over. In the simplest case, only a single pair *VARLIST*, *LIST* is specified and *VARLIST* contains a single variable name. For each value in *LIST*, the command assigns the value to the variable and executes *SCRIPT*.

```
% foreach element {a b c} { puts [string toupper $element] }
→ A
  B
  C
```

When *VARLIST* contains more than one variable name, the variables in *VARLIST* are assigned consecutive elements from the list at the beginning of each iteration. We can use this to iterate through the math_english_scores list we saw earlier.

```
% set math_english_scores
→ Mike 90 85 John 85 90 Michelle 90 92 Ann 92 86
% foreach {name math english} $math_english_scores {
    puts "$name got a score of $math in Math and $english in English."
}
→ Mike got a score of 90 in Math and 85 in English.
  John got a score of 85 in Math and 90 in English.
  Michelle got a score of 90 in Math and 92 in English.
  Ann got a score of 92 in Math and 86 in English.
```

In the case that the number of elements in *LIST* is not a multiple of the number of variable names in *VARLIST*, the extra variables are assigned the empty string on the last iteration.

In its full form, with multiple pairs of *VARLIST* and *LIST* specified, foreach allows simultaneous iteration over multiple lists. Again, revisiting an earlier example with names and scores stored in separate lists,

```
% set scores(names)
→ Mike John Michelle Ann
% set scores(math)
→ 90 85 90 92
% foreach name $scores(names) math $scores(math) {
    puts "$name scored $math."
}
→ Mike scored 90.
  John scored 85.
  Michelle scored 90.
  Ann scored 92.
```

Note that any number of pairs may be specified and in each pair *VARLIST* may contain more than one variable name. If any list is fewer elements that required, the corresponding variables are assigned the empty strings for the remaining iterations.

As for any looping construct, the break and continue commands can be used within a foreach script to terminate the loop or skip to the next iteration.

The foreach command always returns the empty string as its result.

5.10. List utilities

List algorithms often involve some common operations which are not provided as built-in commands in Tcl. Many of these are available in the struct::list module of Tcllib[4].

---

[4] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

```
% package require struct::list
→ 1.8.3
```

Here we describe only a few of the commands provided by the package.

5.10.1. Comparing, differencing and merging

The struct::list equal command returns 1 if two lists are equal and 0 otherwise.

```
set lista {a b c d e f g}      → a b c d e f g
set listb {  b c x y f g h i}  →   b c x y f g h i
struct::list equal $lista $listb → 0
```

More interesting are the commands for differencing and merging lists. The basic command for differencing is struct::list longestCommonSubsequence. Given a pair of lists, the command returns a pair of lists containing indices into the first and second arguments that were passed to the command. The returned lists are of equal length and contain the indices of the elements in the two argument lists that are equal. An example will make this clearer.

```
% set lcs [struct::list longestCommonSubsequence $lista $listb]
→ {1 2 5 6} {0 1 4 5}
```

Element at index 1 in lista equals element at index 0 in listb and so on.

While the above command returns the elements of the two lists that are identical, it is often the case that the **differences** between the lists are of more interest. The struct::list lcsInvert transforms the output of longestCommonSubsequence into a form that details the differences and their type. The first argument to the command is the result of the longestCommonSubsequence command. The following two arguments are the lengths of the two lists.

```
% print_list [struct::list lcsInvert $lcs [llength $lista] [llength $listb]]
→ deleted {0 0} {-1 0}
  changed {3 4} {2 3}
  added {6 7} {6 7}
```

The result of the command is a list of elements each of which is a triple describing a difference. The list can be viewed as the sequence of operations required to transform the first list to the second.

- If the first element of the triple is of type deleted, the second element is the starting and ending indices of the range of elements in the first list that are not in the second. The third element is the range in the second list where those elements were expected to be present. In our example, elements in the range 0:0 of the first list are not present in the second. The index of -1 denotes that the index position in the second list would have been before the first element.

- Conversely, if the first element of the triple is added, the third element is the range of elements from the second list that would be added to the first list at the index range given by the second element.

- If the first element is changed, the second element is the range of elements in the first list that have been replaced and the third element is the range of elements in the second list that serve as their replacements.

Finally, the struct::list lcsInvertMerge returns a combination of the above with the result containing elements that are unchanged as well as those that are changed.

```
% print_list [struct::list lcsInvertMerge $lcs [llength $lista] [llength $listb]]
→ deleted {0 0} {-1 0}
  unchanged {1 2} {0 1}
  changed {3 4} {2 3}
  unchanged {5 6} {4 5}
```

```
added {6 7} {6 7}
```

The result format is the same as that of lcsInvert except the first element of a triple may also be the keyword unchanged to indicate the corresponding ranges are identical.

It should be clear that the above set of commands suffices to implement a simple file differencing program by splitting the files into lists of lines and appropriately formatting the output.

See the package reference for additional options and variations of the above commands.

5.10.2. Permutations and shuffling

An operation that is commonly encountered in games and simulations is generation of permutations of a list. An example would be shuffling a deck of cards. The struct::list module provides a command for this purpose.

The struct::list shuffle command returns the elements of the passed list in a new random order.

```
struct::list shuffle {a b c d e} → b a e c d
struct::list shuffle {a b c d e} → b d c e a
```

Since the returned list is in a randomly generated order, it is possible for a return value to be repeated.

The struct::list permutations command generates all possible permutations of a list without repetition. It returns a list each element of which is a unique permutation of the passed list.

```
% struct::list permutations {a b c}
→ {a b c} {a c b} {b a c} {b c a} {c a b} {c b a}
```

Note the returned list will not have repeated permutations even in the case where the original list contains duplicate elements.

```
% struct::list permutations {a b a}
→ {a a b} {a b a} {b a a}
```

Permutations can also be incrementally generated with the struct::list firstperm and struct::list nextperm pair of commands. The first permutation is obtained by passing the list to firstperm. Subsequent permutations are obtained by passing the previous permutation to nextperm. An empty list is returned when no permutations remain.

```
set perm [struct::list firstperm {b a}] → a b
set perm [struct::list nextperm $perm]  → b a
set perm [struct::list nextperm $perm]  → (empty)
```

A final option is to use the struct::list foreachperm command to iterate through all permutations.

```
struct::list foreachperm perm {b a} { puts $perm }
→ a b
  b a
```

The major advantage of these iterative modes for computing permutations compared to the permutations command is reduced memory requirements when processing large lists.

5.11. Chapter summary

In this chapter, we looked at the first construct for structuring data — lists — where values are accessed based on an integer index. We looked at the many commands Tcl provides for accessing and manipulating lists in various forms.

The next chapter describes another means of structuring data, dictionaries, where values are accessed through keys which may be arbitrary strings.

6

Dictionaries

> If anything is guaranteed to annoy a lexicographer, it is the habit of starting a story with a dictionary definition.
>
> — Eric McKean

We will start by defining dictionaries, just to annoy the many lexicographers who will buy this book. A dictionary in Tcl is a data structure that maps each element of a set of strings, called *keys*, to a value. Other languages may refer to similar structures as associative arrays, maps or hash tables.

The keys in a dictionary are always interpreted as strings so, for example, 1 and 0x1 are different keys. The interpretation of values is up to the application which may treat them as strings, integers, lists or even nested dictionaries.

Like lists, dictionaries play many roles in Tcl programming, for example

- as lookup tables to map keys to values
- as C-style records where the key is a field name
- nested tree like structures, mirroring a file system for example

Correspondingly, Tcl provides commands for a wide variety of operations on dictionaries.

6.1. Constructing dictionaries

6.1.1. Dictionary literals

Dictionaries in string form are exactly like lists with an even number of elements that alternate between the key and the associated value.

```
% set colors {red #ff0000 green #00ff00 blue #0000ff}
→ red #ff0000 green #00ff00 blue #0000ff
% dict get $colors red
→ #ff0000
```

Note again, that as explained for list literals, this assigns a *string* to the variable colors and it is only when acted on by the dict command that it gets interpreted as a dictionary. Thus in

```
% set colors {red #ff0000 green #00ff00 blue}
→ red #ff0000 green #00ff00 blue
% dict get $colors red
∅ missing value to go with key
```

the first statement succeeds as a simple string assignment. The dict command fails because the assigned string could not be interpreted as a dictionary as it has an odd number of elements.

> As for lists, you do not have to worry about the performance impact as the conversion happens only once as long as the value is operated on with dict commands.

6.1.2. The dict create constructor

Like the list constructor for lists, more complex dictionaries are best constructed with the dict create command.

```
dict create ?KEY VALUE KEY VALUE ...?
```

It returns a dictionary containing each of the specified key/value mappings.

```
% set mydict [dict create Key[incr i] Value$i \
              Key[incr i] Value$i \
              Key[incr i] Value$i]
→ Key1 Value1 Key2 Value2 Key3 Value3
```

6.1.3. Creating dictionaries from lists

Any list with an even number of elements can be treated as a dictionary with alternating list elements being treated as keys and values. Thus the following two statements are equivalent.

```
% set ldict [list a 1 b 2 c 3]
→ a 1 b 2 c 3
% set mydict [dict create a 1 b 2 c 3]
→ a 1 b 2 c 3
```

When ldict is accessed using dictionary commands, it will be transparently converted to an internal dictionary form.

```
dict get $ldict b → 2
```

The converse is also true as dictionaries can be manipulated using list commands.

```
% set mydict [lsort -integer -index 1 -decreasing -stride 2 $mydict]
→ c 3 b 2 a 1
```

> This last example brings us to a useful property of dictionaries that often comes in handy. Dictionaries are order preserving so that a sequence like the above can be treated both as an ordered list as well as a dictionary. For example, you can use the list form to order data for display while still being to look it up and modify it via indexed dictionary access *without disturbing the order of items*.

6.1.4. Combining dictionaries: dict merge

The dict merge command creates a new dictionary by combining the content of multiple existing dictionaries.

```
dict merge ?DICTIONARY DICTIONARY ...?
```

The returned dictionary will include every key defined in any of the passed dictionary arguments. The corresponding value in the dictionary will be the value associated with the key in the last dictionary argument that contains that key.

Consider for instance a word processor or Web browser where the appearance of text depends on settings specified at the page, paragraph or individual text span levels. These options can be stored in dictionaries for each level and combined for displaying text using `dict merge`.

```
set page_settings {font-family Helvetica background white foreground black}
set para_settings {font-family Arial}
set link_settings {foreground blue font-style underlined}
set settings [dict merge $page_settings $para_settings $link_settings]
→ font-family Arial background white foreground blue font-style underlined
```

Note the order of merge so the more specific settings in `link_settings` take precedence.

6.1.5. Nested dictionaries

A nested dictionary is nothing but a dictionary in which a value is another dictionary. For example, a dictionary containing student data may look like

```
set students {
    A001 {
        Name Jean
        Grades {Physics A Maths A Spanish B}
        Clubs {Chess Photography}
        Age 17
    }
    A002 {
        Name Pedro
        Grades {Maths A Spanish A History B}
        Clubs {Music}
        Age 16
    }
    A003 {
        Name Laxmi
        Age 17
    }
}
```

At the top level, keys are student ids and the corresponding values are nested dictionaries containing student data. The nested dictionary has keys like Name and Grades where Grades is further a nested dictionary with keys corresponding to subjects. Many `dict` subcommands support *key paths*, such as A001 Grades Physics, that navigate through dictionaries to access a nested element.

The structure and interpretation of dictionaries is entirely up to an application. Different keys within a dictionary may have values with different structure, some scalars, some lists, some nested dictionaries. Nested dictionaries within a parent dictionary need not all have the same structure either as in the example above where student A003 has only two keys, not having taken any courses or joined any clubs as yet.

6.2. Reading values from a dictionary

6.2.1. Retrieving the value for a key: `dict get`

The `dict get` command returns the value corresponding to a key in the dictionary.

```
dict get DICTIONARY ?KEY ...?
```

If no keys are specified, the command returns a list containing the key and value pairs.

```
% set colors {green #00ff00 red #ff0000 blue #0000ff magenta #ff00ff}
```

```
→ green #00ff00 red #ff0000 blue #0000ff magenta #ff00ff
% dict get $colors
→ green #00ff00 red #ff0000 blue #0000ff magenta #ff00ff
```

Specifying a key will return the corresponding value.

```
% dict get $colors red
→ #ff0000
```

The command can also retrieve values from nested dictionaries by specifying multiple keys that define a path through the dictionary.

```
% dict get $students A001 Grades Maths
→ A
```

Note that an attempt to read a key that does not exist in the dictionary will raise a Tcl exception. You can check for the existence of a key before attempting to read it by calling `dict exists`.

6.2.2. Enumerating dictionary keys: `dict keys`

The `dict keys` command returns the keys in a dictionary.

```
dict keys DICTIONARY ?PATTERN?
```

If *PATTERN* is not specified, the command returns a list of all keys in the dictionary. If *PATTERN* is specified, it is treated as a glob pattern (see Section 4.11) and the command returns only the keys matching that pattern.

```
dict keys $colors     → green red blue magenta
dict keys $colors *r* → green red
```

6.2.3. Enumerating dictionary values: `dict values`

The `dict values` command returns the values in a dictionary.

```
dict values DICTIONARY ?PATTERN?
```

If *PATTERN* is not specified, the command returns a list of all values in the dictionary. If *PATTERN* is specified, the command returns only the values matching that glob pattern.

```
dict values $colors     → {#00ff00} #ff0000 #0000ff #ff00ff
dict values $colors #ff* → {#ff0000} #ff00ff ❶
```

❶　Reddish colors

The values are matched as strings no matter whether they are integers, nested strings etc.

> You may be wondering why the first element in the returned list is enclosed in braces. This is because the string representation generated by Tcl for a list whose first element begins with a # always encloses that element in braces. As to the reason why, the short answer is that when a command (or command prefix) is constructed in list form, enclosing the first word of the command in braces prevents it from being mistakenly parsed as a comment. For a fuller explanation see TIP 407 [1]. Note this does **not** invalidate the list contents. So for example retrieving the first element above

---

[1] http://www.tcl.tk/cgi-bin/tct/tip/407

```
lindex [dict values $colors #ff*] 0 → #ff0000
```

correctly retrieves the value #ff0000.

6.3. Modifying dictionaries

6.3.1. Setting values with `dict set`

The `dict set` command operates on a variable presumed to contain a dictionary.

```
dict set DICTVAR KEY ?KEY ...? VALUE
```

If the specified key exists in the dictionary stored in *DICTVAR*, its value is replaced with the new value. If the key does not exist, it is added to the dictionary along with the associated value. The resulting dictionary is returned by the command as well as stored back in the *DICTVAR* variable.

```
set mydict [dict create a 1 b 2 c 3] → a 1 b 2 c 3
dict set mydict a 10                  → a 10 b 2 c 3 ❶
dict set mydict x 100                 → a 10 b 2 c 3 x 100 ❷
set mydict                            → a 10 b 2 c 3 x 100 ❸
```

❶ Modify an existing key
❷ Create a new key
❸ The variable value is also changed

The command will operate on nested dictionaries as well. For example, we can add a key for a new student

```
dict set students A004 {
    Name Mark
    Grades {
        Physics A
    }
}
```

or update an existing student. We could do the update one leaf element at a time,

```
dict set students A003 Grades Physics B
dict set students A003 Grades English A
```

or with the entire nested dictionary

```
dict set students A003 Grades {Physics B English A}
```

> ⚠ The above two sequences are equivalent only because the key Grades did not previously exist for student id A003. If the key did in fact already exist, the first form that updated a leaf at a time would **add** the new keys under the existing Grades key. The second form on the other hand would **replace** the existing contents of Grades.

Note from the example how missing keys in the key path for the nested dictionaries are created as needed.

6.3.2. Removing dictionary elements: `dict unset`

The `dict unset` command is the complement of `dict set` and used to remove individual elements from a dictionary stored in a variable.

```
dict unset DICTVAR KEY ?KEY ...?
```

Thus if Laxmi leaves the school, we can forget about her existence.

```
% dict unset students A003
```

As for `dict set`, a key path can be specified to remove any nested element. If Jean has not actually taken Spanish and her grade was erroneously given we can correct the error.

```
% dict unset students A001 Grades Spanish
```

It is not an error if the last key on the key path (`Spanish` in our example) is missing. However keys other than the last must exist else the command will raise an exception.

6.3.3. Appending values in-place: `dict append`

The `dict append` command appends any specified strings to an element of a dictionary contained in a variable.

```
dict append DICTVAR KEY ?STRING ...?
```

The command concatenates all the supplied *STRING* arguments and appends the result to the value in the dictionary corresponding to the specified key creating it if necessary. The resulting dictionary is returned by the command and also stored back in *DICTVAR*.

```
% set mydict [dict create keyA A]
→ keyA A
% dict append mydict keyA BC ❶
→ keyA ABC
% dict append mydict keyW W XYZ ❷
→ keyA ABC keyW WXYZ
% set mydict
→ keyA ABC keyW WXYZ
```

❶   Append to an existing key
❷   Create a new key and append multiple strings

Note that this command does *not* directly support nested dictionaries as only a single level of keys can be specified.

One way around this limitation is to structure the two-level dictionary as a one-level dictionary stored in an array. We will see this later.

Like the next two dictionary commands we will see, `dict lappend` and `dict incr`, the primary benefit of `dict append` is efficiency as the implementation will append the strings "in place" if possible.

6.3.4. Appending list elements to values: `dict lappend`

The `dict lappend` command is similar to `dict append` except that instead of appending a string, it treats the value in the dictionary as a list and adds additional elements to it.

```
dict lappend DICTVAR KEY ?VALUE ...?
```

The command retrieves the value, which should be interpretable as a list, currently associated with the key *KEY* in the dictionary stored in *DICTVAR*, appends the given elements to it in the same manner as `lappend` and assigns the result back to the key and the resulting dictionary back to *DICTVAR*.

Like `dict append`, `dict lappend` does not support nested dictionaries. Thus to update our students dictionary to reflect that Pedro has joined the Athletics club we have to extract the nested dictionary, modify it and write it back.

```
set pedro [dict get $students A002]
dict lappend pedro Clubs Athletics
dict set students A002 $pedro
```

6.3.5. Incrementing dictionary values: `dict incr`

One other command that updates elements in place is `dict incr`. Whereas `dict append` and `dict lappend` dealt with strings and lists respectively, `dict incr` works with dictionary elements that are integers. Like those commands, `dict incr` operates on a variable and does not support nested dictionaries.

```
dict incr DICTVAR KEY ?INCREMENT?
```

The value of the element in the dictionary contained in *DICTVAR* with key *KEY* is incremented by *INCREMENT* (must be an integer) if specified and by 1 otherwise. The resulting dictionary is stored back in *DICTVAR*. If the key did not exist in the dictionary, it is created with an initial value of 0 before being incremented by the specified amount.

A simple example of maintaining word counts using a dictionary.

```
% foreach word {Do what you can, ignore what you can't.} {
    dict incr word_counts $word
}
% puts $word_counts
→ Do 1 what 2 you 2 can, 1 ignore 1 can't. 1
```

6.3.6. Removing multiple keys: `dict remove`

The `dict remove` command returns a new dictionary formed by removing the specified keys from the provided dictionary value.

```
dict remove DICTIONARY ?KEY ...?
```

It differs from `dict unset` in two respects:

- It operates on a dictionary **value** whereas `dict unset` operates on a **variable**.
- Multiple key arguments refer to the top level keys to be removed, **not** a single key path to a nested element.

```
% set mydict {a 1 b 2 c 3 d 4}
→ a 1 b 2 c 3 d 4
% dict remove $mydict a c
→ b 2 d 4
```

Keys that do not exist in the dictionary are ignored and do not raise an error.

6.3.7. Replacing multiple values: `dict replace`

The `dict replace` command returns a new dictionary formed by replacing the values for specified keys in the dictionary passed in.

```
dict replace DICTIONARY ?KEY VALUE ...?
```

The command creates new entries for keys that do not exist.

```
% set mydict {a 1 b 2 c 3 d 4}
→ a 1 b 2 c 3 d 4
% dict replace $mydict a 10 c 30 e 50
→ a 10 b 2 c 30 d 4 e 50
```

6.3.8. Shadowing dictionaries with local variables: `dict update`

Earlier we saw commands like `dict lappend` that directly update a dictionary entry. In the general case though, updating an entry requires retrieving it with `dict get`, modifying it, and then storing it back which is what we had to do when Pedro joined the Athletics club.

The `dict update` command encapsulates this sequence of retrieval, arbitrary modification via a script and writing back into the dictionary.

```
dict update DICTVAR KEY VARNAME ?KEY VARNAME ...? SCRIPT
```

The command looks up each specified key in the dictionary contained in *DICTVAR* and assigns its value to corresponding variable *VARNAME*. If a specified key is not present, the corresponding variable remains undefined unless it was already defined in the scope. The command then executes the specified script on the completion of which the values in each of *VARNAME* are assigned back to the corresponding key in the dictionary. The modified dictionary is stored back in *DICTVAR*. The return value from the command is the return value of the last statement executed in *SCRIPT*. The example below illustrates the various possibilities.

```
% set xvar X
→ X
% set mydict {a 1 b 2 c 3}
→ a 1 b 2 c 3
% dict update mydict a avar c cvar d dvar x xvar {
    incr avar 10 ❶
    unset cvar ❷
    set dvar [dict get $mydict a] ❸
}
→ 1
% puts $mydict ❹
→ a 11 b 2 d 1
% info exists xvar ❺
→ 0
```

❶ Will change value associated with key a
❷ Will result in key c being removed
❸ Will add a new key d with *old* value of key a as `mydict` is still unchanged at this point
❹ Variable `mydict` is updated *after* completion of script
❺ Since key x did not exist in dictionary, previously existing variable `xvar` is unset

The value of `mydict` is updated even when the script completes with a status other than ok (for example, a `return` or an error exception).

The command can also be used with nested dictionaries. Consider an example similar to one mentioned earlier — here we want to update the dictionary to reflect the fact that Jean joined the archery club on her birthday. Either of the following would do the job, using `dict update` only at the first level as below:

```
set student_id A001
dict update students $student_id student {
    dict lappend student Clubs Archery
    dict incr student Age
}
```

Or using `dict update` at both levels in nested fashion:

```
set student_id A001
dict update students $student_id student {
    dict update student Age age Clubs clubs {
        lappend clubs Archery
        incr age
    }
}
puts [dict get $students A001]
→ Name Jean Grades {Physics A Maths A} Clubs {Chess Photography Archery} Age 18
```

The `dict update` command is really most useful when the update is more complex than the simplistic examples shown here, particularly when the update involves more than one key from the dictionary.

6.3.9. Shadowing nested dictionaries: `dict with`

The `dict with` command is similar to the `dict update` command in that it executes a script with variables that shadow dictionary entries and then writes them back into the dictionary.

```
dict with DICTVAR ?KEY …? SCRIPT
```

If no `KEY` arguments are specified, the command executes `SCRIPT` after assigning the values in the dictionary in variable `DICTVAR` to variables of the same name as the corresponding dictionary keys. When the script completes, whether normally or through a `return` or error condition, any changes made to those variable are written back to the dictionary contained in *DICTVAR*. The result of the script execution is returned as the result of the command.

```
set mydict {a 1 b 2 c 3}
dict with mydict {
    incr a $a
    incr b $b
}
puts $mydict
→ a 2 b 4 c 3
```

Note that unlike `dict update`, dictionary keys are mapped to variables of the same name with no provision to map to variables of a different name. This requires some care to prevent conflict between dictionary keys and existing variables having the same name as we will see in a bit.

Since retrieving key values from dictionaries with `dict get` can be tedious, a trick you will often see employed is to use `dict with` with an empty script to bring the keys into local scope and access them like any other variables. So for instance, instead of

```
% puts "RGB are [dict get $colors red], \
              [dict get $colors blue], \
              [dict get $colors green]"
→ RGB are #ff0000,  #0000ff,  #00ff00
```

we can do this

```
% dict with colors {}
% puts "RGB values are $red, $blue, $green"
→ RGB values are #ff0000, #0000ff, #00ff00
```

which can be convenient in longer scripts.

Another difference between `dict update` and `dict with` is that the latter makes it easy to deal with nested dictionaries. If one or more *KEY* arguments are specified, instead of shadowing the top level keys of the dictionary with variables, the command shadows the nested dictionary identified by the specified key path *KEY ...* .

Here is an example for nested dictionaries that will updates Jean's grades.

```
dict with students $student_id Grades {
    set Physics A-
    set Maths B
}
puts [dict get $students $student_id]
→ Name Jean Grades {Physics A- Maths B} Clubs {Chess Photography Archery} Age 18
```

Note that unlike `dict update`, `dict with` can only update existing keys, not create new ones.

It is important to keep in mind that the variables affected by `dict with` are dependent on the contents of the dictionary. Unexpected behaviour can result if a dictionary key happens to be the same as the name of an unrelated variable which will get overwritten. One way of minimizing this possibility is to adopt a convention where dictionary keys are syntactically different from variable names; for example, making them all upper case or starting with an upper case letter.

Under some circumstances this overwriting of variables can even be a security risk. For instance, some Tcl web servers will return received URL parameters as a dictionary mapping the client supplied parameter to a value. Passing this dictionary to `dict with` will allow the client to overwrite any variable, even global ones, with their own chosen values. In general, avoid using `dict with` and `dict update` with dictionaries constructed from arbitrary input values.

6.4. Iterating over dictionaries: `dict for`

The `dict for` command iterates over every entry in a dictionary.

```
dict for DICTIONARY {KEYVAR VALUEVAR} SCRIPT
```

The command executes *SCRIPT* for every entry in the dictionary **in the order that the keys were inserted into it**. In each iteration, the variables named *KEYVAR* and *VALUEVAR* are assigned the key and the value of next entry in the dictionary. Like other Tcl commands that loop, the iteration can be terminated by a `break` command before all

entries are processed. Likewise, a continue command will skip the rest of the script but continue on with the next entry.

```
dict for {color rgb} $colors {
    puts "The RGB value for $color is $rgb."
}
→ The RGB value for green is #00ff00.
  The RGB value for red is #ff0000.
...Additional lines omitted...
```

6.5. Transforming dictionaries

A dictionary transform produces a new dictionary based on the contents of an existing one.

6.5.1. Filtering dictionaries: dict filter

The dict filter command returns a new dictionary containing entries from an existing dictionary that meet specified matching criteria. It takes one of the following forms

```
dict filter DICTIONARY key ?PATTERN ...?
dict filter DICTIONARY value ?PATTERN ...?
dict filter DICTIONARY script {KEYVAR VALUEVAR} SCRIPT
```

In the first form, the command will return a new dictionary that contains any entry whose key matches at least one of the PATTERN arguments. Note that this means that if no patterns are specified an empty dictionary is returned. The matching is done as described for the string match command in Section 4.11.

For example, we can filter our display settings from our earlier dict merge example to only get the settings related to fonts.

```
% dict filter $settings key font*
→ font-family Arial font-style underlined
```

The second form of the command is similar except that instead of matching against the key for an entry, it matches against the value.

For example, filter *out* all colors that have a green component.

```
% dict filter $colors value #??00??
→ red #ff0000 blue #0000ff magenta #ff00ff
```

The final form of dict filter is the most flexible. It executes SCRIPT for every entry (aside from any break or error conditions) in the dictionary. On each iteration, the key and value are assigned to the variables named KEYVAR and VALUEVAR respectively. The entry is included in the dictionary only if the iteration returns a Boolean true value. The iteration is terminated by a break while a continue is treated the same as a Boolean false return from the script.

The following returns a dictionary containing any color that has either a green or a blue component (ie. at least one of the corresponding bits are non-0).

```
dict filter $colors script {color rgb} {
    expr {![string match #??0000 $rgb]}
}
→ green #00ff00 blue #0000ff magenta #ff00ff
```

6.5.2. Mapping values: `dict map`

The `dict map` command returns a new dictionary formed by mapping each value in the dictionary to a new value returned by a script.

```
dict map {KEYVAR VALUEVAR} DICTIONARY SCRIPT
```

The command executes *SCRIPT* for each entry in the dictionary assigning the key and value to the variables named *KEYVAR* and *VALUEVAR* respectively. On the normal completion of each iteration, a new entry is added to the result dictionary with the key being the current value of *KEYVAR*. The iteration is terminated by a break while a continue is continues with the next iteration without changing the result dictionary for the current iteration.

As an example, consider converting our colors to 8-bit grey scale using a simplistic algorithm that averages the RGB values.

```
set grey_scale [dict map {color rgb} $colors {
    regexp {^#(..)(..)(..)$} $rgb -> r g b ❶
    format "#%x" [expr {("0x$r" + "0x$g" + "0x$b")/3}]
}]
→ green #55 red #55 blue #55 magenta #aa
```

❶ Split into red, green, blue components

If the variable containing the key is modified, the new dictionary will contain a new key corresponding to the modified content of the variable.

```
set grey_scale [dict map {color rgb} $colors {
    regexp {^#(..)(..)(..)$} $rgb -> r g b
    set color "greyscale_$color"
    format "#%x" [expr {("0x$r" + "0x$g" + "0x$b")/3}]
}]
→ greyscale_green #55 greyscale_red #55 greyscale_blue #55 greyscale_magenta #aa
```

6.6. Introspecting dictionaries

6.6.1. Checking for a key: `dict exists`

The `dict exists` command returns a Boolean true value if the specified key exists in the dictionary and a Boolean false otherwise.

```
dict exists DICTIONARY KEY ?KEY ...?
```

Commands like `dict get` will raise an error on an attempt to retrieve the value for a non-existent key. This command should therefore be used to check for existence.

```
dict exists $colors red    → 1
dict exists $colors yellow → 0
```

The command supports nested dictionaries.

```
dict exists $students A001 Grades Physics → 1
dict exists $students A001 Grades Biology → 0
```

6.6.2. Count of entries: `dict size`

The `dict size` command returns the number of entries in a dictionary.

```
% dict size $colors
→ 4
```

6.6.3. Dictionary statistics: `dict info`

The `dict info` command returns a human readable string that provides some information about the internal structure of the dictionary.

```
% dict info $colors
→ 4 entries in table, 4 buckets
  number of buckets with 0 entries: 1
  number of buckets with 1 entries: 2
  number of buckets with 2 entries: 1
  number of buckets with 3 entries: 0
...Additional lines omitted...
```

The command is primarily intended for debugging and performance related analysis.

6.7. Dictionaries and arrays

Because they both provide facilities for mapping keys to values, there is often confusion regarding the differences between arrays and dictionaries and the circumstances in which each is to be preferred. The table below highlights these differences.

Table 6.1. Differences between tables and arrays

Arrays	Dictionaries
Arrays are collections of *variables*. For example, `myarray($key)` is a variable and can be accessed as `$myarray($key)`.	Dictionaries are collections of *values* where individual values cannot be accessed as variables. They must be accessed as `dict get $mydict $key`.
Because they are variables, it is possible to set variable traces on individual array elements.	Variable traces can only be set on the entire dictionary, not individual entries in the dictionary.
Arrays are not values and cannot be directly passed to procedures without using additional mechanisms such as upvar.	Dictionaries are values and can be passed into procedures like any other value.
Arrays cannot be nested as they are not values.	Dictionaries can be nested because they can hold any values including other dictionaries.
Arrays are *unordered* collections and hence the order in which elements are accessed in operations like iteration is not guaranteed.	In iteration and similar operations, dictionary entries are always processed in the order in which the keys for the entries were created.

Dictionaries contain values and therefore cannot contain arrays. On the other hand, dictionaries are values and therefore can be contained in arrays. This fact is often useful in cases where there are two levels of keys. Structuring the data such that the first level of keys are stored in arrays and the second level in dictionaries can make certain accesses more convenient.

For example consider our `students` dictionary data store and convert it to an array form where the array elements indexed by student id will hold a dictionary containing the "record" for that student. This is easy enough to do.

```
% array set student_array $students
% puts $student_array(A001)
```

```
→ Name Jean Grades {Physics A- Maths B} Clubs {Chess Photography Archery} Age 18
```

Now to modify a student's record, we can directly use dictionary in-place commands like dict lappend. In our earlier examples, we could not use these directly because they do not support nested dictionaries causing us to instead do a read/modify/write cycle instead.

```
% dict lappend student_array(A001) Clubs Gymnastics
→ Name Jean Grades {Physics A- Maths B} Clubs {Chess Photography Archery Gymnastics} Age 18
% dict get $student_array(A001) Clubs
→ Chess Photography Archery Gymnastics
```

6.8. Chapter summary

Along with lists, dictionaries form the primary means of structuring data in Tcl. In this chapter, we described the various commands for their manipulation and some simple uses to which they can be applied. We will now move on to the topic of numerical computation in Tcl.

7

Numerics

All which is beautiful and noble is the result of reason and calculation.

— Charles Baudelaire

Although Tcl, like most dynamic languages, is not intended for heavy numeric computation, it provides a full set of operators and functions that should more than suffice for general purpose computing. Additionally, several extension libraries are available which further enhance Tcl capabilities in both performance and functionality.

7.1. Types and representations

Tcl supports operations on boolean, integer and floating point values. In this section we go over these different types in terms of their string representation, acceptable values and conversions.

Internal representation of numbers

At the scripting level, there is no reason to be concerned with the internal representations. However, folks might wonder whether Tcl will convert numerics and strings back and forth on every arithmetic operation and how that might affect performance. The answer is that internally Tcl will keep numbers in the usual native form for the machine. It is only when they are used as strings, for example for printing, are the string representations generated. See Section 10.10 for more on this topic.

7.1.1. The boolean type

Tcl accepts the following values as booleans:

- a boolean false is a numeric value of 0 or the strings `false`, `no`, `off`
- a boolean true is any non-0 numeric value and the strings `true`, `yes`, `on`

The string values are case-insensitive and **unique** abbreviations are also acceptable.

```
% if {1} {puts true!}
→ true!
% if {tRue} {puts true!} ❶
→ true!
% if {fal} {puts true!} else {puts false!} ❷
→ false!
```

❶ Case insensitive
❷ Abbreviated from `false`.

In computed boolean expressions, Tcl returns boolean true as 1 and false as 0.

7.1.2. The integer types

Tcl supports integers of arbitrary size so just in case you wanted to calculate the number of stars in the universe, you could.

```
set ngalaxies 10000000000              → 10000000000
set stars_per 100000000000             → 100000000000
set nstars [expr {$ngalaxies * $stars_per}] → 1000000000000000000000000
```

The advantage over floating point representation is that you don't lose precision and there is no theoretical limit to number of digits (although memory limitations of course apply).

Tcl accepts several representations for integer values:

- a string of decimal digits such as 100
- a string beginning with 0x or 0X is interpreted as an integer in hexadecimal form, e.g. 0x64.
- a string beginning with 0b or 0B is interpreted as an integer in base-2 form[1], e.g. 0b1100100.
- a string beginning with 0o or 0O is interpreted as an integer in octal form, e.g. 0o144. For reasons of backward compatibility, Tcl also treats **any** number beginning with a 0 character as an octal representation. For example, 0144 is treated as an octal representation of 100. You should not use this form in new code as it is very likely to be done away with in the next major Tcl release.

All these forms may be preceded by a - character to represent negative integers.

> ⚠
>
> The treatment of numeric strings beginning with a 0 character as octal numbers can lead to unwanted results. For example, consider a procedure to calculate the minute of the day given a string of the form HH:MM.
>
> ```
> proc minute_of_day {time_of_day} {
>     lassign [split $time_of_day :] h m
>     return [expr {$h * 60 + $m}]
> }
> minute_of_day 12:00
> → 720
> ```
>
> This seems fine until you try this:
>
> ```
> % minute_of_day 09:00
> 0 can't use invalid octal number as operand of "*"
> ```
>
> This fails because 09 is parsed as an octal representation where 9 is not a valid digit.
>
> The best way to get around this pitfall is to use the %d format specifier of scan to parse a string as a decimal integer.
>
> ```
> scan 012 %d → 12 ❶
> scan 09 %d  → 9 ❷
> ```
>
> ❶ If treated as octal, this would have been 10
> ❷ If treated as octal, this would have been an error
>
> As an aside, note that parsing time in this manner was just an illustration and you should prefer the clock scan command for this purpose.

Although Tcl supports arithmetic operations on integers of any size, it does allow for distinction between

---

[1] We use the term base 2, and not binary, to avoid confusion with binary strings

- *integers* which are values that fit in the native word size of the machine as given by the wordSize element of the tcl_platform global array. On most systems these are values that fit in 32 bits.
- *wide integers*, or simply *wides* which are values that fit in 64 bits.
- *bignums* which are integers of unlimited size.

This can be important because although Tcl will itself support arithmetic operations on integers of any size, this may not hold true for all functions, extensions and other programs that interoperate with the application. The string is integer, string is wide and string is entier commands described in Chapter 4 can be used to distinguish between these integer types. We will also have more to say when we discuss numeric conversions in Section 7.1.5.2.

7.1.3. The floating point type

Floating point, or real number, values are represented internally as the C language double type. The string representation takes the form of

- an optional - or + sign
- followed by a string of decimal digits containing at most one decimal point
- optionally followed by the exponent which consists of an e or E character followed by an optional sign, and then a string of decimal digits.

The values 1, 1.0, -1.0e100, 10E-100 are all valid floating point values.

As a special case, the positive and negative infinities are represented by Inf and -Inf respectively.

```
expr 1.0 / 0.0 → Inf
```

Infinities behave as you would expect in most calculations.

```
expr Inf * 2 → Inf
```

The other special case related to floating point representation is the "not a number" value represented by NaN.

```
tcl::mathfunc::sqrt -1 → -NaN
```

We can confirm that both these are treated as floating point values.

```
string is double Inf → 1
string is double NaN → 1
```

7.1.4. Validation of types

The string is command can be used to validate that a passed value is an acceptable string representation for a type.

```
string is integer 123        → 1
string is integer abc        → 0
string is integer ""         → 1 ❶
string is integer -strict "" → 0
```

❶   Note an empty string is acceptable as any type unless -strict is specified.

Other numeric types such as booleans, doubles, etc. can be validated in similar fashion. See Section 4.9 for details.

7.1.5. Number conversions

7.1.5.1. Converting between strings and numbers

A number may have multiple string representations. For example, the integer 10 may be represented as 10, 0xa, 0x0A and so on. The same also applies to floating point numbers.

When a string value is interpreted as a number, Tcl will accept any of these representations as a valid value as described earlier. If you want to validate that the string represents a number in a **specific** form, you can use the scan command which is described in Section 4.7.

Conversely, when converting the numeric result of a computation into a string for display purposes or other reasons, Tcl will generate its "natural" representation. For integers, this is in the form of a string of decimal digits. For floating point numbers, Tcl generates a string that contains the minimal number of digits required to distinguish the number from its nearest floating point neighbours[2].

You can use the format command (see Section 4.2.4) to generate a specific string representation of a number.

> The generation of string representations for floating point numbers can be controlled by the tcl_precision global variable. We do not describe it as it is intended only for use by legacy applications and is full of potential pitfalls. See the Tcl reference documentation for details if you are curious.

7.1.5.2. Converting between numeric types

As we stated in Section 7.1.2, Tcl distinguishes between integers of type integer, wide and bignums based on the number of bits in the internal representation. In addition there is the floating point double type discussed in the previous section. There are some situations where the number representations matter. For instance, the result of a mathematical operation may depend on the type of the operand.

```
expr 3/2    → 1
expr 3/2.0 → 1.5
```

Other instances where the distinctions become important include algorithms that depend on truncation based on integer widths and exchange of data to other programs.

Within integer expressions, Tcl will automatically use a type that is wide enough to hold all values. If an operation has a floating point operand, any integer operands will be converted to floating point, possibly losing precision.

For situations where you want to explicitly control the numeric type, Tcl provides a set of commands to "cast" a value to the desired numeric type. These commands lie within the ::tcl::mathfunc namespace.

The int, wide, entier commands return the integer portion of their argument, truncated to the appropriate width in the case of int and wide. Similarly, the bool and double commands convert their operand to a boolean and floating point value respectively.

```
tcl::mathfunc::int 0x100000000                         → 0 ❶
tcl::mathfunc::int 2.5                                 → 2 ❷
tcl::mathfunc::wide 1.4e10                             → 14000000000 ❸
tcl::mathfunc::double 140000000000000000000000 → 1.4e+23 ❹
```

❶ Truncates wides to native word size
❷ Truncates floating point to native word size
❸ Floating point to wide integers
❹ Wide integers to floating point

---

[2] If you do not understand this statement, it arises from the fact that floating point representations in computer arithmetic are inexact approximations. See http://blog.reverberate.org/2014/09/what-every-computer-programmer-should.html for an explanation.

As we will see shortly, the commands in the `tcl::mathfunc` namespace can be used as functions in Tcl expressions evaluated by `expr` so the above could also be written as

```
expr { int(2.5) } → 2
```

7.2. Mathematical operations

Mathematical operations in Tcl can be executed in one of two ways:

- Each arithmetic operation is implemented as a command in the `::tcl::mathop` namespace
- The `expr` command implements in-fix expressions as found in other languages.

7.2.1. The `tcl::mathop` commands

Commands corresponding to the common arithmetic operations are located in the `::tcl::mathop` namespace. For example, you can add three numbers as

```
tcl::mathop::+ 1 2 3 → 6
```

We have not looked at namespaces yet and will do so in Chapter 12. For now you can either invoke the commands as above or to reduce the typing involved run the following command

```
% namespace path ::tcl::mathop
```

You can then simply type

```
+ 1 2 3 → 6
```

As reflected by the name, the `tcl::mathop` namespace primarily contains commands related to mathematical operations. However, it also includes some operators that work with non-numeric operands, returning boolean values that can then be used in expressions. The discussion below groups the commands into the following categories:

- Arithmetic operators like addition, subtraction etc. that work with any numeric values
- Bitwise operators limited to integer values
- Comparison operators
- String operators
- List operators

7.2.1.1. Arithmetic operator commands

Table 7.1 lists all the commands dealing with arithmetic operations and the operand types to which they apply.

Table 7.1. Arithmetic operators

Operator	Description

! *BOOL*	Negates a boolean value.
	
	`! 2     → 0`
	`! false → 1`

+ ?*NUM* ...?	Returns the sum of the operands.
* ?*NUM* ...?	Returns the product of the operands.

Operator	Description
- NUM ?NUM...?	If a single operand is specified, returns its negative. Otherwise, returns the result of subtracting all subsequent operands from the first.

```
- 10        → -10
- 10 3 2 1 → 4
```

Operator	Description
/ NUM ?NUM ...?	If a single operand is specified, returns its reciprocal. The reciprocal operation is always done as a floating point operation even if the operand is an integer.

```
/ 2 → 0.5
/ 0 → Inf
```

If more than one operand is provided, the command returns the result of successively dividing the first operand by subsequent ones. In this case the operation is done as integer division until the first floating point operand is encountered. All further operations are executed as floating point division. Hence the following results:

```
/ 9 2.0 2 → 2.25
/ 9 2 2.0 → 2.0
```

Operator	Description
% INT INT	Returns the integral remainder when dividing the first operand by the second. The sign of the result will be the same as the sign of the second operand.

```
tcl::mathop::% 13 -3 → -2
```

This ensures that the following two computations return the same result.

```
* [/ 13 -3] -3 → 15
- 13 [% 13 -3] → 15
```

Operator	Description
** ?NUM ...?	Raises the first operand to the power specified by the second operand. The result is then further raised to the power specified by the third operand and so on. If any of NUM is not an integer, the result will be a floating point number.

```
** 2 3 4 → 2417851639229258349412352
** 4 0.5 → 2.0
```

7.2.1.2. Comparison operator commands

The second set of operators compares one or more numbers. **In all cases, the operands are compared numerically if possible. If an operand is not a valid numeric representation, the operands are compared as strings.** These operators are shown in Table 7.2.

Table 7.2. Comparison operators

Operator	Description
== ?ARG ...?	Returns 1 if every argument equals its neighbours and 0 otherwise.

```
set ten 10.0    → 10.0
== 0xa $ten 0o12 → 1
```

Operator	Description
!= *ARG ARG*	Returns 1 if the two arguments are not equal and 0 otherwise.

< ?*ARG* ...?	Returns 1 if every argument is strictly less than the next and 0 otherwise. The following checks if a list is in strictly increasing order.
	
	```% set fruits [list apple banana orange]```
	```→ apple banana orange```
	```% < {*}$fruits```
	```→ 1```

<= ?*ARG* ...?	Returns 1 if every argument is less than or equal to the next and 0 otherwise.
	
	```<= 10 20 30 40 → 1```
	```<= 10 20 40 30 → 0```
	```set val 10 → 10```
	```<= 0 $val 20 → 1``` ❶
	
	❶ Check if a value is within a range

> ?*ARG* ...?	Returns 1 if every argument is strictly greater than the next and 0 otherwise.
>= ?*ARG* ...?	Returns 1 if every argument is greater than or equal to the next and 0 otherwise.

We reiterate that numeric comparisons are used if both operands can be interpreted as numbers. If not, they are compared as strings. This means you have to be careful where you really want string comparisons and there is a chance operands might appear to be numbers (e.g. ZIP codes). In this case use the eq and ne commands described later or the string compare command instead.

Moreover, when more than two operands are specified, each comparison is done in isolation so that one comparison may be numeric and another string. For example, in

```
tcl::mathop::< " a" 12 2 → 0
```

the first comparison is done as a string and evaluates to true. The second comparison would also evaluate to true were if done as a string comparison. But because both 12 and 2 can be interpreted as numbers, it is treated as a numeric comparison and thus returns false. Again, this illustrates the need to be careful about the operand types in such cases.

7.2.1.3. Bit-wise operator commands

The next set of operators deal with bit-wise operations and are shown in Table 7.3. The operands to these must be integers (of any width).

Table 7.3. Bit operators

Operator	Description

~ *INT*	Returns the bit-wise negation of *INT*.
	
	```% format 0x%x [~ 0x0f0f0f0f]```
	```→ 0xf0f0f0f0```

& ?*INT* ...?	Returns the result of a bit-wise AND operation between the specified operands. ```format 0x%lx [& 0xf0f0 0xaaaa 0xcccc] → 0x8080```
| ?*INT* ...?	Returns the result of a bit-wise OR operation between the specified operands.

Operator	Description
^ ?*INT* ...?	Returns the result of a bit-wise XOR operation between the specified operands.
<< *INT SHIFT*	Returns the result of shifting the *INT* operand left by *SHIFT* number of bits.
>> *INT SHIFT*	Returns the result of shifting the *INT* operand right by *SHIFT* number of bits.

7.2.1.4. String operator commands

The eq and ne commands test values for equality as **strings**. These are equivalent to the string equal command.

```
eq ?ARG ...?
ne ?ARG ...?
```

They behave just like the == and != commands described in the previous section with eq returning 1 if all operands are equal and 0 otherwise and ne doing the opposite. The difference relative to == and != is that they always compare the operands as strings even if they can be interpreted as numbers. This is illustrated by the following.

```
== 0xa 10  → 1
eq 0xa 10  → 0
!= NaN NaN → 1
ne NaN NaN → 0
```

The NaN example may surprise you. The NaN value is a valid floating point value and thus the != operator treats it as such. The "specialness" of this *not a number* value is that it will compare as being unequal to all values, **even itself**! Of course, as a string it compares equal to itself.

7.2.1.5. List operator commands

The last set of commands in the tcl::mathop namespace pertain to lists — the in and ni commands.

```
in ELEM LIST
ni ELEM LIST
```

The in command returns 1 if the argument *ELEM* is an element of the list passed in argument *LIST* and 0 otherwise. The ni (*not in*) command does the reverse.

```
in apple {apple banana orange} → 1
ni apple {apple banana orange} → 0
```

The elements are compared purely as strings. These are more concise but less flexible forms of the lsearch command.

7.2.2. Infix expressions: expr

The commands discussed in the previous section provide one means for numeric computation in Tcl. However, unless you are from the Lisp world, you might find it awkard to compute 2+3*4 as

```
% set val [+ 2 [* 3 4]]
→ 14
```

The Tcl expression syntax allows the more common infix syntax

```
2+3*4
```

as an alternative. However, because of Tcl's uniform syntax where interpretation of arguments is entirely up to the command itself, in-fix notation can only be used with commands that interpret arguments in that form as expresssions. This can be a point of confusion so at the risk of belaboring the point, let us take a couple of examples.

```
% set val 2+3*4
→ 2+3*4
% list $val == 14
→ 2+3*4 == 14
```

The first statement above assigns the **string** 2+3*4 to the variable val, not the result of that expression. Similarly, the list command in the second statement creates a list from its arguments 2+3*4, == and 14.

In contrast, commands like expr will treat the arguments as in-fix expressions to be calculated.

```
% expr 2+3*4
→ 14
```

Here the expr command interprets its argument as an expression and returns the result. Other commands that treat an argument as an expression include conditional and looping commands like if and while.

Thus unlike most other languages, Tcl will not necessarily treat a string of characters that looks like a numeric expression as being one. The interpretation is dependent on the command to which it is passed as an argument.

We will describe expressions within the context of expr since it is the most fundamental of these commands.

The general form of the expr command is

```
expr ARG ?ARG ...?
```

The command concatenates the supplied arguments separating them by space characters. The result is then treated as a *Tcl expression* and evaluated. The syntax of a Tcl expression differs from the normal Tcl syntax and is described next.

Expressions consists of

- operands which are the values to be used in the operations
- operators which define the operations to be executed
- parenthesis for grouping operands

We describe each of these in turn.

7.2.2.1. Operands in expressions

An operand in an expression may be one of

- a number in any of the representations described earlier
- a boolean literal in any form such as true, false etc.
- a Tcl variable, dereferenced as usual with a $ prefix
- a double quoted string. The expression evaluator will do the same backslash, variable, and command substitutions on the string as Tcl.
- a brace quoted string which is again parsed as in Tcl.
- a Tcl command enclosed in brackets. The result of the command is used as the operand value.
- a mathematical function that uses the form func(arg,...) where the arguments have the same syntax as other operands but are separated by commas.

Some examples:

```
expr {[clock seconds] + 86400}  → 1499235342  ❶
set exponent 3                  → 3
expr {pow(2,$exponent)}         → 8.0  ❷
set s "bar"                     → bar
expr {"foobar" eq "foo$s"}      → 1  ❸
expr {"foobar" eq {foo$s}}      → 0  ❹
```

❶   Bracketed command
❷   Call to the math function pow with numeric literal and variable arguments
❸   Double quoted literal string operand
❹   Brace quoted literal string operand

One point to make a special note of is that unlike in Tcl, **string literals must be quoted within expressions**. For example,

```
% expr {bar eq $s}  ❶
∅ invalid bareword "bar"
  in expression "bar eq $s";
  should be "$bar" or "{bar}" or "bar(...)" or ...
% expr {"bar" eq $s}  ❷
→ 1
```

❶   Error - string literals must be enclosed in quotes or braces
❷   Ok - bar is placed in quotes

7.2.2.2. Operators in expressions

The operators supported in expressions include those we described previously in Section 7.2.1 and a few more. The table Table 7.4 shows them in order of descending *precedence*. Operators with higher precedence are evaluated before those with lower precedence. For example, in the expression 2+3*4, the 3*4 is evaluated first as * has a higher precedence than + as shown in the table. Operators at the same precedence level are evaluated left to right **excepting the exponentiation operator as noted in the table**.

Table 7.4. Expression operators in precedence order

Operators	Description
-, +, ~, !	Unary operators. The - and + indicate the sign of a numeric operand. The ~ is a bit-wise complement and may only be applied to integer operands of any size. The ! operator is a logical complement and may be applied to both boolean and numeric operands.
**	Exponentiation. Multiple exponentiation operators are evaluated from **right to left** so in the example below the 3**2 (9) is evaluated first and then 4**9. This is an exception to the rule that operators at the same level of precedence are evaluated left to right. `expr {4**3**2} → 262144`
*, /, %	Multiplication, division and remainder operators. See Table 7.1 for details on the sign of the remainder when negative numbers are involved.
+, -	Addition and subtraction.
<<, >>	Left and right shifts. Only valid for integer operands. Right shifts propagate the sign bit.
<, <=, >, >=	Comparison operators return 1 if the comparison holds and 0 otherwise. See Section 7.2.1.2 about how these work with numbers and strings.

Operators	Description
==, !=	Comparison operators return 1 if the condition is true and 0 otherwise. See Section 7.2.1.2 about how these work with numbers and strings.
in, ni	Test for list containment or non-containment. The left side operand is an element value and the right side a list value. See Section 7.2.1.5.
&	Bit-wise AND operation. Operands must be integers.
^	Bit-wise XOR operation. Operands must be integers.
|	Bit-wise OR operation. Operands must be integers.
&&	Logical AND operation on booleans or numbers (interpreted as booleans). The evaluation is "short-circuited" (see below).
||	Logical OR operation on booleans or numbers (interpreted as booleans). The evaluation is "short-circuited".
?:	This is the conditional operator that takes the form

```
CONDITION ? TRUEVAL : FALSEVAL
```

If the *CONDITION* operand evaluates to true, the result of the expression is *TRUEVAL*; otherwise, it is *FALSEVAL*.

```
proc min {a b} { expr {$a <= $b ? $a : $b} }
min 2 -2
→ -2
```

The operators &&, || and ?: undergo "short-circuited" evaluation in that some arguments may not be evaluated if the value of the expression is already determined. For example, in the case of the && operator, if the first operand evaluates to false, the second operand is never evaluated.

```
expr {2 > 3 && [nosuchcommand]} → 0
```

The > has a higher precedence than && is therefore executed first. Since it evaluates to a boolean false, the second operand of the && is never evaluated and consequently no error is raised about the command not existing.

Similarly, in the case of the || operator, if the first argument evaluates to true, the second argument is not evaluated. In the case of the ?: conditional operator, if *CONDITION* evaluates to false, then the *TRUEVAL* operand is never evaluated; likewise for *FALSEVAL* if *CONDITION* is true.

7.2.2.3. Grouping operands with parenthesis

If an expression has multiple operators, it is evaluated in order of the operator precedence. If you want to change the order of evaluation you can use parenthesis to force a different order.

```
expr 2+3*4   → 14
expr (2+3)*4 → 20
```

Needless to say, parenthesis may be nested to any desired level.

7.2.2.4. Braces and double substitution

It is important to note that the arguments passed to expr are parsed twice, once by the Tcl parser and once by the command itself. This may lead to double substitution.

Consider the following code.

```
set elem a
set lst {a b c d}
expr {$elem in $lst}
→ 1
```

Here because the braces protect against substitutions by Tcl, the expr command sees a single argument $elem in $list. It parses this argument as per its rules, doing variable substitution etc. and returns the result.

On the other hand, what happens with the following form.

```
expr $elem in $lst
∅ invalid bareword "a"
  in expression "a in a b c d";
  should be "$a" or "{a}" or "a(...)" or ...
```

Now because there are no braces to protect substitutions, the Tcl parser will substitute the variables. The expr command sees three arguments, a, in and a b c d and as per its defined behaviour concatenates them to form the expression

```
a in a b c d
```

This is not a valid expression and hence the generated error.

This example is illustrative of the fact that you have to be aware of the potential for double substitution. In rare cases this may be desirable, but in most instances **you are strongly advised to use the braced argument form of the expr invocation** for reasons of performance and safety:

- Braced expressions can be compiled and cached internally for significantly better performance. So for example, using the time command for measuring execution time, we can see

```
% set x 1 ; set y 2 ; set z 3
→ 3
% puts [time {expr $x+$y*$z} 10000]
→ 5.3816 microseconds per iteration
% puts [time {expr {$x+$y*$z}} 10000]
→ 0.8748 microseconds per iteration
```

- Double substitution opens your code to unexpected surprises and even security risks if the expression being evaluated comes from some untrusted source similar to SQL injection attacks. Consider writing a program that will print the double of a number input by the user which is stored in the variable input. Either forms of the expr command below returns the right result.

```
set input 3      → 3
expr $input*2    → 6
expr {$input*2}  → 6
```

Now however imagine the user inputs the following string instead.

```
% set input {[puts Hacked!; string cat 3]}
→ [puts Hacked!; string cat 3]
% expr $input*2
→ Hacked!
  6
% expr {$input*2}
∅ can't use non-numeric string as operand of "*"
```

Now you can see the problem with the unbraced version which lands up executing the puts command. Imagine if that command was something more nefarious that formatted your disk via exec. The braced form of the command does not suffer from this vulnerability, generating an error instead.

Generally speaking, you should limit yourself to using the unbraced argument form only in interactive mode where it is a little more convenient.

> The safest way to evaluate expressions from untrusted sources is through **safe interpreters** which we will study in Section 20.6.

7.2.2.5. Expressions in other commands

The expr command is not the only command that uses expression syntax. Several other commands such as if, while and for also use expression syntax for their condition argument and follow the same rules as expr for evaluating it. For example,

```
if {$n > 0} {…some code…}
```

is equivalent to writing

```
if {[expr {$n > 0}]} {…some code…}
```

As described in Section 10.4.1, enclosing the condition in braces has added importance there.

7.2.3. The incr command

For the very common case of incrementing variables holding integers, Tcl offers the incr command.

```
incr VAR ?INCREMENT?
```

Here VAR is the name of a variable that, if already existing, must hold an integer value. If it does not exist, it is created with an initial value of 0. The value is then incremented by INCREMENT which must also be an integer and defaults to 1. The command returns the new value of VAR.

```
incr newvar -2 → -2 ❶
incr newvar    → -1 ❷
```

❶   newvar created with value of 0 and then decremented
❷   Default increment of 1

The command is roughly equivalent to

```
set VAR [expr {$VAR + INCREMENT}]
```

except that it only works with integers and not floating point numbers.

More than just conciseness, the advantages of incr compared to expr is efficiency as incr modifies the variable "in place" as opposed to generating a new value that is assigned back to the variable.

> You will sometimes find the following apparent no-op in Tcl code:.
>
> ```
> incr ival 0
> ```

> The purpose of this statement is to verify that `ival` holds an integer value. If not, an error will be raised. This is both faster and less verbose than
>
> ```
> if {[string is entier -strict $ival]} {
>     error "Expected integer, got \"$ival\""
> }
> ```

7.3. Mathematical functions

Tcl provides some commonly used mathematical functions shown in Table 7.5 as commands in the `::tcl::mathfunc` namespace.

Table 7.5. Mathematical functions

Functions	Description
`bool`, `int`, `wide`, `entier`, `double`	Number conversion. See Section 7.1.5.2.
`abs`, `ceil`, `floor`, `round`, `max`, `min`	Number functions for returning absolute value, ceiling, floor, integer with rounding and the maximum or minimum values in a list.
`exp`, `pow`, `log`, `log10`, `isqrt`, `sqrt`	Functions related to exponents and logarithms.
`sin`, `cos`, `tan`, `sinh`, `cosh`, `tanh`, `asin`, `acos`, `atan`, `atan2`, `hypot`	Trigonometric and geometric functions.
`rand`, `srand`	Functions for random number generation.

You can also enumerate the available functions with the `info functions` command.

```
% info functions
→ round wide sqrt sin log10 double hypot atan bool rand abs acos atan2 entier srand sinh ...
% info functions log*
→ log10 log
```

We will not detail these commands here as their functionality should be obvious. See the Tcl reference documentation of `mathfunc` for details.

These commands can be called like any other command by qualifying with the `tcl::mathfunc` namespace

```
tcl::mathfunc::rand       → 0.2698534490865904
tcl::mathfunc::round 3.5 → 4
```

Alternatively you can import (see Section 12.5.3.1) the commands or add the `::tcl::mathfunc` namespace to the namspace path (see Section 12.5.3.2) to call the commands without qualification.

```
namespace path ::tcl::mathfunc → (empty)
set lst {1.0 -1.1 1.1}           → 1.0 -1.1 1.1
max {*}$lst                      → 1.1
```

7.3.1. Using functions in expressions

Commands in the `::tcl::mathfunc` namespace can be called from expressions using the special function call expression syntax described in Section 7.2.2.1. Note that when called in this manner, the function name does not need to be qualified even if it is not imported or placed on the namespace path.

```
set pi 3.14159     → 3.14159
expr {sin($pi/2)} → 0.9999999999991198
```

7.3.2. Defining custom functions

You can add your own commands to the `::tcl::mathfunc` namespace. Doing so allows you to use the command as a function in an expression.

```
proc ::tcl::mathfunc::signum {n} {
    expr {$n < 0 ? -1 : $n > 0 ? 1 : 0}
}

expr {signum(-5)}
→ -1
```

Naturally, you have to take care that the functions you add do not clash with those that might be added by other libraries in the application or even by Tcl itself in the future. It is best to prefix the name appropriately to reduce the chance of a collision.

7.4. Chapter summary

In this chapter we covered numerical computation in Tcl. Although pure number crunching is not in Tcl's sweet spot, it nevertheless supports a wide variety of mathematical needs in terms of functionality if not performance. In addition, several packages and extensions are available that extend Tcl's numerical computation capabilities in features and performance. See Section A.4.

We have now covered Tcl commands for manipulating data in the form of strings, lists, dictionaries and numbers. One last type of data for which Tcl provides built-in support is time values and we cover that next.

8

Dates and Time

Until we can manage time, we can manage nothing else.

— Peter F. Drucker

Tcl can do many things, but sadly cannot manage time, only time values. It can however do a fair number of things with those values:

- Tell you what time it is, even in different timezones and calendars
- Convert to multiple display formats in different locales
- Parse date time strings in various formats
- Perform date and time arithmetic

All functions related to these time manipulation features are implemented by the `clock` ensemble command.

8.1. Unix time and the epoch

Most Tcl commands dealing with time work with time values expressed as the number of seconds since the *epoch* — January 1, 1970, 00:00 UTC. These values may be negative as well, representing a time before the epoch. This representation originally comes from the Unix operating system and is now commonly used in other computing environments. We will thus refer to these values as *Unix time*.

> Tcl time computations do not take into account leap seconds. Time since the epoch is always calculated assuming every minute has 60 seconds.

8.2. The Julian, Gregorian and alternate calendars

The `clock` command and documentation make reference to three different calendars. Some background information will be helpful in understanding the terminology.

The Julian calendar

Historically, the Julian calendar came on the scene first, introduced in 46 BC. It defined a calendar made up of the 12 months that are in common use today with 365 days in a year with a leap year containing an extra day every four years. The consequent average year length of 365.25 days however did not exactly equal the solar year leading to a three day discrepancy over four centuries.

The Gregorian calendar

The Gregorian calendar is the calendar in accepted international use today. It was introduced in 1582 to resolve the above discrepancy by changing the rules for leap years to be every four years except those divisible by 100 and not by 400. This reduced the average year length to 365.2425 days bringing it closer in duration to the solar year.

In addition, the Gregorian calendar compensated for the accumulated difference by removing 10 days from the calendar. The first day of the Gregorian calendar, 15 October 1582, followed the last day of the Julian calendar, 4 October 1582.

The official Gregorian calendar introduction occured on 15 October 1582. The *proleptic* Gregorian calendar extends the calendar definition backward over time.

An additional complication when comparing dates across the two calendars, or converting Unix time to a time string, is that countries adopted the Gregorian calendar at different times. Converting Unix time representations to dates therefore can also entail specification of a locale in which the conversion is to be done. The GREGORIAN_CHANGE_DATE entry in the localization database contains the date on which the locale changed calendars. The clock command uses this when doing the conversion of Unix time to the Gregorian calendar.

Alternative calendars

Some locales, like the Japanese, have other calendars that are still in common use. The Japanese civil calendar is divided into named *eras* based on the reigning Emperor. Years are then numbered within the era and divided into months and days in month in similar fashion to the Gregorian calendar. The clock command includes formatting codes that specify the use of these alternative calendars when formatting dates and times.

8.3. Time zones

When dealing with time zones, the clock command retrieves the time zone to be used from one of the following sources in order of preference:

- A time zone specified inside the string being parsed
- A time zone specified by command options -timezone or -gmt
- The TCL_TZ and TZ environment variables (in that order)
- The local time zone from system settings (Windows only)
- The C runtime library

The time zone strings may take one of several formats:

- Standardized location names begin with a :, like :America/Argentina/Buenos_Aires. A full list of location names can be found either under lib/tcl*VERSION*/tzdata or under a system specific directory like /usr/share/zoneinfo on Unix systems. The string :localtime is a special case that refers to the local time zone as defined by the C runtime library.
- A second form is a string starting with a + or -, denoting a time zone east or west of Greenwich respectively, followed by a two digit hour offset, a two digit minute offset, and optionally a two digit seconds offset. For example, +0530 is five and a half hours east of Greenwich and -080030 is eight hours and thirty seconds west.
- A string conforming to the Posix definition of the TZ environment variable, for example EST+5 for Eastern Standard Time. Note that the semantics of + and - are opposite from the second form above with + indicating a time zone *west* of the Greenwich and - denoting east. See the POSIX specification for the full syntax which allows for daylight savings start, end and offsets.
- Strings that do not match any of the above are prefixed with a : and attempted to be handled as location names described in the first item above.

We will see examples of time zone use as we discuss each command.

8.4. Retrieving the current time: clock seconds | milliseconds | microseconds

The clock seconds, clock milliseconds and clock microseconds return the current time as the number of seconds, milliseconds and microseconds since the epoch respectively.

```
clock seconds       → 1499148942
clock milliseconds  → 1499148942884
clock microseconds  → 1499148942885092
```

8.5. Interval measurement

While the above commands return time as the number of elapsed time units since the epoch, the `clock clicks` command returns a high-resolution system-dependent value that is not tied to any epoch.

```
clock clicks → 3493118613
```

The return value from the command cannot be converted to a date and time in any calendar. Rather the difference between two return values can be used for measuring intervals with the highest resolution offered by the platform.

8.6. Formatting time for display: `clock format`

The `clock format` command formats a time value, given as number of seconds since the epoch, into a string of a specified format that is suitable for display or for passing to other programs.

```
clock format SECSFROMEPOCH ?OPTIONS?
```

Without any options, the command will return a string using a default format and locale with the local time zone.

```
% set now [clock seconds]
→ 1499148942
% clock format $now
→ Tue Jul 04 11:45:42 IST 2017
```

8.6.1. Formatting for a different time zone: `-timezone`, `-gmt`

By default, the command will format the time using the default time zone. To display the time in the local time zone at the epoch:

```
% clock format 0
→ Thu Jan 01 05:30:00 IST 1970
```

A different time zone can be specified with the `-timezone` option.

```
% clock format 0 -timezone :America/New_York
→ Wed Dec 31 19:00:00 EST 1969
```

The `-gmt` option is an alias for the UTC time zone.

```
% clock format 0 -timezone :UTC
→ Thu Jan 01 00:00:00 UTC 1970
% clock format 0 -gmt 1
→ Thu Jan 01 00:00:00 GMT 1970
```

8.6.2. Formatting for a locale: `-locale`

The `clock format` command accepts the `-locale` option to display the time in a format suitable for a specific locale. The permissible values for the option are any locale identifiers accepted by the `msgcat` (see Section 4.15) command and the values `current` and `system`. The value `current` refers to the locale returned by the `mclocale` command while `system` refers to user preferences if available (such as the registry on Windows), and is synonymous with `current` otherwise.

Note that if `-locale` is not specified, it defaults to the ROOT locale, not to the `current` or `system` locale.

```
% set now [clock seconds]
→ 1499148942
% msgcat::mclocale es ❶
→ es
% clock format $now
→ Tue Jul 04 11:45:42 IST 2017
```

❶   Sets locale to Spanish

Notice that though the locale has been set to Spanish, the time is not formatted as per that locale. This is because if the -locale option is not specified, it defaults to the ROOT locale, not to the current or system locale. To format as per the current locale, we have to explicitly specify that with the -locale option.

```
% clock format $now -locale current
→ mar jul 04 11:45:42 IST 2017
```

The above will format as per the locale returned by mclocale (es in our example).

Alternatively you can specify an explicit locale.

```
% clock format $now -locale fr
→ mar. juil. 04 11:45:42 IST 2017
```

8.6.3. Controlling display format: -format

For exact control over the display format for time, clock format accepts the -format option whose value, the format specification, controls the generated display string. The format specification consists of *format groups*, which are two-character sequences beginning with the % character. Each format group is replaced with a specific time component like day or hour as shown in Table 8.1 and other characters present between the format groups are passed through unchanged.

```
% clock format [clock seconds] -format "The current time is %r."
→ The current time is 11:45:42 am.
```

The format groups supported by clock format, as well as by the clock scan command we will describe later, are shown in Table 8.1.

Table 8.1. Format groups for clock

Format group	Description
%a, %A	Locale dependent day of the week in short and full form respectively.

	```clock format [clock seconds] -format %a → Tue```
	 ```clock format [clock seconds] -format %A → Tuesday```
	 ```clock format [clock seconds] -format %A -locale de → Dienstag```

%b, %B	Locale dependent month short and full form respectively.

	```clock format [clock seconds] -format %b → Jul```
	 ```clock format [clock seconds] -format %B → July```
	 ```clock format [clock seconds] -format %B -locale de → Juli```

%c	Localized representation of date and time of day.
	```% clock format [clock seconds] -format %c```

```
→ Tue Jul  4 11:45:42 2017
% clock format [clock seconds] -format %c -locale de
→ 04.07.2017 11:45:42 +0530
```

%C	Number of the century

```
clock format [clock seconds] -format %C → 20
```

%d	Two digit number of the month.
%D	Synonym for %m/%d/%Y

```
clock format [clock seconds] -format %D → 07/04/2017
```

%e	Day of the month as two digits or a single digit with a leading space.
%Ec, %Ex, %EX, %Ey, %EY	Corresponds to the %c (localized date and time), %x (localized date), %X (localized time of day), %y (two digit year) and %Y (full year) format groups except that the locale's alternative calendar is used. For example, in the Japanese locale, the alternative calendar is the Japanese civil calendar.

```
% clock format 0 -locale ja -format %Y -gmt 1
→ 1970
% clock format 0 -locale ja -format %EY -gmt 1
→ 昭和45
% clock format 0 -locale ja -format %c -gmt 1
→ 1970/01/01  0:00:00 +0000
% clock format 0 -locale ja -format %Ec -gmt 1
→ 昭和45年01月01日 (木) 00時00分00秒 +0000
```

Note for example that the epoch year 1970 is shown as year 45 in the 昭和, or Shōwa, era.

%EC	The locale-dependent era in the locale's alternative calendar

```
clock format 0 -locale ja -format %EC -gmt 1 → 昭和
```

%EE	Either string B.C.E. or C.E., or their localized versions, depending on whether %Y refers to dates before or after Year 1 of the Common Era.

```
clock format 0 -format %EE            → C.E.·
clock format 0 -format %EE -locale de → n. Chr.
```

%g, %G	A 2-digit and 4-digit year suitable for use with the week-based calendar.
%h	Same as %b.
%H, %I	Two digit hour of the day on a 24 and 12 hour clock respectively.

```
% clock format [clock seconds] -format %H
→ 19
% clock format [clock seconds] -format %I
→ 07
```

%j	A 3-digit day of the year.

```
clock format 0 -format %j → 001
```

%J	The Julian day number. This is often useful in calendar calculations.

```
proc julian {secs} {
    return [clock format $secs -format %J]
}
proc days_since {mmddyy} {
    set then [clock scan $mmddyy -format "%Y/%m/%d"]
    return [expr {[julian [clock seconds]] - [julian $then]}]
}
puts "World War II ended [days_since 1945/09/02] days ago."
→ World War II ended 26238 days ago.
```

%k, %l — The one or two digit hour of the day using a 24- and 12-hour clock respectively. Note the single digit hours are left padded with a space.

```
clock format [clock seconds] -format "(%l)" → (11)
```

%m, %N — Number of the month where %m always produces a 2-digit value while %N left pads single digit months with a space.

```
clock format 0 -format (%m) → (01)
clock format 0 -format (%N) → ( 1)
```

%M — A 2-digit minute of the hour (00-59).

%Od, %Oe, %OH, %OI, %Ok, %Ol, %Om, %OM, %OS, %Ou, %Ow, %Oy — These are the same as the corresponding specifiers without the O except that they use locale-dependent alternative numerals

%p, %P — Outputs a locale-specific AM/PM (%p) or am/pm (%P) indicator. If the locale supports both lower and upper case variations, %p and %P select the upper and lower case forms respectively.

```
clock format [clock seconds] -format "%H:%M %p" → 11:45 AM
clock format [clock seconds] -format "%H:%M %P" → 11:45 am
```

%r — Locale-dependent time of day using a 12-hour clock.

```
clock format [clock seconds] -format %r → 11:45:42 am
```

%R — Hours and minutes as 24-hour clock. Same as %H:%M.

%s — Outputs the *SECSFROMEPOCH* argument as a decimal string.

```
% clock format [clock seconds] -format "It is %s seconds since the epoch."
→ It is 1499148942 seconds since the epoch.
```

%S — The 2-digit second of the minute.

%t — Outputs a tab character

%T — Time of day. Alias for %H:%M:%S.

%u, %w — The number for the day of the week. %u conforms with ISO8601 with days Monday-Sunday numbered 1-7 while %w numbers Sunday-Saturday as 0-6.

```
% clock format [clock seconds] -format "Today, %A, is day %u of the week."
→ Today, Tuesday, is day 2 of the week.
```

%U, %V,%W	The ordinal number of the week in the year. %U returns a number in range 00-53 with the first Sunday of the year being the first day of week 01. %W is similar except for week 01 beginning on the first Monday of the year. The prefered grouping %V, which conforms to "ISO8601 week numbering" (see Section 8.9.1.2), returns in the range 01-53.
%x, %X	Locale-dependent date and time representation respectively.

```
clock format $now -format "%x %X"            → 07/04/2017 11:45:42
clock format $now -format "%x %X" -locale be → 4.07.2017 11.45.42
``` |
| %y, %Y | The 2-digit year of the century and 4-digit calendar year respectively. Note that neither yields the correct value for use with ISO8601 week numbers for which %g and %G should be used instead. |
| %z, %Z | Returns the current time zone in +/-hhmm and name format respectively.

```
clock format 0 -format %z -timezone :America/New_York → -0500
clock format 0 -format %Z -timezone :America/New_York → EST
``` |
| %% | Outputs a single % character |
| %+ | Same as %a %b %e %H:%M:%S %Z %Y.

```
clock format 0 -format %+ → Thu Jan  1 05:30:00 IST 1970
``` |

8.7. Parsing dates and times: `clock scan`

The inverse of the `clock format` command is `clock scan` which parses a time string and returns the corresponding count of seconds since the epoch.

```
clock scan TIMESTRING ?OPTIONS?
```

The expected format of *TIMESTRING* is specified by the `-format` option. It is strongly recommended that this option **always** be specified. If this option is not present, the command uses heuristics to guess the format of the string. These may lead to unexpected results due to ambiguities in interpretation of the provided fields. This is described later in Section 8.7.5.

Even with the `-format` option specified, the parsing algorithm used by `clock scan` is necessarily complex for several reasons and we will not detail it here. See the Tcl reference documentation for a full exposition.

8.7.1. Specifying the parse format: `-format`

The `-format` option controls the parsing process by specifying the expected format of the input string. The value for this option takes exactly the same form as described earlier for the `clock format` command except that here the format groups define what time components are expected in the input *TIMESTRING* argument and not how they are to be displayed. The format groups shown in Table 8.1 also apply to `clock scan` so we will not repeat them here but just show some examples.

Parse a full date and time specification:

```
% set t [clock scan "19900613 003000" -format "%Y%m%d %H%M%S"]
→ 645217200
% clock format $t
→ Wed Jun 13 00:30:00 IST 1990
```

If the time is not specified, `clock scan` assumes 00:00:00.

```
% set t [clock scan "1990-06-13" -format "%Y-%m-%d"]
→ 645215400
% clock format $t
→ Wed Jun 13 00:00:00 IST 1990
```

If the date is not specified, the current date is assumed unless the -base option is specified. Note the use of the AM/PM designator in the example.

```
% set t [clock scan "12:30am" -format "%I:%M%p"]
→ 1499108400
% clock format $t
→ Tue Jul 04 00:30:00 IST 2017
```

A feature of clock scan is that it parses embedded fields in strings.

```
% set t [clock scan "October 27, 2004 - a memorable day in history!" \
         -format "%B %d, %Y - a memorable day in history!"]
→ 1098815400
% clock format $t
→ Wed Oct 27 00:00:00 IST 2004
```

However, the fact that the non-time related characters must exactly match the format string limits its usefulness in parsing log files and such.

> ⚠
>
> If the format specification does not contain components that specify the full date (year, month and day), then even the components included in the specification may be ignored and default to the base date. For instance,
>
> ```
> % clock format [clock scan 2014-01-02 -format %Y-%m-%d] ❶
> → Thu Jan 02 00:00:00 IST 2014
> % clock format [clock scan 2014-01 -format %Y-%m] ❷
> → Tue Jul 04 00:00:00 IST 2017
> ```
>
> ❶ Full date specified
> ❷ Full date not specified so defaults to the base date (current date)
>
> Thus it strongly advised to require all date components to be included either directly or indirectly (for example, numerical day in year instead of month and day in month).

8.7.2. Specifying the time zone for parsing: -timezone and -gmt

Just like for clock format, the -timezone option can be specified to indicate which time zone should be assumed for the string being parsed.

```
clock scan "19900613 003000" -format "%Y%m%d %H%M%S"                  → 645217200
clock scan "19900613 003000" -format "%Y%m%d %H%M%S" -timezone :UTC → 645237000
clock scan "19900613 003000" -format "%Y%m%d %H%M%S" -gmt 1          → 645237000
clock scan "19900613 003000" -format "%Y%m%d %H%M%S" -timezone EST   → 645255000
```

8.7.3. Parsing localized time strings: -locale

The clock scan command also accepts the -locale option for parsing localized date and time strings. By default, this is the root locale {} and not the current locale as returned by msgcat::mclocale. The latter can be specified

using -locale current. On some systems, the -locale option also accepts the value system. On Windows this refers to the user's locale settings in the Control Panel.

```
% set tstring_fr [clock format 0 -format "%A, %B %d %Y" -locale fr -gmt 1]
→ jeudi, janvier 01 1970
% clock scan $tstring_fr -format "%A, %B %d %Y" -locale fr -gmt 1
→ 0
% clock scan $tstring_fr -format "%A, %B %d %Y" ❶
Ø input string does not match supplied format
```

❶   Error because default locale is not fr

8.7.4. Changing the defaults for parsing: -base

When a date is not fully specified, the clock scan command uses the *base date* as the default for unspecified components. By default the base date is the current date. In the example below, since the year is not specified, it will default to the current year.

```
% puts "The current year is [clock format [clock seconds] -format %Y]"
→ The current year is 2017
% clock format [clock scan "01/31" -format "%m/%d"]
→ Tue Jan 31 00:00:00 IST 2017
```

This base date can be changed to use a different date by specifying the -base option. The value of the option must be specified as the number of seconds since the epoch. So to use the epoch year as the base date,

```
% clock format [clock scan "01/31" -format "%m/%d" -base 0]
→ Sat Jan 31 00:00:00 IST 1970
```

Or to use year 2000 as the base date,

```
% set secs2000 [clock scan 2000/01/01 -format %Y/%m/%d]
→ 946665000
% clock format [clock scan "01/31" -format "%m/%d" -base $secs2000]
→ Mon Jan 31 00:00:00 IST 2000
```

Note that there is no "base time"; if the time is not specified, it defaults to midnight 00:00:00 in the current locale.

8.7.5. Free form parsing of time strings

When the -format option is not specified to the clock scan command, it attempts to guess the format of the passed argument. This form is now deprecated because of the ambiguity in interpreting strings and we therefore do not discuss it further.

There is however, one useful form of the free form scan that allows specifying relative time using keywords now, today, tomorrow, yesterday, next, last and ago. Some examples:

```
clock format [clock scan now]            → Tue Jul 04 11:45:42 IST 2017
clock format [clock scan tomorrow]       → Wed Jul 05 00:00:00 IST 2017
clock format [clock scan "next week"]    → Tue Jul 11 00:00:00 IST 2017
clock format [clock scan "last month"]   → Sun Jun 04 00:00:00 IST 2017
clock format [clock scan "2 years ago"]  → Sat Jul 04 00:00:00 IST 2015
```

Even here, you have to be careful in use of the words. For example,

```
% clock format [clock seconds]
→ Tue Jul 04 11:45:42 IST 2017
% clock format [clock scan "last 2 months"]
→ Sun Sep 03 23:59:59 IST 2017
```

which is likely not what you expected. The string last 2 months is parsed as 2 months after the "last day". The author's recommendation is to avoid surprises by restricting free form scanning to unambiguous simple keywords like yesterday.

8.8. Time arithmetic: clock add

Tcl provides for basic time arithmetic operations with the clock add command.

```
clock add TIMEVAL ?COUNT UNIT ...? ? OPTIONS?
```

In the basic form, it takes *TIMEVAL*, in the form of Unix time, and one or more pairs of arguments that specify the number and unit by which the *TIMEVAL* is to be changed. For example,

```
set now [clock seconds]                    → 1499148943
clock format $now                          → Tue Jul 04 11:45:43 IST 2017
clock format [clock add $now 1 week]       → Tue Jul 11 11:45:43 IST 2017
clock format [clock add $now -2 hours]     → Tue Jul 04 09:45:43 IST 2017
```

The unit of time may be one of years, months, weeks, days, hours, minutes or seconds and the singular forms of these are accepted as well. The command will accept any number of change specifications.

```
clock format [clock add $now 2 years 1 month 1 day] → Mon Aug 05 11:45:43 IST 2019
clock format [clock add $now 1 day 1 day 1 day]     → Fri Jul 07 11:45:43 IST 2017 ❶
```

❶   Same as clock add $now 3 days

The clock add command implements the -timezone and -locale options which take on the same values as described for the clock format and clock scan options.

The -timezone option is pertinent because it affects arithmetic across daylight savings boundaries as we will see below.

The -locale option controls the date used as the transition from the Julian to the Gregorian calendars which differs in different parts of the world. As a consequence it affects time arithmetic that crosses the transition date as we described in [id_clock_gregorian_change_date].

8.8.1. Clock computations

Because of the various lengths of a time unit (for example, a month might 28, 29, 30 or 31 days), some discussion is warranted in terms of how the arithmetic is done. Here we only provide a summary and refer you to the Tcl command reference for full details and some of the finer points.

Adding seconds, minutes and hours

Hours and minutes are converted to seconds by multiplying with 3600 and 60 respectively and the result added to the *TIMEVAL* argument. Leap seconds are ignored.

Adding days and weeks

Adding days and weeks is done by first converting *TIMEVAL* into a calendar day (**not** date). The days, or weeks multiplied by 7, are then added to the calendar day and then converted back into seconds.

Note that the above means that adding 24 hours and adding 1 day do not always have the same effect. Here is an example from the Tcl command reference that illustrates the difference when the change crosses a Daylight Savings Time boundary.

```
% set t [clock scan {2004-10-30 05:00:00} \
            -format {%Y-%m-%d %H:%M:%S} \
            -timezone :America/New_York]
→ 1099126800
% set tplus1day  [clock add $t 1 day -timezone :America/New_York]
→ 1099216800
% clock format $tplus1day -format %T -timezone :America/New_York
→ 05:00:00
% set tplus24hrs [clock add $t 24 hours -timezone :America/New_York]
→ 1099213200
% clock format $tplus24hrs -format %T -timezone :America/New_York
→ 04:00:00
```

There are some additional special cases for daylight savings changes, such as a local time appearing twice, or the resulting time being "impossible" when the clock jumps forward. See the command reference as to how these are handled.

Adding months and years

Adding months and years works similar to adding of days and weeks except that *TIMEVAL* is first converted to the calendar date, **not** day. The months or years are then added to the calendar date as appropriate. If the resulting date is invalid because the month has fewer days, it is set to the last day of the month.

```
% set t [clock scan {2016-05-31} -format %Y-%m-%d]
→ 1464633000
% set tplus1month [clock add $t 1 month] ❶
→ 1467225000
% clock format $tplus1month
→ Thu Jun 30 00:00:00 IST 2016
```

❶ June 31 would be invalid

Like for arithmetic involving days and weeks, several special cases arise related to daylight savings and calendar changes. We again refer you to the command reference for details.

8.9. Time representation standards

There are several standards that define how dates and times are to be represented to facilitate sharing across applications. We describe two of these here with regards to their handling in Tcl.

8.9.1. The ISO 8601 standard

The ISO 8601 international standard defines standard formats for representing date and time related values.

- The components of dates and times are ordered from the largest to the smallest unit: year, month or week, day of month or day of week, hour, minute, second, and fractions of seconds.
- Components may be left out provided all smaller units are also omitted. For example, if the minutes components is omitted, the seconds and fraction of seconds components must also be omitted.
- Each component has a fixed width and is padded with leading zeroes if necessary.

- The character - is used as the separator between the date components and : between time components. These separators are optional and may be omitted. Thus the strings 1969-07-21 and 19690721 are both valid representations for the date a giant step was taken for mankind.

Tcl itself provides the underlying mechanism for parsing ISO 8601 format dates with the clock scan command as we have seen but that requires knowledge of which specific ISO 8601 format is in use. The clock::iso8601 package in the Tcl standard library Tcllib [1] provides a higher level interface that we will demonstrate below.

```
% package require clock::iso8601
→ 0.1
```

Dates in ISO 8601 can be *calendar dates*, *week dates*, or *ordinal dates*.

8.9.1.1. ISO 8601 calendar dates

Calendar dates are the commonly used form consisting of a 4-digit year, 2-digit month and 2-digit day of month. Valid syntax is one of

```
YYYY-MM-DD or YYYYMMDD
YYYY-MM
YYYY
```

Note YYYYMM is not valid ISO 8601 syntax.

Constructing dates in this representation is straightforward.

```
clock format [clock seconds] -format "%Y-%m-%d" → 2017-07-04
clock format [clock seconds] -format "%Y%m%d"   → 20170704
```

For parsing, we can use the iso8601 parse_date command from the iso8601 package to parse these formats without having to know the format being used up front as would be required for clock scan.

```
clock::iso8601 parse_date 1990-06-13 → 645215400
clock::iso8601 parse_date 19900613   → 645215400
clock::iso8601 parse_date 1990-06    → 644178600
```

8.9.1.2. ISO 8601 week dates

A second format for dates uses a week number, 01-53, in the year and a day in the week instead of the month and day in month. Because weeks can cross year boundaries, the numbering of weeks is not completely straightforward. ISO 8601 treats weeks as belonging to the year in which the majority of days in that week lie. Formally, the standard defines week 01 of a year as the week containing the year's first Thursday.

The ISO 8601 syntax for week dates takes one of the forms

```
YYYY-Www or YYYYWww
YYYY-Www-D or YYYYWwwD
```

where ww is the 2-digit week number and D is the day number in the week.

The definition of week date means that the value of the year component in an ISO 8601 date may differ based on whether calendar dates are being used or week dates. For example, January 1 2005 would be represented as 2005-01-01 as a calendar date but 2004-W53 as a ISO 8601 week date.

---

[1] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

!

When formatting time for ISO 8601 weeks, it is important to keep this difference in mind and use %g/%G to format the year component and **not** %y/%Y. For similar reasons, use %V and %u for the week number and day in week components.

```
set t [clock scan "2005-01-01" -format "%Y-%m-%d"] → 1104517800
clock format $t -format "%G-W%V-%u"                 → 2004-W53-6 ❶
clock format $t -format "%Y-W%W-%w"                 → 2005-W00-6 ❷
```

❶ Correct ISO 8601 week date
❷ **Wrong**

For parsing ISO 8601 week dates, we can just use the iso8601 parse_date command as before.

```
% set t [clock::iso8601 parse_date 2004-W53-6]
→ 1104517800
% clock format $t -gmt 1
→ Fri Dec 31 18:30:00 GMT 2004
```

8.9.1.3. ISO 8601 ordinal dates

Ordinal dates specify the 4-digit year and the 3-digit day number within the year. It takes one of the forms

```
YYYY-DDD or YYYYDDD
```

Ordinal dates are formatted using the %j format group which gives the day of the year.

```
clock format [clock seconds] -format "%Y-%j" → 2017-185
clock format [clock seconds] -format "%Y%j"  → 2017185
```

As always, for parsing iso8601 parse_date does the job for us.

```
set t [clock::iso8601 parse_date 1985-036] → 476389800
clock format $t -gmt 1                      → Mon Feb 04 18:30:00 GMT 1985
```

8.9.1.4. ISO 8601 time

Time in ISO 8061 is represented in any of the following formats.

```
hh:mm:ss.sss or hhmmss.sss
hh:mm:ss or hhmmss
hh:mm or hhmm
hh
```

When combined with a date to denote a single instant, the time and date strings are separated by a single T character.

The time may optionally be followed by a time zone designator. The time zone is specified as Z indicating UTC or as an offset from UTC in one of the forms ±hh:mm, ±hhmm or ±hh where time zones east of Greenwich have a + prefix and those west have a - prefix. If absent, the time is assumed to be in local time.

Formats that do not include fractional seconds can be easily constructed with the clock format command using the %H, %M and %S format groups.

```
% clock format [clock seconds] -format "%H:%M:%SZ" -timezone :UTC
→ 06:15:43Z
```

However, the clock format command does not have support for fractional seconds and any such value needs to be manually formatted.

```
% set t [clock milliseconds]
→ 1499148943025
% format "%s.%sZ" [clock format [/ $t 1000] -format %H:%M:%S -timezone :UTC] [% $t 1000] ❶
→ 06:15:43.25Z
```

❶   Assumes / and % are imported from ::tcl::mathop

The iso8601 parse_time command parses a time or a full date *and* time representation.

```
% set t [clock::iso8601 parse_time 2000-01-01T00:00:00Z]
→ 946684800
% clock format $t -timezone :UTC
→ Sat Jan 01 00:00:00 UTC 2000
% set t [clock::iso8601 parse_time 12:15-05:00] ❶
→ 1499188500
% clock format $t -timezone EST
→ Tue Jul 04 12:15:00 EST 2017
```

❶   Assumes today's date with Eastern Standard Time

Nevertheless there are limitations here as well in that the command does not parse fractional seconds as its output format is always an integral number of seconds.

8.9.2. RFC 2822 format

The IETF standard RFC 2822 defines a syntax for electronic mail and includes a specification for representing time.

As for ISO 8601, Tcllib[2] provides a package, clock::rfc2822, for directly parsing time in this format.

```
% package require clock::rfc2822
→ 0.1
```

The rfc2822 parse_date command can then be used to parse this representation into Unix time.

```
% set t [::clock::rfc2822 parse_date "Fri, 1 April 2000 04:20:00 -0600"]
→ 954584400
% clock format $t -timezone -0600
→ Sat Apr 01 04:20:00 -0600 2000
```

8.10. Localization

We have seen use of the -locale option with various commands to format and parse time values using localized names of months and weeks, different formats and representations and so on. This makes use of Tcl's msgcat facility and supports many locales out of the box. Additional locales may be added defining a set of localized strings as described in the TIP 173[3] specification.

Here we present a small example of extending an existing locale, say fr, to display dates using a different separator, |, when the format group %x is specified. From TIP 173, we find that the relevant entry is DATE_FORMAT and we can set it with the following snippet.

---

[2] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html
[3] http://www.tcl.tk/cgi-bin/tct/tip/173

```
namespace eval ::tcl::clock {
    ::msgcat::mcset fr_xx DATE_FORMAT "%e|%B|%Y"
}
→ %e|%B|%Y
```

And voilà!

```
% clock format 0 -format %x -locale fr
→  1 janvier 1970
% clock format 0 -format %x -locale fr_xx
→  1|janvier|1970
```

The above snippet would normally be placed in the fr_xx.msg file in a directory that is loaded by the msgcat package but could also be executed independently.

See Section 4.15 for more information.

8.11. Chapter summary

Almost any non-trivial software program has to deal with date and time values at some level. In this chapter, we described the comprehensive facilities Tcl provides for this purpose, including retrieving the current time, formatting and parsing time values, localization and time arithmetic. Another function related to time is the scheduling of events. We will defer that topic to Chapter 15. The vast majority of applications will find these built-in commands suffice without the need for any external tools and libraries.

Having covered commands for manipulation of data in various forms ranging from strings to dates and times, we will now move on to another basic operation related to data — input and output.

9

Files and Basic I/O

There are very few programming tasks that do not involve access in some fashion to data stored in files. Correspondingly, Tcl provides a wide variety of commands that deal with related aspects. Some of these are the standard features found in most programming languages including commands for

- Parsing and construction of file paths in a platform-independent manner.
- File system operations for managing volumes and directories, accessing file metadata etc.
- Reading and writing data to files. Both synchronous and asynchronous operations are supported.

In addition, Tcl's channel abstraction provides some unique capabilities not commonly found in other languages:

- Transparent handling of character encodings
- Data transforms, such as compression, during I/O
- Ability to define new channel types, for example to write to in-memory buffers using I/O commands
- A virtual file system API with *plug-ins*, for example to access Web pages or remote files over HTTP or FTP with the standard file commands.

In this chapter, we discuss only the basics, delegating advanced topics like asynchronous I/O, reflected channels, transforms, and virtual file systems to later chapters.

9.1. File paths

The majority of functions related to file paths and file systems are implemented by the `file` ensemble command.

```
file SUBCOMMAND PATH
```

As with most ensemble commands, SUBCOMMAND may be specified as a **unique** prefix. For example, both the following are equivalent since isd is a unique subcommand prefix.

```
file isdirectory C:/Windows  → 1
file isd C:/Windows          → 1
```

9.1.1. Path syntax

Platforms differ in the syntax used for specifying file paths. For example, Unix uses / as the path separator while Windows accepts either / or \ and also has an optional drive or volume component.

On Unix and MacOS X, file paths may contain any character other than a / which is used as the file path component separator. Path components . and .. are special cases that are interpreted as the current directory and its parent respectively. Multiple adjacent / characters are treated as a single / character and trailing ones are ignored.

Windows file paths may start with an optional drive letter or a UNC path of the form \\COMPUTERNAME \SHARENAME. They permit both / and \ as separators and interpret . and .. just like Unix and MacOS X.

When using the \ character as a file path separator in a literal path string, remember that Tcl also treats it as a special character and needs to be escaped by doubling or protected inside braces.

Tcl hides platform differences to the extent possible by two means:

- On all platforms, Tcl supports a generic Unix-like syntax that uses / as the file path separator. The native separator used by a file system can be obtained with the `file separator` command (see Section 9.2.1).
- Tcl provides commands for parsing and constructing file paths from individual path components so applications do not need to be concerned about the exact syntax required.

Nevertheless, applications should be aware that the set of characters supported in paths, path length limits etc. vary between file systems.

9.1.1.1. Absolute and relative paths: `file pathtype`

Tcl supports both absolute and relative paths. The type of a path may be ascertained with the `file pathtype` command.

```
file pathtype PATH
```

The command works purely on a syntactic basis so the specified path does not have to actually refer to an existing file. It returns `absolute` if the path refers to a specific file on a specific volume.

```
file pathtype c:/foo/bar                   → absolute
file pathtype {\\RemoteSystem\C_Drive\foo} → absolute
```

If the path is relative to the current working directory, the command returns `relative`.

```
file pathtype foo/bar     → relative
file pathtype ../foo/bar → relative
```

Finally, on Windows system where file paths can have a drive component, the command returns `volumerelative` if the path is either relative to the current working directory on a specifed volume *or* a specific file on the current working volume.

```
file pathtype {c:foo\\bar} → volumerelative
file pathtype {/foo/bar}    → volumerelative ❶
```

❶   Note that on Unix, this would return the value `absolute`

9.1.1.2. Path normalization: `file normalize`, `fileutil::lexnormalize`, `fileutil::fullnormalize`

A path that is relative can be converted to an absolute path with the `file normalize` command.

```
file normalize PATH
```

The argument *PATH* may be relative, volume relative or absolute. In the first two cases, it is converted to an absolute path. In addition, for all three cases the command takes the following actions to generate a unique string to identify the file.

- Removes all . and .. occurences, adjusting other path components as appropriate.
- Does leading tilde substitution (see Section 9.1.1.3).

- Replaces all links in the path by their targets with the exception that the last component is not replaced even if a link. This exception is by design in case the application wants to manipulate the link itself and not the link target.
- (Windows only) Replaces any path components that use the 8.3 short name format by the long form name. Moreover, if the path component actually exists, the exact case-sensitive version of the name is used.

The following examples illustrate the behaviour on Windows systems.

```
% file normalize a/b ❶
→ C:/temp/a/b
% file normalize c:/temp/foo/../././bar ❷
→ C:/temp/bar
% file normalize c:/WINDOWS/system32 ❸
→ C:/Windows/System32
% file normalize c:/WINDOWSX/system32 ❹
→ C:/WINDOWSX/system32
% file normalize AVERYL~1 ❺
→ C:/temp/A very long file name
```

❶   Convert relative to absolute
❷   Removal of . and ..
❸   Fix character case for path components that exist
❹   Path components that do not exist keep existing character case
❺   Convert file short name to long name

The way `file normalize` handles links may not be suitable for some purposes. First, it replaces any links in the path with the target. Second it does *not* replace a link if it is the last component in the path.

If you do want the last component in the file path to also be replaced if it is a link, you can use the following trick. Append a dummy non-existent file name, such as ... , and normalize the resulting path. Then use `file dirname` to retrieve the original path in normalized form. For example, if `$path` is the path to be normalized,

```
file dirname [file normalize [file join $path ...]]
```

The `fileutil` module of Tcllib[1] provides two alternatives to the `file normalize` functionality:

- The `fullnormalize` command implements the above trick, normalizing all links in the path.
- The `lexnormalize` command performs normalization purely based on the syntactic structure of its argument with no special consideration for links and without converting 8.3 names to their long form.

9.1.1.3. Tilde substitution and the home directory

The Tcl file commands treat file names starting with a tilde, ~, in special fashion. If the tilde is immediately followed by a path separator, it is replaced by the value of the HOME environment variable. Otherwise, all characters between the tilde and the next path separator are treated as the name of a user on the system and the path component is replaced with that user's home directory.

```
file normalize ~/foo      → C:/Users/ashok/Documents/foo
file normalize ~ashok/foo → C:/Users/ashok/foo
```

---

[1] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

The ~USER form does not look at the specified user's HOME environment variable. Thus the two forms ~ and ~USER may not give the same results even when USER is the same as the one owning the current process

Tilde substitution can be a problem, particularly on Windows where, unlike Unix shells, tilde substitution is not common practice. Thus a file name beginning with a tilde is perfectly legitimate but using it directly will lead to Tcl interpreting it as a user name. To work around this, the following fragment is useful.

```
if {[file pathtype $path] eq "relative"} {
    set path "./$path"
}
```

Since tilde substitution is only done on the *first* component in the path, it will not be effected here.

9.1.2. Parsing paths: `file split|extension|rootname|dirname|tail`

To allow programming without being bothered by platform-specific details of path syntax, Tcl provides a number of commands for parsing and extracting components from a path.

The first of these, `file split` breaks a path into separate components.

```
file split PATH
```

The return value from the command is a list of path components. Note that PATH may be relative or absolute.

```
% file split c:/dir/file
→ c:/ dir file
% file split {\\RemoteSystem\ShareName\dir\file}
→ //RemoteSystem/ShareName dir file
% file split dir/file
→ dir file
```

In addition, several subcommands return parts of a file path. The specified path passed to the commands may be either absolute or relative.

The `file extension` command returns the extension of a path or an empty string if the path does not have an extension. Conversely, `file rootname` returns the entire path except the extension.

```
file extension /dir/subdir/file.ext  → .ext
file extension /dir/subdir/file       → (empty)
file rootname /dir/subdir/file.ext    → /dir/subdir/file
```

Similarly, the complementary commands `file dirname` and `file tail` return the directory component of the path and the name of the file respectively.

```
file dirname /dir/subdir/file.ext → /dir/subdir
file tail /dir/subdir/file.ext     → file.ext
file dirname foo                   → .
file tail foo                      → foo
file tail foo/                     → foo ❶
```

❶   Note trailing separators are completely ignored.

9.1.3. Constructing paths: `file join`

Just as for parsing paths, Tcl provides a command, `file join`, for path construction in a platform-independent manner.

```
file join PATH ?PATH …?
```

If a `PATH` argument is a relative path, it is joined to the path being constructed using a path separator. If the argument is itself an absolute path, the path constructed so far is discarded and the argument becomes the initial value of the constructed path when processing the remaining arguments.

```
file join dir subdir file.ext          → dir/subdir/file.ext
file join dir/sub1 sub2\\sub3 file.ext → dir/sub1/sub2/sub3/file.ext ❶
file join dir /subdir file.ext         → /subdir/file.ext ❷
```

❶   Note \ replaced with the Tcl canonical separator /.
❷   Absolute path arguments result in previous arguments being ignored.

9.1.4. Converting paths to native form: `file nativename`

When passing a file path to an external program, for example the command shell on Windows, the path must be converted to the native form for that platform. The `file nativename` command can be used for the purpose.

```
file nativename PATH
```

The command takes whatever actions are necessary to convert the specified path to platform-specific syntax, for example replacing tilde, converting canonical path separator / to platform specific ones and so on. For example, on Windows,

```
file nativename /dir/subdir/file.ext → \dir\subdir\file.ext
file nativename ~/file.ext           → C:\Users\ashok\Documents\file.ext
```

The `file join` command can be used as a complementary command to convert in the other direction.

```
% set native_path [file nativename c:/dir/file.ext]
→ c:\dir\file.ext
% file join $native_path
→ c:/dir/file.ext
```

9.2. File system operations

A number of `file` subcommands are related to operations related to the file system, such as retrieving metainformation about a file, directory creation, manipulation of file attributes etc. We describe these facilities here.

Discussion of Tcl support for Virtual File Systems (VFS) is postponed to Chapter 19 and not included here.

9.2.1. File system information: `file volumes|system|separator`

Tcl allows access to some rudimentary information about the file systems and volumes present.

The `file volumes` command returns the list of volumes mounted on the system.

```
file volumes
```

On Windows, the command returns the list of drives, remote shares and mounted VFS volumes. On Unix, the returned list consists of the single entry / and VFS volumes.

```
file volumes → C:/ D:/
```

The type of a file system can be ascertained with the `file system` command.

```
file system PATH
```

The argument *PATH* can be any path on the filesystem of interest. The command returns a list containing one or two elements, the first element indicating the file system and the second element, if present, being the specific type. For example,

```
file system c:/ → native NTFS
```

shows that file system for C: is `native`, meaning the file system of the underlying platform, and of type NTFS. For a virtual file system, the return value would be `tclvfs` if it was implemented with the `tclvfs` package.

One final piece of information provided is the path separator used by the file system. This is returned by the `file separator` command.

```
file separator ?PATH?
```

Without any arguments, the command returns the separator used by the native file system for the platform. If an argument is supplied, the command returns the separator used by the file system containing the specified path.

```
file separator            → \
file separator C:/Windows → \
```

9.2.2. File accessibility: `file exists|readable|writable|executable|owned`

Another set of `file` subcommands pertain to determining whether a file can be accessed in a specific mode like reading or writing. All these commands take a file path and return 1 if the file exists **and** is accessible for the specified mode and 0 otherwise.

The simplest check is for the existence of a file using `file exists`.

```
file exists c:/windows → 1
```

The `file readable`, `file writable` and `file executable` commands indicate whether the file can be read, written to or executed by the current real (not effective) user id.

```
file readable c:/windows/system32/cmd.exe   → 1
file writable c:/windows/system32/cmd.exe   → 0
file executable c:/windows/system32/cmd.exe → 1
file writable nosuchfile                    → 0
```

The commands also apply to directories where they indicate whether the directory contents can be listed, whether files can be created in the directory and whether it can be traversed.

```
file readable c:/windows   → 1
```

```
file writable c:/windows   → 0
file executable c:/windows → 1
```

Finally, the `file` owned command indicates if the specified file is owned by the current user id.

```
file owned [info nameofexecutable] → 1
```

> ⚠️ There are several caveats that apply to the use of all these access check commands. A return value of 1 from `file readable` does not always allow the file to be actually read for several reasons. For example,
>
> - The file may be opened for exclusive access by some other process
> - Local permissions permit reading but the file resides on a remote system which does not
> - File system permissions allow access but the Windows integrity levels do not.
>
> Thus, in the author's opinion it is better to just attempt the desired operation, like opening the file for read access, instead of using these commands.

9.2.3. File types: `file isdirectory|isfile|type`

The `file` type command returns the type of a file. The command returns one of `file`, `directory`, `characterSpecial`, `blockSpecial`, `fifo`, `link`, or `socket`.

```
file type c:/windows/system32/cmd.exe → file
file type c:/windows/system32         → directory
file type CON                         → characterSpecial ❶
```

❶ The special Windows built-in console interface

For the common cases where we need to know if a file is a regular file or a directory, the `file isfile` and `file isdirectory` commands offer a convenient alternative. These commands return 1 if the path exists **and** is of the corresponding type and 0 otherwise.

```
file isfile nosuchfile                 → 0 ❶
file isfile c:/windows/system32/cmd.exe → 1
file isfile $env(WINDIR)               → 0
file isdirectory $env(WINDIR)          → 1
```

❶ Note the commands return 0 for non-existent files as opposed to raising an error.

9.2.3.1. File content type: `fileutil::fileType`

The `file` type command returns the type of a file based on the file system or operating system perspective. It is often useful to know the type of the **content** stored in a file. Though Tcl itself does not provide a command for this in the core, the `fileutil` module of Tcllib[2] includes the `fileType` command that provides this functionality. The command is a variation of the Unix `file` program and guesses the type of content in a file, such as text, GIF image etc.

```
% package require fileutil
→ 1.15
% fileutil::fileType c:/windows/system32/cmd.exe
→ binary executable pe
```

---

[2] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

```
% fileutil::fileType images/icons/tip.png
→ binary graphic png
```

9.2.4. File properties

The file system stores several properties and attributes of files and several file subcommands return, and in some cases set, the values of these properties.

9.2.4.1. File size: file size

The file size command returns the size of the specified file in bytes.

```
file size [info nameofexecutable] → 65536
```

In case of links, the command returns the size of the link target, not the link itself.

9.2.4.2. File timestamps: file atime|mtime

For files on file systems that keep track of access and modification times for a file, this information can be retrieved and set with the file atime and file mtime commands respectively.

```
file atime PATH ?TIMESTAMP?
file mtime PATH ?TIMESTAMP?
```

If the TIMESTAMP argument is not specified, the commands return the access or modification time, respectively, as the number of seconds since the epoch January 1, 1970 (see Section 8.1).

```
clock format [file atime [info nameofexecutable]] → Tue Jul 04 11:23:42 IST 2017
clock format [file mtime [info nameofexecutable]] → Thu Jul 28 21:22:28 IST 2016
```

If TIMESTAMP is specified, the corresponding timestamp is set to this value which must be specified as seconds since the epoch. This value is also returned from the command.

```
clock format [file atime [info nameofexecutable] [clock seconds]] → Tue Jul 04 11:45:43 IST 2017 ❶
```

❶   Similar to the Unix touch utility

If the specified PATH is a link, the commands operate on the link target, not the link itself.

> Not all file systems maintain access and/or modification times or permit them to be set. In such cases the commands will raise an error.

9.2.4.3. File information: file stat|lstat

The commands file stat and file lstat are a direct interface to the stat and lstat system calls.

```
file stat PATH VAR
file lstat PATH VAR
```

The two calls only differ when PATH refers to a symbolic link. In that case, file stat returns information about the file that is the target of the link whereas file lstat returns information about the link itself.

Both commands store the result of the system call in an array of name *VAR* in the caller's context. The elements of the array are shown in Table 9.1.

Table 9.1. File stat array elements

Element	Description
`atime`	The last access time of the file in seconds since January 1, 1970.
`ctime`	The creation time of the file in seconds since January 1, 1970.
`dev`	The device id of the device on which the file resides.
`gid`	Group id of the file owner.
`ino`	The inode number of the file.
`mode`	The mode bits from the directory entry for the file.
`mtime`	The last modification time of the file in seconds since the epoch.
`nlink`	The number of hard links to the file.
`size`	The number of bytes stored in the file.
`type`	The type of the file.
`uid`	User id of the file owner.

```
% file stat [info nameofexecutable] stat
% parray stat
→ stat(atime) = 1499148943
  stat(ctime) = 1470287503
  stat(dev)   = 2
  stat(gid)   = 0
  stat(ino)   = 16605
  stat(mode)  = 33279
  stat(mtime) = 1469721148
  stat(nlink) = 1
  stat(size)  = 65536
  stat(type)  = file
  stat(uid)   = 0
```

> These commands are very much reflective of Unix file systems and many elements do not make sense for all platforms. Their use should therefore be avoided in portable code.

9.2.4.4. File attributes: `file attributes`

File systems may store attributes associated with a file. For example, Windows stores a short 8.3 version of a file name along with its real name. The `file attributes` command provides a means of retrieving, and in some cases storing, these file system specific attributes.

```
file attributes PATH
file attributes PATH ATTRIBUTE
file attributes PATH ?ATTRIBUTE VALUE ...?
```

The *PATH* argument specifies the file whose attributes are to be accessed. If it refers to a link, the attributes are those of the link's target, not the link itself.

The first form returns all attributes for the specified file. The second form returns the value of the specified attribute. The third form sets the values of one or more attributes. The permitted values of *ATTRIBUTE* are platform-dependent and we discuss them for the common platforms below.

Windows file attributes

The possible values of file attributes for Windows systems are shown in Table 9.2.

Table 9.2. Windows file attributes

Attribute	Description
-archive	Retrieves or sets the value of the *archive* file attribute.
-hidden	Retrieves or sets the value of the *hidden* file attribute.
-longname	Returns the long name for the file path. Each component of the path is converted to its long name. This attribute cannot be set.
-readonly	Retrieves or sets the value of the *readonly* file attribute.
-shortname	Returns the 8.3 format short name for the file path. Each component of the path is converted to its short name. This attribute cannot be set.
-system	Retrieves or sets the value of the *system* file attribute.

Some sample output from the command:

```
% array set attrs [file attributes c:/windows/system32/cmd.exe]
% parray attrs
→ attrs(-archive)   = 1
  attrs(-hidden)    = 0
  attrs(-longname)  = C:/Windows/System32/cmd.exe
  attrs(-readonly)  = 0
  attrs(-shortname) = C:/Windows/System32/cmd.exe
  attrs(-system)    = 0
```

The following shows the equivalence between the short name and the long name for a file.

```
set long_path "c:/temp/A long name.long extension"     → c:/temp/A long name.long extension
close [open $long_path w]                               → (empty)
file exists $long_path                                  → 1
set short_path [file attributes $long_path -shortname] → C:/temp/ALONGN~1.LON
file attributes $short_path -longname                   → C:/temp/A long name.long extension
file normalize $short_path                              → C:/temp/A long name.long extension ❶
file exists $short_path                                 → 1
file delete $short_path                                 → (empty)
file exists $long_path                                  → 0
```

❶ Remember that in addition to converting paths to absolute paths, file normalize also maps paths to their long name format.

Retrieving the short name of a file is often useful when executing external programs with exec. Since short names never contain spaces, it obviates the need for escaping space characters in any file name passed to the external program.

Unix file attributes

On Unix and Linux systems, file attributes supports the attributes shown in Table 9.3.

Table 9.3. Unix file attributes

Attribute	Description
-group	The group name for the file. When being set, either the group name or id can be passed. The return value is always the group name.
-owner	Name of the user owning the file. When being set, either the name or id can be passed. The return value is always the name.
-permissions	The octal code as accepted by the chmode system call. When being set, the command will accept a symbolic form as well. For example, u+rw,go-rwx or rwxr-r-- . See the reference documentation for details.
-readonly	The readonly attribute for a file on Unix systems that support the uchg flag for the chflags system call.

OS X file attributes

On Mac OS X systems, file attributes supports the attributes shown in Table 9.4.

Table 9.4. Mac OS X file attributes

Attribute	Description
-creator	The Finder creator type of the file.
-hidden	Retrieves or sets the value of the *hidden* file attribute.
-readonly	Retrieves or sets the value of the *readonly* file attribute.
-rsrclength	Length of the resource fork of the file. If setting this value, only 0 is accepted and results in the resource fork being stripped from the file.

9.2.5. Creating directories: `file mkdir`

The file mkdir command creates one or more directories.

```
file mkdir ?DIR ...?
```

For each argument specified, the command will create a directory with that path including any intermediate directories if required. If a path already exists, no action is taken if it is a directory and an error raised otherwise. Note that the arguments are processed in order and in case of errors, processing of further arguments is aborted while any previous directories that have already been created are not removed.

```
% file exists /tmp/dirA
→ 0
% file mkdir /tmp/dirA/dirB ❶
% file exists /tmp/dirA/dirB
→ 1
```

❶ Intermediate directory will also be created.

9.2.6. Removing files and directories: `file delete`

The file delete command deletes files and directories.

```
file delete ?-force? ?--? ?PATH ...?
```

Each *PATH* argument may refer to a file or a directory. For arguments that are symbolic links, the link itself is removed and not its target. If the argument specifies a file or directory that does not exist, it is ignored without the command raising an error.

The -force option affects two failure modes. Normally if the path corresponds to a non-empty directory or is the working directory of the current process, the command will raise an error. If -force is specified, the command will delete non-empty directories along with their content and also change the working directory, if required, so as to allow the directory deletion to proceed.

```
cd /tmp/dirA                    → (empty)
file delete /tmp/dirA           ∅ error deleting "/tmp/dirA": permission denied ❶
pwd                             → C:/tmp/dirA
file exists /tmp/dirA/dirB      → 1
file delete -force /tmp/dirA    → (empty)
pwd                             → C:/tmp ❷
file exists /tmp/dirA/dirB      → 0
```

❶   Will fail for two reasons - current directory and not empty
❷   Notice current directory changed

The optional -- argument indicates the end of options causing all remaining arguments to be treated as paths. In particular, if -force follows the --, it will be treated as a path argument and not as an option.

9.2.7. Copying and renaming: `file copy|rename`

The file copy and file rename commands are similar to each other in their behaviour so we describe them together. Both commands conceptually (not necessarily how they are implemented) make a copy of existing files or directories but file rename also deletes the original source after making the copy.

```
file copy ?-force? ?--? FROMPATH ?FROMPATH ...? TOPATH
file rename ?-force? ?--? FROMPATH ?FROMPATH ...? TOPATH
```

The optional -- sequence indicates the end of options in cases where the first *FROMPATH* argument might begin with a - causing it to be misinterpreted as an option.

The behaviour of these commands is slightly involved due to different variations depending on whether files or directories are being copied, whether the destination path *TOPATH* already exists, the number of arguments supplied and so on.

In this description, "copying" a directory also involves recursively copying all files and subdirectories contained within it.

If exactly one *FROMPATH* argument is specified, it may be either a file or a directory. The file copy command behaves as follows:

- If *TOPATH* does not exist, a copy of *FROMPATH*, whether a file or a directory, is stored as *TOPATH*.
- If *TOPATH* exists and is a **directory**, a copy of *FROMPATH*, again irrespective of whether it is a file or directory, is made and placed **under** *TOPATH*.
- If *TOPATH* exists and is **not a directory**, it is overwritten with a copy of *FROMPATH* if the latter is also not a directory **and** the -force option is specified. Otherwise (if *FROMPATH* is a directory or -force is not specified) an error is raised.

When the source file is a symbolic link within the same file system as the destination, the link itself is copied and not the link target.

The following examples illustrate the above scenarios. We use the glob command to list the contents of directories.

```
% file copy /temp/fromDir /temp/newDir ❶
% glob /temp/newDir/*
→ C:/temp/newDir/fileA.txt C:/temp/newDir/subDir
% file copy /temp/fromDir /temp/toDir ❷
% glob /temp/toDir/*
→ C:/temp/toDir/fromDir
% file copy /temp/fromDir/fileA.txt /temp/newDir ❸
∅ error copying "/temp/fromDir/fileA.txt" to "/temp/newDir/fileA.txt": file already exists
% file copy -force -- /temp/fromDir/fileA.txt /temp/newDir ❹
% file copy -force -- /temp/fromDir/subDir /temp/newDir ❺
∅ error copying "/temp/fromDir/subDir" to "/temp/newDir/subDir": file already exists
```

❶   Creates a recursive copy of fromDir **as** newDir.
❷   Creates a recursive copy of fromDir **under** toDir as toDir already exists.
❸   Fails because file exists.
❹   Option -force forces overwrite of existing file.
❺   Option -force will **not** overwrite an existing directory if it is not empty.

> Irrespective of whether the -force option is specified or not, the command will never overwrite a directory that is not empty (as illustrated above), or overwrite a file with a directory or vice versa.

The above description applies when there is exactly one FROMPATH argument. If more than one FROMPATH argument is specified, TOPATH must be an **existing directory** and file copy behaves as the second case above, placing a copy of each FROMPATH argument, whether a file or a directory, under the TOPATH directory.

```
% file copy /temp/fromDir/subDir/fileB.txt /temp/toDir /temp/newDir
% glob /temp/newDir/*
→ C:/temp/newDir/fileA.txt C:/temp/newDir/fileB.txt C:/temp/newDir/subDir
   ↳ C:/temp/newDir/toDir
% file copy /temp/fromDir/subDir/fileB.txt /temp/toDir /temp/newDir2 ❶
∅ error copying: target "/temp/newDir2" is not a directory
```

❶   Fails because /temp/newDir2 is not an existing directory.

The file rename command behaves similarly except that for all successful copies, file rename will delete the original file. As an implementation detail, when both the source and destination are on the same file system, this "copy and delete" operation may in fact be a single "move" or "rename" operation.

9.2.8. Enumerating files: glob

The glob command returns a list of all files matching any of one or more patterns.

```
glob ?OPTIONS? ?--? ?GLOBPAT ...?
```

The returned list contains matching files in an unspecified order. Applications should not assume file names are sorted or that names matching an earlier pattern will appear in the list before names matching later patterns.

The optional special sequence -- is used to indicate the end of options with remaining arguments being treated as patterns. This is useful when there may be confusion that a pattern is interpreted as an option, for example when it is passed in a variable and potentially begins with a -.

Each *GLOBPAT* is a pattern as described for the `string match` command with two additional features. A pair of braces containing strings separated by commas can be used to enclose alternatives in a pattern. Secondly, a pattern ending in a / will only match directories, not ordinary files. The list of special characters is shown in Table 9.5.

Table 9.5. Glob patterns

Character	Description
*	Matches any number (including zero) of characters except directory separators.
?	Matches one occurrence of any character except a directory separator.
[...]	Matches one occurrence of any character between the brackets except directory separators. A range of characters can also be specified. For example, [a-z] will match any lower-case letter.
{*STRING*?,...?}	Matches any of the *STRING* character sequences separated by commas within the braces except directory separators.
\	The backslash escapes the following character such as * or ? so that it is treated as an ordinary character. This allows you to write patterns that match literal glob-sensitive characters, which would otherwise be treated specially.

To illustrate the use of glob and its various options, let us first create a simple directory and file structure.

```
file mkdir /tmp/tcl-book
close [open /tmp/tcl-book/foo.txt w] ❶
close [open /tmp/tcl-book/fubar.doc w]
close [open /tmp/tcl-book/foohidden w] ❷
file attributes /tmp/tcl-book/foohidden -hidden 1
file mkdir /tmp/tcl-book/foodir
close [open /tmp/tcl-book/foodir/foo.txt w]
file mkdir /tmp/tcl-book/f{}dir     ❸
close [open /tmp/tcl-book/f{}dir/bar.txt w]
```

❶ This creates an empty file using open and close we look at later.
❷ A hidden file
❸ Directory name with special characters

The following examples illustrate basic glob usage.

```
% glob /t*/*book/f* ❶
→ C:/temp/book/files.adocgen C:/tmp/tcl-book/foo.txt C:/tmp/tcl-book/foodir C:/tmp/tcl-bo...
% glob /tmp/tcl-book/*.txt /tmp/tcl-book/foodir/* ❷
→ C:/tmp/tcl-book/foo.txt C:/tmp/tcl-book/foodir/foo.txt
% glob /tmp/tcl-book/f*/ ❸
→ C:/tmp/tcl-book/foodir/ C:/tmp/tcl-book/f{}dir/
% glob /tmp/tcl-book/f{oodi,uba}r ❹
→ C:/tmp/tcl-book/foodir
```

❶ Wild cards can appear in any path component
❷ Multiple patterns
❸ Trailing / will only match directories and returned values will also have / appended
❹ Example of alternation

If the list of matching files is empty, glob will raise an error by default. You can pass the -nocomplain option to have it return an empty list instead.

```
% glob nosuchfil*
0 no files matched glob pattern "nosuchfil*"
% glob -nocomplain nosuchfil*
```

9.2.8.1. Matching based on type: -type option

In addition to matching files based on file name patterns, glob can also further qualify matches based on the type of the file and access attributes through the -type option. The option value is a list of type and permissions specifiers. These fall into two categories where glob will return a file name if it matching **any** specifier from the first category and **all** specifiers from the second category.

The specifiers in the first category are shown in Table 9.6. Returned files will match one of these specifiers that are included in the value for -type.

Table 9.6. Glob category 1 type specifiers

Specifier	Description
b	Must be a block-special file.
c	Must be a character-special file.
d	Must be a directory.
f	Must be an ordinary file.
l	Must be a symbolic link.
p	Must be a named pipe.
s	Must be a socket.

The specifiers in the second category are shown in Table 9.7. Returned files will match **all** these specifiers that are included in the value for -type.

Table 9.7. Glob category 2 type specifiers

Specifier	Description
r	File has read permission.
w	File has write permission.
x	File has execute permission.
readonly	File has the read-only attribute.
hidden	File has the hidden attribute. By default, glob will not include hidden files. With this specifier, glob will include **only** hidden files.
XXXX	(Mac OS only) File has the 4-character type, e.g. TEXT
{macintosh type *XXXX*}	(Mac OS only) File has the 4-character type, e.g. TEXT
{macintosh creator *XXXX*}	(Mac OS only) File has the creator *XXXX*.

Some examples of filtering based on the types:

```
% glob -type {f d} /tmp/tcl-book/fo*    ❶
→ C:/tmp/tcl-book/foo.txt C:/tmp/tcl-book/foodir
% glob -type d /tmp/tcl-book/fo*    ❷
→ C:/tmp/tcl-book/foodir
% glob  -type {f hidden} /tmp/tcl-book/fo*    ❸
```

```
→ C:/tmp/tcl-book/foohidden
```

❶   Lists both files (f) and directories (`d).
❷   Lists only directories.
❸   Lists only ordinary files that are hidden.

9.2.8.2. Changing glob locations: `-directory`, `-path`

By default, the `glob` command uses the current directory as the starting point for file matching. For example `glob *` will return the files in the current directory.

The `-directory` and `-path` options allow this "starting location" from where the command looks for files to be changed without changing the current working directory.

```
% pwd
→ C:/temp/book
% glob /tmp/tcl-book/*
→ C:/tmp/tcl-book/foo.txt C:/tmp/tcl-book/foodir C:/tmp/tcl-book/fubar.doc C:/tmp/tcl-boo...
% glob -directory /tmp/tcl-book -- *
→ /tmp/tcl-book/foo.txt /tmp/tcl-book/foodir /tmp/tcl-book/fubar.doc /tmp/tcl-book/f{}dir
```

These options are useful when the directory component of the path contains characters that treated as special characters by `glob`. This is illustrated by the following example.

```
% glob /tmp/tcl-book/f{}dir/*
Ø no files matched glob pattern "/tmp/tcl-book/f{}dir/*"
% glob -dir /tmp/tcl-book/f{}dir *
→ /tmp/tcl-book/f{}dir/bar.txt
% glob -dir /tmp/tcl-book *.txt *.doc ❶
→ /tmp/tcl-book/foo.txt /tmp/tcl-book/fubar.doc
```

❶   If multiple patterns are specified, the directory applies to all.

Our funnily named directory `f{}dir` will get interpreted as `fdir` when passed as the `glob` pattern as there are no elements listed between the braces (see Table 9.5). When passed via the `-directory` option (shortened to `-dir` in the example), the braces are no longer interpreted as glob patterns.

The `-path` option has a similar effect except that whereas `-directory` specifies a directory, `-path` specifies **any** prefix. The difference is illustrated by the following:

```
% glob -directory /tmp/tcl-book/f{}dir/b *
Ø no files matched glob pattern "*"
% glob -path /tmp/tcl-book/f{}dir/b *
→ /tmp/tcl-book/f{}dir/bar.txt
```

In the first case, `glob` looks for a *directory* called `/tmp/tcl-book/f{}dir/b` which is not found. In the second case, `glob` uses the passed option value as a *prefix*.

9.2.8.3. Stripping path names: `-tails`

The above examples returned full paths of the listed commands. In many cases, only the name of the file is desired and not the full path. The `-tails` option provides an easier alternative to iterating over the returned list invoking the `file tail` command for each file. Note that the `-tails` option requires either `-directory` or `-path` to also be specified. Here is an example showing the effect of the option.

```
% glob -dir C:/tmp/tcl-book *
→ C:/tmp/tcl-book/foo.txt C:/tmp/tcl-book/foodir C:/tmp/tcl-book/fubar.doc C:/tmp/tcl-boo...
% glob -dir C:/tmp/tcl-book -tails *
```

```
→ foo.txt foodir fubar.doc f{}dir
% glob -path C:/tmp/tcl-book -tails *
→ tcl-book
```

9.2.8.4. Combining path component patterns: `-join`

Sometimes the various path components to be used in the pattern match are supplied as separate arguments. We can use the `file join` command to combine these before passing to `glob`. Alternatively, we can use the `-join` option to `glob` which indicates that all patterns are to be combined as path components.

```
% glob -join /tmp tcl* f*
→ C:/tmp/tcl-book/foo.txt C:/tmp/tcl-book/foodir C:/tmp/tcl-book/fubar.doc C:/tmp/tcl-boo...
```

9.2.8.5. Recursive listing of files

The `glob` command does not recurse into directories. A command such as

```
glob /*/*/*
```

will return file names exactly three levels deep from the root.

Writing a recursive version using `glob` is trickier than you might think at first glance having to take into account links, circular references and so on. Fortunately, Tcllib[3] provides two commands, `find` and `findByPattern` in its `fileutil` package which do the needful.

```
% package require fileutil
→ 1.15
% join [fileutil::findByPattern c:/tmp/tcl-book *.txt] \n
→ c:/tmp/tcl-book/foo.txt
  c:/tmp/tcl-book/foodir/foo.txt
  c:/tmp/tcl-book/f{}dir/bar.txt
```

The commands have several options, including matching based on regular expression instead of glob patterns, that make them sufficient for most purposes.

9.2.8.6. Special considerations for `glob`

When using `glob` there are a few considerations to be taken into account because of platform differences and some debatable quirks in the command behaviour. These are listed in this section. **It is important to be aware of these to avoid unexpected results.**

9.2.8.6.1. Case sensitivity

The case-sensitivity of `glob` matching depends on the underlying file system. On Windows for example, the pattern `foo*` will match file FOOBAR as well whereas on Unix it will not.

9.2.8.6.2. Short names on Windows

For file names that do not fit the 8.3 filename format, Windows creates a corresponding 8.3 format short name. The `glob` command does *not* pay any heed to these short names when matching special characters. It will however match if the exact file name is specified. For example,

```
% set long_path "/tmp/tcl-book/a long directory name"
→ /tmp/tcl-book/a long directory name
% file mkdir $long_path
```

---

[3] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

```
% set short_name [file attributes $long_path -shortname]
→ /tmp/tcl-book/ALONGD~1
% glob /tmp/tcl-book/ALONG* ❶
∅ no files matched glob pattern "/tmp/tcl-book/ALONG*"
% glob $short_name ❷
→ C:/tmp/tcl-book/ALONGD~1
```

❶ Pattern * will not be matched against the short name version of file name.

❷ However, an exact match against the short name with no wildcard patterns will succeed.

One might consider this a quirk of the implementation.

9.2.8.6.3. Enumerating hidden files

You might expect the command glob * to return all files in the current directory. It does not. In particular, by default the glob return value does not include any "hidden" files where the term has a platform-dependent meaning.

On Windows platforms, hidden files are those that have the hidden file attribute set. On Unix, hidden files are those whose names begin with a period (.). On either platform, the -types hidden option must be specified. Furthermore, when this option is used, **only** hidden files will be included in the returned list.

Thus (for example) to get a list of all files in a directory, one must concat both the lists.

```
% concat [glob C:/tmp/tcl-book/*] [glob -types hidden C:/tmp/tcl-book/*]
→ {C:/tmp/tcl-book/a long directory name} C:/tmp/tcl-book/foo.txt C:/tmp/tcl-book/foodir ...
```

On Unix platforms, you can also use the .* pattern to return hidden files in which case you may also retrieve all files with the following single command.

```
glob * .*
```

> There is one additional complication with hidden files on Unix. Use of the -types hidden option or .* pattern also returns the special directory entries . and ... In most cases, you will want to filter these out.

9.2.8.6.4. Interaction with tilde expansion

The use of the glob command also has some tricky interaction with the tilde expansion done by other file commands. To illustrate, let use create a file called ~someuser in our test directory.

```
close [open /tmp/tcl-book/~someuser w] → (empty)
```

Now we loop through and print access times for all files in the directory. We find that when we encounter the file named ~someuser, an error is generated.

```
% cd /tmp/tcl-book
% foreach fn [glob *] {
    puts $fn:[file atime $fn]
}
∅ a long directory name:1499148943
  foo.txt:1499148943
  foodir:1499148943
  fubar.doc:1499148943
  f{}dir:1499148943
  user "someuser" doesn't exist
```

What happened? The issue is that `glob`, quite naturally, returns ~someuser as one of the elements in the file list. When passed to `file atime`, the file name gets interpreted as the home directory of user ~someuser which does not exist, and hence the error.

Our example is relatively innocuous. But consider if instead we wanted to clean up files in /tmp and coded it as follows:

```
cd /tmp
foreach fn [glob *] {
    file delete $fn
}
```

A malicious user placing a file called ~, or any other valid user name, would lead to disaster.

This example is simplistic but the issue is real. Of course, the issue is not really with the `glob` command itself but with Tcl's treatment of ~. See Section 9.1.1.3 for a workaround.

9.2.9. Links: `file link`, `file readlink`

Some file systems support *hard links* where a new directory entry is created for a file, in effect giving the file another alternative name by which it can be accessed. Modification through one name will also be reflected when the file is accessed through another name. However, deletion of a hard link only removes the corresponding directory entry. The file can still be accessed via its other directory entries (file names). Hard links are not distinguishable from the "original" name of the file and Tcl commands do not (and cannot) distinguish between the two either.

File systems may also support the concept of *soft links*, (also called *symbolic links*) where the directory entry does not point to the target file content. Rather the link content is actually a reference to another file, the link target. When passed an argument that is a symbolic link, some commands like `file delete`, operate on the link itself. Other commands, like `file size` and open, operate on the link target. Still others, like `file copy`, may work either way depending on the context. These specifics are discussed in the description of each command. Here we only discuss commands that operate specifically on links.

Unix and Unix-like platforms support both hard and soft links to files and directories and correspondingly Tcl supports both. Although newer Windows versions support both types of links, older versions only supported soft links to directories and hard links to files and Tcl on Windows support for links is likewise limited to the same.

The `file link` command has two forms.

```
file link LINK
file link ?-symbolic|-hard? LINK TARGET
```

The first form returns the target path referenced by the argument *LINK*. If *LINK* is not a path to a symbolic link, an error is generated.

The second form allows creation of a link with path *LINK* to the file or directory specified by *TARGET*.

If either `-symbolic` or `-hard` is specified, the created link is of soft (symbolic) or hard type respectively. Otherwise, the command chooses a link type that is appropriate for the platform and file systems.

When *TARGET* is a relative path, the behaviour of the command is platform-dependent. On Unix-like platforms, the relative path is referenced as-is and will be interpreted by the system as relative to *LINK* and not relative to the current working directory. On other platforms, *TARGET* is normalized to an absolute form and *LINK* is set up to point to this absolute path. *TARGET* is also subject to tilde expansion (see Section 9.1.1.3).

The `file readlink` command is identical to the first form of the `file link` command returning the same data.

```
file readlink LINK
```

Some simple examples illustrating difference between links to directories and files on Windows:

```
% file link /tmp/tcl-book/dirlink /tmp/tcl-book/foodir
→ /tmp/tcl-book/foodir
% file readlink /tmp/tcl-book/dirlink ❶
→ C:\tmp\tcl-book\foodir
% file link /tmp/tcl-book/dirlink ❷
→ C:\tmp\tcl-book\foodir
% file link /tmp/tcl-book/filelink /tmp/tcl-book/foo.txt
→ /tmp/tcl-book/foo.txt
% file readlink /tmp/tcl-book/filelink ❸
Ø could not read link "/tmp/tcl-book/filelink": not a directory
```

❶ Succeeds because on Windows directory links are soft links
❷ Same as above
❸ Fails because on Windows file links are hard links

The last command fails because because hard links cannot be read. Nevertheless we can still verify it is indeed a link by writing to it and reading back from the target path.

```
set fd [open /tmp/tcl-book/filelink w] → file3fb0430
puts $fd "Hee haw"                      → (empty)
close $fd                               → (empty)
set fd [open /tmp/tcl-book/foo.txt]     → file3762b80
read $fd                                → Hee haw
close $fd                               → (empty)
```

9.2.10. Temporary files: `file tempfile`, `fileutil::tempfile`, `fileutil::tempdir`

For programs needing temporary files, Tcl provides the `file tempfile` command.

```
file tempfile ?NAMEVAR? ?TEMPLATE?
```

The command creates a temporary file in a system-specific directory, and returns a read-write channel to it. If *NAMEVAR* is provided, the command will set the variable of that name to the full path to the temporary file. If *NAMEVAR* is **not** provided, Tcl will attempt to delete the file on channel closure. *TEMPLATE* is an optional path template from which the temporary file path is generated. This is system specific and not discussed here.

```
set fd [file tempfile temppath]     → file38887a0
puts $temppath                      → C:/Users/ashok/AppData/Local/Temp/TCL45924.TMP
puts $fd "This is a temporary file" → (empty)
close $fd                           → (empty)
set fd [open $temppath]             → file3f7b990
read $fd                            → This is a temporary file
close $fd                           → (empty)
file delete $temppath               → (empty)
```

An alternate set of commands for temporary files is provided by the `fileutil` package of the Tcllib [4] library. The `tempdir` command from the package returns the directory to be used for temporary files and also allows it to be set for this process. The `tempfile` command is similar to `file tempfile` but only returns the name of a unique temporary file without also opening a channel to it.

---

[4] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

```
package require fileutil         → 1.15
set tempfile [fileutil::tempfile] → C:/Users/ashok/AppData/Local/Temp/EXjmfb9sGv
file exists $tempfile            → 1
```

9.3. Channels and File I/O

Tcl's input and output facilities revolve around the *channel* abstraction. Channels provide a uniform interface to all forms of I/O, so that reading and writing data to files, network connections, pipes and even serial ports, can all be treated uniformly. In addition, the channels provide additional functionality related to automatic character encoding and end of line translation.

Tcl even provides the ability for applications and extensions to define their own channel types, and to "stack" data transforms, a capability we will discuss in detail in Section 17.3.

The general sequence for doing I/O from Tcl is

- Open a channel for reading and/or writing, with a command such as open or socket.
- Optionally configure the channel parameters such encoding, buffer sizes and so on.
- Read and/or write to the channel using commands such as gets, read, or puts.
- Close the channel when done with the close command.

With the exception of commands that open channels, all commands related to channels are implemented by the chan ensemble command with subcommands for the various operations. Many of these are also available as independent commands for historical reasons. For example, the chan read and read are equivalent commands.

In this chapter, we introduce the use of channels and describe only the basic forms of input and output to files using synchronous calls. More advanced capabilities including asynchronous I/O, network communication, pipes are described in later chapters.

9.3.1. Standard channels: stdin, stdout, stderr

When a process starts up, most operating systems create streams for reading and writing data. These are commonly known as *standard input*, from which data is read, *standard output* where data is written, and *standard error*, where error messages are written. Generally, these are connected to the terminal or console if the process is attached to one, or to a pipe to another process.

Correspondingly, Tcl creates three standard channels by default, named stdin, stdout and stderr respectively. Commands that read and write data operate on stdin and stdout respectively if a channel is not explicitly specified. Thus

```
puts foo
puts stdout foo
```

are equivalent, both writing to the standard output.

With the exception noted below for Windows, these channels can be used in the same exact fashion as channels that are explicitly opened by the application. If a standard channel is closed, the very next channel that is opened is assigned to the standard channel as illustrated below.

```
% chan names ❶
→ stdout stdin stderr
% close stderr
% chan names
→ stdout stdin
% set ch [open error.log w]
→ stderr
% chan names
```

```
→ stderr stdout stdin
```

❶   Returns the list of currently open channels.

> Closing and reopening stdout in a similar fashion is one way you can change where commands like puts write by default. Note you need to make sure the new channel is opened for writing.

Standard channels on Windows

On Windows systems, applications may be console-mode programs that run in a command shell or have a graphical user interface (GUI). For console-mode programs, Windows creates standard channels in a manner similar to Unix systems. For GUI programs however, Windows itself does not create standard channels as all interaction is expected to be graphical. Thus in GUI programs like wish, Tcl creates "pseudo" channels via the console command in Tk that emulate the standard channels. However this emulation is only partial since more advanced features like asynchronous I/O will not work with these emulated channels.

9.3.2. Creating file channels: open

The open command returns a channel for doing I/O to a file, a process pipeline or a serial port. We only describe the first of these in this chapter.

```
open PATH ?ACCESS? ?PERMISSIONS?
```

The PATH argument specifies the path to the file whose content is to be read or written. If the ACCESS argument is not specified, or if it indicates the file is to be opened only for reading, PATH must reference an existing file.

The optional ACCESS argument specifies the desired access to the file and takes one of two forms. The first form is one of the string values shown in Table 9.8.

Table 9.8. Access mode for open - string form

Mode	Description
r, rb	The file is to be opened only for reading in text and binary modes respectively.
r+, rb+, r+b	The file must exist and is to be opened for both reading and writing with r+ indicating text mode with the other two being equivalent and indicating binary mode.
w, wb	The file is to be opened only for writing in text and binary mode respectively. The file will be created if it does not exist and truncated if it does.
w+, wb+, w+b	The file is to be opened for both reading and writing. The file will be created if it does not exist and truncated if it does. The value w+ specifies text mode and the others binary.
a, ab	Similar to w, wb except that all writes to the file are appended to the content irrespective of the current file pointer.
a+, ab+, a+b	Similar to w+, wb+, w+b except that all writes to the file are appended to the content irrespective of the current file pointer.

The following examples illustrate the use of this string form of access mode specification. Note all channels are closed after use with the close or chan close commands.

Create a new file and write a line to it:

```
% set chan [open /tmp/tcl-book/newfile.txt w] ❶
→ file322def0
```

```
% puts $chan "Line one" ❷
% gets $chan ❸
0 channel "file322def0" wasn't opened for reading
% close $chan ❹
```

❶ Open for write
❷ Write to the returned channel
❸ Fails because not open for read
❹ Channels must be closed when done.

Note the above will truncate the file if it already existed.

Open an existing file for reading and writing:

```
% set chan [open /tmp/tcl-book/newfile.txt r+] ❶
→ file3784e70
% read $chan
→ Line one
% close $chan
% set chan [open /tmp/tcl-book/newfile.txt w+] ❷
→ file3140530
% read $chan                                    ❸
% puts $chan "Line one again"                   ❹
% close $chan
```

❶ Open for read and write **without** truncating
❷ Open for read and write. Will truncate file.
❸ Notice empty string returned as file truncated.
❹ Write to the file

Append to a file:

```
% set chan [open /tmp/tcl-book/newfile.txt a] ❶
→ file382dd80
% puts $chan "Line two"                         ❷
% close $chan
% set chan [open /tmp/tcl-book/newfile.txt] ❸
→ file385de10
% read $chan
→ Line one again
  Line two
% close $chan
```

❶ Open for append. Will **not** truncate file.
❷ Line will be written at the end of the file
❸ Open for read only

On Windows systems, an attempt to open a file with the w or w+ modes will fail if the file has the hidden or system attributes set. To get around this, you can either reset those attributes with the file attributes command before opening the file, or open the file using the r+ mode and then use chan truncate to truncate the file content.

The second form that the *ACCESS* argument can take is that of a list whose elements are flag values from Table 9.9 with at least one among RDONLY, WRONLY or RDWR being present.

Table 9.9. Access mode for open - list form

Mode	Description
RDONLY	Open the file only for reading.
WRONLY	Open the file only for writing.
RDWR	Open the file for reading and writing.
APPEND	All writes are appended to the end of the file. Note this must be specified **in addition** to WRONLY or RDWR.
BINARY	All I/O is to be done in binary mode. Text mode is used if this flag is not specified.
CREAT	The file is to be created if it does not exist. Without this flag an error is raised on attempts to open non-existent files.
NOCTTY	The opened file is not to be made the controlling terminal for the process. Only relevant if the *PATH* corresponds to a terminal device.
NONBLOCK	Prevents the process from blocking while opening the file. The exact semantics are system dependent. See the reference documentation for details.
TRUNC	Specifies that if the file exists, it is to be truncated.

The following examples illustrate the use of this second form for the *ACCESS* argument that correspond to the examples for the first form above.

Create a new file and write a line to it:

```
% set chan [open /tmp/tcl-book/newfile.txt {WRONLY CREAT}] ❶
→ file3753580
% puts $chan "Line one"
% gets $chan                                                ❷
0 channel "file3753580" wasn't opened for reading
% close $chan
```

❶ Open for write, creating the file if necessary.
❷ Fails because not open for read

Open an existing file for reading and writing:

```
% set chan [open /tmp/tcl-book/newfile.txt RDWR] ❶
→ file39ed950
% read $chan
→ Line one
% close $chan
% set chan [open /tmp/tcl-book/newfile.txt {RDWR TRUNC}] ❷
→ file3784e70
% read $chan
% puts $chan "Line one again"
% close $chan
```

❶ Open for read and write **without** truncating
❷ Open for read and write. Will truncate file.

Append to a file:

```
% set chan [open /tmp/tcl-book/newfile.txt {WRONLY APPEND}] ❶
→ file384e150
```

```
% puts $chan "Line two"
% close $chan
% set chan [open /tmp/tcl-book/newfile.txt RDONLY] ❷
→ file37afa20
% read $chan
→ Line one again
  Line two
% close $chan
```

❶ Open for append. Will **not** truncate file.

❷ Open for read only

The *PERMISSIONS* parameter to the open command is only used if the file did not previously exist and has to be created. It specifies the access permissions for the newly created file together with the process' file mode creation mask. By default, *PERMISSIONS* has the value (octal) 0666 permitting both read and write access for all users unless limited by the process' file mode creation mask.

9.3.3. Closing a channel: chan close, close

All channels should be closed with chan close, or the equivalent close, once no longer required.

```
chan close CHANNEL ?DIRECTION?
close CHANNEL ?DIRECTION?
```

If only a single argument is given to the command, the channel is closed for both input and output. Otherwise, *DIRECTION* must be read or write and the channel (presumed to be bidirectional) is only "half-closed" with no further operations permitted of the specified type (read or write). In the case of files, *DIRECTION* may not be specified but we will see examples of its use when we discuss process pipelines (see Section 16.4).

When a channel is closed for input, any input data not read by the application is discarded. When closed for output, all output data buffered internally by Tcl is written out to the file (or pipe, socket etc. as the case may be). If the channel is a blocking channel, the command only returns once the data has been written out and the operating system file descriptor or handle has been closed. For non-blocking channels, the command returns immediately and flushing of data and closing of handles happens in the background.

> ⚠️ Tcl 8.6 will not automatically flush any non-blocking channels that are open when the process exits without explicitly closing the channels. See the close reference documentation for methods to achieve this.

9.3.4. Channel configuration: chan configure, fconfigure

There are several configuration properties associated with a channel. All these properties can be retrieved or set with the chan configure, or equivalent fconfigure, commands.

```
chan configure CHANNEL
chan configure CHANNEL OPTION
chan configure CHANNEL OPTION VALUE ?OPTION VALUE?
fconfigure CHANNEL
fconfigure CHANNEL OPTION
fconfigure CHANNEL OPTION VALUE ?OPTION VALUE?
```

When called with only one argument, the commands return a dictionary containing the current values for the configuration options for the channel.

```
% chan configure stdout
```

```
→ -blocking 1 -buffering line -buffersize 4096 -encoding cp1252 -eofchar {} -translation ...
```

If two arguments are supplied, the second argument must be one for the configuration option names. The commands then return just the value of that configuration option.

```
chan configure stdout -buffersize → 4096
fconfigure stdin -encoding        → cp1252
```

In the final form, one or more option and value pairs may be specified. The corresponding configuration options for the channel are then set to the new values.

Note that in addition to configuration options that are common to all, channels may have additional configuration options that are specific to that channel type. For example, network sockets will have a -peername configuration option corresponding to remote endpoint of a connection.

We will describe the various common configuration options over the next few sections. Options specific to channel types will be discussed when we discuss those types.

9.3.5. Writing to channels: chan puts, puts

The commands chan puts, and the equivalent puts, write data to a channel.

```
chan puts ?-nonewline? ?CHANNEL? DATA
puts ?-nonewline? ?CHANNEL? DATA
```

The commands write DATA to the specified channel. If CHANNEL is not specified, the data is written to the standard output channel stdout. The commands will append an additional newline character to the output data unless the -nonewline option is specified.

```
% puts "This will go to the standard output"
→ This will go to the standard output
% set chan [open /tmp/tcl-book/myfile.txt w]
→ file384e150
% puts $chan "Line one"
% puts -nonewline $chan "This is "  ❶
% puts $chan "the second line"
% close $chan
% fileutil::cat /tmp/tcl-book/myfile.txt
→ Line one
  This is the second line
```

❶ Note the -nonewline option

If the -nonewline option is specified when writing to channels that are **line buffered**, such as standard output or standard error, you may need to do a flush on the channel for the output to show up on the device. See Section 9.3.5.1.

The data passed to puts or chan puts is not necessarily the exact data written to the file or output device for a number of reasons:

- The channel is configured to do "end of line translation" (see Section 9.3.9).
- The channel is not in binary mode (see Section 9.3.10).
- The channel has transforms applied (see Section 17.2).

We will explore all these possibilities as we go along.

9.3.5.1. Output buffering

Data written to a channel is not necessarily written out to the underlying file or device right away. By default, Tcl maintains an output buffer for each channel and the size of this buffer and flushing of data stored in it is controllable through various channel options.

9.3.5.1.1. Buffering mode: `chan configure`, `fconfigure -buffering`

The `-buffering` option to `chan configure`/`fconfigure` controls when data is written out from the channel to the file or other device. The various values of this option and their effect is shown in Table 9.10.

Table 9.10. Buffering policy option values

Value	Description
none	Buffering is disabled. Any `puts` invocation results in the data being "immediately" written out to the system.
line	Data is flushed from the buffer whenever a newline character is written to the channel.
full	Data is fully buffered and flushed when the buffer is full.

By default, all channels are configured for full buffering except for terminal-like devices which are configured to be line buffered. The `stdout` and `stderr` standard channels are initially set to `line` and `none` respectively.

```
chan configure stdout -buffering       → line
chan configure stdout -buffering none → (empty) ❶
```

❶   Reset standard output to flush on every write.

9.3.5.1.2. Buffer flushing: `chan flush`, `flush`

In addition to the automatic flushing of data from output buffers as described above, an application can also explicitly force a channel's buffer to be flushed with the `chan flush` or `flush` commands. These take the form

```
chan flush CHANNEL
flush CHANNEL
```

9.3.5.1.3. Controlling the buffer size: `chan configure`, `fconfigure -buffersize`

The size of the buffer maintained for a channel can be retrieved or set with the `-buffersize` option to the `chan configure` or `fconfigure` commands.

```
chan configure stdout -buffersize      → 4096 ❶
fconfigure stdout -buffersize 10000 → (empty) ❷
```

❶   Retrieves the current buffer size
❷   Set the buffer size to 10,000 bytes

This configuration setting applies to both the input and the output sides of the channel.

9.3.5.2. A wrapper for writing files: `fileutil::writeFile`

A common sequence of operations is to create a file, write its content in one shot and close it. Instead of doing explicit open, `puts`, `close` and possibly even `fconfigure` operations, you may find it more convenient to use the `writeFile` command from the `fileutil` package in Tcllib[5].

---

[5] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

```
::fileutil::writeFile ?OPTIONS? PATH DATA
```

The command accepts the -encoding, -translation and -eofchar fconfigure options and configures the channel accordingly before writing the file.

```
% fileutil::writeFile /tmp/tcl-book/myfile.txt "Line one\nLine two"
```

The command will create the file or overwrite its contents if it already exists.

9.3.6. Reading from channels

Tcl provides two ways to read data from a channel, one line at a time, or a specified number of characters.

9.3.6.1. Reading lines from a file: chan gets, gets

The chan gets and the equivalent gets commands retrieve a line at a time from a channel. Here we describe only the blocking mode operation postponing discussion of non-blocking mode to Section 17.1.

```
chan gets CHANNEL ?VARNAME?
gets CHANNEL ?VARNAME?
```

In the single argument form, the commands return a complete line from the specified channel not including the end-of-line character sequence.

```
set chan [open /tmp/tcl-book/myfile.txt] → file306a620
gets $chan                               → Line one
```

If a second argument is specified, it is the name of a variable in the caller's context. The commands store the line in this variable and return the number of characters in the line.

```
gets $chan line → 8
puts $line      → Line two
```

If the end of the file is reached before finding a end of line sequence, the remaining characters are returned as a complete line. If no end of line is found without reaching the end of the file, for example when reading from a network socket, the command will block (assuming a blocking mode channel) until additional data containing an end of line arrives on the channel or the channel is closed from the remote end. Note this situation does not arise when reading from files.

After the last line is read from the channel, subsequent calls will return an empty string if the *VARNAME* argument is not specified. Note this cannot be distinguished from an empty line and the eof command must be used to distinguish between the two cases. On the other hand, if *VARNAME* is specified, the two cases are immediately distinguished as the end of file condition will result in the command returning -1 versus 0 for an empty line.

```
gets $chan line → -1
```

The foreachline utility described in Section 9.3.6.5 provides a convenient means of executing a script for every line in a file.

9.3.6.2. Reading characters from a file: `chan read`, `read`

Unlike `chan gets`, the `chan read` and `read` commands return a specified number of characters from a channel without any regard for line endings. Again, the command behaviour differs between blocking and non-blocking mode and here we only describe the former.

In the first form of the command,

```
chan read ?-nonewline? CHANNEL
read ?-nonewline? CHANNEL
```

the commands return all data from the channel until the end of file is reached. If the `-nonewline` option is specified, the last character read is discarded if it is a newline character.

The second form only reads the specified number of characters.

```
chan read CHANNEL NUMCHARS
read CHANNEL NUMCHARS
```

In this case the command returns the specified number of characters from the channel unless the end of file is reached first in which all remaining characters are returned. If the specified number of characters is not available in the channel and end of file is not reached, the command will block (again assuming blocking mode is in effect). This situation cannot happen with file based channels.

> Because of various data transforms that can happen as part of I/O, the character count is not necessarily the same as the number of raw bytes read from the file or device.

Some simple examples of reading characters:

```
% set chan [open /tmp/tcl-book/myfile.txt]
→ file381c2c0
% read $chan 1 ❶
→ L
% read $chan ❷
→ ine one
  Line two
% read $chan ❸
% eof $chan
→ 1
```

❶   Read a single character
❷   Read all remainining data
❸   End of file reached (empty string returned)

9.3.6.3. Input buffering

Like the output side, the input side is also buffered. However, since there is no meaningful flush operation on input, only the `-buffersize` option affects input buffering.

9.3.6.4. A wrapper for reading files: `fileutil::cat`

Just as `writeFile` simplifies writing of data to a file in a single command, the `cat` command from the Tcllib [6] `fileutil` module does the same for the input side.

---

[6] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

```
::fileutil::cat ?OPTIONS? PATH ?OPTIONS? PATH …
```

The command takes one or more paths, each preceded by options, and returns a string formed from the concatenation of the contents of all specified files. Valid options include -encoding, -translation and -eofchar and have the same effect as when passed to chan configure. Note that options are cumulative and when specified only affect files listed later in the arguments.

```
% fileutil::writeFile /tmp/tcl-book/myfile2.txt "Line A\nLine B"
% fileutil::cat /tmp/tcl-book/myfile.txt /tmp/tcl-book/myfile2.txt
→ Line one
  Line twoLine A
  Line B
```

9.3.6.5. Iterating over file contents: `fileutil::foreachLine`

Another convenience command provided by the fileutil package is foreachLine which iterates over all lines in a file executing a script for each.

```
fileutil::foreachLine line /tmp/tcl-book/myfile.txt {
    puts [string toupper $line]
}
→ LINE ONE
  LINE TWO
```

The command takes care of all I/O operations and error handling such as closing the file in case of errors during execution of the script.

9.3.7. Detecting end of file: `chan eof, eof`

There are several situations where an application needs to explicitly check if a channel is at end of file. We mentioned one such earlier where there is ambiguity in one form of the chan gets command between an empty line and end of file. Other situations arise when working with non-blocking channels where there is need to distinguish between end of file and data not being available yet.

The chan eof and eof commands check for the end of file condition on a channel.

```
chan eof CHANNEL
eof CHANNEL
```

We saw an example in the previous section and we will see less trivial examples when we discuss non-blocking I/O in later chapters.

> The chan eof and eof command will return 1 on an end of file condition only *after* an attempt has been made to read beyond the last character. Thus it should be checked only when gets or read indicate a potential end of file condition.

9.3.7.1. The end of file character: `chan configure, fconfigure -eofchar`

Some systems use a special end-of-file (EOF) character, generally Ctrl-Z, to mark the end of data in a file. Channels can be configured to recognize this through the -eofchar option to chan configure or fconfigure. The value of this option is a list containing up to two elements, the first element being the EOF character for the input side of the channel and the second being the EOF character for the output side. If the list contains a single element, it is used for both the input and output side. If an element is an empty string, EOF character recognition is disabled.

If an EOF character is configured for input, the appearance of of that character in the input stream is treated as end of file. If configured for output, Tcl will output the character when the channel is closed.

The EOF character defaults to the empty string (and thus disabled) except for Windows **files** (not other channel types) where it defaults to Ctrl-Z on input and the empty string on output.

The following example illustrates the behaviour on input when E is specified as the EOF character.

```
fileutil::writeFile /tmp/tcl-book/eofchar.txt "abcdEfghi" → (empty)
set chan [open /tmp/tcl-book/eofchar.txt]               → file31d8a80
chan configure $chan -eofchar E                         → (empty)
read $chan                                              → abcd ❶
read $chan                                              → (empty)
eof $chan                                               → 1
```

❶ Reading all data from channel only returns characters before E

9.3.8. Channel encoding: `chan configure, fconfigure -encoding`

We saw in Section 4.14 that Tcl strings are conceptually sequences of Unicode characters which need to be converted to a sequence of physical bytes using a well defined encoding when storing to files or communicating with other programs.

To revisit the example there, consider the Portugese word Olá. If we wanted to write this to a file that was to be read by another program that expected UTF-8 encoding, we would have to first convert the string to UTF-8 encoding before writing to the file.

Remembering to do the encoding for every write to the file is inconvenient (for example, consider when the writes happen from different procedures) and error prone. Instead we can use the `-encoding` option to automatically do the conversion to UTF-8. As always, the current value of the configuration setting can be obtained by not specifying a value for the option.

```
chan configure stdout -encoding → cp1252
```

Specifying a value will set the encoding for the channel. The above example could be written as

```
set greeting "\u004f\u006c\u00e1"
set chan [open /tmp/tcl-book/portugese.txt w]
chan configure $chan -encoding utf-8
puts $chan $greeting
close $chan
```

Note we no longer have to explicitly code the encoding command every time we write to the channel.

Although the above example shows output to a channel, the encoding applies to input as well. Data read from the channel will be expected to be in UTF-8 encoding and will be converted to a Unicode sequence automatically without necessitating a encoding convertfrom call.

The value supplied for the `-encoding` option may be any of the encoding names returned by the `encoding names` command. In addition, the special value `binary` is also accepted to write out raw binary data. This is discussed in Section 9.3.10.

9.3.9. End of line translation: `chan configure, fconfigure -translation`

Internally, Tcl uses the linefeed (LF) character ASCII 10, \n as the newline character. This is also the convention followed on Unix platforms. On Windows however, newlines are indicated by the character sequence carriage

return (CR) ASCII 13, \r followed by LF. Tcl's channel implementation provides for the various conventions through the -translation option to the chan configure and fconfigure commands.

```
chan configure stdout -translation → crlf
chan configure stdin -translation  → auto
```

The option value must be a one or two element list consisting of values shown in Table 9.11. The first element of the list applies to the input side of a channel. If a second element is present, it applies to the output side. If the list has only one element, it applies to the output side as well.

Table 9.11. Option -translation values

Value	Description
auto	On the input side, a setting of auto causes any occurence of the LF by itself, CR by itself, or a CRLF pair to be converted to the LF character. On the output side, the translation depends on the platform and channel type. On all channel types on Windows platforms, **and sockets on all platforms**, newlines are output as CRLF pairs. In all other cases, a single LF is output.
cr	On input CR characters are treated as new lines and converted to Tcl's internal LF based newlines. On output, the reverse conversion is done.
lf	The external representation matches Tcl's representation and thus no conversions are performed for new lines.
binary	This sets the translation mode to lf **and** in addition modifies other channel options to support binary data I/O. This is described in the next section.

The following will set the output translation to the Windows CRLF format even on Unix systems. This will allow programs like Notepad to interpret line endings correctly if the file is moved to a Windows system.

```
set chan [open /tmp/tcl-book/lf.txt w]
fconfigure $chan -translation lf
puts $chan "Line one\nLine two"
close $chan
```

9.3.10. Binary I/O

Much of the Tcl's channel system assumes files contain text content and the automatic translation of line endings, encodings etc. are directed towards convenient and portable text I/O.

When reading or writing binary strings (see Section 4.13) however, we need to turn off these features discussed earlier so that the data is written out as-is without any modifications:

- Character encoding
- End of line translation
- End of file character interpretation

In the case of files, specifying either the b qualifier for the access mode or BINARY for the list form access mode will do the needful.

```
% set chan [open /tmp/tcl-book/myfile.txt r]  ❶
→ file385e590
% chan configure $chan
→ -blocking 1 -buffering full -buffersize 4096 -encoding cp1252 -eofchar ^Z -translation
  ↳ auto
% close $chan
% set chan [open /tmp/tcl-book/myfile.txt rb]  ❷
```

```
→ file3762100
% chan configure $chan
→ -blocking 1 -buffering full -buffersize 4096 -encoding binary -eofchar {} -translation lf
% close $chan
```

❶   Text mode
❷   Binary mode

Notice the difference in the value for the -eofchar, -translation and -encoding options.

Another alternative to setting up a channel for binary I/O is to user chan configure to set the -translation option to binary. This is useful when the channel creation command, for example socket, does not have a means to specify binary mode or on channels where applications may need to switch between text and binary modes, for example on an HTTP connection that embeds binary data.

Thus the above example may also be written as

```
% set chan [open /tmp/tcl-book/myfile.txt r]
→ file31409f0
% chan configure $chan -translation binary
% chan configure $chan
→ -blocking 1 -buffering full -buffersize 4096 -encoding binary -eofchar {} -translation lf
% close $chan
```

Note how setting the -translation option to binary actually sets its value to lf and also causes the -eofchar and -encoding options to change.

9.3.11. The file access pointer

Every channel has an associated file access pointer that tracks the current position in the file. Any subsequent reads and writes occur starting at this position. The pointer is then updated to the offset just after the read or write. This pointer can be read and set with the chan tell and chan seek commands.

9.3.11.1. Retrieving the current file position: chan tell, tell

The chan tell and tell commands return the value of this pointer.

```
chan tell CHANNEL
tell CHANNEL
```

For channels that do not support this operation, the command returns -1.

The following example shows how the file pointer is moved with each I/O operation.

```
set chan [open /tmp/tcl-book/myfile.txt]   → file3872660
chan tell $chan                            → 0
gets $chan                                 → Line one
chan tell $chan                            → 10
gets $chan                                 → Line two
chan tell $chan                            → 18
```

It is important to note that the return value from these commands is an offset in **bytes, not in characters**, from the beginning of the file. The difference arises from multi-byte encoding and end of line translations. To illustrate,

```
file size /tmp/tcl-book/portugese.txt        → 6
set chan [open /tmp/tcl-book/portugese.txt]  → file379d950
fconfigure $chan -encoding utf-8             → (empty)
string length [read $chan]                   → 4
```

```
chan tell $chan                        → 6
close $chan                            → (empty)
```

Note the difference between the number of characters read and the position of the file pointer (which equals the size of the file).

9.3.11.2. Setting the file access position: chan seek, seek

The chan seek and seek commands change the file pointer so that the next I/O operation begins at a different position from where the last one ended.

```
chan seek CHANNEL OFFSET ?ORIGIN?
seek CHANNEL OFFSET ?ORIGIN?
```

The *OFFSET* argument must be an integer, positive or negative. The file pointer will be moved by these many bytes from the position specified by *ORIGIN*. *ORIGIN* may be one of the values shown in Table 9.12.

Table 9.12. Origin values for seek

Origin	Description
start	*OFFSET* is with respect to the start of the file and so effectively an absolute offset into the file. This is the default if the *ORIGIN* argument is not specified.
current	*OFFSET* is with respect to the current file pointer position.
end	*OFFSET* is with respect to the end of the file and is usually a negative number in this case.

For channels that do not support the seek operation, an error is raised.

The following example illustrates the use of seek and tell.

```
set chan [open /tmp/tcl-book/myfile.txt] → file385e590
gets $chan                             → Line one
set pos [chan tell $chan]              → 10 ❶
gets $chan                             → Line two
chan seek $chan $pos                   → (empty) ❷
gets $chan                             → Line two
close $chan                            → (empty)
```

❶   Note position of second line
❷   Return to beginning of second line

The next example overwrites the last few characters from a file.

```
% set path /tmp/tcl-book/seek.txt
→ /tmp/tcl-book/seek.txt
% fileutil::writeFile $path "1234567890"
% set chan [open $path r+b]
→ file385e4d0
% chan seek $chan -5 end ❶
% puts -nonewline $chan abc
% chan seek $chan 0 end
% puts -nonewline $chan def
% close $chan
% fileutil::cat $path
→ 12345abc90def
```

❶   Note the negative offset

> ❗ When a channel is configured for binary I/O, you can use any integer values for the *OFFSET* argument since there is a one-to-one correspondence between the file position pointer and read/write counts. In text mode however, this correspondence does not hold and the offsets specified should be either a value returned by `tell` or 0. Otherwise, the results may not be what you expect.

9.3.12. Truncating files: `chan truncate`

The `chan truncate` command truncates the file or other data stream open in a channel to a specific number of bytes (**not characters**).

```
chan truncate CHANNEL ?LENGTH?
```

If the *LENGTH* argument is specified, the file or data stream length is set to that value. If *LENGTH* is not specified, it defaults to the current file pointer value for the channel.

9.3.13. Copying data between channels: `chan copy, fcopy`

Data may be copied between channels simply by doing `read` on the source channel and `puts` on the destination channel. A more efficient method is to use the `chan copy` or `fcopy` commands.

```
chan copy FROMCHAN TOCHAN ?-size SIZE? ?-command CALLBACK?
fcopy FROMCHAN TOCHAN ?-size SIZE? ?-command CALLBACK?
```

The advantage of `chan copy` over the `read` and `puts` method is primarily efficiency. It minimizes both CPU and memory usage by avoiding buffer copies.

If the `-size` option is not present, all data from the input channel *FROMCHAN* until end of file is copied to *TOCHAN*. Otherwise, only the number of bytes specified by the option are copied.

If the `-command` option is not specified, the commands will block until the copy is complete. If the option is specified, the commands will return immediately and the data copying will continue in the background via the event loop. On completion of the copy, the callback command is invoked. Note that the event loop (see Chapter 15) must be running for this to work.

The command respects the encoding and translation settings of each channel. The following will convert the UTF-8 encoded file to one using UCS-2.

```
set from [open /tmp/tcl-book/portugese.txt]
chan configure $from -encoding utf-8
set to [open /tmp/tcl-book/portugese.ucs w]
chan configure $to -encoding unicode
chan copy $from $to
close $from
close $to
```

9.3.14. Enumerating open channels: `chan names`

The list of channels that are currently open can be obtained with the `chan names` command.

```
chan names ?PATTERN?
```

The command returns the list of channels with names matching *PATTERN* in the form accepted by the `string match` command. *PATTERN* defaults to * resulting in all channel names being returned.

```
% chan names
```

```
→ stdin rc9 stdout stderr
% chan names stdo*
→ stdout
```

9.3.15. Tcllib fileutil module

The Tcllib[7] library has a package fileutil that includes a number of utilities dealing with files. We have already seen some of the commands which provide additional flexibility or variations on the built-in commands. Other commands in this module provide higher level commands often modeled on Unix programs.

For example, here is a poor man's grep text search utility.

```
% join [fileutil::grep e.*t [glob -dir /tmp/tcl-book *.txt]] \n
→ /tmp/tcl-book/lf.txt:2:Line two
  /tmp/tcl-book/myfile.txt:2:Line two
  /tmp/tcl-book/newfile.txt:2:Line two
```

The module contains many other file related commands for in-place editing of files, additional path manipulation, etc. but here we only mention two generally useful facilities.

The first, the traverse command returns an object that can be used for traversing a directory hierarchy in a very flexible fashion. The application provides a script to be matched against each file and only files for which the script returns a boolean true value are included in the traversal. Additionally, the returned object provides multiple ways to iterate through the results. Here is a simple example, where we want to find files less than 10 bytes in size.

```
package require fileutil::traverse
proc filematch {path} {
    if {[file isfile $path] && [file size $path] < 10} {
        return 1
    } else {
        return 0
    }
}
set walker [fileutil::traverse %AUTO% c:/tmp -filter filematch]
→ ::traverse1
```

This creates an object that we can then invoke methods on to retrieve the list of files.

```
% $walker files ❶
→ c:/tmp/tcl-book/portugese.txt c:/tmp/tcl-book/eofchar.txt c:/tmp/backup/portugese.txt c...
% $walker foreach path { puts [file nativename $path] }  ❷
→ c:\tmp\tcl-book\portugese.txt
  c:\tmp\tcl-book\eofchar.txt
  c:\tmp\backup\portugese.txt
  c:\tmp\backup\eofchar.txt
% $walker destroy ❸
```

❶   Returns the whole list
❷   Iterates one at a time
❸   Traverse object must be destroyed when done

The other general purpose facility in fileutil is the multi command which implements a domain specific language for specification of file operations. Since the language is large, we only provide a small example of its flavor.

---

[7] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

```
package require fileutil::multi
fileutil::multi copy \
    the *.txt \
    from /tmp/tcl-book \
    into /tmp/backup \
    but not lf.txt \
    the lf.txt \
    as linefeed.txt
```

The example should be self explanatory. The `multi` and underlying `multi::op` DSL is very powerful and versatile. Again, see the Tcllib[8] reference documentation for full details.

9.4. Chapter summary

In this chapter, we looked at the Tcl commands for working with the file system and basic facilities for reading and writing files. We introduced the channel abstraction and various modes of operation including binary I/O, end of line translation and character encodings.

In later chapters, we will look at more advanced features including non-blocking and asynchronous I/O, network communications, implementing reflected channels and virtual file systems.

---

[8] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

<div style="text-align: right">

10

</div>

Code Execution

Tcl is distinguished by the flexibility and versatility of its execution model that makes it amenable to be used in a wide variety of software architectures and patterns.

We will study the basics of code execution in this chapter including evaluation of scripts, loops, conditional statements, run-time code construction, the structure of the call stack and more. This will provide the background for the more sophisticated material in later chapters dealing with events, coroutines and threads.

10.1. Evaluating strings: `eval`

One of the major features of dynamic languages is the ability to execute scripts constructed "on-the-fly" in the course of a program's execution. This capability is useful in diverse situations, for example

- Applications where users can write macros to automate tasks.
- Parsers for data and text where the input is transformed into a script that produces the desired output
- Implementation of domain specific languages for which Tcl code is generated at runtime.

We start off by looking at the most basic of these commands, `eval`.

> In modern Tcl, use of `eval` is generally deprecated in favor of other facilities like argument expansion. See http://wiki.tcl.tk/1017 for a discussion. Here we start with `eval` because it is the basic mechanism for executing dynamically generated code and serves as an introduction to issues related to quoting and double substitutions.

The command accepts one or more arguments, concatenates these in the same manner as the `concat` command and then executes the result as a standard Tcl script.

```
eval ARG ?ARG ...?
```

The result of the command is the result of the script execution. In its simplest form

```
eval {puts foo} → foo
```

the command executes the script `puts  foo` and is effectively no different than had we just said

```
puts foo → foo
```

Let us look at an example that illustrates the difference. We will define a variable `bar` with a value `hello` and a second variable `foo` which references it.

```
set bar "hello" → hello
set foo {$bar}  → $bar ❶
```

❶   Note the braces cause $bar to be treated as a literal string.

Now compare the following commands.

```
puts $foo       → $bar
eval puts $foo → hello
```

By now you should have understood why the puts $foo command on the first line prints $bar — Tcl does not reparse the words comprising a command after any substitutions are made (see Section 3.2). The eval on the other hand prints hello. Let us look at this eval statement step by step:

- When it is parsed by Tcl, the $foo variable reference is replaced by its value $bar.
- The eval command receives two arguments, the strings puts and $bar.
- As per its defined behaviour, it first concatenates its arguments to form the string puts $bar.
- This string is then executed as a Tcl script. The Tcl parser now sees the command puts $bar and as usual breaks it up into words using the usual rules of substitution replacing $bar with hello.
- The command puts is then invoked with the argument hello and does its thing.

10.1.1. Double substitutions in eval

We see in the above example that it appears that the variable reference $foo undergoes double substitution in the command eval puts $foo, first to $bar and then to hello. Note this does not go against what we stated earlier about the Tcl parser not reparsing substituted values. Here it is the eval command that is invoking the Tcl parser a second time. Remember we said Tcl commands can do whatever they want with their arguments? Here eval chooses to treat its (concatenated) arguments as a Tcl program to be parsed and executed.

Consider if we had braced the arguments to eval in either of the following forms instead:

```
eval {puts $foo} → $bar
eval puts {$foo} → $bar
```

In both these cases, the braces prevent the **initial** round of substitutions. The eval command still does its concatenation and substitution, but because it is now passed $foo as its argument, and not the value of foo, a single round of substitution results.

It needs to be emphasized that eval executes a **script** and not a **command**. Thus both the following

```
% eval {puts foo ; puts bar}
→ foo
  bar
% eval "puts foo" ";" puts bar
→ foo
  bar
```

are parsed as a script with two commands and not as a single command puts with four arguments foo, ;, puts and bar.

Issues around double substitutions and quoting come up with several other commands and can be confusing so we will take up a couple of additional examples. We first define some variables used in the examples.

```
% set cmdA llength
→ llength
% set cmdB "string length"
→ string length
% set arg "foo bar"
→ foo bar
```

Now compare the results of the various commands below.

```
$cmdA $arg              → 2
eval {$cmdA $arg}       → 2 ❶
$cmdB $arg              Ø invalid command name "string length" ❷
eval {$cmdB $arg}       Ø invalid command name "string length" ❸


eval $cmdA $arg         Ø wrong # args: should be "llength list" ❹
eval $cmdB $arg         Ø wrong # args: should be "string length string" ❺

eval $cmdA {$arg}       → 2
eval $cmdB {$arg}       → 7
eval $cmdA [list $arg]  → 2
eval $cmdB [list $arg]  → 7
```

❶ Effectively same as above because enclosed in braces
❷ Fails because `string length` is parsed as a *single* word and there is no command of that name.
❸ Fails for the same reason as above.
❹ Fails because double substitution causes `foo bar` to be treated as two arguments `foo` and `bar`.
❺ Fails for the same reason

Make a note of the last two forms. The braces around the argument prevent the first round of variable substitution. The second round of substitution via the `eval` is still effected. The `list` command form on the other hand allows the first round of substitution but wrapping the argument in list form protects against substitution in the second round. In our example, the two are effectively the same because the `eval` command does not change the variable context and none of the arguments have side effects. However, as we will see in later sections, the difference is important for commands like `uplevel` that can execute in different variable contexts.

Prefer argument expansion to eval where possible

In earlier versions of Tcl, a common use for `eval` was to expand a list value into its constituent elements. We saw one example above:

```
 eval $cmdB {$arg} → 7
```

In modern versions of Tcl (8.5 and later) the recommended method is to use the argument expansion syntax instead.

```
 {*}$cmdB $arg → 7
```

See http://wiki.tcl.tk/1017 for the rationale.

10.2. Evaluating file content: `source`

We have already seen in Section 2.2.2.1.2 how a Tcl program stored in a file can be executed by passing the file name as a command line argument to the `tclsh` or `wish` applications. The `source` command is another means of evaluating the contents of a file as a Tcl script.

```
source ?-encoding ENCODING? PATH
```

The command reads the file identified by *PATH* and evaluates its content as a Tcl script is a manner similar to `eval`. If *PATH* is a relative path, it is treated as relative to the **current** working directory of the process, **not** relative to the file containing the `source` command.

The file content is expected to be in the encoding specified by the `-encoding` option. If the option is not specified, the content is assumed to be in the system encoding.

The presence of a Ctrl-Z character in the file content is treated as the end of the file by the `source` command. Any data beyond the Ctrl-Z character is ignored. This feature is sometimes used to store binary data beyond the end of the Tcl script. The script can use the `info script` command to identify its containing file, read it in and then locate the binary data by searching for the first Ctrl-Z character. This is often more convenient than having to distribute a separate file containing the data. We will see an example of this use in Section 13.5.5.

The result of the `source` command is the result of the last command executed in the read script. A `return` command within the script will cause the rest of the commands in the script to be skipped with the argument to `return` being returned as the result.

> It is perfectly legal to source a file multiple times. This is particularly useful during interactive development where you might edit the source code to fix bugs or add features and then re-source the file into the interpreter.

Most large applications and packages are structured as a single "main" script with supporting data and procedure definitions stored in other files. Running the application or loading a package involves executing this script which in turn uses `source` to pull in the other files.

It is often useful in these cases (among others) for a script to know its own path so that it can locate the other scripts it needs to source. The `info script` command provides this information.

```
info script ?SCRIPTPATH?
```

In the usual case, where the *SCRIPTPATH* argument is not specified, the command returns the full path of the innermost file being sourced. For example, if file `a.tcl` is being sourced and it in turn sources `b.tcl` which in turn sources `c.tcl`, the result of the command **while** `c.tcl` **is being sourced** will be the full path to `c.tcl`.

The command can be used in the main script in a fashion similar to the code fragment below.

```
namespace eval myapp {
    # Remember the directory we are located in.
    variable script_dir [file dirname [info script]]
}
# Using apply only to not pollute global namespace with temporary variables
apply {paths {
    foreach path $paths {
        source [file join $::myapp::script_dir $path]
    }
}} {a.tcl b.tcl}
```

If no file is being sourced when `info script` is called, the command returns an empty string.

> The following procedure to return the directory where the file containing the procedure is located will **not** work as you might expect.
>
> ```
> proc get_my_dir {} { return [file dirname [info script]] }
> ```
>
> If the procedure is called **after** the file has been sourced, `info script` returns the empty string which is not what you would want. You must make sure `info script` is actually executed at the time the file is being sourced and the result saved somewhere as shown in the earlier example.

When the command is supplied the *SCRIPTPATH* argument, further calls to the `info script` command will return *SCRIPTPATH* instead of the real file name **for the duration of the current sourcing**.

<div align="center">Dual mode scripts</div>

In many instances, a Tcl script may run as a main application or be loaded as a library module. For example, a script may run as a command line Web client when invoked from the command line or simply provide a library for retrieving Web pages when loaded as a package into an application. The `info script` command can be used to distinguish the two cases as shown in the following snippet.

```
if {[info exists ::argv0] &&
    [file dirname [file normalize [info script]/…]] eq [file dirname [file normalize $argv0/…
]]} {
    … Script file was passed as an argument on the command line …
    … Parse command line options and retrieve web pages …
} else {
    … Script was sourced as a library
}
```

The only points to be noted are the need to normalize before comparing the command line argument `argv0` which contains the path of the script invoked from the command line. This normalization takes care of relative and absolute path differences as well links. See Section 9.1.1.2 regarding the above normalization pattern.

10.3. Conditional execution

Tcl supports two commands that provide for execution of code only when certain conditions are met. The `if` command executes scripts based on arbitrary expressions, whereas `switch` is limited to pattern matching comparisons.

10.3.1. Evaluating scripts based on an expression: `if`

The `if` command has the form

```
if IFEXPR then BODY ?elseif ELSEIFEXPR then BODY …? ?else BODY?
```

The expression *IFEXPR* is evaluated in the same manner as the `expr` command. It is expected to yield a boolean otherwise an error exception will be raised. If the boolean has a true value, the associated script *BODY* is evaluated in the context of the caller. Otherwise, each *ELSEIFEXPR* expression is evaluated in turn in the same manner and the corresponding *BODY* evaluated if true. If none of the expressions evaluate to a boolean true value, the *BODY* script associated with the `else` clause is evaluated if present.

```
if {$i > 0} then {
    puts "$i is positive"
} elseif {$i < 0} {
    puts "$i is negative"
} else {
    puts "$i is zero"
}
```

The `elseif` and `else` clauses are optional and there may be multiple `elseif` clauses.

Moreover, the keywords then and else are also optional so the above could also have been written as

```
if {$i > 0} {
    puts "$i is positive"
} elseif {$i < 0} {
    puts "$i is negative"
} {
  puts "$i is zero"
}
```

In most Tcl scripts the then keyword is left out while else is explicitly specified.

The result of the if command is the result of the evaluated script or an empty string if no expressions yielded true and no else clause was present.

```
set x 2 ; set y 1                    → 1
set x [if {$x > $y} {set x} else {set y}] → 2
```

Coming from other languages, you may try to write your if statement as follows:

```
if {$x > $y}
{
    …do something…
}
else
{
    …do something else…
}
```

This will not work. Remember if is a command like any other; it is **not** a special keyword with special syntax. Its expressions and script bodies are just arguments as for any other command and have to be quoted appropriately. The syntactic rules for separating commands also apply. So in the above example, Tcl will see the first line as a complete command with two words and invoke if with a single argument, the braced expression. It is the if command that will then raise an error as it expects at least two arguments.

Of course, this syntactic requirement also applies to commands like while, for etc.

10.3.2. Evaluating scripts based on patterns: switch

The switch command offers a more convenient and readable alternative to the if command when the condition for executing a script is based on matching a value. It has two syntactic forms

```
switch ?OPTIONS? ?--? VALUE LIST
switch ?OPTIONS? ?--? VALUE PATTERN BODY ?PATTERN BODY …?
```

In the first form LIST is a list of PATTERN and BODY elements specified as a single argument. In the second form, each pair is separately specified.

The command compares the VALUE argument against each PATTERN in turn and evaluates the BODY argument corresponding to the first pattern that matches. If the **last** pattern is the string default, it matches all values and the corresponding BODY is executed if no previous pattern matched. If BODY is the - character, the BODY corresponding to the next argument is executed.

As always, the optional -- character sequence is used to separate options from the VALUE argument in case of any ambiguity arising from the latter beginning with a - character.

An example using the first form of the command:

```
switch $image_format {
    png { save_as_png $image }
    jpg -
    jpeg { save_as_jpeg $image }
    gif { save_as_gif $image }
    default {
        error "Unsupported image type $image_format"
    }
}
```

The same example using the second form of the command would read as

```
switch $image_format png {
    save_as_png $image
} jpg - jpeg {
    save_as_jpeg $image
} gif {
    save_as_gif $image
} default {
        error "Unsupported image type $image_format"
}
```

This second form is more convenient when the patterns being compared are not literals but rather passed through variables. For example, if the patterns being matched were stored in variables x, y and z, the first form would have to be written as

```
switch $val [list $x { ..take x action.. } $y {.. take y action ..} ...]
```

since use of braces would result in the switch command seeing the literal string $x instead of the contents of the variable of that name. On the other hand, the second form can be simply written as

```
switch $val $x { .. take x action .. } $y {.. take y action ..} ...
```

The return value from the switch command is the result of the executed script or the empty string if no pattern matched and no default handler was specified. Thus the command can be used in assignments and such.

```
proc print_weekday {when} {
    set day [switch $when {
        today     { clock seconds }
        tomorrow  { clock add [clock seconds] 1 day }
        yesterday { clock add [clock seconds] -1 day }
        default {
            error "Don't understand \"$when\"."
        }
    }]
    puts [clock format $day -format %A]
}
print_weekday tomorrow
→ Wednesday
```

Yes, we could have done that as a one-liner using the clock command

```
clock format [clock scan tomorrow] -format %A
→ Wednesday
```

The switch command takes several options that control the type of matching performed. These are shown in Table 10.1.

Table 10.1. Matching options for switch

Option	Description
-exact	Exact string matching. This is the default.
-glob	Treats *PATTERN* arguments as glob patterns.
-nocase	Modifies the matching to be case-insensitive. By default matching is case-sensitive.
-regexp	Treats *PATTERN* arguments as regular expressions.

Here is an example of using switch with glob patterns.

```
set url "https://www.example.com"
set port [switch -glob -nocase -- $url {
    http://*  { string cat 80 }
    https://* { string cat 443 }
    ftp://*   { string cat 21 }
    default   { error "Unknown URL type" }
}]
→ 443
```

Here we are using string cat as an identity function.

Note that the order of patterns is important when multiple patterns match the value since the script associated with the first match is evaluated.

In the case of regular expression matching, there are two additional options that may be specified:

- The -matchvar option takes an additional argument that is the name of the variable in which to store the matched substrings. The content of this variable will be a list whose first element is the entire substring of *VALUE* that matched the regular expression pattern. The remaining elements of the list contain the substrings matched by the capturing parenthesis (see Section 4.12.1.10.1) in the expression, if any.

- The -indexvar option is similar except that instead of a list of matched substrings, the variable will contain a list of sublist pairs containing the starting and ending indices of the substrings.

The example below illustrates the -matchvar option.

```
proc connect_url {url} {
    switch -regexp -nocase -matchvar connection -- $url {
        "http://([-_%:.[:alnum:]]*)" {
            puts "Connecting to [lindex $connection 1] on port 80"
        }
        "https://([-_%:.[:alnum:]]*)" {
            puts "Connecting to [lindex $connection 1] on port 443"
        }
        default { error "Unknown protocol" }
    }
}
connect_url http://www.example.com
→ Connecting to www.example.com on port 80
```

Remember that just as in the `regexp` command, the match succeeds if the pattern matches any **substring**, not necessarily the entire string.

```
% connect_url "Please connect to http://www.example.com"
→ Connecting to www.example.com on port 80
```

If the desired behaviour is to match the entire string, use the ^ and $ anchors.

10.4. Looping

Tcl has several commands for executing code in a loop. Two of these, `foreach` and `lmap` are specific to lists. Similarly, the `dict for` command is specific to dictionaries. They are discussed in the related chapters. Here we describe the general purpose looping commands `while` and `for`.

If you are unhappy with the variety of looping and control statements in Tcl, it is almost trivial to write your own constructs. See Section 11.7.

10.4.1. Looping: `while`

The `while` command executes a script as long as a given expression is true.

```
while EXPR BODY
```

The argument *EXPR* is evaluated as an expression. If the result is a boolean true value, the script argument *BODY* is executed. This process is repeated until *EXPR* evaluates to a false value. The command always return the empty string as its result.

An example of some sophisticated computation using `while`:

```
proc sum {n} {
    if {$n < 0} { error "$n is negative" }
    set sum 0
    while {$n > 0} {
        incr sum $n
        incr n -1
    }
    return $sum
}
sum 3
→ 6
```

The *EXPR* argument should almost always be enclosed in braces as above to protect it from the Tcl parser. Otherwise, the parser will substitute the variable values before passing them to the `while` command. The result may be an error or worse. For example, if the above loop loop were written in either of the following forms

```
while "$n > 0" {
    …
}
while $n {
    ….
}
```

> the first argument seen by the `while` command (when called as `sum 3`) would be `3 > 0` or just `3` respectively. Since these expressions always evaluate to boolean true values, the loop would run forever. Or at least until the system runs out of electrons.

The `break` (see Section 10.4.3.1) and `continue` (see Section 10.4.3.2) loop control commands may be used within *BODY* to terminate the loop or to skip iterations.

10.4.2. Looping: `for`

The other generic looping command is the `for` command.

```
for INIT EXPR NEXT BODY
```

The command starts off by executing the script *INIT*. It then evaluates *EXPR* as an expression. If the result is a boolean true value, the command executes the script *BODY*, followed by the script *NEXT*. It then repeats this sequence as long as the *EXPR* expression evaluates to true. The result of the command is always the empty string.

The `while` loop from the previous section could also be written with `for`:

```
proc sum n {
    for {set sum 0} {$n > 0} {incr n -1} {
        incr sum $n
    }
}
```

Note that *INIT* and *NEXT* are also scripts like *BODY*, not necessarily single commands, and can be script blocks spread over multiple lines.

As always, the `break` and `continue` commands can be used to control the loop execution. In the case of `continue`, the commands in *BODY* after the `continue` will be skipped but the *NEXT* script is still executed.

10.4.3. Loop control

The commands `break` and `continue` can be used to prematurely terminate or skip iterations when looping. They can be used within the script bodies of any Tcl looping or iteration commands like `while`, `for`, `foreach`, `lmap` and `dict for`.

Both `break` and `continue` are a special case of a more general Tcl exception mechanism that we will discuss in Section 11.2.1 and may even be used outside looping constructs in some specific situations.

10.4.3.1. Terminating loops: `break`

The `break` command is used for prematurely terminating a loop.

```
break
```

Here we copy files to a floppy drive until there is insufficient space. Yes, I'm dating myself and no, not the smartest algorithm, does not consider disk allocation quanta and so on.

```
foreach file [glob -nocomplain *] {
    set size [file size $file]
    if {$size > $floppy_size} {
        break
    }
    file copy $file $floppy_drive
    set floppy_size [expr {$floppy_size - $size}]
}
```

The break command is sometimes used in contexts outside of loops as well. Examples of this use are given in Section 11.2.1.

10.4.3.2. Skipping loops: continue

The continue command is used for skipping an iteration, or a part, of a loop.

```
continue
```

Let us rewrite our previous example to be slightly smarter.

```
foreach file [glob -nocomplain *] {
    set size [file size $file]
    if {$size > $floppy_size} {
        continue
    }
    file copy $file $floppy_drive
    set floppy_size [expr {$floppy_size - $size}]
}
```

Instead of breaking out of the loop, we now skip the file and move on to trying the next one.

10.5. Frames and the call stack

As is true for practically all programming languages, at each stage of a computation, Tcl has to keep track of the execution context to be used for resolving names of commands and variables, both local and non-local. Where Tcl differs from most languages is that it allows programmatic control of the different contexts in which code is executed, the utility of which will become apparent later. Going over the management of these contexts will help in understanding some of the related commands.

10.5.1. The call stack

Consider the following program

```
namespace eval areas {
    variable pi 3.142
    proc circle {radius} {
        variable pi
        set area [expr {$pi*$radius*$radius}]
        return $area
    }
}
areas::circle 2
→ 12.568
```

When Tcl begin execution, it does so in the *global* context where all variables and commands resolve to the global namespace (see Chapter 12) unless they are explicitly qualified with a namespace. Being outside a procedure context, there are no local variables. This execution context is stored in a *call frame* as shown in Figure 10.1.

Figure 10.1. Initial call frame

When the areas::circle command in invoked, the areas relative namespace name is resolved in the current (global) context resulting in the ::areas::circle procedure being called. A procedure call has (potentially) local variables as well as (again, potentially) a different namespace context. Thus a new call frame reflecting these is added on every procedure call. This collection of frames is known as the *call stack* and each frame is at a *level* based on its position in the stack.

While areas::circle is executing, the call stack then looks as shown in Figure 10.2.

Figure 10.2. Level 1 call frame

The local variables include radius, passed in as an argument, and a procedure-local variable area. Any references to these variable names will be resolved from this list of locals. Similarly, the namespace context is now areas and correspondingly the variable command results in a local variable called pi that is linked to the variable of the same name in the areas namespace.

The invocation of the expr command on the other hand does **not** result in a new call frame. New call frames are only created by commands that may change the namespace context or have local variables, such as procedures and namespace eval. Since expr does neither, like most commands, it executes in the context of its caller and does not necessitate a new call frame.

When a procedure completes execution, its call frame is *popped* off the call stack. The call stack then again looks like the initial frame.

The above is a simplified, not quite accurate depiction, but sufficient for our purposes.

10.5.2. Inspecting the call stack: info level

We can examine the stack at any point of time with the info level command.

```
info level ?LEVEL?
```

Without the optional LEVEL argument, the commands return the current depth of the current call stack.

```
proc cmdA {} {
    puts "cmdA info level: [info level]"
    cmdB
}
proc cmdB {} {
    puts "cmdB info level: [info level]"
}
puts "Global info level: [info level]"
cmdA
→ Global info level: 0
  cmdA info level: 1
  cmdB info level: 2
```

If the optional *LEVEL* argument is specified, the command returns a list containing the command name and arguments that were specified for the command executing at that level in the call stack. If *LEVEL* is a positive number, it specifies an absolute call stack level. Otherwise, it specifies a stack level relative to the calling procedure where 0 is the level of the procedure itself, -1 is the procedure one level above (i.e. the one that invoked the current one) and so on.

```
proc cmdA {a {b 0}} {
    puts "cmdA: I was called as '[info level 0]'"
    cmdB $a
}
proc cmdB {a} {
    puts "cmdB: I was called as '[info level 0]'"                    ❶
    puts "cmdB: My caller was called as '[info level -1]'"           ❷
    puts "cmdB: The command invoked at the global level was '[info level 1]'"  ❸
}
cmdA 1 2
→ cmdA: I was called as 'cmdA 1 2'
  cmdB: I was called as 'cmdB 1'
  cmdB: My caller was called as 'cmdA 1 2'
  cmdB: The command invoked at the global level was 'cmdA 1 2'
```

❶ Relative level 0
❷ Relative level -1 (the caller)
❸ Absolute level 1

Note the information returned is the command that the caller invoked, not the command that is executed. What is the difference? If the procedure has optional default arguments that are not specified by the caller they will not be shown in the result from the info level command.

```
cmdA 1
cmdA 1 2
→ cmdA: I was called as 'cmdA 1'
  cmdB: I was called as 'cmdB 1'
  cmdB: My caller was called as 'cmdA 1'
  cmdB: The command invoked at the global level was 'cmdA 1'
  cmdA: I was called as 'cmdA 1 2'
...Additional lines omitted...
```

The information returned by info level makes it easy to print out the entire call stack for any procedure invocation for debugging or troubleshooting purposes. In a later section we will see how we can do this at runtime without modifying any application code.

10.5.3. Commands that create call frames

So which Tcl commands create call frames? As stated earlier, essentially any command that executes a script and supports local variables or (potentially) changes namespace contexts need new call frames. We have already seen that procedure calls fall in this category. Other such commands include namespace eval and object method calls (see Chapter 14). On the other hand, the commands eval, try, source and control statements execute scripts but do not need new call frames.

It is easy enough to check whether a command adds a call frame.

```
info level                     → 0  ❶
eval {info level}              → 0  ❷
namespace eval ns {info level} → 1  ❸
apply {{} {info level}}        → 1  ❹
```

❶ Current frame level
❷ eval runs in the namespace of the caller and has no local variables.
❸ namespace eval has no local variables but changes the namespace context.
❹ Anonymous procedure calls are just procedure calls.

10.5.4. Referencing variables in call frames: upvar

The upvar command offers the ability for a procedure to reference a variable defined anywhere in its call stack, even local variables in other procedures.

```
upvar ?LEVEL? ?VARNAME LOCAL ...?
```

The *LEVEL* argument specifies the level in the call stack. If a non-negative integer, *LEVEL* specifies the number of levels up the call stack that the variable to be referenced resides. A level of 0 references the current frame itself. If *LEVEL* begins with # immediately followed by an non-negative integer, it gives the **absolute** level in the call stack with #0 referring to the global context. If unspecified, *LEVEL* defaults to 1. However, it is strongly recommended that it be specified in case the first *VARNAME* matches one of the forms used to specify a level.

> ⚠ Note that the syntax used for the *LEVEL* argument differs from that for the info level command.

For each *VARNAME LOCAL* pair, the command will create a local variable *LOCAL* and link it to a variable *VARNAME* in the referenced call frame. All accesses to the local variable will then be passed on to the linked variable.

Time for a few examples to clarify the variations. In the script below, we define a variable named myvar in multiple contexts and then examine the call stack to see how each is referenced.

```
set myvar "Global"
proc gproc {} {
    set myvar "gproc"
    upvar #0 myvar var#0
    upvar #1 myvar var#1
    upvar 1  myvar var1 nsvar nsvar
    upvar 0  myvar var0
    puts "var#0 = ${var#0}, var#1 = ${var#1}, var1 = $var1, var0 = $var0"
    set nsvar "Created via linked variable"
    unset var#0
}
namespace eval ns {
    variable myvar "ns"
    proc nproc {} {
        variable nsvar
        set myvar "nproc"
        gproc
    }
}
```

Now if we were to run the command

```
namespace eval ns nproc
```

the call stack when the puts command in gproc is invoked will look as shown in Figure 10.3.

```
Level: 0
Namespace: global
Globals: myvar ◄──────────┐

Level: 1 (namespace eval)
Namespace: ns
Locals:   myvar ◄────────┐

Level: 2 (nproc)
Namespace: ns
Locals:   myvar ◄────────┐
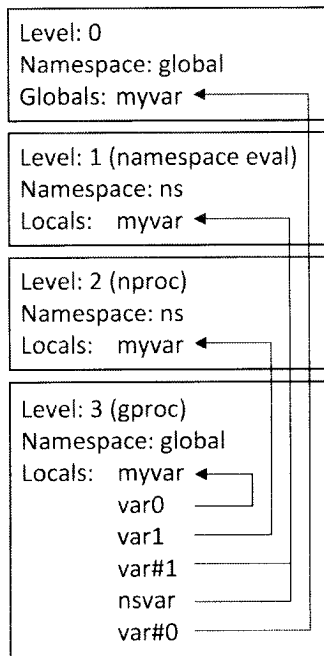
Level: 3 (gproc)
Namespace: global
Locals:   myvar ◄───┐
          var0   ───┘
          var1   ───┐
          var#1  ───┐
          nsvar  ───┐
          var#0  ───┐
```

Figure 10.3. Call stack and upvar

A few points to be noted from the figure:

- Call levels can be referenced using absolute levels (#0, #1) or relative levels (0, 1).
- The referenced variables are all named myvar (of course, this need not be the case) but are distinguished by the fact that they all appear in different call frames or namespace contexts.
- The referenced name may be that of a global variable, a namespace variable, a local variable in a procedure on the stack or itself be a linked variable (e.g. nsvar)
- **The referenced variable need not actually exist** (e.g. nsvar). It will be created if written to. Conversely, unsetting a linked variable (var#0) unsets the variable to which it is linked (global myvar).

For confirmation, we will run the command to verify the output.

```
% namespace eval ns nproc
→ var#0 = Global, var#1 = ns, var1 = nproc, var0 = gproc
% info exists ::myvar  ❶
→ 0
% puts $::ns::nsvar    ❷
→ Created via linked variable
```

❶ Was unset via the linked variable
❷ Was created via the linked variable

Using upvar

Now that we know how upvar works, where is it actually of use?

As a first example, consider implementing a command lprepend, which works like the lappend command except that it adds an element to the front of a list contained in a variable instead of the end. The command has the signature

```
lprepend VAR ?ELEM ...?
```

where VAR is the name of a variable, which may be a local, global or a namespace variable. Such a command could be implemented using upvar:

```
proc lprepend {varname args} {
    upvar 1 $varname var
    set var [linsert $var 0 {*}$args]
}
set lvar {1 2}
lprepend lvar 3 4
puts $lvar
→ 3 4 1 2
```

Notice here that unlike our earlier examples, **the referenced variable name is not hardcoded into the** upvar **invocation but rather is itself passed through a variable**.

Use of upvar in this fashion allows a "call by name" facility similar to that found in other languages. Another example where this is useful is when a procedure has to modify the contents of an array. Remember that arrays are themselves variables, not values. To modify an array then, we can just pass its name to the procedure.

Here is a procedure to change values of all array elements to uppercase.

```
proc upcase_array {arrayvar} {
    upvar 1 $arrayvar arr
    foreach {key val} [array get arr] {
        set arr($key) [string toupper $val]
    }
}
array set myarr {1 one 2 two}
upcase_array myarr
parray myarr
→ myarr(1) = ONE
  myarr(2) = TWO
```

Another situation where upvar is useful is to create an alias purely as a convenience to reduce typing effort or increase readability. For example,

```
proc myproc {} {
    upvar 0 ::ns::nsvar nsvar
    upvar 1 ::myarr(0) elem
    puts $nsvar
    set elem zero
}
```

Notice that by using a LEVEL argument of 0, we are not really changing the call frame or the variable context. We are simply creating a new name and linking it to a variable that was already available in the current context (using fully qualified names).

Variable aliases created with upvar cannot be used with commands like trace or vwait and with the -textvariable option associated with Tk widgets. You need to provide these commands with the name of the original variable instead.

10.5.5. Executing scripts in a call frame: uplevel

Having looked at upvar which allows access to variables in any frame on the call stack, we now turn our attention to the more general purpose and powerful uplevel command which allows **execution** of code within the context of any frame on the stack. It is this command that underlies some of Tcl's most dynamic features such as the ability to define of new control constructs that are on par with the built-in ones like while or switch.

```
uplevel ?LEVEL? ARG ?ARG …?
```

The command is very similar to eval in its behaviour in that it concatenates its *ARG* arguments and executes the result as a Tcl script. It differs from eval in that it accepts the *LEVEL* argument which specifies the frame on the stack within whose context the constructed script is to be executed.

The format of the *LEVEL* argument is the same as that for upvar with non-negative integers specifying the number of levels above the current call frame in which the code is to be executed. Thus a *LEVEL* of 0 will execute the script in the current context and would be equivalent to the eval command. Absolute frame numbers are specified starting with a # character followed by the level number of the frame.

This similarity to eval also implies that care needs to be taken to properly protect against double substitution when and where appropriate. See that discussion for details.

If unspecified, *LEVEL* defaults to 1. However, as for upvar it is strongly recommended that it be specified in case the first *ARG* matches one of the forms used to specify a level. Unlikely, but remember command names in Tcl can pretty much take on any form.

Time for some examples again. This time instead of showing the context using boring old variables within each procedure as we did for upvar, we will print out the command being executed at each level (refer back to info level if you don't understand this code).

```
proc cmdA {} { cmdB }
proc cmdB {} { cmdC }
proc cmdC {} {
    uplevel 0 {puts [info level]:[info level [info level]]} ❶
    uplevel 1 {puts [info level]:[info level [info level]]} ❷
    uplevel 2 {puts [info level]:[info level [info level]]} ❸

    uplevel #1 {puts [info level]:[info level [info level]]} ❹
}
cmdA
→ 3:cmdC
  2:cmdB
  1:cmdA
  1:cmdA
```

❶ Execute in the current frame (cmdC itself)
❷ Execute in caller's context
❸ Execute in frame 2 levels above
❹ Execute in context of Level 1

The call stacks when cmdC is running look as shown in Figure 10.4.

Call frame when cmdC begins execution or is evaluating the uplevel 0 command.

Call frame when cmdC is evaluating the uplevel 2 or uplevel #1 commands.

Figure 10.4. Call stack and uplevel

For the duration that cmdC is running, there are four frames on the call stack as shown. The *current frame* though, which holds the context to resolve unqualified variable and command names, changes through the execution of the procedure. On entry to the procedure, as well as during execution of uplevel 0, the current frame is the level 3 frame as we have seen before. All variables and commands will be resolved in the context of cmdC. On the other hand, when the uplevel commands with a level other than 0 are executed, this current frame pointer moves that many levels up the stack. Thus as shown, with uplevel 2 the current frame will be that for cmdA and all names will be resolved in that context.

The most common values for *LEVEL* passed to uplevel are #0 to execute in the global context and 1 to execute in the caller's context. Let us see a "real world" examples of each.

Use of uplevel to implement an interactive shell

In the first case, consider an application that allows the user to type in commands in Tcl to (for example) query a database, fetch files over the network and so on. If tclsh is passed such a script, it will run the script and exit when done. So what we need is a means to set up a "read, eval, print, loop" and call it from the script. The repl procedure below implements such a loop.

```
proc repl {} {
    set command ""
    set prompt "% "
    puts -nonewline stdout $prompt
    flush stdout
    while {[gets stdin line] >= 0} {
        append command "\n$line"
        if {[info complete $command]} {
            catch {uplevel #0 $command} result
            puts stdout $result
            set command ""
            set prompt "% "
        } else {
            set prompt "(cont)% "
        }
        puts -nonewline stdout $prompt
        flush stdout
    }
}
```

Ignoring the bulk of the code which primarily concerns I/O, the key thing to note is that users expect commands to execute in the global context. This is what the uplevel #0 command does. Had we used eval instead, the command would have been executed in the context of the repl procedure which is not what the user would expect.

We have not seen the catch command as yet. For now, it suffices to know that it is used here to handle any exceptions that may be raised during the execution of the entered command.

We saw earlier the means for checking if a file is being sourced as the main application. We can use that in conjunction with our repl procedure as follows.

```
if {[info exists ::argv0] &&
    [file dirname [file normalize [info script]/…]] eq [file dirname [file
 normalize $argv0/…]]} {
    repl
}
```

You will often find similar code at the bottom of the library scripts. If the script is being sourced as an embedded module from a main application, the code is effectively disabled. On the other hand, if the file is being sourced directly from the command line as the main application, it enters the prompt loop. This is an easy means to try out commands in the library script interactively for purposes of debugging or experimentation.

Using uplevel to implement new control structures

Another common use of uplevel is to implement new control statements that exhibit all the characteristics of the ones like while and switch that Tcl provides out of the box. For instance, let us define a command repeat that will execute a script a given number of times. A sample use might look like

```
set sum 0
repeat i 10 {
    incr sum $i
}
```

The repeat command might be implemented as below

```
proc repeat {loopvar count body} {
    upvar 1 $loopvar iter
    for {set iter 0} {$iter < $count} {incr iter} {
        uplevel 1 $body
    }
    return
}
```

The loop variable passed has to be updated in the caller's context so we use upvar to link to it. In addition, the loop body also has to execute in the caller's context so that both variable names and commands will resolve as expected. This is accomplished by the uplevel command as shown. We can try out out our new control structure.

```
% set sum 0
→ 0
% repeat i 5 { incr sum $i }
% puts "The sum of the first $i natural numbers is $sum"
→ The sum of the first 5 natural numbers is 10
```

However, this implementation is not complete. It will not behave in the same manner as the built-in control statements in the presence of errors, break or return statements and the like within the loop body. A complete implementation will have to wait until after we discuss return codes and exception handling in Tcl.

Finding the caller's namespace

There is one other common use of uplevel that you will see in commands that themselves create new commands, for example in object frameworks. When these new commands are created, care has to be taken that their names are placed within the proper namespace context.

Let us assume we want to implement such a framework where the command newobj will construct a new command of a given name (we don't really care what it does) that can be used as follows.

```
newobj cmd1          ❶
newobj ns::cmd2      ❷
namespace eval ns {newobj cmd3}      ❸
namespace eval ns {newobj ::cmd4}    ❹
```

❶ Should create ::cmd1
❷ Should create ::ns::cmd2
❸ Should create ::ns::cmd3
❹ Should create ::cmd4

We can write this procedure as follows.

```
proc newobj {name} {
    if {[string match ::* $name]} {
        set cmdname $name
    } else {
        set ns [uplevel 1 {namespace current}]
        if {$ns eq "::"} {
            set cmdname ::$name
        } else {
            set cmdname ${ns}::$name
        }
    }
    if {[namespace which -command $cmdname] ne ""} {
        error "command $name already exists"
    }
    proc $cmdname {} "puts {I am $cmdname}"
    return
}
```

In our simplistic example where the created commands simply print their name, all our newobj procedure really has to do is to ensure the command name is created in the correct namespace context:

- If the name is already fully qualified, it can be used as is.
- Otherwise, the name is relative to the **caller's** namespace so we use uplevel to retrieve that and qualify the constructed command with the namespace name.
- Finally, as a matter of policy we check that the name does not already exist. (It is really a matter of choice whether to allow commands to be overwritten.)

To verify our command name generation,

```
% newobj cmd1
% cmd1
→ I am ::cmd1
% namespace eval ns {newobj cmd3}
```

```
% cmd3 ❶
∅ invalid command name "cmd3"
% ns::cmd3
→ I am ::ns::cmd3
% newobj ns::cmd3 ❷
∅ command ns::cmd3 already exists
```

❶ Error. cmd3 was not created in the global namespace
❷ Error. Command by that name already exists!

10.5.6. The internal C stack

We have talked about the call stack that keeps track of the execution contexts through a chain of procedure calls. These contexts control how variables, commands and namespaces are resolved. Tcl also maintains another stack, maintained internally and not directly visible at the scripting level, that keeps track of (among other things) the currently executing command and the location to continue from when it completes. For the lack of a better term, we will call this the internal C stack [1].

This internal C stack will become relevant when we discuss more sophisticated programming models in Tcl including recursion, the event loop and coroutines.

To illustrate the relationship between the call stack and the internal C stack, consider execution of the following script which prints the call stack level at which each procedure is executing.

```
proc demo1 {} {
    puts "[info level [info level]]: Level [info level]"
    demo2
}
proc demo2 {} {
    puts "[info level [info level]]: Level [info level]"
    uplevel 1 {
        puts "uplevel: Level [info level]"
        demo3
    }
}
proc demo3 {} {
    puts "[info level [info level]]: Level [info level]"
}
demo1
→ demo1: Level 1
  demo2: Level 2
  uplevel: Level 1
  demo3: Level 2
```

The states of the call stack and the C stack at two stages of evaluation are shown in Figure 10.5. The left side shows the state during execution of puts in the demo2 procedure while the right side shows the state during execution of puts in the demo3 procedure.

Note the following points illustrated by the figure:

* The puts command does not create a new call frame in the call stack as it resolves names within the context of its caller. Nevertheless it adds a slot to the C stack where Tcl stores its caller and return information.

* Likewise, the uplevel 1 command adds a slot to the C stack as well. On the other hand, its associated context level is actually less than that of its caller.

---

[1] In reality, Tcl maintains multiple internal stacks but we will not concern ourselves with that as it is an implementation detail.

- When demo3 is called via `uplevel`, the context for demo2 does not even appear on the call stack. (This does not mean it has disappeared. It simply is not accessible through the call stack until control returns to evaluation of demo2.)
- Notice how the depth of the C stack has grown even while the call stack depth stays the same.

This last point is the most important one to note because it impacts the maximum depth of recursive algorithms and serves as the motivation for the `tailcall` command we discuss next.

Figure 10.5. C stack and call frames

10.5.7. Recursing in place: `tailcall`

The growth of the internal C stack that we described in the previous section is generally not an issue because procedure calls rarely nest deep enough for it to be a problem. The one common situation where it can be a factor is in the implementation of recursive algorithms. Let us illustrate with a simple command that calculates the sum of the first N natural numbers. We will use a recursive command instead of a simple iterative loop because the latter would not impress anyone.

```
proc sum {n {total 0}} {
    if {$n == 0} { return $total }
    sum [expr {$n-1}] [incr total $n]
}
```

This works well enough for small numbers.

```
sum 4 → 10
```

However, see what happens when we try to sum the first 1000 integers.

```
sum 1000 0 too many nested evaluations (infinite loop?)
```

The error you see comes from Tcl aborting the evaluation to guard against an overflow of the C stack which would lead to the process crashing. Although the `interp limit` command can be used to change the limit of recursion,

this merely postpones the problem. Morever, changing the recursion limit with `interp limit` is dangerous without recompiling or relinking Tcl to increase the process stack size.

The problem of stack growth can be solved for certain kinds of recursive algorithms where the recursion is the last operation in the execution of the function or procedure. Under these circumstances, the context of the **calling** procedure need not be maintained because there is nothing to be done after the called procedure returns. Thus the stack space occupied by the calling procedure can be **reused** for the called procedure.

This is what the `tailcall` command effects.

```
tailcall COMMAND ?ARG ...?
```

The `tailcall` command invokes COMMAND, passing it any supplied arguments, overwriting the context of its caller with that of COMMAND.

Before we go into a detailed explanation, let us rewrite our example using the `tailcall` command.

```
proc sum {n {total 0}} {
    if {$n == 0} { return $total }
    tailcall sum [expr {$n-1}] [incr total $n]
}
```

You can now sum without running into recursion limits.

```
sum 100000 → 5000050000
```

A look at the internal C stacks, shown in Figure 10.6, in the computation of sum 2 in the two cases will tell us why.

Figure 10.6. Call stack with tailcall

As sum recurses, the tailcall version reuses the caller's stack slot keeping the stack space constant.

There are a few instances where `tailcall` is useful even without any recursion being involved. For instance we saw in a previous chapter a simple method for wrapping a procedure by renaming it and then calling it from the redefined procedure. That method had the drawback that it increased the call stack depth and would not work with commands like `foreach` that use `uplevel` or the equivalent to execute in their caller's context. The failure mode is demonstrated by the following example.

```
rename while _builtin_while
proc while args {
    puts "while called"
    _builtin_while {*}$args
}
set n 2
while {$n > 0} {puts $n ; incr n -1}
Ø can't read "n": no such variable
  while called
```

The command failed because our `while` wrapper executed the built-in command within its own context where there was no variable n. Although that could be fixed via an explicit `uplevel`, an easier and faster solution is to use `tailcall` to delegate the invocation. This way the stack depth remains the same when the original command is called and the body of the `while` is evaluated in the context of the original caller.

```
proc while args {
    puts "while called"
    tailcall _builtin_while {*}$args
}
set n 2
while {$n > 0} {puts $n ; incr n -1}
→ while called
  2
  1
```

This then works as expected. The command is similarly useful in scenarios like delegation of object methods.

Be aware that `tailcall` executes a **command**, not a **script**, so there are no issues around double substitution as there are with `eval` or `uplevel`. If you do need to call a script in tail-recursive fashion, you can use `try` in combination with `tailcall`.

```
 tailcall try { Your script }
```

The `try` command is preferable here to `eval` because it is faster to execute being byte-compiled in Tcl 8.6 whereas `eval` is not.

With that introduction under our belt, we can move on to detailing exactly what `tailcall` does.

> The details behind the operation of `tailcall` are only important when you are using it in conjunction with other commands like `uplevel` that affect call stacks or with control structures like `try`. For its primary use for simple recursion as in the above example, these details are not important and may be skipped.

The `tailcall` command works by

- first arranging for its command argument to be invoked after the completion of the call frame within which the `tailcall` was invoked,
- and then forcing its caller to complete immediately with a return code value of 2 / `return`. **Note the return code from a command invocation is not the same as the command result.** Return codes are discussed in Section 11.2.1.

As we will see, this two step process can be a bit tricky when the call frame is not directly that of the caller but let us start with a simple example.

```
proc demo1 {} {
    puts "demo1 enter"
    tailcall puts "tailcalled puts"
    puts "demo1 exit"
}
proc demo {} {
    puts "demo enter"
    demo1
    puts "demo exit"
}
```

Here is the output when we invoke demo.

```
% demo
→ demo enter
  demo1 enter
  tailcalled puts
  demo exit
```

The tailcall arranges for the puts command to be invoked after the completion of the current call frame, which is that of demo1. Then it forces the caller, again demo1, to immediately complete at which point the puts command set up by the tailcall is invoked. The puts "demo1 exit" line never gets executed.

This is all fairly straightforward. Now for the trickier example we mentioned. Let us rewrite demo1 as follows.

```
proc demo1 {} {
    puts "demo1 enter"
    uplevel 1 {
        tailcall puts "tailcalled puts"
    }
    puts "demo1 exit"
}
```

Now the output below is somewhat puzzling (maybe) for a couple of reasons. First, you might expect that the demo exit line would not be printed as the tailcall is executed in the context of demo. Even stranger is that, unlike the previous example, the tailcalled puts is printed **after** demo exit.

```
% demo
→ demo enter
  demo1 enter
  demo exit
  tailcalled puts
```

Here is the explanation. Because it is run via uplevel, the tailcall runs in the call frame for demo, not that of demo1. Thus it schedules its argument to run after completion of demo and not demo1. Then it forces its caller to complete immediately with the return return code. The caller here is uplevel which propagates the return code causing demo1 to immediately return. Evaluation of demo then continues as normal resulting in the demo exiting line being printed and finally when demo completes, the command set up by the tailcall gets to run.

Rarely will you need this level of minutiae but there you have it.

10.5.8. Hidden frames: info frame

We learnt about call frames in Section 10.5.2 and how the info level command provides access to the different call frames in the call stack. There are in fact a few hidden frames that are not visible via info level. These are hidden because they do not introduce new local variable scopes and as such do not have much programming

significance. They contain metainformation about the script being executed, such as the method by which it is being executed (through `eval`, procedure calls, etc.), the source file it was defined in and so on. The `info frame` command provides access to this information.

```
info frame ?FRAMENUMBER?
```

If *FRAMENUMBER* is not provided, the command returns the frame level for the command. Otherwise, it returns a dictionary containing the metainformation for the frame at that level. If *FRAMENUMBER* is positive, it specifies the absolute frame level; if negative, it is relative to the current frame.

First, let us contrast `info level` and `info frame`.

```
proc demo {} {
    puts "demo level: [info level], frame: [info frame]"
    eval {
        puts "eval level: [info level], frame: [info frame]"
    }
    uplevel 1 {
        puts "uplevel level: [info level], frame: [info frame]"
    }
}
puts "global level: [info level], frame: [info frame]"
demo
→ global level: 0, frame: 1
  demo level: 1, frame: 2
  eval level: 1, frame: 3
  uplevel level: 0, frame: 3
```

If we look at the output, notice that

- In the global context, `info level` returns 0, `info frame` returns 1.
- The procedure call to demo increments both.
- The `eval` command increments only the `info frame` value as it does not add a call frame with a new variable scope.
- The `uplevel` command increments the `info frame` value but **decrements** the `info level` as we described in Section 10.5.5.

Other commands that evaluate scripts but do not introduce a new local variable scope, such as `source`, `try`, `if`, `while` etc. also behave in a manner similar to `eval`.

Let us now look at the information returned by the command. Write the following script to a file and then use `source` to evaluate it.

```
proc demo {} {
    demo2 "argument"
}
proc demo2 {arg} {
    puts "Frame: [info frame]"
    print_dict [info frame 2]
}
```

Then running demo will show the following output.

```
% demo
→ Frame: 3
  cmd      = demo2 "argument"
  file     = C:/Users/ashok/AppData/Local/Temp/TCL57053.TMP
  level    = 1
```

```
line   = 2
proc   = ::demo
type   = source
```

The cmd element of the returned dictionary is the command being executed in that frame. Note we have introspected the frame one level up from where the info frame call is made. Thus the cmd entry shows the call to the demo2 procedure.

The type element of the returned dictionary is source indicating that the command was defined inside a sourced file. It may also be proc for dynamically created procedure bodies, eval if located within a script being evaluated by eval, uplevel, try etc., or precompiled indicating it is a pre-compiled script loaded by tbcload.

The other entries of the dictionary are dependent on the value of the type element. In our example, these entries are

- file, containing the path to the file containing the command definition
- line, the line number within the file
- proc, the name of the procedure within whose body the command was invoked
- level, corresponding to the info level command.

See the reference pages for information about the dictionary elements for other values of type. We do not go into further detail because of the limited utility of the info frame command. Its primary purpose is for building debuggers and similar tools. Unlike info level, you will rarely find it present in Tcl code.

10.6. Traces

One of the features in Tcl not commonly found in other languages is the ability to have program actions like variable access or command invocation trigger the execution of code (in addition to the command being invoked of course!). This capability, which we call *tracing*, can be used with great effect in a wide variety of scenarios:

- Implementation of custom language features like read-only variables or value validators, or modifying the behaviour of commands without making internal code changes.
- A data flow style of programming where data modifications are propagated to other parts of the application. Spreadsheets are an example of this. So also user interfaces where programmatic updates and updates from the user are propagated in both directions.
- Development tools like profiles and debuggers that by their very nature need to be able "hook" into program and data flow to track and display changes.
- Resource cleanup in situations that cannot be handled with the normal Tcl exception handling capabilities.

We will see rudimentary illustrative examples of these throughout this section.

10.6.1. Tracing variables

The trace command implements this tracing facility for both variables and commands. We will start with exploring traces for variables.

10.6.1.1. Creating a variable trace: trace add variable

Tracing of a variable is enabled with the trace add variable command.

```
trace add variable VARNAME OPS COMMANDPREFIX
```

The VARNAME argument specifies the name of the variable to be traced. OPS specifies the operations of interest. COMMANDPREFIX is a command prefix to be invoked when the variable undergoes one of these operations.

There is no restriction on the number of traces you may add to a variable. If multiple traces are created, they will all be invoked unless one of them raises an exception. The order of invocation is the reverse order of their creation with the last added trace being invoked first.

The *OPS* argument must be a list whose elements are amongst the operations shown in Table 10.2. The operations control the types of variable access that will trigger the trace.

Table 10.2. Trace operations on variables

Operation	Description
array	Triggered when the variable is accessed or modified via the array command.
read	Triggered just **before** the variable is read. This means the variable need not exist and can in fact be set by the trace callback.
unset	Triggered when the variable is unset, either explicitly or implicitly when its containing scope is exited.
write	Triggered just **after** the variable is written to. The trace callback can change the variable afterwards.

When a trace is triggered, *COMMANDPREFIX* is invoked with three additional arguments. The first is the name of the variable on which the trace triggered, the second is the key of the element being accessed if the variable is an array or an empty string otherwise, and the third is an operation value from the above table.

Let us start with a basic example to illustrate some finer points of traces. First some set up. We need a procedure that we will use as the callback for the traces.

```
proc tracer {varname elemname op} {
    puts "Trace: $op operation on variable $varname"
}
```

Now for the actual trace itself. Note that we are setting up a trace on a variable that does not even exist yet.

```
% unset -nocomplain myvar
% trace add variable myvar {read write unset} tracer
% namespace which -variable myvar
→ ::myvar
% info exists myvar
→ Trace: read operation on variable myvar
  0
```

Notice the differing output between namespace which and info exists. The former verifies that the name is created while the latter confirms that the variable itself does not exist. Also note that info exists also triggered our trace causing the message to be printed.

We can now try the various operations on the variable.

```
% set myvar "foo"
→ Trace: write operation on variable myvar
  foo
% set myvar
→ Trace: read operation on variable myvar
  foo
% unset myvar
→ Trace: unset operation on variable myvar
% set myvar "bar"
→ bar
```

We can see our `tracer` procedure is triggered as expected in each case. Notice however that once the variable is unset, the trace is also deleted so when we write to a variable of that name again, there is no trace in place.

A cautionary note is in order. The variable name that is passed to the trace callback is not necessarily that on which the trace was applied. It is the name used to access the variable in the caller's context. For example,

```
% trace add variable myvar {read write unset} tracer
% upvar 0 myvar linked_var
% set linked_var foo
→ Trace: write operation on variable linked_var
  foo
```

Notice that the callback sees the variable name `linked_var`, and not `myvar`. In most cases, where the callback uses `upvar` to access the variable if needed (see examples later), this does not matter. But if you are for example logging variable access to a file for debugging purposes this might cause some confusion. In such instancs, pass the "real" name to the callback as an additional argument at the time the `trace` command is called.

Let us make our trace callback a little more sophisticated. This time instead of merely printing the operation, we will actually **affect** it.

```
proc tracer {varname elemname op} {
    upvar 1 $varname var
    switch $op {
        read {
            puts "Trace: op=$op, $varname=$var"
            set var [string reverse $var]
        }
        write {
            puts "Trace: op=$op, $varname=$var"
            set var [string toupper $var]
        }
        unset {
            puts "Trace: \[info exists $varname\]=[info exists var]"
        }
    }
    return "This result of the callback is ignored"
}
```

The following points about trace callbacks should be noted in our example:

- Trace callbacks are invoked within the same context as the command that operates on the variable. Since our trace prefix is itself a procedure, it adds a call frame and thus we have to use `upvar` to access the variable name in the context of the caller.

- Both `read` and `write` traces can modify the variable. The new value of the variable is what will be returned by the traced operation.

Let us try this new version.

```
% trace add variable myvar {read write unset} tracer
% set myvar "foo"
→ Trace: op=write, myvar=foo
  FOO
```

Notice writing the variable stores the upper case form of the original value being assigned. Now we can try reading it back.

```
% set myvar
→ Trace: op=read, myvar=FOO
  OOF
% set myvar
→ Trace: op=read, myvar=OOF
  FOO
% unset myvar
→ Trace: [info exists myvar]=0
```

Every read on the variable reverses the stored value! A bit of imagination is sufficient to see how you drive a hated colleague bananas without even touching their code.

The output leads to some additional points of note:

- The write trace is called **after** the operation sets the new value of the variable. Our callback then updates it with the upper case version.

- The unset trace is also called **after** the variable is unset.

- The result of the original command reflects the new value of the variable **after** it is updated by the trace handler.

- The result of the trace callback `tracer` does not show up anywhere. It is ignored. Note however raised errors are treated differently as described below.

> Given that we have traces set on the variable, a legitimate question to ask is won't those traces recursively fire when we modify the variable within our callback? The answer is that Tcl is smart enough to disable read and write traces on a variable while a read or write callback is in progress. **However, note that this is not so (by design) if the callback an unset operation.** Any unset traces will be triggered as usual. Moreover, traces are not disabled if the callback itself is in response to an unset operation.

If a read or write callback raises an exception, the original command also completes with the same exception. However, exceptions raised during an unset callback are ignored.

One non-obvious point with variable unset traces is that the command creates the specified variable if it is not already created. This means you can create traces on non-existent variables and have them fire as a means to detect when a variable scope is deleted. For example,

```
% proc demo {} {
    trace add variable NOSUCHVAR unset print_args
}
% demo
→ Args: NOSUCHVAR, , unset
```

Notice our trace fired when the demo procedure returned. Section 21.4.1 describes an application of this feature.

We now show some examples of the different ways variable traces might be put to use.

Lazy initialization

The fact that variables need not exist when traces are registered as well as the fact that they can be modified by the traces allows us to do lazy initialization.

Consider our simple sum procedure that returns the sum of the first N natural numbers. We can use traces to allow the application to access these values as array elements. For example, we should be able to say

```
puts "Sum of 1:5 is $sums(5)."
```

Now clearly we cannot predict which numbers **might** be of interest and even if we did, it might computationally expensive to pre-fill the array with values that might only be **potentially** used. Both these issues are easily solved with lazy initialization using traces.

First we create the empty array and attach a variable trace to it.

```
array set sums {}
trace add variable sums read calculate_sum
```

We will discuss array traces in the next section but for now it suffices to know that our trace procedure, which we define next, is called every time an array element is read.

```
proc calculate_sum {varname elem op} {
    upvar 1 $varname var
    if {! [info exists var($elem)]} {
        puts "Calculating sum of $elem"
        set var($elem) [sum $elem]
    }
}
```

Now we can go ahead with our calculations.

```
% puts "Sum of 1:5 is $sums(5)."
→ Calculating sum of 5
  Sum of 1:5 is 15.
% puts "Sum of 1:3 is $sums(3)."
→ Calculating sum of 3
  Sum of 1:3 is 6.
% puts "Sum of 1:5 is $sums(5)." ❶
→ Sum of 1:5 is 15.
% parray sums
→ sums(3) = 6
  sums(5) = 15
```

❶   Note no computation message

As you can imagine, this technique can be put to use in other scenarios, caching of URLs for instance.

Constant variables

Here is an example of defining a variable as "read-only" or a constant. We will define a const command for the purpose.

```
const VARNAME VALUE
```

Any attempt to modify the variable will raise an error while keeping the variable unchanged.

```
proc const {varname value} {
    upvar $varname var
    trace add variable var write \
        [lambda {constval name element op} {
            upvar 1 $name var
            set var $constval ❶
            throw {CONST MODIFY} "Attempt to modify a constant."
        } $value]
}
```

❶   Restore original value since it would have already been modified.

The above code implements the trace callback as an anonymous procedure (see Section 3.5.8.4). When the callback is invoked, the value of the variable would have already changed so we have to pass the original constant to the callback separately as its first parameter.

Now any attempt to modify a `const` variable is rejected.

```
% const e 2.71828
% set e 0
∅ can't set "e": Attempt to modify a constant.
% set e
→ 2.71828
```

Data flow programming

Our last example is an outline of how a data flow program structure might be implemented. We will have a 2x2 spreadsheet with cells numbered A1-A2..B1-B2 and store cell values in global variables of the same name. Spreadsheet formulas are then almost trivial to implement in declarative style. Suppose the user has defined cell contents of B1 and B2 to be

```
B1 = A1 + A2
B2 = B1**2
```

This would translate to the following code

```
proc getval cell {
    upvar #0 $cell var
    return [expr {[info exists var] ? $var : 0}]
}
proc updateB1 {args} {
    set ::B1 [expr {[getval ::A1] + [getval ::A2]}]
}
proc updateB2 {args} {
    set ::B2 [expr {[getval ::B1]**2}]
}

trace add variable A1 {write unset} updateB1
trace add variable A2 {write unset} updateB1
trace add variable B1 {write unset} updateB2
```

Now we can see how updates are automatically propagated between cells.

```
set A1 3 → 3
set A2 4 → 4
set B2   → 49
set A2 2 → 2
set B2   → 25
```

This example is beyond simplistic but should give you a flavor for how you might put variable traces as the basis for such a system There are several examples of such use in the Tcler's Wiki [2].

The above example demonstrates "push" traces where changes to a variable are propagated to its dependents when it is written to. In some cases, a "pull" model can be more convenient. Instead of adding write traces to A1, A2 we could add a read trace to B1 and B2 instead. When this trace fired, the **current** values of A1, A2 would be used to compute a new value for B1 / B2.

---

[2] http://wiki.tcl.tk

10.6.1.2. Tracing array variables

There are some special cases to be considered for tracing of array variables. Let us modify our tracing callback to just print its arguments.

```
proc tracer {varname elem op} {
    puts "Trace: varname=\"$varname\", elem=\"$elem\", op=\"$op\""
}
```

Tracing a specific element of an array is just like tracing a variable except that the name of the element is supplied as the second argument to the callback.

```
% trace add variable arr(x) {read write unset} tracer
% set arr(x) 100
→ Trace: varname="arr", elem="x", op="write"
  100
% set arr(x)
→ Trace: varname="arr", elem="x", op="read"
  100
% array set arr {x 0 y 1}
→ Trace: varname="arr", elem="x", op="write"
% array get arr
→ Trace: varname="arr", elem="x", op="read"
  x 0 y 1
% unset arr
→ Trace: varname="arr", elem="x", op="unset"
```

Notice that our trace fires irrespective of whether the element is individually operated on or is part of an operation on the entire array.

If you want to track changes to the entire array, and not just specific elements, in addition to providing the name of the array as the variable name, you have to specify array as the operation of interest (along with other operations if desired). Operations on the array with commands such a array set and array get will trigger the callback.

```
% array set arr {x 0 y 1}
% trace add variable arr {read write unset array} tracer
% array set arr {a1 2 a2 3}
→ Trace: varname="arr", elem="", op="array"
  Trace: varname="arr", elem="a1", op="write"
  Trace: varname="arr", elem="a2", op="write"
% array unset arr a*
→ Trace: varname="arr", elem="", op="array"
  Trace: varname="arr", elem="a1", op="unset"
  Trace: varname="arr", elem="a2", op="unset"
% set arr(x) 10
→ Trace: varname="arr", elem="x", op="write"
  10
% array get arr
→ Trace: varname="arr", elem="", op="array"
  Trace: varname="arr", elem="x", op="read"
  Trace: varname="arr", elem="y", op="read"
  x 10 y 1
% unset arr
→ Trace: varname="arr", elem="", op="unset"
```

Points to be noted from the above:

- When the array callback is made, it applies to the whole array and hence the second argument of the callback, the element, is set to the empty string.
- The array callback does not indicate the type of operation, get, set etc. This is unfortunate.
- Operations using the array command will also trigger read, write and unset traces on individual elements if these were included in the operations list for the trace command. Moreover, setting individual elements will also trigger the trace. Note the individual element traces are invoked **after** the array trace.
- Unsetting the entire array with unset does **not** trigger traces for **individual** elements unlike array unset.

10.6.1.3. Resource management using variable traces

We saw earlier the use of the try..finally command to ensure resources are released even in case of errors. There are however circumstances where that technique is not viable because the finally clause never gets to run. Two such cases are

- Deletion of an entire namespace.
- Deletion of a coroutine with the rename command.

In such cases, variable traces can be put to good use to release resources at an appropriate time. This is described in detail with respect to coroutines in Section 21.4.1. The technique described there may be used to deal with namespace deletion as well.

10.6.2. Tracing commands

Tracing facilities for commands fall into two categories:

- tracing the lifetime of a command definition
- tracing command execution

We describe these in turn.

10.6.2.1. Tracing command lifetimes: trace add command

The trace add command command registers a callback to be invoked when the specified command is renamed or deleted.

```
trace add command NAME OPS COMMANDPREFIX
```

The argument *NAME* is the name of the command to be traced. Unlike for variable traces, this command must already exist. The argument *COMMANDPREFIX* is a command prefix to be invoked when the command is renamed or deleted. The *OPS* argument must be a list whose elements are shown in Table 10.3.

Table 10.3. Trace operations on commands

Operation	Description
rename	Triggered when a command is renamed. Note that renaming a command to an empty string is treated as a deletion and does not trigger this trace.
delete	Triggered when the command is deleted. This may happen if it is explicitly deleted by renaming it to the empty string or if the containing namespace is deleted.

When a trace is triggered, *COMMANDPREFIX* is invoked with three additional arguments. The first is the **fully qualified** name of the command, the second is either the empty string if the operation is delete or the new name of the command, again fully qualified, if the operation is rename. The third argument is either rename or delete and indicates the operation.

As for variable read and write traces, command traces for a command are also disabled when a command trace callback for **that** command is in progress.

Here is a simple example illustrating command traces.

```
% namespace eval ns { proc demo {} {} }
% trace add command ns::demo {rename delete} tracer
% rename ns::demo demo2
→ Trace: varname="::ns::demo", elem="::demo2", op="rename"
% rename demo2 ""
→ Trace: varname="::demo2", elem="", op="delete"
```

The above example also demonstrates that traces stay attached to a command even when it is renamed.

Redefinition of a procedure is treated as a deletion and the trace fires accordingly. The new definition will however **not** have the trace attached.

Command traces tend to be less common than variable traces but there are still situations where they are put to good use. One example is in packages where a command is a "proxy" for an external object, such COM on Windows or CORBA. The package can ensure the external object is released appropriately by placing a trace on the command deletion.

10.6.2.2. Tracing command execution: `trace add execution`

The `trace add execution` command is a very powerful tool that can provide insight into the exact sequence of commands executed in a program.

Powerful as they are, execution traces also extract a large performance penalty because they prevent inlining of byte coded commands (see Section 10.10.2.5). Their use should therefore be limited to debugging and troubleshooting purposes and is not recommended as part of program flow in normal operation.

The execution trace can track both invocation of a specified command or **all** command execution for the duration of that command.

```
trace add execution NAME OPS COMMANDPREFIX
```

The argument `NAME` is the name of an existing command whose execution is to be traced and `COMMANDPREFIX` is a command prefix (see Section 10.7.1) to be invoked when the trace is triggered. The `OPS` argument must be a list whose elements are shown in Table 10.4.

Table 10.4. Trace operations on command execution

Operation	Description
enter	Triggered just before the **specified** command begins execution.
leave	Triggered just after the **specified** command completes. The callback will be invoked even if the command completed with an exception.
enterstep	Triggered just before **any** command is invoked during the execution of the specified command. This include any nested calls to any depth.
leavestep	Like enterstep but triggered just **after** the completion of every command for the duration of execution of the specified command. This include any nested calls to any depth.

When a trace is invoked for enter and enterstep triggers, `COMMANDPREFIX` is invoked with two additional arguments. The first is the full command string and the second is enter or enterstep indicating the trigger.

When invoked for `leave` and `leavestep` triggers, four arguments are added. The first is the command string, the second is the return code (see Section 11.2.1) from the command invocation, the third is its result and the fourth is the trigger operation, `leave` or `leavestep`.

```
proc tracer args {
    puts "Trace: [join $args {, }]"
}
proc demo {args} { demo2 X Y }
proc demo2 {args} { demo3 }
proc demo3 {} {return "result" }
trace add execution demo {enter leave} tracer
demo
→ result
  Trace: demo, enter
  Trace: demo, 0, result, leave
```

Notice our tracer logs the invocation and completion of the demo command. On the other hand, adding traces for `enterstep` and `leavestep` would log **all** commands while demo was executing.

```
trace add execution demo {enterstep leavestep} tracer
demo
→ result
  Trace: demo, enter
  Trace: demo2 X Y, enterstep
  Trace: demo3, enterstep
  Trace: return result, enterstep
  Trace: return result, 2, result, leavestep
  Trace: demo3, 0, result, leavestep
  Trace: demo2 X Y, 0, result, leavestep
  Trace: demo, 0, result, leave
```

Let us redefine demo3 to raise an error instead.

```
proc demo3 {} {error "Something horrible happened."}
demo
∅ Something horrible happened.
  Trace: demo, enter
  Trace: demo2 X Y, enterstep
  Trace: demo3, enterstep
  Trace: error {Something horrible happened.}, enterstep
  Trace: error {Something horrible happened.}, 1, Something horrible happened., leavestep
  Trace: demo3, 1, Something horrible happened., leavestep
  Trace: demo2 X Y, 1, Something horrible happened., leavestep
  Trace: demo, 1, Something horrible happened., leave
```

Again, note the trace triggers on completion of each procedure even on exceptions where the return code is shown as 1 as opposed to 0 for a normal return.

Execution traces are not normally used because of their performance impact. Nevertheless, they can be indispensible for fault diagnosis, particularly in the field since they can be configured with no changes to the application source.

10.6.2.3. Deleting a trace: `trace remove`

All three forms of traces can be deleted with the `trace remove` command.

```
trace remove variable NAME OPS COMMANDPREFIX
```

```
trace remove command NAME OPS COMMANDPREFIX
trace remove execution NAME OPS COMMANDPREFIX
```

The *NAME* and *OPS* arguments have the same semantics as for the trace add command — they identify the variable or command and the operations for which the trace is to be deleted. Since there may be multiple traces on a variable or command, *OPS* and *COMMANDPREFIX* identify the specific trace to be removed. They must match the corresponding arguments that were used to initiate the trace.

```
trace remove execution demo {enterstep leavestep} tracer
demo
Ø Something horrible happened.
  Trace: demo, enter
  Trace: demo, 1, Something horrible happened., leave
```

Notice that only the specified triggers were removed. The enter and leave remained active.

To reiterate the point about both the *OPS* and *COMMANDPREFIX* arguments having to be the same as in the initiating trace command, suppose we only wanted to remove the enterstep trigger. The following would **not** work.

```
trace remove execution demo enterstep tracer
```

The *OPS* argument would not match the initiating trace so the trace removal would be silently ignored. For the desired effect you have to remove the complete trace as in the prior example and add a new one with just the leavestep trigger specified.

An attempt to remove a trace on a non-existent variable is silently ignored but not so for commands. If the command *NAME* is not defined, the trace remove will raise an error exception.

10.6.2.4. Inspecting traces: `trace info`

The trace info command can be used to retrieve all traces active on a variable or command.

```
trace info variable NAME
trace info command NAME
trace info execution NAME
```

The result of the command is a list of pairs each of which contains the *OPS* and *COMMANDPREFIX* arguments that were supplied to the trace add command.

Let us check the execution traces on our demo procedure after adding back the trace that we previously removed for demonstration purposes.

```
trace add execution demo {enterstep leavestep} tracer
puts [trace info execution demo]
→ {{enterstep leavestep} tracer} {{enter leave} tracer}
```

We can use this information to remove all traces from a variable or command.

```
foreach trace [trace info execution demo] {
    trace remove execution demo {*}$trace
}
puts [trace info execution demo]
→
```

10.7. Code construction

In dynamic languages like Tcl, it is common for code fragments to be passed around, evaluated and even constructed on the fly. Examples include

- Operation of commands like `lsort` and `dict filter` can be customized through callbacks.

- Event handlers and traces use callbacks for notification purposes.

- Although it may not be obvious if you are coming from other languages, even the "body" of commands like `eval`, `try`, `if`, and `while` are just arguments and not any special syntactic constructs. You can thus pass dynamically constructed code as their bodies.

- Metaprogramming, which we discuss in Section 10.8, is based on the ability to construct and execute code at runtime.

In this section, we provide some hints and tips related to these aspects of Tcl programming.

10.7.1. Scripts versus command prefixes

For starters, we need to distinguish between arguments that are *scripts* versus arguments that are *command prefixes*. Commands that take script arguments evaluate them as Tcl scripts with (potentially) multiple commands and following the usual Tcl syntax and substitution rules. On the other hand, commands that take a command prefix argument treat the argument as a **single** command which is in a list form containing the command name and possibly some arguments to be passed to it. In both cases, when the callback is invoked additional arguments **may** be appended containing specific information about why it is being invoked.

The following mock procedures illustrate the difference. The `script_cb` procedure is written to accept a script callback as an argument while `cmd_cb` takes a command prefix. We will pass the same `callback` argument to both.

```
set callback {print_args A ; print_args B C}
proc script_cb {script} {
    uplevel 1 $script "(script)"
}
proc cmd_cb {cmdprefix} {
    tailcall {*}$cmdprefix "(command)"
}
```

Notice the difference in the output below in the two cases.

```
% script_cb $callback
→ Args: A
  Args: B, C, (script)
% cmd_cb $callback
→ Args: A, ;, print_args, B, C, (command)
```

This difference arises because the `script_cb` command executes the callback as a script where the `;` character is treated as a command separator. In the case of `cmd_cb` it is treated simply as an argument.

Both forms of callback arguments are commonly seen and you have to just be aware of what type of callback is expected by a command.

10.7.1.1. Constructing command prefixes

In the example above, we passed a (brace enclosed) string as an argument for the purposes of contrasting command prefixes with scripts. However, the recommended way to pass a command prefix as a callback is by constructing it as a list. For example,

```
% set some_value "First arg"
→ First arg
```

```
% cmd_cb [list print_args $some_value ";" "Third arg"]
→ Args: First arg, ;, Third arg, (command)
```

Providing the callback as an interpolated string would require more care, such as escaping whitespace and special characters, to ensure it is parsed correctly as a list of arguments.

It is also common to use an anonymous procedure as a command prefix instead of defining a named procedure for a one time use. For example, the custom sorting example from Section 5.7 can be written as

```
set part_numbers {part_100_b PART_100_C PART_20_B}
lsort -command [lambda {s1 s2} {
    return [expr {[string length $s1] - [string length $s2]}]
}] $part_numbers
→ PART_20_B part_100_b PART_100_C
```

where we have used the lambda utility procedure from Section 3.5.8.4 to define an anonymous procedure for comparing strings in order of their length.

10.7.1.2. Constructing scripts

Constructing scripts is more involved than command prefixes because while the latter have a limited structured form, scripts can be full blown Tcl programs. Scripts are not only used for callbacks but also in metaprogramming as discussed in the next section.

In their simplest form, scripts are enclosed in braces as a literal string. We have seen this frequently as in the definition of a procedure body, if statement and so on. This is not particularly useful in dynamic script construction though, because in most cases the script is at least partly built from runtime information and not completely known at the time it is written. Enclosing the script in braces precludes use of variable and command substitutions in the generation of the script.

There are several alternatives for building scripts by combining "static" fragments with dynamic ones at runtime:

- The script can be composed from a series of commands that append static literals and variable fragments. This is the most flexible alternative but suffers from a lack of readability where the structure and purpose of the script is not readily apparent.

- Alternatively, the script can be constructed as a literal string in double quotes instead of braces. The variable parts of the scripts can then simply be variable references or bracketed commands that are replaced through the normal string interpolation rules. This is a reasonable approach when the constructed script is simple. For even moderately complex scripts however, several issues arise. Variable references and bracketed commands that are part of the *generated* script need to be escaped so they are not substituted at script generation time. This escaping of special characters and newlines can become tricky. There is also a loss of readability as in the previous alternative.

- For more complex scripts, it is often easiest to write the script as a template with "place holders" for the dynamic parts. These are then replaced at runtime through commands such as subst, format and string map.

This last method is illustrated in Section 10.8.1.

10.7.2. Capturing namespace contexts in callbacks

One consideration that arises when constructing scripts that will be passed as callbacks is definition of the namespace in which the callback should execute. Most commands such as after execute scripts in the global context. To execute the callback in another context, some additional steps are needed. Since we have not discussed namespaces yet, we will postpone a discussion of this topic to Section 12.2.1.

10.8. Metaprogramming

What is *metaprogramming*? Roughly speaking, metaprogramming involves writing a program **that in turn writes a program** to do the desired task. In some cases metaprogramming makes for simpler code while in others

it optimizes performance by generating specialized code at runtime. Tcl lends itself naturally to this style of programming as we already seen with some simple examples involving procedure redefinitions and such. We will now present some additional illustrations of metaprogramming. We will see another example in Section 20.9.2.

10.8.1. Procedures with initializers

In Section 3.5.6 we saw how a procedure could redefine itself to do one-time initialization. That required some boilerplate code to be written for every procedure that wanted to do this. This boilerplate followed the pattern

```
proc NAME {ARGLIST} {
    INITCODE
    proc NAME {ARGLIST} {
        BODY
    }
    tailcall {*}[info level 0]
}
```

We can generalize this by introducing an enhanced form of the proc command, which we will imaginatively call proc_ex, that will generate this boilerplate for us. Our new command takes an additional argument which is the initialization script and has the form

```
proc_ex PROCNAME ARGS INIT BODY
```

The command will create a new procedure called *PROCNAME* just as the proc command does except that the **first** time it is called the created procedure will run the *INIT* script before running *BODY*.

We can implement proc_ex by just using the above pattern as a template for defining the target procedure and substituting for the variable parts in the template. For example, the following implementation uses string map to do the needful.

```
proc proc_ex {name arglist initcode body} {
    if {![string match ::* $name]} {
        set ns [uplevel 1 {namespace current}]
        set name ${ns}::$name
    }

    set template {
        proc NAME {ARGS} {
            INIT
            proc NAME { ARGS } { BODY }
            tailcall {*}[info level 0]
        }
    }
    set replacements [list NAME $name ARGS $arglist INIT $initcode BODY $body]
    eval [string map $replacements $template]
}
```

The first part of the procedure merely ensures the name of the procedure to be defined is appropriately qualified irrespective of the namespace context of the caller. In the second part, we take the generalized procedure template we laid out above, replace the variable parts of the template with the actual values using string map, and then execute the generated procedure definition.

To understand how it works, let us use it in an example and introspect the generated code.

```
proc_ex say_hello {message} {
    puts "Loading package msgcat"
    package require msgcat
} {
    puts [msgcat::mc $message]
```

```
}
```

The above defines a `say_hello` procedures whose implementation we can examine with `info body`.

```
% info body say_hello
→

        puts "Loading package msgcat"
        package require msgcat

                proc ::::say_hello { message } {
        puts [msgcat::mc $message]
    }
                tailcall {*}[info level 0]
```

The formatting of the generated code is a bit of a mess as we have not bothered to prettify it. However, we can still follow what's going on. The implementation of `say_hello` (generated by `proc_ex`) runs the initialization code and then redefines itself with the main procedure body. It finishes by calling this redefined version of itself.

Let us call our procedure for the first time.

```
% say_hello "Hello World!"
→ Loading package msgcat
  Hello World!
```

As you can see the initialization code is executed before the main body. Moreover, if we examine the body of the procedure, we find it has changed.

```
% info body say_hello
→
        puts [msgcat::mc $message]
```

And naturally, when we invoke it a second time, there is no attempt to load the `msgcat` package.

```
% say_hello "Hello again!"
→ Hello again!
```

To recap the benefits of our `proc_ex` command,

- We can postpone any expensive initialization (loading `msgcat` in our example) until the time it is actually needed.
- Subsequent calls after the first are streamlined as they neither attempt to load the package nor even have to check for the same.
- Because we have wrapped this one-time initialization within our `proc_ex` procedure, it is simple to use. A procedure that requires one-time initialization does not need to reinvent the wheel.

The `string map` command is only one way of generating a script from a template. You could also use commands like `subst` or `format`. An implementation of `proc_ex` that uses `format` is shown below.

```
proc proc_ex {name arglist initcode body} {
    if {![string match ::* $name]} {
        set ns [uplevel 1 {namespace current}]
        set name ${ns}::$name
    }
    eval [format {
        proc %1$s { %2$s } {
            %3$s
            proc %1$s { %2$s } { %4$s }
            tailcall {*}[info level 0]
        }
    } $name $arglist $initcode $body]
}
```

10.8.2. Parsing data by transmutation to code

A common task in programming is parsing of structured data or text. One technique used for this purpose that you will see in the Tcl world is to transform the data into a Tcl script that embeds Tcl commands within the data and then execute the generated script.

This is easiest explained through an example. We will use the following Tcler's Wiki[3] code derived from Stephen Uhler's famous 4-line HTML parser.

```
proc html_parse {html callback} {
    set re {<(/?)([^ \t\r\n>]+)[ \t\r\n]*([^>]*)>}
    set sub "\}\n[list $callback] {\\2} {\\1} {\\3} \{"
    regsub -all $re [string map {\{ \&ob; \} \&cb;} $html] $sub script
    eval "$callback PARSE {} {} \{ $script \}; $callback PARSE / {} {}"
}
```

The intent is to transform the HTML text to a Tcl script where each HTML tag results in the invocation of a command which is passed the tag as a parameter. To understand what html_parse is doing, let us **interactively** execute a slightly simplified version of the above implementation line by line.

We start off by defining variables corresponding to the arguments passed to html_parse. These will serve as the "arguments" to our interactive execution of the procedure.

```
% set html {
    <p class='important'>Something <b>really</b> important.</p>
    <p>A second paragraph</p>
}
→
    <p class='important'>Something <b>really</b> important.</p>
    <p>A second paragraph</p>
% set callback html_cb ❶
→ html_cb
```

❶   We will define the html_cb callback procedure later

The html_parse procedure first defines a regular expression that matches both opening and closing HTML tags.

```
% set re {<(/?)([^ \t\r\n>]+)[ \t\r\n]*([^>]*)>}
→ <(/?)([^ \t\r\n>]+)[ \t\r\n]*([^>]*)>
```

Next, html_parse defines the substitution that will convert each tag to a call to the callback command.

---

[3] http://wiki.tcl.tk

```
% set sub "\}\n[list $callback] {\\2} {\\1} {\\3} \{"
→ }
  html_cb {\2} {\1} {\3} {
```

Our intent is that a tag of the form <body> will result in a call to html_cb (the callback procedure passed in) with the <body> passed as an argument along with the succeeding text. We will see this in a minute.

Now we use the regsub command to transform the HTML text into an equivalent Tcl script fragment.

```
% regsub -all $re $html $sub script
→ 6
% puts $script
→
        }
  html_cb {p} {} {class='important'} {Something }
  html_cb {b} {} {} {really}
  html_cb {b} {/} {} { important.}
...Additional lines omitted...
```

This is actually a script fragment, not an entire script (hence the leading and trailing brace characters). It calls the specified command passing four parameters. The complete script that is passed to eval (the last command in html_parse) would then look like this:

```
% puts "$callback PARSE {} {} \{ $script \}; $callback PARSE / {} {}"
→ html_cb PARSE {} {} {
        }
  html_cb {p} {} {class='important'} {Something }
  html_cb {b} {} {} {really}
  html_cb {b} {/} {} { important.}
...Additional lines omitted...
```

Thus invoking our 4-line HTML parser as follows

```
html_parse $html html_cb
```

will result in the script printed above being generated and evaluated. We can now gain a better understanding of how the script works. It transforms the passed HTML text such that each HTML begin and end tag is converted to a call to the passed callback command, html_cb in our example, with four arguments:

- the name of the tag, such as p or b,
- an argument that is empty if it is the beginning of the tag and / if it corresponds to the tag termination
- any attributes for the tag
- The text content until the start of the next tag.

The script uses a special tag name PARSE that allows the callback command to recognize the start and end of parsing for any required state initialization or finalization. All we need to do now is define our callback command to do the desired parsing action. Let us define a trivial callback to simply convert all tags to upper case.

```
proc html_cb {tag place attrs content} {
    if {$tag ne "PARSE"} {
        if {$attrs ne ""} {
            set attrs " $attrs"
        }
        puts -nonewline "<$place[string toupper $tag]$attrs>$content"
    }
}
```

Now examine the output when our sample HTML fragment is processed as below.

```
% html_parse $html html_cb
→ <P class='important'>Something <B>really</B> important.</P>
      <P>A second paragraph</P>
```

As another example, here is a HTML to Latex markup converter (that only understands two tags and ignores the rest). Again, this is only illustrative of the technique and does not consider even basic requirements such as escaping of special characters.

```
proc html_latex {tag place attrs content} {
    switch -exact -nocase -- $tag {
        B {
            if {$place eq ""} {
                append_paragraph "\{\\em $content"
            } else {
                append_paragraph "\}$content"
            }
        }
        P {
            flush_paragraph
            append_paragraph $content
        }
        PARSE {
            if {$place eq ""} {
                append_paragraph $content
            } else {
                flush_paragraph
            }
        }
    }
}

proc append_paragraph {content} {
    if {[string length $content]} {
        append ::paragraph $content
    }
}

proc flush_paragraph {} {
    if {[info exists ::paragraph]} {
        set para [string trim $::paragraph]
        if {[string length $para]} {
            puts "$para\n"
        }
        unset ::paragraph
    }
}
```

We can then convert HTML to Latex input like this

```
% html_parse $html html_latex
→ Something {\em really} important.

  A second paragraph
```

Our HTML parser is simplistic and does not take into account all of HTML syntax details and idiosyncracies. It is meant to illustrate the "data to script" transform technique. The htmlparse module in Tcllib [4] provides a more robust HTML parsing solution based on the same technique. An even faster, standards compliant solution for parsing HTML is the tdom package described in Section A.6. This is however a binary extension.

> ⚠ This technique of transforming data to a script has potential security issues when the data comes from an unknown (and potentially malicious) source. Although this can be guarded against with proper escaping and quoting of the input data, it is advisable to execute the generated script in a safe interpreter as discussed in Chapter 20.

10.8.3. Metaprogramming for specialization

Another use of metaprogramming that you will find in advanced Tcl scripts is for specializing code for specific situations. Again, this is best illustrated with an example, in this case from the implementation of Tcl itself.

One of the functions of the clock command in Tcl is to generate a string representation of time using a caller-specified formatting string. For example,

```
% clock format [clock seconds] -format "The time is %H:%M" -locale en -gmt 1
→ The time is 06:15
```

Formatting the time as above requires parsing of the passed format string based on appropriate time zone, locale and calendar information. The requested values such as month, hour of the day, etc. are then filled in based on the time value passed in to the command. Since this can be an expensive operation relatively speaking, the clock implementation includes an optimization where it generates separate procedures **on demand** that are customized for a specific combination of format string, time zone and localization. The next time the command is called with the same combination of formatting and locale arguments, this constructed procedure is directly called without the need for the first parsing step. In practice, an application generally uses only a few different combinations of formatting and therefore this optimization is fairly effective.

Here is the pseudocode for the clock format procedure that leaves out the details that are not directly relevant. The explanations follow below.

```
proc ::tcl::clock::format {args} {
    variable FormatProc

    lassign [ParseFormatArgs {*}$args] format locale timezone
    set clockval [lindex $args 0]

    .. Initialize / configure time zone settings ..

    set procName formatproc'$format'$locale

    if {[info exists FormatProc($procName)]} {
        set procName $FormatProc($procName)
    } else {
        set FormatProc($procName) \
            [ParseClockFormatFormat $procName $format $locale]
    }

    return [$procName $clockval $timezone]
}
```

The command parses its arguments to extract the format string, locale, time zone and time value to be formatted. It then constructs a procedure name based on the format string and locale and checks to see if such a procedure

---

[4] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

has already been constructed. If not, it calls `ParseClockFormatFormat` to construct the procedure and caches it in the `FormatProc` array. Finally, the cached procedure is invoked to do the real work.

We can dump the `FormatProc` array to see the names of the constructed procedures.

```
% print_list [array names ::tcl::clock::FormatProc]
→ ::tcl::clock::formatproc'The time is %H\:%M'en
```

Invoking the format command with another format will add to this array.

```
% clock format [clock seconds] -format "Today is %D"
→ Today is 07/04/2017
% print_list [array names ::tcl::clock::FormatProc]
→ ::tcl::clock::formatproc'The time is %H\:%M'en
  ::tcl::clock::formatproc'Today is %D'c
```

Finally, we can use the `reconstruct` procedure that we defined in Chapter 3 to dump the procedure definitions.

```
% foreach proc_name [array names ::tcl::clock::FormatProc] {
    puts [reconstruct $proc_name]
}
→ proc {::tcl::clock::formatproc'The time is %H\:%M'en} {clockval timezone} {

                variable TZData
                set date [GetDateFields $clockval  $TZData($timezone)  2361222]

        return [::format {The time is %02d:%02d}  [expr { [dict get $date localSeconds]...
                                  / 60
                                  % 60 }]]
    }
  proc {::tcl::clock::formatproc'Today is %D'c} {clockval timezone} {

                variable TZData
                set date [GetDateFields $clockval  $TZData($timezone)  2299161]

        return [::format {Today is %02d/%02d/%04d}  [dict get $date month] [dict get $d...
    }
```

The key point to note is that these constructed procedures contain no code for parsing at all making them very efficient for subsequent calls using the same format string and locale.

The hard work of constructing these specialized procedures is done by the `ParseClockFormatFormat` procedure. It uses the techniques we described in Section 10.7.1.2. We will not describe it here but you would do well to study its implementation in the `clock.tcl` file in the Tcl installation.

10.8.4. Metaprogramming for generalization

Consider writing a procedure that given a pair of lists, returns a list of all possible pairs containing an element from each list.

```
proc pairs {la lb} {
    set res {}
    foreach a $la {
        foreach b $lb {
            lappend res [list $a $b]
        }
    }
    return $res
}
```

An example invocation would be

```
% pairs {a b} {1 2 3}
→ {a 1} {a 2} {a 3} {b 1} {b 2} {b 3}
```

What if we wanted to generate triples from three lists instead? That would be easy — we just add another nested loop. But what if we wanted to generalize the procedure to be able to take an **arbitary** number of list arguments? And not limit the generalization to just generating list pairs? This metaprogramming example [5], shown below, from Tcler's Wiki [6] is one such generalization.

The syntax of the `forall` command that we will write is very similar to that of the built-in `foreach` command. The difference is that `forall` processes the lists in **nested** fashion while the latter processes them in parallel in the same iteration.

```
forall VAR LIST ?VAR LIST ...? BODY
```

The implementation of the command basically constructs a script containing `foreach` loops nested to the required depth with the innermost loop containing the *BODY* script to be executed. The constructed script is then evaluated in the caller's scope.

```
proc forall args {
    if {[llength $args] < 3 || [llength $args] % 2 == 0} {
        return -code error "wrong \# args: should be \"forall varList list ?varList list \
            ...? body\""
    }
    set body [lindex $args end]
    set args [lrange $args 0 end-1]
    while {[llength $args]} {
        set varName [lindex $args end-1]
        set list    [lindex $args end]
        set args    [lrange $args 0 end-2]
        set body    [list foreach $varName $list $body]
    }
    uplevel 1 $body
}
```

We can emulate out `pairs` command as below.

```
% forall x {a b} y {1 2 3} { lappend res [list $x $y] }
% set res
→ {a 1} {a 2} {a 3} {b 1} {b 2} {b 3}
```

But now, with this generalized procedure, we are not limited to producing pairs. We can do something more creative, like producing strings instead! And this time with a different number of lists.

```
% set res ""
% forall x {a b} y {1 2 3} z {M N} { append res $x$y$z }
% set res
→ a1Ma1Na2Ma2Na3Ma3Nb1Mb1Nb2Mb2Nb3Mb3N
```

Finally, here is the generalization of our `pairs` procedure. Instead of producing pairs, it will produce tuples composed of elements from an arbitrary number of lists. It uses `forall` under the covers to generate the combinations.

---

[5] http://wiki.tcl.tk/2546
[6] http://wiki.tcl.tk

```
proc tuples args {
    set res {}
    set listargs {}
    set body "lappend res \[list"
    foreach arg $args {
        set loopvar v[incr i]
        append body " \$$loopvar"
        lappend listargs $loopvar $arg
    }
    append body "\]"
    forall {*}$listargs $body
    return $res
}
```

And to prove it works as desired,

```
% tuples {1 2} {a b c}
→ {1 a} {1 b} {1 c} {2 a} {2 b} {2 c}
% tuples {1 2} {a b c} {X Y}
→ {1 a X} {1 a Y} {1 b X} {1 b Y} {1 c X} {1 c Y} {2 a X} {2 a Y} {2 b X} {2 b Y} {2 c X}...
```

Now, for this specific problem, there are easier ways to code directly without metaprogramming but what the forall procedure has done for us is **to make it easy to iterate over lists in nested fashion in a very generalized way without having to write custom code every time**.

10.9. Command history: history

When running in interactive mode, Tcl keeps a history of all commands entered interactively. These commands can then be recalled or otherwise manipulated with the history command. This is an ensemble command with subcommands for various operations.

> The history commands described in this section are almost never directly used by applications. Really the only time they are used is in writing a custom shell, similar to tclsh, wish or tkcon, that accepts input commands from the user and wants to provide command history and recall similar to those shells. You can therefore comfortable skip this section.

When invoked without any arguments, the command returns a human-readable representation of the history.

```
% puts "This is the first command"
→ This is the first command
% set i 1
→ 1
% incr i
→ 2
% history
→     1  puts "This is the first command"
      2  set i 1
      3  incr i
      4  history
```

The above history command is just a short form of the history info command which allows you to optionally specify the number of entries to be returned (the latest entries are returned).

```
% history info 2
→     4  history
      5  history info 33
```

Note each command has a sequence number associated with it. We can check the sequence number of the **next** entry that will be added with the history nextid command.

```
% history nextid
→ 7
```

The command returned 7 because the history nextid command was itself the sixth command.

Entries in the command history are referred to as command history events (not to be confused with events as described in Chapter 15). They can be referenced in multiple ways:

- A positive integer is interpreted as the sequence number of an entry in the command history
- A negative integer is interpreted relative to the current sequence number
- Any other string is first searched for backward as an exact prefix of an entry in the history and if not found, is searched as a glob pattern.

We can use these forms with any of the history subcommands that operate on individual entries. For example, the history event command returns the corresponding entry from the history.

```
% history event ❶
→ history nextid
% history event 2
→ set i 1
```

❶ By default the previous entry in the history is returned

Any entry in the command history can be re-executed with the history redo command. Again, you can use any of the forms above for referencing the target entry.

```
% history redo -8 ❶
→ This is the first command
% history redo 2 ❷
→ 1
% history redo incr ❸
→ 2
% history redo ❹
→ 3
```

❶ Execute command 8 entries back
❷ Execute command with sequence number 2
❸ Execute command matching incr
❹ Repeat previous command

It is also possible to modify the command history in various ways. The history add command adds a command to the history and optionally executes it if the exec argument is specified.

```
% history add [list puts "This will not print"] ❶
% history add [list puts "This will print"] exec ❷
→ This will print
```

❶ Command is not evaluated
❷ Command is evaluated

The command history change on the other hand modifies an existing command.

```
% history event 15
→ history add [list puts "This will not print"]
% history change [list puts "This will now also print"] 15
→ puts {This will now also print}
% history event 15
→ puts {This will now also print}
```

The entire command history can be erased by calling history clear. Further commands added to the history will begin with sequence number 1.

```
% history clear
% puts "History never repeats itself!"
→ History never repeats itself!
% history
→      1  puts "History never repeats itself!"
       2  history
```

The limit on the size of the command history can be retrieved with history keep. An optional argument can be supplied to change this limit.

```
% history keep
→ 20
% history keep 100
→ 100
```

10.10. Tcl internals

Before ending this chapter, let us take a brief look at some of Tcl's internals; or to be more precise, internals of the "official" Tcl implementation. This material is mostly for the benefit of those like to peek under the hood, although some of it serves as background for our discussion of Tcl performance in Chapter 24.

We will interactively explore Tcl internals with the help of two commands, tcl::unsupported::representation and tcl::unsupported::disassemble. Notice both lie within the tcl::unsupported namespace which implies their functionality, or even presence, in future releases should not be counted on. However, they are perfectly fine for our purposes. We will alias (see Section 20.3) them into the global namespace to save some typing.

```
interp alias {} representation {} tcl::unsupported::representation → representation
interp alias {} disassemble {} tcl::unsupported::disassemble      → disassemble
```

If you are following along and typing commands in an interactive shell, you should disable Tcl's command history feature as it will interfere with our exploration below. The easiest way to do this is by redefining the history command to do nothing.

```
% rename history ::tcl::history
% proc history args {}
```

10.10.1. How values are stored

The representation command dumps the internal representation of a Tcl value in human readable form. We will use it to examine, as a simple case, how a constructed string might be stored internally and contrast that with

a constructed list value. (We say **might** because as we shall see, this can change depending on how the value is used.)

Compare the output of the following two commands:

```
% representation [string repeat abc 2]
→ value is a pure string with a refcount of 2, object pointer at 0000000004298AA0, string
    ↳ representation "abcabc"
% representation [lrepeat 2 abc]
→ value is a list with a refcount of 2, object pointer at 000000000349EEE0, internal
    ↳ representation 00000000041EB450:0000000000000000, no string representation
```

The *object pointer* in the output is the memory address where the internal structure for the constructed value is stored. This structure has the type Tcl_Obj in the Tcl source code and has the following primary fields:

- An internal type, shown above as a *pure string* and *list* respectively.

- An **optional** *string representation* as seen in the output of the first command above.

- An **optional** type-specific *internal representation* as seen in the output of the second command above.

- A *refcount* field that holds the reference count for that Tcl_Obj. Tcl uses reference counting internally for memory management.

We expand on these fields below.

10.10.1.1. Understanding internal representations

Although Tcl semantics are defined in terms of *everything is a string*, for performance reasons Tcl maintains an internal representation more suitable for computation when necessary. Tcl stores values internally in a Tcl_Obj C structure. The internal representation is stored as two fields within this Tcl_Obj structure and may change based on the operations invoked on the value.

We illustrate this for integer values.

```
% set ival 99
→ 99
% representation $ival
→ value is a pure string with a refcount of 4, object pointer at 0000000004298A40, string
    ↳ representation "99"
```

Because we simply assigned a literal to ival, and have not invoked any operations on it, its type is shown as a *pure string*, the *pure* indicating that there is no other representation associated with it. Also note that there is no mention of an internal representation in the command's output.

Now we look at how that changes once we invoke an integer operation like incr on it. **Type both commands on the same line, separated by a semi-colon, for reasons we detail below.**

```
% incr ival ; representation $ival
→ value is a int with a refcount of 3, object pointer at 0000000004298A40, internal
    ↳ representation 0000000000000064:0000000004299010, no string representation
```

The invocation of an integer operation creates an internal representation of type int held within the Tcl_Obj structure. Thus the next time an integer operation is to be performed on the value, there is no cost incurred converting a string to an integer.

Note the presence of an *internal representation* field where the first value 0x64 directly stores the integer value. The second field is not used for integers and contains a random value.

Furthermore, there is no string representation for the incremented value. Tcl will only generate one when required. A string or I/O operation will force the string representation to be created. This is also why we placed the

representation command on the same line as the incr. Otherwise, when the shell printed the result of incr, the string representation would have been generated and we would not have been able to show the intermediate step.

If we were to now print the value of ival, a string representation of its value would be created.

```
% puts $ival ; representation $ival
→ 100
   value is a int with a refcount of 3, object pointer at 0000000004298A40, internal
     ↳ representation 0000000000000064:0000000004299010, string representation "100"
```

Note there is now a string representation **in addition** to the internal integer representation.

Invoking a list operation, like llength will transform the internal representation yet again.

```
% llength $ival ; representation $ival
→ value is a list with a refcount of 3, object pointer at 0000000004298A40, internal
     ↳ representation 0000000003186F30:0000000000000000, string representation "100"
```

Again the internal representation field is present but has changed. It now holds a pointer to a list structure in memory.

Throughout the above sequence of operations, the object pointer has stayed the same, as has the reference count. Only the type of the internal representation has changed. This is commonly referred to as *shimmering*.

Here is a short example of how shimmering to an appropriate type on the fly leads to more efficient operation.

```
% proc rgb {color} {
      set colors {
          red    0xff0000
          green 0x00ff00
          blue   0x0000ff
      }
      puts [representation $colors]
      return [dict get $colors $color]
}
% rgb red
→ value is a pure string with a refcount of 5, object pointer at 000000000349E6D0, string
     ↳ representation "
            red ..."
  0xff0000
```

When the procedure is compiled, the value assigned to colors is stored as a string as we see in the output from representation on the call to rgb. However, a second call reveals that the internal representation is now a dictionary.

```
% rgb red
→ value is a dict with a refcount of 5, object pointer at 000000000349E6D0, internal
     ↳ representation 00000000037ABD50:0000000000000000, string representation "
            red ..."
  0xff0000
```

The dict get operation on the first call shimmered the internal representation to a dictionary. Thus on subsequent calls, the command is spared the expense of converting the string to a dictionary before looking it up.

In the above example, the shimmering of the colors value happens just once — on the first call to rgb. Thereafter it is accessed as and remains internally stored as a dictionary. On the other hand repeated shimmering of values between different types not only entails a performance hit, it is often indicative of a design or conceptual flaw, for example using a string operation like append in place of the list operation lappend.

Tcl uses internal representations for many types of objects. There is no means to enumerate them all since this is supposed to be an implementation detail and not intended to be visible to scripts at all. A small subset of these is shown below.

```
% representation  [dict create key val]  ❶
→ value is a dict with a refcount of 4, object pointer at 00000000034A51E0, internal repr...
%
% representation [pwd]                    ❷
→ value is a path with a refcount of 3, object pointer at 00000000032A8370, internal repr...
%
% set pos end-1 ; lindex {1 2} $pos
→ 1
% representation $pos                     ❸
→ value is a end-offset with a refcount of 3, object pointer at 00000000034A51E0, interna...
%
% set code "set x 1" ; eval $code
→ 1
% representation $code                    ❹
→ value is a bytecode with a refcount of 3, object pointer at 00000000032A90F0, internal ...
%
% set re ".*" ; regexp $re "a string"
→ 1
% representation $re ❺
→ value is a regexp with a refcount of 3, object pointer at 0000000003497AF0, internal re...
```

❶   Dictionary
❷   File paths
❸   List indices
❹   Compiled byte code
❺   Compiled regular expression state machine

Before leaving our discussion of internal representations, we note once again that these multiple internal representations are primarily for execution efficiency. At the script level, you generally do not need to be aware of them except in performance-sensitive areas where shimmering between multiple internal representation can be detrimental. This is never a problem if type-appropriate commands are used (for example, using list commands to operate on list values).

10.10.1.2. Data storage and reference counting

Having looked at using representation for the purpose of exploring the different internal types, we now use it to delve into Tcl's memory management. Tcl uses reference counting to manage its values. When a variable is assigned to another, rather than making a copy of the value contained in it, the reference count for the Tcl_Obj holding the value is incremented and the same Tcl_Obj value is assigned to the target variable.

```
% set avar "some value"
→ some value
% set bvar $avar
→ some value
```

Now when we look at the representations for avar and bvar, we will see that both point to the same Tcl_Obj structure in memory.

```
% representation $bvar
→ value is a pure string with a refcount of 4, object pointer at 000000000348D550, string
    ↳ representation "some value"
% representation $avar
→ value is a pure string with a refcount of 4, object pointer at 000000000348D550, string
    ↳ representation "some value"
```

The reference count for the Tcl_Obj value includes references from each of the two variables. In addition, any time a value is passed as an argument to a command (including to representation), its reference count is incremented to reflect its presence on the call stack. Correspondingly, it is decremented when the command returns. This "hidden" reference can have performance implications as we discuss in Section 24.2.3.

If we were to assign a different value to one of the variables, the reference count for the current assignment would be decremented.

```
% set bvar "some other value"
→ some other value
% representation $avar
→ value is a pure string with a refcount of 3, object pointer at 000000000348D550, string
    ↳ representation "some value"
% representation $bvar
→ value is a pure string with a refcount of 3, object pointer at 000000000349EA60, string
    ↳ representation "some other value"
```

Notice that the two variables now point to different Tcl_Obj locations in memory and reference counts have been adjusted accordingly.

10.10.1.3. The literal table

The representation command can also be used to look at another aspect of the current Tcl implementation — the literal table.

Tcl internally maintains a table of all literal values encountered. In an effort to save memory, when compiling a procedure Tcl checks if any literals it encounters are already in this table and if so simply references them instead of creating a new Tcl_Obj with the same literal value. The following snippet demonstrates this.

```
% proc p1 {} {representation "just a literal"}
% p1
→ value is a pure string with a refcount of 4, object pointer at 000000000349E1F0, string
    ↳ representation "just a literal"
% proc p2 {} {representation "just a literal"}
% p2
→ value is a pure string with a refcount of 5, object pointer at 000000000349E1F0, string
    ↳ representation "just a literal"
```

Notice that even though the two literals are used in different procedures, they both point to the same Tcl_Obj in memory.

10.10.2. How code is executed

Having looked at data, let us turn to code execution and look at the sequence of steps required to execute a procedure. In simplifed terms,

1. At the time of procedure definition, Tcl does very little work per se. The body of the procedure is simply stored in a location associated with the procedure name. It is a plain string at this point; there is no check any kind for syntax errors etc.

2. When a procedure is invoked, a check if made to see if it has a **valid** compiled form. If not, the procedure body is compiled into byte code. You can think of byte codes as being an instruction set for a virtual machine — the Tcl byte code interpreter. Compiling into byte code saves on the parsing step the next time the procedure is invoked resulting in faster execution.

3. Any supplied arguments are pushed onto the call stack and the procedure is invoked.

Of these steps the second one is the one of most interest and we can use the disassemble command to examine what the compiled body of a procedure looks like. We will look at the byte code generated for a simple procedure that returns the first word in a string.

```
proc demo {line} {
    set words [split $line $::splitter]
    return [lindex $words 0]
}
```

We can then look at the compiled form of this procedure with the disassemble command.

There is also a getbytecode command that is similar but returns the disassembled code as a dictionary instead of in human readable form.

After defining the above procedure, type the following command at the Tcl shell prompt and examine the resulting output.

```
% tcl::unsupported::disassemble proc demo
→ ByteCode 0x000000000403D300, refCt 1, epoch 16, interp 0x0000000003B661A0 (epoch 16)
    Source "\n    set words [split $line $::splitter]\n    return [..."
    Cmds 4, src 70, inst 30, litObjs 2, aux 0, stkDepth 3, code/src 0.00
    Proc 0x0000000003F89A10, refCt 1, args 1, compiled locals 2
        slot 0, scalar, arg, "line"
        slot 1, scalar, "words"
    Commands 4:
        1: pc 0-11, src 5-39      2: pc 0-8, src 16-38
        3: pc 12-28, src 45-68     4: pc 21-27, src 53-67
    Command 1: "set words [split $line $::splitter]..."
    Command 2: "split $line $::splitter..."
      (0) push1 0  # "split"
      (2) loadScalar1 %v0  # var "line"
      (4) push1 1  # "::splitter"
      (6) loadStk
      (7) invokeStk1 3
      (9) storeScalar1 %v1  # var "words"
      (11) pop
    Command 3: "return [lindex $words 0]..."
      (12) startCommand +17 2  # next cmd at pc 29, 2 cmds start here
    Command 4: "lindex $words 0..."
      (21) loadScalar1 %v1  # var "words"
      (23) listIndexImm 0
      (28) done
      (29) done
```

Going through the disassembled listing line by line will provide us with a basic understanding of the byte code interpreter.

The first part of the disassembly provides summary information about the compiled procedure. The fields are shown in Table 10.5.

Table 10.5. Disassembly header

Field	Description
ByteCode	Memory address where the compiled byte code is stored.
interp	Memory address of the Tcl interpreter structure owning the byte code
refCt	The number of references to the byte code structure which is reference counted.
epoch	The compiler epoch. See Section 10.10.2.1.
Source	The Tcl source code from which the byte code was compiled.
Cmds	The number of script level commands in the compiled source
src	The number of characters in the script
inst	The number of bytes in the generated byte code
litObjs	The size of the local literals table. See Section 10.10.2.2
stkDepth	The maximum stack depth utilized by this byte code fragment. This is used at run time to preallocate the required stack space before the byte code is executed.
Proc	The address of the procedure descriptor associated with the byte code
args	The number of parameters defined by the procedure.
compiled locals	Size of the local variable table. See Section 10.10.2.3.
slot	Entries in the local variable table. See Section 10.10.2.3.

The remaining lines in the disassembly are discussed in the sections below.

10.10.2.1. Compiler epochs

The epoch field denotes a *compiler epoch*. Earlier we mentioned in passing that a procedure may have associated byte code which is invalid and needs to be recompiled. This can happen because of Tcl's dynamic nature. For instance, consider what happens when a built-in command like lindex is redefined. Calls to many built-in commands are compiled to a sequence of inline byte code instructions like listIndexImm above. If this command is redefined, the compiled byte code for our procedure would not longer be correct as listIndexImm would (likely) not be the correct implementation of the redefined lindex. Tcl detects this situation by maintaining a compiler epoch. This epoch is incremented on any action, such as command definition, that would invalidate any compiled byte code. The epoch at the time a procedure is compiled is also stored with the compiled byte code as shown above. When the procedure is called, the procedure is recompiled if its compilation epoch is not the same as the current compiler epoch.

10.10.2.2. The local literals table

As we saw in Section 10.10.1.3, the compiler maintains a table of literals so that they can be shared across the interpreter. Correspondingly, the compiled byte code unit maintains a table that points to locations in this global literal table. The purpose of this indirection is that the corresponding byte code instructions can use a single byte to index into the local literal table which is not possible for the global table which tends to be quite large. For example, the byte code instruction

```
push1 0
```

pushes the first (i.e. index 0) from the literal table on to the evaluation stack.

Note that the literal table is not just used for literal operands that appear in the script but also for things like names of commands and variables. In our example, there are two literals in the table as indicated by the value of the litObjs field. The disassembler-generated comments for the push1 instructions at locations 0 and 4 indicate these correspond to the command name split and global variable ::splitter respectively. Notice though that there is no literal object corresponding to the lindex command. We shall shortly see why that is so.

10.10.2.3. The local variable table

Local variables and arguments are stored in *slots* allocated in the call frame when a procedure is invoked. Our disassembly shows two slots defined for our procedure, one for the line argument and the other for the words local variable.

```
slot 0, scalar, arg, "line"
slot 1, scalar, "words"
```

The key thing to note about local variables is that access to slots is much faster than access to global and namespace variables. We will see why in the next section.

10.10.2.4. The stack machine

The Tcl byte code interpreter is stack-based (as opposed to register-based) where byte code instructions operate on operands placed on an *evaluation stack*. This evaluation stack is internal to the byte code interpreter and not to be confused with the C-stack or procedure call stack discussed earlier. Moreover, each Tcl interpreter (not to be confused with the byte code interpreter!) has its own evaluation stack.

The call to split in our example is compiled to the byte code

```
Command 2: "split $line $::splitter..."
  (0) push1 0  # "split"
  (2) loadScalar1 %v0  # var "line"
  (4) push1 1  # "::splitter"
  (6) loadStk
  (7) invokeStk1 3
  (9) storeScalar1 %v1  # var "words"
  (11) pop
```

Here the actual call to split takes place via the invokeStk1 instruction. The corresponding three arguments to be pushed on the stack are the **name** of the command to be invoked, the **value** of the line local variable, and the **value** of the ::splitter global variable.

- The command name, split, is stored in the local literals table at index 0. The push1 0 instruction pushes it onto the evaluation stack.

- The local variable line is stored at in the local variable slot 0. The loadScalar1 instruction pushes the value of the variable at this slot onto the stack.

- The global variable ::splitter needs to be handled differently. The variable **name** is stored at index 1 in the local literal table. However, we need to pass the **value** of this variable and not the name. This is done in two steps. First, the push1 1 instruction places the literal table entry at index 1, i.e. the literal ::splitter, on to the top of the stack. The loadStk instruction then looks up the variable by name and replaces the top of the stack with the value of the variable.

This now explains why access to local variables is much faster than to globals. The former is a direct indexed look-up into the local slots whereas the latter involves locating the global via a hash table (as part of execution of loadStk) and then retrieving its value.

The invokeStk1 instruction looks up the supplied command name and executes it, passing the arguments on the stack. The result of the command replaces **all** arguments passed in. The storeScalar1 instruction then stores the value on the top of the stack into the local variable slot at index 1. Finally, the stack is cleaned up by popping the topmost entry with pop.

10.10.2.5. Inlined byte code

The byte code instruction invokeStk1 used to invoke commands needs to resolve the command name in the current namespace context to locate the appropriate C function to call, wrap arguments into a form accessible to the C function, arrange for exception handling and so on. These operations make invocation of commands

relatively expensive. As an optimization therefore, Tcl will directly inline byte code for many built-in commands at the call site itself. In our example, contrast the call to split with the call to lindex. The latter does not involve a call to invokeStk at all. Instead the byte code sequence implementing the command, a single instruction listIndexImm in this case, is directly inserted in the procedure body.

```
(21) loadScalar1 %v1   # var "words"
(23) listIndexImm 0
```

At the time of compilation, before inlining the command, the compiler ensures that the name resolves to the built-in command and not some procedure of the same name. Moreover as discussed before, redefinition of the command invalidates the generated byte code by bumping the compiler epoch. This ensures the inlined code is valid whenever in use.

You might have noticed the inlined command is prefixed with a startCommand instruction. This is present to take care of some special checks that are not needed with non-inlined invocation because the invokeStk1 command internally takes care of them. These checks include verifying that the inlined byte code is still valid by checking the compiler epoch and that any set interpreter limits are not crossed.

Note that at some point in the future, it is possible that the split command may have an inline implementation as well. This would depend on the complexity of the implementation and how commonly the command is used in programs.

10.10.3. Precompiled byte code: tbcload package

ActiveState's commercial Tcl Development Kit has the ability to save compiled byte code in files. These files can then be loaded with the tbcload package similar to how the source command evaluates Tcl scripts.

It should be noted that this functionality is primarily for the purposes of hiding the source code in commercial Tcl applications. It does **not** improve performance of Tcl applications in any way.

Because this facility is not available in any open source distributions of Tcl, we do not discuss it further.

10.11. Chapter summary

We have come to the end of a necessarily long chapter, its length arising from the breadth of facilities Tcl offers for code execution. We covered basic control structures, dynamic evaluation of scripts, the call stack, command and variable tracing, metaprogramming and more.

We are still not done with Tcl's execution model though. In the next chapter we explore Tcl's powerful exception handling features and further in the book we will examine the more advanced facilities like the event loop, coroutines and threads.

10.12. References

TIP327
> *TIP #327: Proper Tailcalls*, Miguel Sofer, David S. Cargo, Tcl Improvement Proposal #327, http://www.tcl.tk/cgi-bin/tct/tip/327

11

Errors and Exceptions

> An error does not become truth by reason of multiplied propagation, nor does truth become error because nobody sees it.
>
> — Mahatma Gandhi

Dealing with unexpected failures is an important part of any non-trivial program. Failures may result from programming errors such as attempting to access an undefined variable, user actions such as attempts to open a protected file, hardware conditions etc. In this chapter we examine Tcl's facilities for handling errors and special conditions.

11.1. Dealing with failures

When such a failure or error occurs, software may deal with it in any of the following ways:

1. Ignore the issue. Stout denial, as Wodehouse would say, that a problem could possibly exist. This is not uncommon, for example failing to check for integer overflow even when the possibility exists. We will pretend we never want to do this.

2. Terminate the program. This is a fair strategy in some circumstances. If a file copy program does not have access to the target directory, it is perfectly reasonable for it to simply exit.

3. Report the error through a special result value, for example an empty string. This requires that there is some value that could not possibly be a valid result. The caller can then check for this value to determine if the command completed successfully. Alternatively, if no such "impossible" value exists, a global is checked or an additional call is made to check for an error.

4. The command may explicitly pass back a status code in addition to the command result. This may be through an additional named parameter into which the status is stored or by returning the status and result as a pair.

5. One last alternative is the subject of this chapter — *exceptions*. When an error or failure is detected, the code detecting the error *throws* or *raises* an exception. This causes the normal flow of execution to be aborted and control is passed back up the call stack until an *exception handler* is found that is defined for that exception. This exception handler is expected to take the appropriate actions to deal with the error condition.

Exceptions are the preferred mechanism for error reporting for several reasons:

- We really should not be considering the first alternative.
- Alternative 2 is a viable alternative only at the top application level.
- Pairing an explicit status code with every result, as in Alternative 4 makes for awkard programming. So also if all possible values are valid results in Alternative 3.
- Alternatives 3 and 4 both require an explicit check for errors. Unfortunately, it is all too common for the caller to forget to do this. Exceptions on the other hand cannot be ignored "by default".
- Exceptions do not clutter up the main logic with explicit error checks, making it easier to read and reason about the program.
- Exceptions make it easy to handle errors at an appropriate point in the call hierarchy, which is not necessarily at the point the error is discovered.

In addition to error handling, the exception mechanism has other uses in Tcl as well, such as for implementation of custom control structures that are on par with the built-in ones like for or while.

We will start our discussion of error handling and exceptions by describing the underlying return code mechanism on which they are based.

11.2. Return codes and the option dictionary

So far we have been happily working under the assumption that Tcl commands return a single result value (though the value itself may be a collection of multiple values). In reality, every command completion actually yields *three* values, the command *result*, an integer *return code*, and a *return options dictionary*.

The command result is the result value from invocation of a command that we have been making use of all along. For instance, the command

```
string toupper abc → ABC
```

completes with a result of ABC. Additionally, the above command completion also returns a return code of 0 and an associated return options dictionary. The return code can be roughly thought of as a status, with 0 indicating normal completion of the command. The return options dictionary contains additional information that is usually relevant only in the case of errors.

The command result is directly available to a script as we have seen. We will now look at how the return code and return options dictionary are retrieved, assigned and the associated semantics.

11.2.1. Return codes

Tcl executes a script, including procedures, eval arguments, if and while statements etc., as a sequence of commands. Each command completes with a result, a return code and the return options dictionary. A return code of 0 signifies normal completion of a command and Tcl continues execution with the next command in the script. For any other value of the return code, execution of the script stops and the result and return code from the last executed command becomes the result and return code of the script.

What happens next depends on the caller of the script. The caller may be the Tcl procedure evaluation code, a built-in command, looping or conditional commands like while and if, or even a user defined one. This caller may choose to take a specific action for certain return codes. Other codes that it chooses not to handle are then passed further up the call stack where they are handled in the same manner.

Let us illustrate the use of return codes and associated semantics through an example — the break command used for early termination of loops. Like all other commands in Tcl, break is not a special keyword as in other languages. It is simply a command like any other and returns a result and a return code. The **result value** for the break command is always an empty string and the **return code** is always 3 (we will show this in a bit). In the case of the break command, both values are implicit though that is not the case for all commands. **Now, how this return code is dealt with by the calling command is entirely up to it**. Tcl itself does **not** mandate any semantics on a return code value of 3. Any of the following actions might be taken in response:

- When invoked within a looping construct like while or foreach, the implementation of those commands treat the return code of 3 from the invocation of **any** command, not just break, as a signal to terminate the loop.

- In the Tk GUI extension, bindings for events such as mouse clicks allow multiple scripts to be registered. These are run sequentially on the occurence of the event. If any script returns a code of 3, Tk treats this as a directive to skip the remaining registered scripts.

- Within a catch command, a return code of 3 will result in the catch command returning 3 as its **result** (not its return code).

- Within the **outermost** level of a procedure body evaluation, any command returning a code of 3 will result in the procedure returning immediately with a return code of 1 / error (we will see what this means momentarily).

Thus each invoking command will treat a break command within its scope in a slightly different manner. The semantics are completely up to the command. Now naturally, the behaviour has to be documented by the command and similar commands should treat the same return code value in consistent fashion. It would be no good if the foreach command treated the return code 3 from break as a loop termination command while the lmap command treated it as a signal to repeat the last iteration!

The return code returned by a command may be any integer value but Tcl defines five specific values, and associated mnemonics, shown in Table 11.1. All other return codes have no defined semantics and are treated as custom return codes (see Section 11.3.2).

Table 11.1. Tcl-defined return codes

Code	Mnemonic	Description
0	ok	The command completed normally.
1	error	Indicates an error condition. We go into errors and error handling in great detail later in this chapter.
2	return	This return code signals the caller that it should stop its own execution and return control back to **its** own caller.
3	break	This return code expects callers to be looping constructs and signifies termination of the loop. As we saw in our introductory example, it may also be used in other situations where a sequence of commands is being executed to skip the remaining commands in the sequence. We stress again that this behaviour is **not** built into Tcl. It is dependent on how the command receiving the return code chooses to handle it.
4	continue	Like the break return code, this is used with looping constructs which are then expected to skip the remaining part of the current iteration and move on to the next one.

We will use the term *normal return* whenever a command completes with a return code of 0 / ok. Any other return code value will be termed as an *exceptional return* or just *exception*. Note that an exception is not necessarily an error (e.g break).

11.2.2. Return code propagation

Let us now take a closer look at how these return codes are propagated and how they control the flow of execution in a Tcl program.

The break and continue return codes

We will use the following simple while loop to demonstrate.

```
set i 0
while {1} {
    incr i
    if {$i == 1} {
        incr i
        continue
    }
    puts "i = $i"
    if {$i >= 4} break
}
→ i = 3
  i = 4
```

We will focus on the execution of the while body. In the first iteration of the loop,

- The command incr i is invoked. This command completes normally with a **result** of 1 (the value of i) and a **return code** of 0 / ok that signals the normal completion.

- Because the command completed normally, the evaluation of the while body continues with the next command, the if {$i == 2} … statement.

- As its condition evaluates to true, the if statement begins executing its body. (Here we are actually glossing over the fact that the condition evaluation itself involves return codes.)

- The first statement in the if body is an incr which as before completes successfully. Its return code is therefore 0 / ok and evaluation moves on to next command in its body.

- This is now where things get more interesting. The continue command's sole purpose in life is to complete with a return code of 4 / continue. When a script evaluation is handed a exceptional return code (i.e. any other value than 0 / ok), instead of continuing with the next command in the script, it returns the same return code to its caller which in our example is the if command.

- The if command also does not know how to deal with a return code of 4 / continue, and it is therefore propagated up the call stack to the evaluation of the while body and then to the while command itself.

- The while command does incorporate special handling for the break and continue return codes. When it gets back a continue here, it proceeds with the next iteration of its body. Note once again that this is the choice of the while command implementation, **not** Tcl.

The second iteration behaves in similar fashion:

- As before the incr command return code is ok.

- The condition for the if command is false so its body is not executed. The condition boolean has nothing to do with its return code however which remains ok.

- The puts command also completes normally with a return code of ok.

- The second if statement condition is true. Its body is simply the break command which as we said completes with a return code of break. As we described for the continue command above, this is propagated up through the evaluation of the if body, the if command itself, the evaluation of the while body until it is passed to the while. At that point, the while command on receiving the break return code, reacts by terminating the loop iterations.

To summarize the above then, every command evaluation returns a return code independent of the command result. If the return code is ok the caller proceeds as normal. It may also have specific handling for one or more specific exception codes (like break and continue above). All codes that it does not handle are propagated up the call stack.

The return return code

Let us move on to a discussion of the return code value 1 / return which has its own subtleties. This code is returned either explicitly via the return statement or by the implicit return command at the end of every procedure body. It is the fundamental basis for the underlying mechanism by which procedures return to their caller.

Practically all commands, except those specifically dealing with manipulation of return codes, such as catch, try and the like, propagate the return code up the call stack like any other exceptional code. The special handling for the return return code occurs in two instances:

- In the evaluation of a procedure body, a return code value of return will terminate execution of the procedure. *However, rather than propagating this return code value, the procedure evaluation will complete with a return code 0/ok so its caller sees a normal completion.

- Similarly, when the source command is used to execute the contents of a script, a return code of return from any command will terminate execution of any further commands from the sourced file. The source command itself will return the ok return code to the caller.

Let us go through a concrete example of nested procedure calls to see how a return code of 2 / return is handled.

```
proc cmdB {} {
    return "a value"
    puts "cmdB returning"
}
proc cmdA {} {
    cmdB
    puts "cmdA returning"
}
cmdA
→ cmdA returning
```

In procedure cmdB, the return command itself completes with the result a value and a return code of 2 / return. Since this return code is something other than ok, execution does not continue with the subsequent puts statement. Rather the Tcl procedure evaluation implementation sees the return code 2 and treats it specially. The corresponding result a value is returned as the result of cmdB **and** instead of propagating the code 2 / return, a return code of 0 / ok is returned to the cmdA invocation. The ok return value allows the cmdA procedure to invoke its puts command before returning.

Note how the 2 / return return code is transformed to 0 / ok when the procedure completes. If the invocation of cmdB had propagated 2 / return, the procedure evaluation of cmdA would itself have returned immediately after cmdB returned without executing its puts command. As we will see later, it is also possible to accomplish the latter, effectively returning to the caller several levels up the stack.

Regarding the aforementioned subtleties with respect to the return return code, consider the following script.

```
proc cmdB {} {
    set x "cmdB"
    uplevel 1 {
        puts "x = $x"
        return
    }
    puts "cmdB returning"
}

proc cmdA {} {
    set x "cmdA"
    cmdB
    puts "cmdA returning" ❶
}
```

❶ Is this line printed?

What output would you expect when procedure cmdA is called? One might think that since the uplevel command called from cmdB executes in the context of cmdA, the return command within the uplevel would cause cmdA to return without printing the cmdA returning line. What actually happens is

```
% cmdA
→ x = cmdA
  cmdA returning
```

Based on our prior discussion, this behaviour should be, umm, obvious. If not, remember that all that the return statement does is to return the code 2 / return. Since uplevel itself does not treat this return code specially, it propagates to the **caller** of the uplevel command which is cmdB, **not** cmdA. On receiving this return code, the procedure invocation of cmdB terminates with a return code of ok as described above. The cmdA procedure thus only sees a cmdB return code of ok and continues on to invoke its puts command.

The `error` return code

The one standard exception code we have not discussed is 2 or `error`. This is because we have an separate section coming up soon devoted to error generation and handling in Tcl.

11.2.3. The return options dictionary

Along with the command result and return code, every command completion also includes a *return options dictionary*. This is a dictionary with keys

- `-code` and `-level` which together determine the return codes at each level of the call stack. These keys are not just informational but control the unwinding of the call stack and we describe how they are set and used in Section 11.3.
- `-errorinfo`, `-errorline`, `-errorcode` and `-errorstack`, which provide additional information when an error exception is raised. We detail these in Section 11.4.2.
- Any other application defined keys whose use and interpretation is entirely up to the application.

Note that only the keys `-code` and `-level` are guaranteed to exist on every completion.

We will be revisiting the return options dictionary as we proceed through the chapter. For now, we move on to a discussion of how they are generated alongside return codes.

11.3. The `return` command

We have seen the use of the `return` command to return a result from a procedure. This is the most common use by far but the `return` command in Tcl is far more flexible and powerful than demonstrated by this typical use. It can be used to generate a return code, a custom return options dictionary and even skip levels in the call stack when returning from a procedure. We explore these capabilities in this section.

So far we have seen several commands that generate a specific return code:

- The `break`, `continue` commands generate their respective return codes.
- The `error` and `throw` commands that we describe later generate the `error` return code.
- Procedure invocations that complete normally do so with a return code of `ok`.

There is no way for these to generate other return codes and nor do they have any means of manipulating the return options dictionary.

The `return` command on the other hand provides a general-purpose mechanism to set all three values — the result, the return code and the return options dictionary — associated with a command completion. Additionally, the command has the ability to control the return codes generated at any level of the call stack, a facility that is important in construction of new control statements.

The `return` command has the syntax

```
return ?OPTION VALUE ...? ? RESULT?
```

The `return` command **is just a command like any other** and as such it completes with the same three values — result, return code and return options dictionary — as any other command. Unlike other commands though, it allows the caller to specify the values to be returned for all three of these:

- The result of the `return` command is the *RESULT* argument which defaults to the empty string if unspecified.
- The return code from the `return` command itself is usually the code 2 / `return` but as we see in a bit, this is dependent on the `-level` and `-code` options.
- The return options dictionary resulting from an call to `return` is composed from the specified *OPTION VALUE* pairs. As we stated in Section 11.2.3, this dictionary may have any number of keys. Tcl defines the names and semantics of certain keys but applications can add their own as well to pass along additional information. We will see examples later. In addition, the key `-code` with a value of 2 / `return`, and the key `-level` with a value

of 1, are added to the dictionary if they are not already specified by the option value list. Finally, the option name -options is treated specially. Its value is treated as a dictionary whose content is merged with the other options to form the return options dictionary. We will see examples of this below and a real world use case in Section 11.6.1.

Let us now take a closer look at the -level and -code options to the return command. We will start with a somewhat simplified explanation. The basic form of the return command

```
return "foo"
```

is equivalent to

```
return -code ok -level 1 "foo"
```

corresponding to the default values for the -code and -level options. With one important exception that we note below, the return command always completes with a return code of 2 / return. Note this is true irrespective of the value of the -code option. Since this return code is not a normal ok return code, when called within a procedure the procedure evaluation code will stop processing further commands in the procedure body. Moreover, since the procedure evaluation code affords special treatment to the code return, it will not be propagated as is. Rather the procedure evaluator treats the return code as a special case and completes the procedure itself with the return code value specified by the -code option, which is ok in our default case.

The sole exception we mentioned above with regards to the return code from the return command is when the -level option is specified with a value of 0. In that case, the return command will itself complete with the code specified by the -code option and not with the 2 / return code.

To help clarify the difference in behaviour when -level 0 option is specified versus the default -level 1 value, let us contrast the following two procedures.

```
proc demo1 {} {
    puts "demo1 enter"
    return -code ok -level 1 ❶
    puts "demo1 exit"
}
proc demo0 {} {
    puts "demo0 enter"
    return -code ok -level 0 ❷
    puts "demo0 exit"
}
```

❶   Equivalent to a plain return
❷   Notice -level specifies 0

Now if we invoke the two procedures, you see the difference in the output.

```
% demo1
→ demo1 enter
% demo0
→ demo0 enter
  demo0 exit
```

The behaviour of the demo1 procedure is what you might expect. The second line is equivalent to the default return command and therefore the demo1 exit line is not printed.

The demo0 procedure behaviour is different. Despite the return command appearing before it, the second puts is still invoked and the demo0 exit line printed. This is explained by the fact that the -level 0 option to return causes that command to complete with the return code specified by the -code option, which is ok in our example,

instead of the return return code. The procedure evaluation sees this as a normal command completion, not an exception, and therefore continues with the next statement in the procedure, the puts command.

Under what circumstances might one use a value of 0 for the -level option? There are a couple that are commonly seen. One is when the return command is used, in lieu of the error or throw commands, to raise an error exception. This is discussed in Section 11.5.2.

The other common use of -level 0 in Tcl is as an *identity* function whose result is simply the value passed in [1].

```
set n 0
set reciprocal [if {$n == 0} {return -level 0 Inf} else {expr {1/$n}}]
→ Inf
```

The -level option has more utility than just this though. It can be actually used to control the unwinding of the procedure call stack. This requires us to detail exactly how a return code of 2 / return from a command is handled during evaluation of a procedure body. When a command returns this return code during a procedure evaluation,

- First, the -level element of the return options dictionary is decremented.

- If the post-decrement value of -level is 0, the procedure completes and returns to its caller with completion return code being set to the value of the -code element in the return options dictionary. The caller of the procedure will handle this return code as appropriate. For example, if set to ok, it will continue with the next command.

- If the post-decrement value is greater than 0, the procedure is completed but with a return code of 2 / return, and **not** the code specified by the -code element of the return options dictionary. The caller of the procedure will see this return code and thus repeat this sequence of steps to handle it (assuming it is also a procedure). Note the value of the -level element will have been decremented in the passed return options dictionary.

The point in all this is that the -level command can be used to force a return **from any point in the call stack with any desired return code** as illustrated in the following example.

```
proc demo1 {levels} {
    puts "demo1 enter"
    demo2 $levels
    puts "demo1 exiting"
    return "demo1 return value"
}
proc demo2 {levels} {
    puts "demo2 enter"
    demo3 $levels
    puts "demo2 exiting"
    return "demo2 return value"
}
proc demo3 {levels} {
    return -level $levels "demo3 return value"
}
```

If we call demo1 with an argument of 1, the return command executed in demo3 is essentially the default form of the command. As expected, all puts statements are executed and we can see the corresponding outputs as the call stack unwinds. The result of our demo1 call is demo1 return value.

```
% demo1 1
→ demo1 enter
  demo2 enter
  demo2 exiting
  demo1 exiting
  demo1 return value
```

---
[1] In the latest version of Tcl, the string cat command can also be used as an identity function

Now if we call demo1 with an argument of 2, the situation is different and we can see the output has changed.

```
% demo1 2
→ demo1 enter
  demo2 enter
  demo1 exiting
  demo1 return value
```

Notice that the demo2 exiting statement is no longer printed. This is a consequence of the command executed in demo3 being

```
return -level 2 "demo3 return value"
```

Let us follow the description earlier of how the -level element is processed to see how this works:

- The return command in demo3 completes with a -level of 2. This is decremented and since it is not 0, the demo3 procedure itself completes with a return code of 2 / return **and not** 0 / ok **as in the normal case**.

- The evaluation of demo2 sees this 2 / return code and instead of executing the next command in the procedure as it would if the code were ok, it also completes as per the usual handling of the return return code. However, now the result of decrementing of the -level element is 0 and as per our -level processing rules, the procedure completes with a return code as specified by the -code element of the return options dictionary, which in this case is 0 / ok. The exiting puts statement as well as the final return command in demo2 are never reached.

- The caller demo1 sees this ok and moves on to processing the next command in its body. (Note that the **result** of demo2 is not used and discarded.)

To go one step further, see what happens with the command

```
% demo1 3
→ demo1 enter
  demo2 enter
  demo3 return value
```

Now notice that neither the demo1 nor the demo2 exiting statements are printed. In addition, the result of the demo1 command is the value originally returned from demo3 and not the return value from demo1 itself. You can extend the steps above to understand why that is.

Thus we see that the return command can be used to not only unwind the call stack but to also set the return value at that level. In case you were wondering, this also took place in our previous demo1 2 example. The result of demo3 was also the result of demo2. However, there the demo1 procedure discarded the result of demo2.

In our illustration, we used the default -code value of 0 / ok. This is not mandated as shown in our next example where we write a utility command that checks if an argument is an integer and raises an error exception otherwise. We may write the procedure as follows:

```
proc check_integer {arg} {
    if {![string is integer -strict $arg]} {
        error "$arg is not an integer."
    }
}

proc tohex {arg} {
    check_integer $arg
    return [format %x $arg]
}
```

Passing it a non-integer raises an error exception.

```
% tohex abc
0 abc is not an integer.
```

This works but the error stack (see Section 11.4.2.1) is a bit messy.

```
% puts $::errorInfo
→ abc is not an integer.
      while executing
   "error "$arg is not an integer.""
      (procedure "check_integer" line 3)
      invoked from within
   "check_integer $arg"
      (procedure "tohex" line 2)
      invoked from within
   "tohex abc"
```

It is difficult to spot the line where the mistake was **made** as opposed to where it was **detected**. We can instead write the check_integer procedure as follows.

```
proc check_integer {arg} {
    if {![string is integer -strict $arg]} {
        return -level 2 -code error "$arg is not an integer."
    }
}
```

The error stack now looks much cleaner. We immediately know the call that needs to be fixed.

```
% tohex abc
0 abc is not an integer.
% puts $::errorInfo
→ abc is not an integer.
      while executing
   "tohex abc"
```

> The use of -level does not change the fact that all return codes can be caught as we will discuss in Section 11.4. A -level simply propagates a return code up the stack a specified number of steps. This return code can be trapped at any of those levels through a catch or trap command.

11.3.1. Emulating other commands with return

Given our discussion of the -code option for the return command and an example of using it with the error return code, you might guess that the return command can be used to emulate other commands, like break or continue, that generate return codes. Here is a stop procedure that emulates the break command.

```
proc stop {} {return -code break}
```

And to show it works,

```
% foreach char {a b c} {
    puts $char
    stop
}
→ a
```

The stop procedure effectively completes with a return code of break. The foreach command has special handling for this return code and duly terminates the loop iterations.

In a sense, commands like break and continue are really just syntactic sugar for specific common uses of the return command.

11.3.2. Custom return codes

Table 11.1 showed the return codes defined by Tcl which range from 0-4. You are free to use any integer outside this range as a custom return code. This can be caught with catch or try just as any other return code.

```
proc ret5 {result} {return -code 5 $result} → (empty)
catch {ret5 "Code 5!"} result            → 5
set result                               → Code 5!
```

The interpretation of custom codes is of course entirely up to the application or package and different ones might interpret the same code differently. Of course, this can be a problem when multiple libraries are in use and thus such extensions must be used in very controlled fashion. See Section 11.7 for one possible use.

11.3.3. Custom return options dictionary

The other way return handling can be customized is through additional custom entries in the return options dictionary. This is done simply by specifying the custom options as options to the return command. For example, suppose your code catches an error from an invoked command and before passing it on, wants to add some additional information, such as the timestamp. You can do this by passing it as an additional element in the return options dictionary as below.

```
proc badcode {} { error "Did something bad!" }
proc demo {} {
    if {[catch {badcode} result ropts]} {
        return -options $ropts -timestamp [clock seconds] $result ❶
    } else {
        return $result
    }
}
```

❶   See Section 11.6.1 for an explanation of this idiom

The code passes on the error after adding a new element -timestamp to the return options dictionary.

```
% catch demo result ropts
→ 1
% set result
→ Did something bad!
% dict get $ropts -timestamp
→ 1499148945
```

The above works because the return command will treat any option not known to it as a custom entry to be added to the return options dictionary.

11.4. Trapping exceptions

We saw in Section 11.2.2, how return codes are propagated up the call stack and how commands like for and while trap and handle special return codes like break and continue. We now describe the general purpose commands which can trap exceptions arising from **any** return code. Error exceptions are just a special case where the return code is 1 or error.

Let us start by looking at what happens when a command raises an error exception, i.e. it completes with a return code of error. As detailed in Section 11.2.2, all exception return codes are propagated up the call stack until handled by a command and the error return code is no exception (pardon the pun). If no such command appears in the call stack, the exception return code is propagated all the way up to the outermost level where the default error handler will terminate the program or if it is a background error (see Section 15.4.1) in code run from the event loop, an action like displaying an error message is invoked.

We saw how the looping constructs handle return codes of break and continue preventing the propagation of these codes up the call stack. Similarly, the catch and try commands allow the **trapping** of **any** return code from the execution of a command or script enabling further appropriate action to be taken.

11.4.1. Trapping exceptions: catch

We will start off by looking at the catch command.

```
catch SCRIPT ?RESULTVAR? ?OPTSVAR?
```

The catch command executes the specified script returning as its result the **return code** from the script, **not the script result**.

We can use catch in its simplest form to show the return codes of various commands.

```
catch {set x "Normal completion"}        → 0
catch {error "This is an error message"} → 1
catch {return "A result"}                → 2
catch {break}                            → 3
catch {continue}                         → 4
catch {return -code 5 -level 0}          → 5
```

If the *RESULTVAR* argument is specified, the variable of that name will hold the result of the script. The return code as before will be the result of the catch command itself. Note that the result may be from a normal completion or an exceptional one.

```
% catch {set x 100} result  ❶
→ 0
% puts $result
→ 100
% catch {set x $nosuchvar} result
→ 1
% puts "Error: $result"      ❷
→ Error: can't read "nosuchvar": no such variable
```

❶   Here result is the result of the script on normal completion
❷   Here result is the error message on an error exception

11.4.2. The error stack and return options dictionary

The optional *OPTSVAR* argument to the catch command is the name of a variable to hold the return options dictionary we discussed in Section 11.2.3. As we stated there, this dictionary always contains at least the two keys -level and -code that we have already elaborated on in previous sections. In the case of an error exception, the dictionary also contains the additional keys -errorinfo, -errorcode, -errorstack and -errorline.

Let us define a simple procedure that raises an error exception as a sample and print what we get back from the catch.

```
proc badproc {} {
```

```
    set y $nosuchvar
}
catch {badproc} result ropts
→ 1
```

As we saw earlier, the `result` variable will hold the error message.

```
puts "result: $result" → result: can't read "nosuchvar": no such variable
```

The `ropts` variable holds additional information described in the following sections.

11.4.2.1. Error stack trace: `-errorinfo` element, `errorInfo`

The `-errorinfo` element contains a complete call stack dump that shows the sequence of calls up to the point the error exception was raised. The content is meant for human consumption and is primarily a debugging and troubleshooting aid.

```
% dict get $ropts -errorinfo
→ can't read "nosuchvar": no such variable
      while executing
   "set y $nosuchvar"
      (procedure "badproc" line 2)
      invoked from within
   "badproc"
```

This information is also stored in the `errorInfo` global variable.

```
% puts $::errorInfo
→ can't read "nosuchvar": no such variable
      while executing
   "set y $nosuchvar"
      (procedure "badproc" line 2)
      invoked from within
...Additional lines omitted...
```

11.4.2.2. Error line number: `-errorline` element

The `-errorline` element gives the line number where the error was raised.

```
dict get $ropts -errorline → 1
```

Again, this is primarily for debugging purposes.

11.4.2.3. Error codes: `-errorcode` element, `errorCode`

The `-errorcode` element in the dictionary on the other hand contains additional information about the error that is in a form convenient for programmatic consumption. It is by convention structured as a list of elements, the first of which is the module generating the error followed by module specific information. In our example, it indicates the error was generated by Tcl itself, on a failed read operation on a variable. This information is also available via the `errorCode` global variable.

```
dict get $ropts -errorcode → TCL READ VARNAME
puts $::errorCode        → TCL READ VARNAME
```

The error code can often be parsed during execution to programmatically ascertain the specific cause and whether corrective action can be taken.

```
if {[catch {open options.ini} result ropts]} {
    if {[lindex $::errorCode 0] eq "POSIX" &&
        [lindex $::errorCode 1] eq "ENOENT"} {
        # File does not exist
        .. use default options ..
    } else {
        return -code error -options $ropts $result; # Explicitly propagate the error
    }
} else {
    # $result holds opened channel
    .. read options from $result ..
    close $chan
}
```

11.4.2.4. Error stack: -errorstack element, info errorstack

The final error related element in the return dictionary is -errorstack.

```
dict get $ropts -errorstack → INNER loadScalar1 CALL badproc
```

This is similar to the -errorinfo element except it is in a form more suitable for programmatic consumption. It consists of alternating token and parameter pairs where the token may be one of INNER, CALL or UP indicating an internal command or byte code instruction, a procedure call, or a call frame change via uplevel and the like. The associated parameter gives the specifics. For example, in the above output, the CALL parameter indicates badproc as the name of the procedure that was called. It will also show the actual argument values in the invocation unlike -errorinfo which shows invocations before variable substitutions.

The information in the -errorstack element is also available with the info errorstack command.

```
info errorstack → INNER loadScalar1 CALL badproc
```

This returns the stack corresponding to the last error encountered.

11.4.3. Trapping exceptions: try

The try command offers a functionally equivalent alternative to the catch command for handling exceptions and errors. The latter is convenient when a single set of actions is to be taken on any non-normal return. The try command on the other hand makes it easier to break out handlers for different return codes or failure modes and when common set of actions, such as releasing resources, need to be taken for both normal and exceptional conditions. The choice is usually a matter of personal preference.

```
try BODY ?HANDLER …? ?finally FINALSCRIPT?
```

Each HANDLER specifies a completion status and/or an error code pattern along with a Tcl script. The command evaluates BODY and matches its completion status against the specification of each HANDLER. On finding a match, the corresponding handler script is evaluated. Remaining handlers are ignored.

If the finally clause is specified, the FINALSCRIPT argument is evaluated before the try command completes **irrespective of the completion status or whether any handlers were invoked**.

The completion status and result of evaluation of BODY is propagated as that of the try command unless a handler is executed in which case the completion status and result of the handler is propagated instead. If FINALSCRIPT completes normally, its result is thrown away. If it completes with an exception, that exception is propagated. In the cases where a handler or FINALSCRIPT generate an exception, the return options dictionary from the evaluation of BODY is added to the new return options dictionary under the -during key.

The *HANDLER* clauses themselves may take one of the following forms:

```
on CODE VARLIST HANDLERBODY
trap ERRORPATTERN VARLIST HANDLERBODY
```

For the on clause, *CODE* must be an integer or one of the mnemonic return code values shown in Table 11.1. The clause will match if the try command's *BODY* script completes with that return code.

A trap clause will match if *BODY* completes with a return code of error **and** *ERRORPATTERN* matches the return options dictionary's -errorcode element. The match is done by interpreting *ERRORPATTERN* as a list of words each of which must match the corresponding element of the -errorcode value. Additional elements in -errorcode are ignored so an empty *ERRORPATTERN* will match any value of -errorcode.

In both cases, *VARLIST* is a list of up to two variable names. On a match

- the result of the evaluation of *BODY*, is assigned to the first name in *VARLIST* if not the empty string.
- the return options dictionary is assigned to the second name in *VARLIST* if not the empty string.
- the *HANDLERBODY* script is then executed. If *HANDLERBODY* is -, the *HANDLERBODY* of the next clause is used instead.

The handlers are matched in sequence and only the first matched one is evaluated. If no match is found, the completion return code and result are propagated to the caller.

As always, some examples will make all this clearer.

A simple try without any additional clauses acts similar to eval. If the *BODY* script completes normally, the result for the evaluation is the result of the try command. On any exceptional completions, the return code is propagated up **since the try command does not have any handlers specified**.

```
try {set x 1}         → 1
try {set x $nosuchvar} ∅ can't read "nosuchvar": no such variable ❶
```

❶ Error return code is propagated up to the command shell

> If you need to evaluate a single script in the current context with eval, consider using try without any clauses instead. It is faster due to its being byte compiled which eval is not.

The on handlers can be used to trap completions with specific return codes. Any return codes that are trapped in this manner are **not** automatically propagated. We can use catch to see the propagation of return codes when handler clauses are present.

```
% catch {
    try { error "Error!" } on error result {puts Trapped!}     ❶
}
→ Trapped!
  0
% catch {
    try { break } on error result {puts Trapped!}             ❷
}
→ 3
```

❶ Completes normally as error return code trapped.
❷ Propagates break as no handler defined for it.

Note that as for catch, **any** return code can handled, even ok, return or non-standard numeric values.

Let us now look at the *VARLIST* argument in a bit more detail. This lets us retrieve the result of evaluation of *BODY* and the return options dictionary in a similar manner to the two optional arguments to the `catch` command.

```
try {
    set x $nosuchvar
} on error {result ropts} {
    puts "result = $result"
    puts "Return options dictionary:"
    print_dict $ropts
}
→ result = can't read "nosuchvar": no such variable
  Return options dictionary:
  -code       = 1
  -errorcode  = TCL LOOKUP VARNAME nosuchvar
  -errorinfo  = can't read "nosuchvar": no such variable
...Additional lines omitted...
```

The other form that a `try` handler specification can take is specifically for trapping completions with an `error` return code. The utility of the `trap` handler over the on `error` handler is that it directly allows distinction between different error codes without having to separately check for them within an on `error` handler.

Consider the following commands.

```
/ 4 0    Ø divide by zero ❶
/ 4 0.0 → Inf
```

❶   Assumes `tcl::mathop::/` is on the namespace path

If we wanted the two to behave the same, we could define a `div` procedure as follows.

```
% proc div {a b} {
    try {
        return [/ $a $b]
    } trap {ARITH DIVZERO} result {
        return [/ $a 0.0]
    }
}
```

Our handler is invoked only when

* the return code is `error`, **and**
* the error code in the return options dictionary is a list starting with `ARITH DIVZERO`.

All other cases work as before including normal operation and other types of errors.

```
div 4 2    → 2
div 4 0    → Inf
div 4 xyz Ø can't use non-numeric string as operand of "/"
```

> ⚠ Note that an on `error` handler is equivalent to `trap {}` and handles all completions with a return code of `error`. Therefore any `trap` clauses should appear before an on `error` clause. Trap clauses placed after it will not have effect.

We are left with the `finally` clause to discuss. The most common use of this clause is to ensure that resources are freed irrespective of whether a script completes normally or not. Thus the clause is used in the fashion similar to the following.

```
set fd [open data.xml]
try {
    return [parse_data [read $fd]]
} finally {
    close $fd
}
```

The code ensures that the opened file channel is closed irrespective of any errors in reading or parsing the data.

11.5. Raising exceptions

In Section 11.3 we saw how we can specify any arbitrary value as the return code on completion of a script or procedure. Raising an exception is nothing other than specifying a value other than 0/ok for the return code. Thus, we do not need to say anything more about this general case.

However, the error return code differs from from other return codes in that Tcl takes some additional implicit actions such as generating the stack trace back we saw earlier. The following sections describe this special case.

11.5.1. Raising errors: throw, error

Tcl has two dedicated commands, throw and error, in addition to the general purpose return command which can generate any desired return code.

```
throw ERRORCODE MESSAGE
error MESSAGE ?ERRORINFO? ?ERRORCODE?
```

Both commands complete with a return code of 1 / error and a **result** of MESSAGE. As is true for all command completions, this result is captured by the first variable name argument to the catch command. The ERRORCODE element supplies the error code that will be stored in the -errorcode element of the return options dictionary and the errorCode global variable.

When an error exception is thrown, Tcl accumulates a stack trace of the calling sequence in the -errorinfo element of the return dictionary and the errorInfo global. By default, this stack trace starts at the point of the call to error or throw. However, the error command allows specification of the ERRORINFO argument to "seed" the stack trace. We will see an example of its use for propagating a caught exception in a later section.

Raising an error exception is staightforward using either throw or error.

```
proc change_password {name pass} {
    set len [string length $pass]
    if {$len < 8} {
        throw [list OAUTH PASSLEN $len]   "Password length must be at least 8."
    }
    db_update $name $pass
}
```

A couple of points to note about the above example. The general convention for the format of the error code is a word that identifies the module or package (OAUTH), then one or more failure "reason" codes (PASSLEN) and possibly some detail about the error, in our case the length of the supplied password.

```
% change_password user abc
0 Password length must be at least 8.
% puts $::errorCode
→ OAUTH PASSLEN 3
```

We could have replaced the throw command in the procedure with the equivalent error command.

```
error "Password length must be at least 8." "" [list OAUTH PASSLEN $len]
```

The last two arguments to error are optional so we could have also raised an error as

```
error "Password length must be at least 8."
```

In this case the error code is set to an empty string.

> Because throw is a relatively new addition to Tcl, you will find error used more often. Moreover, it is tempting to be lazy and use the one argument form of error. However, it is now considered good practice to always specify an error code which makes throw syntactically a little more convenient and preferable for the common case where a initial value for the errorInfo stack trace is not required to be passed.

11.5.2. Raising errors: return -code

We described the return command in detail in Section 11.3. Here we detail additional considerations when using the command to raise an error by returning the error return code value.

```
return -code error ?-errorcode ERRORCODE? ?-errorinfo ERRORINFO? -errorstack ERRORSTACK? MESSAGE
```

The ERRORCODE, ERRORINFO and MESSAGE have the same semantics as for the throw or error commands. The -errorstack option sets the -errorstack element in the return option dictionary.

Here is a short example demonstrating the equivalent use of throw versus return.

```
proc check_boolean_1 {arg} {
    if {![string is boolean -strict $arg]} {
        throw {TYPECHECK BOOLEAN} "$arg is not a boolean"
    }
}
proc check_boolean_2 {arg} {
    if {![string is boolean -strict $arg]} {
        return -code error -errorcode {TYPECHECK BOOLEAN} \
            "$arg is not a boolean"
    }
}
```

If you run both procedures however, you will notice a small difference in the error stack.

```
% check_boolean_1 abc
0 abc is not a boolean
% puts $errorInfo
→ abc is not a boolean
    while executing
  "throw {TYPECHECK BOOLEAN} "$arg is not a boolean""
    (procedure "check_boolean_1" line 3)
    invoked from within
...Additional lines omitted...
% check_boolean_2 abc
0 abc is not a boolean
% puts $errorInfo
→ abc is not a boolean
    while executing
  "check_boolean_2 abc"
```

11.6. Forwarding exceptions

There are circumstances where we need to trap an exception, handle it if we can, and if not, forward or re-throw the exception with the **same error code and error stack** as in the original. We can use the `return` or `error` command for this purpose.

11.6.1. Forwarding exceptions: `return`

The complete control you have over the result, return code, level and return options dictionary makes the `return` command ideal for forwarding any kind of exception. Here is an example of its use for this purpose.

```
proc recover args {return 0}
proc do_something {} {
    set x $nosuchvar
}
proc demo {} {
    if {[catch {do_something} result ropts]} {
        if {![recover]} {
            return -options $ropts $result ❶
        }
    }
}
```

❶ Note there is no `-code` option specified because it is already contained in the `ropts` return dictionary

Let us confirm that the caught exception information is preserved when we raise the exception again.

```
% demo
∅ can't read "nosuchvar": no such variable
% puts $::errorCode
→ TCL READ VARNAME
% puts $::errorInfo
→ can't read "nosuchvar": no such variable
      while executing
  "set x $nosuchvar"
      (procedure "do_something" line 2)
      invoked from within
...Additional lines omitted...
```

Notice that all information in the original exception is preserved. This includes cases where the returned option dictionary may contain additional elements as we described in Section 11.3.3.

11.6.2. Forwarding exceptions: `error`

Alternatively, we can use the `error` command in the special case where we want to forward error exceptions by specifying the original `errorCode` and `errorInfo` values as arguments to the `error` command.

```
proc demo {} {
    if {[catch {do_something} result]} {
        if {![recover]} {
            error $result $::errorInfo $::errorCode
        }
    }
}
```

If we invoke demo, notice again that the original error code and stack trace were preserved.

```
% demo
0 can't read "nosuchvar": no such variable
% puts $::errorCode
→ TCL READ VARNAME
% puts $::errorInfo
→ can't read "nosuchvar": no such variable
      while executing
   "set x $nosuchvar"
      (procedure "do_something" line 2)
      invoked from within
...Additional lines omitted...
```

This use of error to forward an error exception is seen in legacy code as **use of the** return **command for this purpose is preferred** for the following reasons:

- The error command cannot forward exceptions other than errors
- There is no means to preserve the full contents of the return options dictionary

11.7. Custom control statements

We saw in Section 10.5.5 an attempt at implementing a new control statement repeat. That implementation was incomplete because it did not handle exceptional conditions like break and errors. Having described return code and error handling, we are now in a position to present a full implementation. As a reminder, we want to implement a repeat command that can be used as follows

```
set sum 0
repeat i 10 {
    incr sum $i
}
```

This command can be implemented as

```
proc repeat {loopvar count body} {
    upvar 1 $loopvar iter
    for {set iter 0} {$iter < $count} {incr iter} {
        set ret_code [catch {uplevel 1 $body} result ropts]
        switch $ret_code {
            0 {}
            3 { return }
            4 {}
            default {
                dict incr ropts -level
                return -options $ropts $result
            }
        }
    }
    return
}
```

Exceptions like break, continue, return and errors are now handled appropriately.

Having implemented a custom loop command let us extend its functionality further with another control command, skip that behaves like continue but lets you specify how many iterations of the loop are to be skipped.

```
proc skip {skip_count} { return -code 5 $skip_count }
```

We have now introduced a new return code 5 and have to account for it in our repeat procedure.

```
proc repeat {loopvar count body} {
    upvar 1 $loopvar iter
    for {set iter 0} {$iter < $count} {incr iter} {
        set ret_code [catch {uplevel 1 $body} result ropts]
        switch $ret_code {
            0 {}
            3 { return }
            4 {}
            5 { incr iter $result }
            default {
                dict incr ropts -level
                return -options $ropts $result
            }
        }
    }
    return
}
```

We can now use it to skip a given number of iterations.

```
repeat n 5 {
    puts "Iteration $n"
    if {$n == 1} {skip 2}
}
→ Iteration 0
  Iteration 1
  Iteration 4
```

Of course, only our repeat custom iteration command understands this new skip construct.

11.8. Chapter summary

In this chapter, we explored the mechanisms Tcl provides for special conditions, including errors, through the exception handling facilities. These in turn are based on Tcl's generalized framework for returning computational state from script execution, a feature which lends itself not only to error handling but to creation of new first-class custom control structures as well.

We will now move on to some practical aspects of programming in Tcl such as modularization and packaging libraries.

11.9. References

TIP90

TIP #90: Enable [return -code] in Control Structure Procs, Don Porter, Donal K. Fellows, Tcl Improvement Proposal #90, http://www.tcl.tk/cgi-bin/tct/tip/90

12

Namespaces

> I, sir, am Dromio; command him away.
>
> I, sir, am Dromio; pray, let me stay.
>
> — William Shakespeare *Comedy of Errors*

If Shakespeare understood namespaces, there would have been no confusion between Syracuse::Dromio and Ephesus::Dromio. Much havoc could have been avoided!

Most modern languages support the concept of *namespaces* as a means to resolve conflicts between multiple libraries or components defining the same name, for a variable, function or any programming construct. This is even more of an issue for dynamic and scripting languages where there is no separate compile/link step that can be used to limit name visibility to file scope. A common convention before the advent of namespaces was to prefix names with the name of the module or library so libA_state and libB_state could be distinguished. Given that most references to names are to those within the same module, this is not just unnecessary typing but a hindrance to readability as well.

Namespaces are a solution to this issue. They provide a means to partition names and define a scope within which they are visible so there is no confusion as to which construct is being referenced.

Tcl's support for namespaces is dynamic and scriptable. It goes further than most languages in its capabilities and flexibility. We explore these features in this chapter.

12.1. Namespace basics

A *namespace* is a mechanism for grouping together variables and commands under an identifier, the name of the namespace. It also creates a *scope* for execution of code wherein names within the same namespace can be referenced without further qualification while requiring names outside the namespace to be qualified with the name of their containing namespace.

12.1.1. A simple namespace example

A simple script will clarify the concepts.

```
namespace eval nsA {
    variable my_var "variable in nsA"
}
```

The above command creates a namespace nsA and evaluates the passed script within the *context* of the namespace. Thus the script

```
variable my_var "variable in nsA"
```

is executed in the context of namespace nsA.

The command variable is used to declare and optionally initialize (as here) a variable inside the namespace context within which it is executed, in this case namespace nsA.

Next we will create another namespace, nsB, in the same fashion.

```
namespace eval nsB {
    variable my_var "variable in nsB"
}
```

And finally we will call variable **outside** of any namespace.

```
variable my_var "global variable"
```

This variable command is executed outside of any namespace and hence defaults to the global namespace context. The variable is a global variable similar to that created by the global command.

We are now ready to give some examples of scope and context.

```
% puts $my_var
→ global variable
```

Because the puts is executing outside any namespace, or to be precise in the global namespace, the reference my_var is to the global variable of that name.

On the other hand, if the same command were to be executed within the context of namespace nsB

```
% namespace eval nsB { puts $my_var }
→ variable in nsB
```

the name my_var would refer to the variable defined in the nsB context.

In both cases, the variable references were *unqualified* and hence referred to the namespace context in which the code was executing. To refer to variables **outside** the context in which the code is executing, the name must be *qualified* with the name of the containing namespace. For example, to access the variables in the global and nsA namespace contexts from code running under the nsB context,

```
% namespace eval nsB {
    puts $::my_var
    puts $::nsA::my_var
}
→ global variable
  variable in nsA
```

Although the above example used variables to demonstrate context, the same also applies to command definitions and invocations as we will see as we proceed.

It is completely legal and often very useful to store a namespace name itself in a variable and use the variable to refer to the contents of the namespace. However, you have to be careful in the syntax used.

```
% set my_namespace nsA
→ nsA
% puts $my_namespace::my_var
Ø can't read "my_namespace::my_var": no such variable
```

The above generates an error because the parser treats my_namespace::my_var as the name of the variable and tries to resolve. You need to therefore use one of several alternative syntaxes instead.

You can use the ${} syntax to constrain the parsing of the variable name and then use set to retrieve the value.

```
puts [set ${my_namespace}::my_var] → variable in nsA
```

Alternatively, you can use nested set commands.

```
puts [set [set my_namespace]::my_var] → variable in nsA
```

Finally, you can link a local variable to the namespace variable using the namespace upvar command described in Section 12.5.1.3.

```
namespace upvar $my_namespace my_var linked_var → (empty)
puts $linked_var                                → variable in nsA
```

12.1.2. Namespace names and hierarchy

Namespaces may nest in a hierarchical fashion similar to the paths in a file system except for the use of :: as the separator instead of / or \. So for example the identifier a::b::c consists of the namespace a, the namespace b contained within a, and an identifier c which may be a variable, command name or even another namespace inside a::b. The root of the hierarchy is the global namespace whose name is the empty string so ::a refers to an identifier a in the global namespace.

Just like file paths, names may be absolute or relative. An absolute name always starts with a :: and defines a path through the namespace hierarchy starting with the global namespace. An example is ::a::b::c. A relative name does not start with a :: and defines a path through the namespace hierarchy relative to the namespace in which it is referenced. The name a::b::c is a relative name and is **not** the same as ::a::b::c unless the reference occurs in the global namespace.

Let us rework our earlier example to include nested namespaces. The namespace current command, which we will see later, returns the name of the current namespace context.

```
namespace eval nsA {
    variable my_var "[namespace current] variable"
    namespace eval nsB {
        variable my_var "[namespace current] variable"
    }
}
namespace eval nsB {
    variable my_var "[namespace current] variable"
}
variable my_var "[namespace current] variable"
```

With these definitions in place, we can see how the different variables might be referenced from within the nsA namespace.

```
namespace eval nsA {puts $my_var}        → ::nsA variable ❶
namespace eval nsA {puts $::my_var}      → :: variable ❷
namespace eval nsA {puts $nsB::my_var}   → ::nsA::nsB variable ❸
namespace eval nsA {puts $::nsB::my_var} → ::nsB variable ❹
```

❶ Current namespace
❷ :: is a synonym for the global namespace
❸ Relative namespace
❹ Absolute namespace

We will have more to say about name resolution in Section 12.5.

There are a couple of points to be noted about nested namespaces.

First, a nested namespace can be directly defined with a single namespace eval so that instead of

```
namespace eval nsA {
    namespace eval nsB {
        .. Some code ..
    }
}
```

we could have said

```
namespace eval nsA::nsB {
    .. Some code ..
}
```

which would have resulted in the whole hierarchy being created if necessary.

The other point to be noted is that each namespace eval for a namespace does not overwrite any existing namespace of that name; it modifies or adds to it as we saw in the above examples.

12.1.2.1. Inspecting namespace hierarchies: namespace current, namespace parent, namespace children

The namespace hierarchy can be traversed with the namespace current, namespace parent and namespace children commands.

The namespace current command returns the current namespace context.

```
namespace eval nsA {
    proc whereami {} {return [namespace current]}
}
puts [nsA::whereami]
→ ::nsA
```

In the above fragment, the proc definition is inside the nsA namespace and consequently, the whereami procedure is also created within that namespace.

The namespace parent command returns the name of the namespace containing the specified namespace.

```
namespace parent ?NAMESPACE?
```

The command returns the fully qualified name of the parent. If NAMESPACE is not specified, it returns the parent of the current namespace from which the command is invoked.

```
namespace eval nsA { namespace parent }     → ::  ❶
namespace eval nsA { namespace parent nsB } → ::nsA ❷
namespace parent nsB                        → ::  ❸
namespace parent ::                         → (empty)
```

❶ Parent of current namespace context
❷ nsB child of nsA
❸ nsB child of global namespace

Conversely, the namespace children command returns a list of namespaces that are the children of a specified namespace.

```
namespace syntax ?NAMESPACE?
```

Again, *NAMESPACE* defaults to the current namespace if unspecified.

```
namespace eval nsA {namespace children}  → ::nsA::nsB
namespace children nsA                    → ::nsA::nsB
namespace children ::                     → ::cmdline ::zlib ::nsA ::fileutil ::pkg ::oo ::nsB ::tcl
```

12.1.2.2. Manipulating names: `namespace qualifiers`, `namespace tail`

Unlike static languages, programs in dynamic languages like Tcl often construct namespaces on the fly at runtime. Tcl therefore provides some commands to make manipulation of namespace names easier.

The `namespace qualifiers` command returns the leading namespace qualifiers from an identifier.

```
namespace qualifiers IDENTIFIER
```

Correspondingly, `namespace tail` returns the last component of an identifier.

```
namespace tail IDENTIFIER
```

Both commands work purely on a syntactic basis. There is no requirement for the namespaces in *IDENTIFIER* to actually exist.

```
set nshead [namespace qualifiers ::no::such::namesp]  → ::no::such
set nstail [namespace tail ::no::such::namesp]         → namesp
```

The above commands deconstruct an identifier into namespace components. There are no complementary commands to *construct* namespace paths because use of normal string interpolation or commands like `join` is sufficient.

```
set my_ns "::${nshead}::$nstail"          → ::::no::such::namesp
set my_ns [join [list $nshead $nstail] ::]  → ::no::such::namesp  ❶
```

❶ Useful when the namespace components are already in list form

> When constructing namespace identifiers, it is useful to know that Tcl will treat more than two `:` characters as namespace separators as well.
>
> ```
> puts $::nsA::::::::::my_var → ::nsA variable
> ```
>
> Thus when interpolating or joining you do not need to worry about trailing namespace separators in the identifier.

12.1.3. Deleting a namespace: `namespace delete`

As is always the case with Tcl, program elements can be created and destroyed at will and namespaces are no exception. We have seen how namespaces are created with `namespace eval`. The complementary command to destroy namespaces is `namespace delete`.

```
namespace delete ?NAMESPACE ...?
```

The command takes zero or more namespace names and deletes each namespace along with all its contained program elements, including variables, commands and even nested namespaces.

12.1.4. Checking namespace existence: `namespace exists`

Given that they can appear and disappear on the fly, there needs to be a means of checking whether a namespace exists. The `namespace exists` command provides this functionality.

```
namespace exists NAMESPACE
```

The command returns 1 if the specified namespace exists and 0 otherwise.

For both commands, each `NAMESPACE` argument is resolved as per the rules stated in Section 12.5.2.

```
namespace delete nsA → (empty)
namespace exists nsA → 0
namespace exists nsB → 1
```

12.2. Executing code in a namespace: `namespace eval|inscope`

We have already seen how code is executed in the context of a namespace with the `namespace eval` command.

```
namespace eval NAMESPACE SCRIPT ?SCRIPT ...?
```

The command will create the namespace of the specified name if it does not already exist. It then concatenates the remaining arguments, separating them with spaces, and evaluates the result in that namespace.

`NAMESPACE` is resolved as we detail later in Section 12.5.2. If `NAMESPACE` is a hierarchical namespace, any intermediate namespaces are also created if necessary. So for example,

```
namespace eval ns1 {
    namespace eval ns2::ns3 {}
    namespace eval ::ns4 {}
}
```

evaluated in the global scope will create namespaces `::ns1`, `::ns1::ns2`, `::ns1::ns2::ns3` and `::ns4`.

What does execution in the context of a namespace mean? It has primarily to do with how names (variables, commands, namespace names) are resolved as we summarized earlier and will go into detail in Section 12.5.

The `namespace inscope` command is very similar to `namespace eval`.

```
namespace inscope NAMESPACE SCRIPT ?ARG ...?
```

Like `namespace eval`, `namespace inscope` will execute `SCRIPT` in the context of the specified namespace but with two important differences:

- The first is that `namespace inscope` will **not** create the namespace if it does not already exist.
- The other difference is that the arguments are not all concatenated before execution as is done by the `namespace eval`. Rather, `SCRIPT` is executed after appending the remaining arguments as proper list elements. In effect, the additional arguments do not undergo a second round of substitution as is the case with `namespace eval`.

The following code snippet illustrates the difference. First, a small procedure to print arguments defined inside a namespace:

```
namespace eval ns1 { proc print_args {args} {puts [join $args ,]} }
```

Now we evaluate a call to the procedure via both namespace eval and namespace inscope.

```
% set arg1 "First argument"
→ First argument
% set arg2 {$arg1}
→ $arg1
% namespace eval ns1 { print_args } $arg1 $arg2
→ First,argument,First argument
% namespace inscope ns1 { print_args } $arg1 $arg2
→ First argument,$arg1
```

As seen from the output, with namespace eval the arguments undergo two rounds of substitution whereas with namespace inscope they only undergo a single round.

The namespace inscope command is rarely used directly in Tcl programming. Rather its primary purpose is to form the basis of the namespace code command which serves a specific purpose that we now describe.

12.2.1. Namespace contexts in callbacks

Tcl programming often involves callbacks — scripts and commands that are invoked from the event loop or other contexts. Examples include scripts scheduled via the after command and invoked by keyboard handlers in Tk, both of which are evaluated in the global context. For such cases, callback scripts that expect to be run in the context of a specific namespace will fail, for example the following snippet:

```
namespace eval ns1 {
    variable avar "Some value"
    after 100 {puts $avar}
}
```

This will fail because the callback script puts $avar will execute in the global context where there is no variable avar defined. We really want to execute the script in the context of ns1. The namespace code provides a convenient mechanism to accomplish this.

```
namespace code SCRIPT
```

The command result is a script that can be evaluated in **any** scope, global or any other namespace, and will still result in SCRIPT being invoked in the same namespace context in which the namespace code was invoked.

So the above fragment would work correctly if written as

```
namespace eval ns1 {
    variable avar "Some value"
    after 100 [namespace code {puts $avar}]
}
```

You can examine the result of the command to see how it works.

```
% namespace eval ns1 { namespace code {puts $avar} }
→ ::namespace inscope ::ns1 {puts $avar}
```

As you can see the passed script is wrapped in a namespace inscope to achieve the desired result.

The following utility procedure is useful as syntactic sugar for capturing the namespace scope when the callback script consists of a single command.

```
proc callback {args} {tailcall namespace code $args}
```

Then the above call can be written as

```
after 100 [callback puts $avar]
```

12.3. Defining variables in a namespace: `variable`

There are two ways a variable can be defined within a namespace. The first, and **recommended**, way is through the explicit use of the `variable` command.

```
variable ?NAME VALUE ...? ? NAME?
```

The command takes a list of alternating variable name and value arguments with the value for the last variable name being optional.

The command may be invoked directly from a `namespace eval` script or from within a procedure. In the former case,

- if a variable of a specified name does not exist, it is created. If a corresponding initializing value is specified, it is assigned to the variable. Otherwise, the variable is created but left undefined (see Section 3.6.5.3).

- if the variable of that name already existed within the namespace, it is assigned the initializing value if specified and left unaltered otherwise.

When the command is invoked from a procedure, the behaviour is similar except that the command creates variables local to the procedure but linked to namespace variables of the same name. This is detailed in Section 12.5.1.2 when we discuss name resolution in procedures.

An example of `variable` use.

```
namespace eval nsA {
    variable var_a "abc"
    variable var_b [clock seconds] var_c
    variable var_d
}
```

Here the variable `var_a` and `var_b` are created (assuming they did not already exist) and initialized while `var_c` and `var_d` are created but remain undefined.

```
set nsA::var_a          → abc
info exists nsA::var_c → 0
```

The other way to create namespace variables is to directly assign to them from within the namespace context without explicitly declaring them with the `variable` command.

```
namespace eval nsA {
    set var_e 42
}
puts $::nsA::var_e
→ 42
```

However, there is a historical quirk that you have to be aware of and guard against. Suppose you have set the value of a global variable somewhere to hold the application version.

```
set version 1.0
```

Then within some independently written package, the following code sets the package version and description **for that package**.

```
namespace eval mypackage {
    set description "My Package"
    set version 2.0
}
```

Now having loaded the package let us print out the package description.

```
puts $::mypackage::description → My Package
```

So far, so good. Let us then print out the application and package versions.

```
% puts $::version
→ 2.0
% puts $::mypackage::version
0 can't read "::mypackage::version": no such variable
```

Huh? The application version has mysteriously changed and what's more the package version is not even defined! The problem arises because the statements

```
set description "My Package"
set version 2.0
```

look alike but have very different results. In the first case, the variable description is not found in either the mypackage namespace or in the global one. The command therefore creates a new variable of that name in the current namespace mypackage. On the other hand, the name resolution sequence finds a variable version in the global context and modifies that instead of creating a variable of the same name in the mypackage context.

This is an acknowledged misfeature which is currently preserved for backward compatibility and will probably be fixed in the next major Tcl release. To protect against this, always explicitly declare namespace variables with the variable command even if there is no need to immediately initialize them.

12.4. Defining commands in a namespace

So far our examples have dealt with defining variables in namespaces. We now look at the same for defining procedures.

If the name passed to the command is fully qualified, it defines the containing namespace no matter where the procedure definition is placed.

```
namespace eval nsA::nsB {} ❶
proc ::nsA::nsB::demo_a {} {return [namespace current]}
namespace eval nsC {
    proc ::nsA::nsB::demo_b {} {return [namespace current]}
}
puts "[::nsA::nsB::demo_a], [::nsA::nsB::demo_b]"
→ ::nsA::nsB, ::nsA::nsB
```

❶   Make sure the namespaces exist

If the name does not have any namespace qualifiers or is not fully qualified, it is treated as relative to the current namespace.

```
namespace eval ::nsA {
    proc demo_c {} {return [namespace current]} ❶
    proc nsB::demo_d {} {return [namespace current]} ❷
    puts [demo_c]
    puts [nsB::demo_d]
}
→ ::nsA
  ::nsA::nsB
```

❶   Defined in namespace ::nsA
❷   Defined in namespace ::nsA::nsB

Other Tcl commands, such as TclOO object constructors, which create new commands also behave as above. The notable exception is the interp alias which always resolves relative names in the context of the global namespace even when invoked from within another namespace.

12.4.1. Namespace contexts in procedures

When a procedure is executed, its code runs in the context of the namespace in which the procedure is defined. Use of variable inside the procedure ties the specified name to the variable of the same name in the procedure's namespace. Calls to other procedures that are not fully qualified first look up procedures defined in the same namespace context.

```
proc demo {} {return "Proc in [namespace current]"}
namespace eval nsA {
    variable my_var "Variable in [namespace current]"
    proc demo {} {return "Proc in [namespace current]"}
    proc test_proc {} {
        variable my_var
        puts "Calling namespace proc: [demo]"
        puts "Calling global proc: [::demo]"
        puts "Value of my_var=$my_var"
    }
}
nsA::test_proc
→ Calling namespace proc: Proc in ::nsA
  Calling global proc: Proc in ::
  Value of my_var=Variable in ::nsA
```

12.5. Name resolution

When a name being referenced in a script begins with a :: sequence, it is a fully qualified, or absolute name that uniquely identifies its target by specifying a path through the namespace hierarchy starting at the root (global) namespace. These names obviously do not need to be resolved. Further discussion in this section thus only pertains to names that are not fully qualified.

For relative names, both simple names that have no :: separators as well as those that do but do not begin with ::, the manner of resolution depends on whether the name corresponds to a variable, a namespace or a command.

12.5.1. Resolving variable names

We will first look at how names of variables are resolved in various types of contexts.

12.5.1.1. Variable resolution outside a procedure

Both simple name references and names that are not fully qualified are first resolved in the current namespace and if not found, in the global namespace. The example below illustrates this process.

```
namespace eval nsA { variable my_var "nsA variable" }
namespace eval nsB {
    namespace eval nsC {
        variable my_var "nsB::nsC variable"
    }
    puts $nsC::my_var  ❶
    puts $nsA::my_var  ❷
}
→ nsB::nsC variable
  nsA variable
```

❶ Resolved within current namespace
❷ Not resolvable in current namespace so resolved in global namespace

12.5.1.2. Variable resolution in a procedure

Variable name resolution within a procedure is slightly different because there are procedure-local and argument names to deal with. Moreover, there are differences between resolution of simple names, i.e. names without any :: separators, and names which have at least one namespace component (but are not fully qualified).

A simple name is local to the procedure, or an argument, unless previously linked via a variable or upvar command. If linked via variable, it is linked to the variable of the same name defined in the context of the namespace in which the procedure is defined. In the case of upvar it is linked to a variable defined further up the call stack as described in Section 10.5.4.

A variable that is a relative name with namespace components is first resolved within the namespace in which the procedure resides and then if not found there, it is resolved in the context of the global namespace.

The following example illustrates the different cases. Assume we have the following namespace structure.

```
set my_var "global variable"
namespace eval nsC {
    variable my_var "nsC variable"
}
namespace eval nsA {
    variable my_var "nsA variable"
    namespace eval nsB {
        variable my_var "::nsA::nsB variable"
    }
}
```

The various ways names are resolved is illustrated in the following procedure.

```
proc nsA::demo {} {
    variable my_var  ❶
    set local_var "local"
    puts "local_var = $local_var"           ❷
    puts "my_var = $my_var"                  ❸
    puts "nsB::my_var = $nsB::my_var"        ❹
    puts "nsC::my_var = $nsC::my_var"        ❺
}
nsA::demo
→ local_var = local
```

```
my_var = nsA variable
nsB::my_var = ::nsA::nsB variable
nsC::my_var = nsC variable
```

❶   Creates a local my_var linked to ::nsA::my_var
❷   Variable local to procedure
❸   Variable linked to ::nsA::my_var
❹   Relative name successfully resolved from current namespace
❺   Relative name successfully resolved from global namespace

12.5.1.3. Linking to variables in another namespace: namespace upvar

The namespace upvar command allows linking of local variables or variables in one namespace to variables in any namespace.

```
namespace upvar NAMESPACE ?NSVAR LOCALVAR ...?
```

The NAMESPACE argument specifies the name of the namespace whose variables are to be linked. It is resolved as described in Section 12.5.2.

Each NSVAR LOCALVAR pair specifies a variable name in the NAMESPACE namespace and the corresponding variable to which it is to be linked. When the command is invoked outside a procedure, LOCALVAR names a namespace variable in the namespace in which the command is invoked.

```
namespace eval nsC {
    namespace upvar ::nsA::nsB my_var linked_var
}
puts $::nsC::linked_var
→ ::nsA::nsB variable
```

Within a procedure, LOCALVAR is a procedure-local variable.

```
proc demo {} {
    namespace upvar ::nsA my_var linked_var
    puts $linked_var
}
demo
→ nsA variable
```

In all cases, the LOCALVAR variable must not already exist.

12.5.2. Resolving namespace names

Resolution of namespace names that are not absolute is very simple. They are always resolved with respect to the current namespace.

```
namespace eval nsA {
    namespace eval childNS {}
}
```

The unqualified name childNS results in creation of a namespace of that name in the current namespace context, ie. nsA::childNS.

12.5.3. Resolving command names

Resolution of command names differs from that of variable and namespace names in that additional mechanisms, *name imports* and *namespace paths*, are available to control the ways names are resolved.

Resolution proceeds in the following manner:

1. The current namespace is checked first.
2. If not found there, all namespaces on the namespace path, which is a list of namespaces, are checked in the order of their appearance.
3. If the command is still not found, the global namespace is looked up.
4. As a final resort, the `namespace unknown` handler is called.

In steps 1-3 above, a command may exist in a namespace either because it is defined there **or because it has been imported into the namespace**.

12.5.3.1. Importing names: `namespace export|import|forget`

Namespace export and import is a convenience feature that allows a namespace to mark selected commands as exported and callable without requiring any namespace qualifiers from any other namespace that imports them.

The namespace from where commands are being exported uses one or more `namespace export` commands to designate which commands are to be exported.

```
namespace export ?-clear? ?PATTERN ...?
```

If no arguments are specified, the command returns the list of names that are currently exported from the namespace. Otherwise the list of *PATTERN* arguments are **appended** to the current list of patterns exported from the namespace. Any command in the namespace whose name matches any pattern in this export list using glob pattern (see Section 4.11) matching can be imported into other namespaces. If the `-clear` option is specified, the export list is reset to empty before the patterns are added to it.

The complementary command to `namespace export` is `namespace import` which is invoked from the namespace into which commands are to be imported.

```
namespace import ?-force? ?PATTERN ...?
```

If no arguments are supplied, the command returns a list of the commands that have been imported into the current namespace. Otherwise, for every command that matches any of the *PATTERN* arguments a new command is created in the current namespace that points to the original command. *PATTERN* may be fully or partially qualified but only the last component is treated as a glob pattern.

By default, if there is an existing command of the same name as the command being imported, an error is generated. If the `-force` option is specified, then instead of generating an error, the imported command overwrites the existing one.

Here is a simple example to illustrate the basic working of export and import of names.

```
namespace eval nsA {
    proc aproc {} {puts "aproc called"}
    proc bproc {} {puts "bproc called"}
    proc cproc {} {puts "cproc called"}
    namespace export a* b*
}
namespace eval nsB {
    namespace import {::nsA::[ac]*}
}
```

Let us check what commands actually land up being exported and imported.

```
namespace eval nsA { namespace export } → a* b*
namespace eval nsB { namespace import } → aproc
```

And when we try to invoke commands in the nsA from nsB without using any namespace qualifiers:

```
namespace eval nsB { aproc } → aproc called
namespace eval nsB { cproc } Ø invalid command name "cproc" ❶
namespace eval nsB { bproc } Ø invalid command name "bproc" ❷
```

❶ Fails because cproc is not exported from nsA
❷ Fails because bproc is not imported into nsB

The namespace export command is "sticky" in that even a new command defined **after** it has been executed will also be exported if its name matches a pattern in the list of exported command patterns. On the other hand, the namespace import command only imports those commands that already existed at the time it was invoked. For example, let us define a new command that matches a pattern we previously exported.

```
namespace eval nsA {
    proc acommand {} { puts "acommand called" }
}
```

Now let us invoke it from nsB.

```
% namespace eval nsB { acommand } ❶
Ø invalid command name "acommand"
% namespace eval nsB { namespace import {::nsA::[ac]*} }
% namespace eval nsB { acommand } ❷
→ acommand called
```

❶ Fails because namespace import takes a snapshot
❷ Succeeds after we invoke namespace import again

Notice that we did not have to re-export the command but we did have to do a re-import.

Another point to note is that imported commands can be re-exported from the importing namespace.

```
namespace eval nsB { namespace export aproc }
namespace eval nsC {
    namespace import ::nsB::aproc
    aproc
}
→ aproc called
```

Finally, you can undo the effect of a namespace import with the namespace forget command.

```
namespace forget ?PATTERN ...?
```

The PATTERN arguments are of the same form as accepted by namespace import except that they can also be simple names not be qualified by namespaces. If namespace qualifiers are present, the argument is matched against exported commands from all matching namespaces and the commands imported into the current namespace, if any, are removed. If no namespace qualifiers are present, any command matching the pattern in the current namespace are removed if they were imported.

Thus either of the following would undo the effect of the `namespace` `import` into the nsB namespace.

```
namespace eval nsB { namespace forget ::nsA::aproc }
namespace eval nsB { namespace forget acommand }
```

As we can see both commands are removed from nsB.

```
namespace eval nsB { aproc }      ∅ invalid command name "aproc"
namespace eval nsB { acommand } ∅ invalid command name "acommand"
```

12.5.3.2. Namespace paths: `namespace` `path`

The other way to set up access to another namespace's commands without requiring qualification for every call is through namespace paths. A *namespace path* is a list of namespaces that should be searched to locate a command if it is not found in the current namespace. This list is specific to a namespace and can be set up with the `namespace` `path` command.

```
namespace path ?NAMESPACELIST?
```

If the `NAMESPACELIST` argument is specified, it should be a list of namespace names and the namespace path for the context in which the command is called is set to this value. If no argument is specified, the command just returns the current namespace path.

The example below illustrates several points about namespace paths.

```
proc global_proc {} {puts "global_proc called"}
namespace eval nsA {
    proc nsA_proc {} { puts "nsA_proc called" }
    namespace eval nsB { proc nsB_proc {} { puts "nsB_proc called" } }
}

namespace eval nsC {
    namespace path [list ::nsA ::]
    puts "The namespace path is now [namespace path]."
    proc nsC_proc {} { nsB::nsB_proc }
    global_proc
    nsA_proc
    nsC_proc
}
→ The namespace path is now ::nsA ::.
  global_proc called
  nsA_proc called
  nsB_proc called
```

Note from this example that

- The global namespace is like any other namespace and can be explicitly placed at any position on the namespace path for a namespace if so desired. Keep in mind though that commands in the global namespace will automatically be resolved in any context even if they do not appear on the namespace path. Adding it to the path only makes sense if you want it to be searched **before** some of the namespaces in the path.

- The namespace path is searched not only for simple names but for relative names with namespace components.

- The namespace path is effective not only within a `namespace` `eval` but also within procedures defined in that namespace.

12.5.3.3. Comparing namespace imports and paths

Though similar in their ability to reference program elements in one namespace from another without explicit qualification, the namespace import/export and path features work differently.

Importing a name into a namespace with namespace import actually **creates** a command in that namespace which points to the command in the exporting namespace. On the other hand, namespace path does **not** create a new command. The following example will clarify the differences.

```
namespace eval nsA {
    proc aproc {} { puts "aproc called" }
    namespace export aproc
}
namespace eval importer { namespace import ::nsA::aproc }
namespace eval pathfinder { namespace path ::nsA }
```

The command nsA::aproc can be accessed from both namespace without qualification.

```
namespace eval importer { aproc }   → aproc called
namespace eval pathfinder { aproc } → aproc called
```

However, the two are not equivalent as the following fragments illustrate.

```
importer::aproc    → aproc called
pathfinder::aproc Ø invalid command name "pathfinder::aproc"
```

In the first case, importer::aproc can be directly called because importing actually creates a command of that name in importer. The second call raises an error because there is no aproc in pathfinder and the namespace path only applies to commands invoked from within pathfinder. To confirm,

```
info commands importer::*   → ::importer::aproc
info commands pathfinder::* → (empty)
```

Here is a slightly different effect of the same.

```
namespace eval nsB { namespace path ::importer }
namespace eval nsC { namespace path ::pathfinder }
```

If we try to invoke aproc from the nsB and nsC, the first works and the second does not.

```
namespace eval nsB { aproc } → aproc called
namespace eval nsC { aproc } Ø invalid command name "aproc"
```

Another way of viewing the difference is that imports link to the original command whereas the path mechanism searches for the command by name along the search path. Thus if we were to rename the original command or the one in the importing namespace, imports would continue to work.

```
rename ::nsA::aproc ::nsA::aproc2                → (empty) ❶
namespace eval importer {aproc}                  → aproc called
rename ::importer::aproc ::importer::a_better_name → (empty) ❷
importer::a_better_name                          → aproc called
```

❶   Rename the original procedure
❷   Rename the procedure in the importing namespace

On the other hand, the path mechanism would no longer locate a command of that name in the defining namespace.

```
% namespace eval pathfinder {aproc}
Ø invalid command name "aproc"
```

12.5.3.4. Handling unknown commands: `namespace unknown`

If all the mechanisms discussed above to resolve a command fail, Tcl will call the *unknown command handler* for the namespace. This handler is set independently for each namespace by calling the `namespace unknown` command from the context of the namespace to which it is to be applied.

```
namespace unknown ?COMMANDPREFIX?
```

If specified, `COMMANDPREFIX` should be a list consisting of the name of a command and optionally zero or more arguments. When a command cannot be resolved within the namespace, the entire command including arguments is appended to `COMMANDPREFIX` and invoked. The result is then returned as the result of the original command.

In our example, if a command is not located when called from the `nsA` namespace, we will try invoking it as an external program instead.

⚠️ | This example is for pedagogic purposes only. It is not safe programming practice!

```
% namespace eval nsA { ls *.adocgen }
Ø invalid command name "ls"
% namespace eval nsA { namespace unknown [list exec -keepnewline --]}
→ exec -keepnewline --
% namespace eval nsA { ls *.adocgen}
→ basics.adocgen
  code.adocgen
...Additional lines omitted...
```

If `COMMANDPREFIX` is not specified, the command returns the current handler for the namespace.

```
% namespace eval nsA {namespace unknown}
→ exec -keepnewline --
```

Note that the handler for the namespace is only executed when a command lookup fails within the specified namespace context. It will not be invoked either when lookups fail in some other context or even when an attempt is made to call a non-existent command within the handler's context from outside the context. Thus neither of the following will invoke our handler.

```
namespace eval nsB { ls *.ad} Ø invalid command name "ls" ❶
nsA::ls                        Ø invalid command name "nsA::ls" ❷
```

❶ Fails because `nsB` does not have an unknown handler.
❷ Fails because call is made from outside the `nsA` namespace context.

> If no unknown command handler is set for a namespace, the global handler `::unknown` will be called instead (see Section 3.5.1.2).

12.5.4. Introspecting name resolution: `namespace which`, `namespace origin`

Tcl provides two commands `namespace which` and `namespace origin` which map names to their fully qualified versions. We will use a slightly modified version of the example in our previous section. We add an additional namespace `middleman` which imports and re-exports from `nsA`.

```
namespace eval nsA {
    proc aproc {} { puts "aproc called" }
    namespace export aproc
}
namespace eval middleman {
    namespace import ::nsA::aproc
    namespace export aproc
}
namespace eval importer { namespace import ::middleman::aproc }
namespace eval pathfinder { namespace path ::nsA }
```

We will start with `namespace which`.

```
namespace which ?-command? ?-variable? NAME
```

The command returns the fully qualified version of *NAME* as per the name resolution rules discussed earlier. The switches `-command` and `-variable` indicate whether *NAME* refers to a command or a variable respectively. If neither switch is specified, `-command` is assumed.

Let us see how the command works with imported names and namespace paths.

```
namespace eval importer { namespace which -command aproc}   → ::importer::aproc
namespace eval pathfinder { namespace which -command aproc} → ::nsA::aproc
```

Notice that in the first instance, the returned fully qualified name is within the current namespace. This makes sense since the import of a name actually creates a command of that name in the importing namespace as we saw in the previous section.

In the second instance, there was no command of that name created in the `pathfinder` namespace. Hence the namespace path of `pathfinder` is searched and the fully qualifed name of `aproc` is returned corresponding to the namespace in which it was found.

The `-variable` switch works similarly except that instead of following the name resolution rules for commands, it follows the name resolution rules for variables. It returns the fully qualified name of the variable if it has been created and an empty string otherwise.

```
namespace eval nsA {
    variable avar
    proc demo {} {
        variable avar
        namespace which -variable avar
    }
}
nsA::demo
→ ::nsA::avar
```

Note that it suffices for the variable to have been **created**, it need not be **defined** (see Section 3.6.5.3 for the distinction).

We leave it to the reader to experiment further with it and move on to the namespace origin command.

```
namespace origin NAME
```

While namespace which locates a command and returns the fully qualified path, namespace origin serves a different purpose. The fully qualified name it returns is that of **original** command even if there are "intermediate" namespaces importing and re-exporting the name. Contrast the two in our example:

```
namespace eval importer { namespace which -command aproc} → ::importer::aproc
namespace eval importer { namespace origin aproc}          → ::nsA::aproc
```

We see that namespace which returns the current namespace importer since the import of aproc resulted in the creation of a command of that name in the importer namespace itself. On the other hand, namespace origin traverses back through all intermediate links (middleman in our case) to the original command nsA::aproc.

The command will work with namespace paths as well.

```
% namespace eval pathfinder { namespace origin aproc}
→ ::nsA::aproc
```

> The namespace which command can be used in lieu of info commands to check for the existence of a command. It is often preferred because unlike info commands, it does not treat its argument as a **pattern**. This makes its use safer when checking for existence of commands whose names may contain wildcard characters [1].

12.6. Namespace ensembles

In most languages, namespaces are limited to a single (albeit important) purpose — that of preventing conflicts between identifiers defined by indepedent modules. In Tcl, namespaces also provide the basis of another piece of useful functionality, *ensemble commands*.

12.6.1. Ensemble commands

An ensemble command is a command that has subcommands that collectively perform a set of related functions. Tcl itself has several built-in commands that are ensembles, such as string that operates on strings, and clock that implements date and time related functions.

Namespaces offer a means for you to construct your own ensemble commands.

12.6.2. Creating a simple ensemble: `namespace ensemble create`

Assume we want to encapsulate various operations related to Fibonacci sequences under the command fib. We will support three simple commands,

```
fibonacci sequence N
fibonacci nth N
fibonacci sum N
```

that return a sequence of length N, the N'th number in the sequence and the sum of a sequence of length N respectively.

---

[1] Use of non-alphabetic characters in procedure names is not uncommon. For example, you will find ? suffixes used for procedure names that return booleans, or * for extended forms of standard commands.

```
package require math
namespace eval fib {
    proc nth {n} { return [math::fibonacci $n] }
    proc sequence {n} {
        set seq {}
        for {set i 1} {$i <= $n} {incr i} {
            lappend seq [nth $i]
        }
        return $seq
    }
    proc sum {n} {return [::tcl::mathop::+ {*}[sequence $n]]}
}
```

We can now call it using the standard namespace syntax

```
fib::sequence 3 → 1 1 2
fib::sum 3       → 4
```

To convert this to an ensemble command we need to make use of the `namespace ensemble create` command.

```
namespace ensemble create ?OPTION VALUE?
```

For our example, this is very simple. By default, when no options are specified, `namespace ensemble create` will create an ensemble command of the same name as the namespace from which it is called. The subcommands will be the exported commands from the namespace. So in our case, all we need to do is

```
namespace eval fib {
    namespace export *
    namespace ensemble create
}
→ ::fib
```

This creates an ensemble command of the same name as the namespace, `fib` in our example, which we can then invoke in any of the following forms.

```
fib nth 4        → 3
fib sequence 3 → 1 1 2
fib sum 5        → 12
```

12.6.2.1. Naming an ensemble command

Readers who are at least half-awake will object that the name of the command is wrong; we wanted it to be `fibonacci`, not `fib`. The obvious way to fix this would be to change the name of the containing namespace itself to `fibonacci`. We will instead follow a different path of configuring the ensemble as that provides more flexibility in cases where the ensemble construction is not based on a single namespace. The `-command` option allows us to define the name to be used for the ensemble command.

```
% namespace eval fib {namespace ensemble create -command ::fibonacci}
→ ::fibonacci
% fibonacci nth 6
→ 8
```

Note the value we passed to the `-command` option was fully qualified. Otherwise we would have created the command `fibonacci` inside the `fib` namespace instead of at the global level as we wanted.

The -command option is also useful when defining an ensemble command without creating a new namespace as we will see in a later example.

The namespace ensemble create command also takes additional options. These are the same as described below for the namespace ensemble configure command.

12.6.3. Configuring ensembles

Having looked at the simplest method for creating ensembles, we will now look at configuration options that allow for more flexible construction of ensembles. Ensembles are configured using the namespace ensemble configure command.

```
namespace ensemble configure COMMAND ?OPTION ?VALUE? ...?
```

The COMMAND argument is the name of the ensemble command being configured. If no options are specified, the command returns the current values of the options.

```
% namespace ensemble configure ::fibonacci
→ -map {} -namespace ::fib -parameters {} -prefixes 1 -subcommands {} -unknown {}
```

If a single option is specified, with no associated value argument, the command returns the value of the option.

```
namespace ensemble configure ::fibonacci -namespace → ::fib
```

If more than one argument is specified after the ensemble name COMMAND, they are interpreted as option and value pairs and the ensemble command is configured as per the specified values. An exception is -namespace which is a read-only option and cannot be modified.

12.6.3.1. Subcommand configuration: -subcommands, -map

In our example above, all **exported** commands from the fib namespace became subcommands of the command ensemble. There are times when this is not the desired behaviour. The -subcommands option allows specification of exactly which subcommands are available through the ensemble.

```
% namespace ensemble configure ::fibonacci -subcommands {nth sum}
```

Now only the two listed commands are callable through the ensemble.

```
% fibonacci nth 4
→ 3
% fibonacci sum 5
→ 12
% fibonacci sequence 3 ❶
∅ unknown or ambiguous subcommand "sequence": must be nth, or sum
```

❶   Error because sequence was not included in the -subcommands value

> The commands in subcommand list need not be exported commands.

By default, the value of the -subcommands option is the empty list in which case, as we saw earlier, all exported commands from the namespace become ensemble subcommands. We can reset to that default behaviour so sequence becomes a subcommand again.

```
namespace ensemble configure ::fibonacci -subcommands {} → (empty)
fibonacci sequence 3                                    → 1 1 2
```

While the -subcommands option lets us control which commands in the namespace are exposed as subcommands in the ensemble, what if we want a subcommand that is not implemented within the namespace at all? This is where the -map option comes in. It lets an ensemble subcommand be mapped to any command prefix.

For example, consider the procedure fib::nth which does nothing other than call the math::fibonacci command from the math library. Instead of defining that procedure, we could have used the -map option to directly invoke math::fibonacci. Let us use this method to define a new subcommand term that does the same thing as nth. Additionally, making note that the mapping target is a **command prefix** and not just a command, we will define another subcommand term4 which always returns the fourth number in the sequence.

```
namespace ensemble configure ::fibonacci -map {
    term ::math::fibonacci
    term4 {::math::fibonacci 4}
} -subcommands {term term4 sequence sum}
```

And of course, it all works as advertised.

```
fibonacci term 4 → 3
fibonacci term4  → 3
```

The -map and -subcommands options options together control the subcommands available in an ensemble and the implementations to which they are mapped.

- If neither -subcommands nor -map is configured (or are empty) the ensemble subcommands are exactly those exported by the namespace.
- If the -map option is specified but -subcommands was an empty list (or unspecified), the ensemble commands are exactly the keys of the dictionary passed as the -map option value.
- If -subcommands is specified (and not empty) the ensemble subcommands are exactly those listed in the option value. The corresponding implementation is that supplied in the -map dictionary argument if the subcommand is found there, or a procedure of the same name in the namespace linked to the ensemble.

12.6.3.2. Subcommand prefixes: option -prefixes

A feature of ensemble commands is that by default they accept unique prefixes for subcommands. For example,

```
% fibonacci su 4 ❶
→ 7
% fibonacci s 4 ❷
∅ unknown or ambiguous subcommand "s": must be sequence, sum, term, or term4
```

❶ Uniquely identifies subcommand sum.
❷ Error because ambiguous prefix.

This feature can be controlled with the -prefixes option which is enabled by default. If you want exact matching of subcommands, you can disable the feature by setting the option to false.

```
% namespace ensemble configure ::fibonacci -prefixes false
% fibonacci su 4 ❶
∅ unknown subcommand "su": must be sequence, sum, term, or term4
```

❶ Unique prefixes no longer work

12.6.3.3. Subcommand positioning: option -parameters

For additional flexibility, the position in which the subcommand appears in the ensemble command can be controlled with the -parameters option. For example, suppose we wanted to implement an ensemble command arith for simple arithmetic using infix notation. So instead of commands like

```
arith + 3 4
arith - 5 2
```

we would be able to write

```
arith 3 + 4
arith 5 - 2
```

Thus in effect we want the subcommands +, - to be positioned **after** the first argument to the command.

The first step is to define a simple namespace implementing the commands.

```
namespace eval arith {
    proc + {operand increment} {expr {$operand + $increment}}
    proc - {operand decrement} {expr {$operand - $decrement}}
}
```

Then we use the -parameters option to specify the number of parameters that appear **before** the subcommand in the ensemble. The option value is a list of elements corresponding to the number of arguments that should appear before the subcommand. The actual values of the list elements are only used to generate meaningful error messages and do not have any relevance otherwise.

For our example, we want a single argument before the subcommand.

```
namespace eval arith {
    namespace export + -
    namespace ensemble create -parameters {operand}
}
→ ::arith
```

We can now use infix notation for the ensemble.

```
arith 3 + 4 → 7
arith 5 - 2 → 3
arith         Ø wrong # args: should be "arith operand subcommand ?arg ...?" ❶
```

❶   Note use of operand in error message

For more substantive examples of how -parameters might be used, see Tcl Improvement Proposal #314[2] which in addition to specifying the behaviour, also provides motivation for the feature and examples.

12.6.4. Handling unknown subcommands: option -unknown

Just as for global commands and commands within a namespace, Tcl provides a means for an application to handle errors when an ensemble command is not defined. This is done with the -unknown ensemble configuration option.

---

[2] http://www.tcl.tk/cgi-bin/tct/tip/314.html

If the ensemble's -unknown option has the default value of the empty string, any attempt to invoke a subcommand that is not defined will result in an error.

```
arith 2 * 3 Ø unknown or ambiguous subcommand "*": must be +, or -
```

If the -unknown option is configured for the ensemble and is not the empty string, it is registered as the unknown handler for subcommands for that ensemble and is called when a subcommand cannot be resolved as described in the previous sections. The entire attempted command is appended to the unknown handler before its invocation.

The return value from the unknown handler must be a valid (possibly empty) list. If the returned list is not empty, Tcl replaces the original ensemble command as well as the original subcommand with the words from the returned list and re-executes the replacement with the additional arguments from the original invocation.

An example will clarify how this works. Let us assume that for our arith ensemble command, if the subcommand has not been defined we will attempt to treat it as a standard operator defined in the tcl::mathop namespace. So we define the following unknown handler delegator for the ensemble.

```
namespace eval arith {
    proc delegator {args} {
        if {[llength $args] != 4} {
            error "Wrong number of arguments: should be \"[lindex $args 0] operand \
                operator operand\""
        }
        return ::tcl::mathop::[lindex $args 2]
    }
}
namespace ensemble configure ::arith -unknown ::arith::delegator
```

Now, if we try to execute a subcommand that has not been defined for arith, say *, delegator will be invoked with the name of the ensemble and all additional arguments. So when we execute

```
arith 2 * 3 → 6
```

delegator is invoke with arguments ::arith, 2, * and 3. After some error checking, the command returns ::tcl::mathop::*. Tcl then executes this command passing it the additional arguments 2 and 3 and returning the result.

If the unknown handler returns a list that is empty, Tcl will then attempt to run the **original** command again. What this does is to allow the unknown handler to add appropriate commands "on the fly".

Continuing with our example, considering going through the unknown handler for every invocation of * would be inefficient, we can instead add each subcommand the first time it is referenced.

Let us see how this might be implemented by redoing our example from scratch. This also illustrates that we can create an ensemble without an explicit namespace by using the global one in conjunction with the -command option.

```
namespace delete ::arith ❶
proc delegator {args} {
    if {[llength $args] != 4} {
        error "Wrong number of arguments: should be \"[lindex $args 0] operand operator \
            operand\""
    }
    lassign $args cmd - op
    set escaped_op [string map {* \\* ? \\? [ \\[ ] \\] \\ \\\\} $op]  ❷
    if {[llength [info commands ::tcl::mathop::$escaped_op]] == 0} {
        error "Invalid operator \"$op\""
    }
    set map [namespace ensemble configure $cmd -map]
```

```
    dict set map $op ::tcl::mathop::$op
    namespace ensemble configure $cmd -map $map
    return ""
}
namespace ensemble create -command arith -map {} -parameters {operand} -unknown [namespace \
    current]::delegator -prefixes false ❸
→ ::arith
```

❶ Get rid of our prior example
❷ The `string map` is used to escape operators that might also be interpreted as special characters by `info` commands
❸ Prefixes disabled so (for example) `=` will not be treated as a valid prefix of `==`.

We have created the ensemble command as before but with **no subcommands defined**. Now everytime we invoke a new (valid) operator, it will be added as a subcommand. We can see this in the following sequence.

```
namespace ensemble configure arith -namespace → :: ❶
namespace ensemble configure arith -map       → (empty) ❷
arith 2 == 3                                  → 0
arith 2 * 3                                   → 6
arith 2 = 3                                   → Ø Invalid operator "=" ❸
namespace ensemble configure arith -map       → == ::tcl::mathop::== * ::tcl::mathop::* ❹
```

❶ Notice linked namespace is the global namespace
❷ Initial subcommand map is empty
❸ `=` is not a valid operator
❹ The subcommand map is dynamically filled so `delegator` is called only once per operator

This method of updating subcommands on the fly is often seen in code that implements object systems based on namespaces where object method names are discovered dynamically[3].

12.6.5. Checking for ensembles: `namespace ensemble exists`

The command `namespace ensemble exists` returns 1 if its argument is a ensemble command and 0 otherwise.

```
namespace ensemble exists string       → 1
namespace ensemble exists puts         → 0
namespace ensemble exists nosuchcommand → 0
```

12.6.6. Nested ensembles

Ensembles can be nested so that the command takes multiple subcommand arguments that form a hierarchy.

```
COMMAND SUBCOMMAND SUBCOMMAND ...  ARGUMENTS
```

Suppose there is a image manipulation package which implements a `image` command that handles PNG and JPEG image formats and some set of operations like resizing or rotating. A possible interface it presents might look like

```
image png resize PNGDATA WIDTH HEIGHT
image jpeg rotate PNGDATA DEGREES
```

This interface is easy to create with nested namespaces as shown below.

---

[3] The Windows COM IDispatchEx interface being one example.

```
namespace eval image::png {
    proc rotate {imagedata degrees} {
        puts "Rotating PNG image"
    }
    proc resize {imagedata height width} {
        puts "Resizing PNG image"
    }
    namespace export *
    namespace ensemble create
}

namespace eval image::jpeg {
    proc rotate {imagedata degrees} {
        puts "Rotating JPEG image"
    }
    proc resize {imagedata height width} {
        puts "Resizing JPEG image"
    }
    namespace export *
    namespace ensemble create
}
namespace eval image {
    namespace export *
    namespace ensemble create
}
→ ::image
```

We can then call the commands in straightforward fashion.

```
% image png rotate "Some binary PNG" 90
→ Rotating PNG image
% image jpeg resize "Some binary JPEG" 640 480
→ Resizing JPEG image
```

12.6.7. Examples of ensembles

Namespace ensembles are commonly used for a variety of purposes so we present some small examples illustrating their use.

12.6.7.1. Enhancing existing commands

There are times when there is some commonly used functionality you wish was provided by a built-in command. For example, a common pattern seen when using dictionaries is to use a default value if a key is not present in a dictionary. Because the dict get command raises an error on an attempt to access to a key that is not present, a check for existence is needed. This is such a commonly used idiom that most programmers have some version of the following code.

```
proc dict_get_with_default {dictval key {defval ""}} {
    if {[dict exists $dictval $key]} {
        return [dict get $dictval $key]
    } else {
        return $defval
    }
}
```

Now if you wanted to add this functionality into the dict command itself for a more "integrated" feel, you could add it to the dict ensemble.

```
set map [namespace ensemble configure ::dict -map]
dict set map lookup dict_get_with_default
namespace ensemble configure ::dict -map $map
```

This now allows a more natural access to the functionality.

```
set adict [dict create a 1 b 2] → a 1 b 2
dict lookup $adict a 0              → 1
dict lookup $adict c 0              → 0
```

There is one caveat though. Someone may have the same bright idea and define a new subcommand of the same name which has a different function. It is probably wise to use a prefix or some other means to prevent name clashes.

12.6.7.2. Indexing lists by name

Data records in Tcl are often stored as lists with individual fields accessed or set using lindex or lset. For example, a student record might be stored as a list containing the student name, age and college.

```
set rec {Manute 18 {College of Engineering}}
puts "[lindex $rec 0] is [lindex $rec 1]."
→ Manute is 18.
```

Accessing fields using list indices can be tedious, particularly when the number of fields is large. One can use dictionaries or write accessor functions to access the list. The former is not always under your control depending on the interface returning the data. The latter is tedious to do for every record "type" in the application.

A more convenient way might be if we could define a student record as a list of fields,

```
record student {Name Age College}
```

and then be able to access fields using names instead of indices making it more readable.

```
puts "[student $rec name] is [student $rec age]."
```

Let us see how to provide such a facility through a generic command, record, that will create an ensemble to retrieve fields by name.

The record command first ensures that there are no conflicts with existing commands of the same name as the record being created. It then creates an ensemble command of the given record name whose subcommands are field names and map to an anonymous procedure (defined in the variable accessor) that returns or updates the appropriate field.

```
proc record {recname fields} {
    if {[uplevel 1 [list namespace which $recname]] ne ""} {
        error "can't create command '$recname': A command of that name already exists."
    }

    set index -1
    set accessor [list ::apply {
        {index rec args}
        {
            if {[llength $args] == 0} {
                return [lindex $rec $index]
            }
```

```
            if {[llength $args] == 1} {
                return [lreplace $rec $index $index [lindex $args 0]]
            }
            error "Invalid number of arguments."
        }
    }]
    set map {}
    foreach field $fields {
        dict set map $field [linsert $accessor end [incr index]]
    }

    uplevel 1 [list namespace ensemble create -command $recname -map $map -parameters rec]
}
```

As an aside, note that the namespace ensemble is created in the context of the caller via uplevel.

We can now access fields more conveniently.

```
% record student {name age college}
→ ::student
% student $rec age                   ❶
→ 18
% set rec [student $rec age 19] ❷
→ Manute 19 {College of Engineering}
```

❶   Retrieve a field
❷   Update a record

Since the code is generic, we could of course define other record types as well.

```
% record automobile {manufacturer model color}
→ ::automobile
% automobile {Ferrari CaliforniaT red} color
→ red
```

12.6.7.3. Command objects using ensembles

Our final example uses namespace ensembles to illustrate implementation of a "command as an object" idiom. We will implement an ordered set that can be used as follows

```
set oset [ordered_set::new] ❶
$oset add foo ❷
$oset contents ❸
$oset remove foo ❹
$oset destroy ❺
rename $oset "" ❻
```

❶   Creates a new empty ordered set
❷   Adds an element to the set if it does not exist
❸   Returns the contents of the set
❹   Removes element foo from the set if it exists
❺   Destroy the ordered set
❻   Alternate means of destroying the set

An ordered set is ordered in that elements are preserved in the order that they are added. It so happens that Tcl dictionaries are order preserving so our implementation is very simple. We keep a nested dictionary, the first level being indexed by an object identifier with the corresponding value being the content of the corresponding set, also stored as a dictionary thanks to the order preserving properties.

```
namespace eval ordered_set {
    variable nextid 0
    variable sets {}

    proc add {id elem} {
        variable sets
        dict set sets $id $elem $elem
        return
    }

    proc remove {id elem} {
        variable sets
        if {[dict exists $sets $id $elem]} {
            dict unset sets $id $elem
        }
        return
    }

    proc contents {id} {
        variable sets
        return [dict keys [dict get $sets $id]]
    }

}
```

Now we just have to arrange for creating ordered sets and cleaning up when they are destroyed. The latter is easy, we just have to get rid of the data, so we show that first. The only non-obvious part is the additional args argument, the reason for which will soon be clear.

```
proc ordered_set::cleanup {id args} {
    variable sets
    dict unset sets $id
}
```

We are left with needing a means to create an ordered set command object which was more or less the whole point of this whole exercise.

We first generate a unique name for the command object using the namespace variable nextid as a identifier counter and initialize the corresponding content of the sets dictionary to empty.

We then create a map of ensemble subcommands that are mapped to a command prefix consisting of the corresponding procedure name with the identifier of the object being passed as the first argument. This map is then used to create an ensemble command whose name is the name of the object.

The last thing we have to deal with is object destruction. Any command can be deleted by using rename to rename it to the empty string. The same will also hold for our object command so we need to arrange for the related data to be cleaned up from the sets dictionary when the command is deleted. We do this by setting a trace on the command object to invoke our cleanup procedure when it is deleted. The trace callback appends additional arguments (that we do not use) which is why cleanup definition had an unused args parameter. Note that we had also added a destroy subcommand to our map that does the same thing as syntactic sugar.

Here is the command object creation procedure.

```
proc ordered_set::new {} {
    variable nextid
    variable sets
    set objname "::oset#[incr nextid]"
    dict set sets $nextid [dict create]
    set map [dict create \
                add [list add $nextid] \
                contents [list contents $nextid] \
                remove [list remove $nextid] \
                destroy [list ::rename $objname ""]]   ❶

    namespace ensemble create -command $objname -map $map
    trace add command $objname delete [list [namespace current]::cleanup $nextid]
    return $objname
}
```

❶   Note rename is fully qualified else it will default to the current namespace.

We can try out our ordered sets.

```
set oset [ordered_set::new]   → ::oset#1
$oset add fee                 → (empty)
$oset add fie                 → (empty)
$oset add fo                  → (empty)
$oset contents                → fee fie fo
$oset add fie                 → (empty) ❶
$oset contents                → fee fie fo
$oset remove fee              → (empty)
$oset contents                → fie fo
$oset destroy                 → (empty)
$oset contents                Ø invalid command name "::oset#1" ❷
```

❶   Duplicate element, preserves existing order
❷   Object has been destroyed

Of course, our toy implementation only illustrates some basic techniques. A real implementation would be generalized, support inheritance and other features. The Tcl Wiki has several examples of object-oriented programming systems built on top of namespaces. With the advent of TclOO there are few reasons to roll your own.

12.7. Chapter summary

In this chapter we examined namespaces and their use in the modularization of large code bases. The dynamic nature of Tcl widens their role beyond that in other languages to form the basis of simple object based systems, nested command structures and such. In Chapter 14 we will study Tcl's native object-oriented features which build on some of these facilities to offer some of the most flexible object-oriented designs found in languages.

While namespaces are targeted towards one aspect of modularization, there is another aspect — packaging common functionality into libraries — that we will look at next.

13

Libraries and Packages

One of the basic principles of software development is to collect implementations of commonly useful and widely applicable functionality into *libraries* that can be shared amongst multiple applications. The simplest mechanism for implementing such a library is as a set of procedures in a file that is then sourced by any application that makes use of its functionality. In the case of a large library, this file could be a "main" file that optionally sources other files that implement parts of the library. This simple approach needs to be enhanced to provide a means of locating the library on the file system, loading on demand, versioning etc.

For historical reasons, Tcl includes multiple mechanisms for working with libraries:

- An index based system where procedure names are stored in an index file that is looked up and loaded on procedure invocation
- Tcl *packages* which define a structure for versioning, locating and loading libraries implemented through multiple scripts and extensions
- Tcl *modules* that implement a simpler and more performant way to locate and load libraries implemented in a single file

We will describe the particulars pertaining to all three in this chapter. Yet another alternative, *tclkits*, is dealt with in Section 19.4.

13.1. The Tcl system library

Some of the Tcl core commands, for example `clock`, are themselves implemented as a library of Tcl scripts. The name of the directory where this system library resides is stored in the `tcl_library` global variable.

```
set tcl_library → c:/tcl/866/x64/lib/tcl8.6
```

The same information is also available via the `info library` command which simply returns the value of the `tcl_library` global variable.

```
info library → c:/tcl/866/x64/lib/tcl8.6
```

When Tcl starts up, it sets the value of the `tcl_library` variable by checking the following locations in order for library scripts:

- The directory specified by the `TCL_LIBRARY` environment variable if it exists and references an appropriate directory
- Directories relative to a default location that is defined when the Tcl executable was compiled
- Directories relative to the location of the Tcl executable
- Directories relative to the current working directory

The above locations are checked in order and the first one that contains the expected library scripts is used to initialize `tcl_library`.

13.2. Loading libraries on demand: `auto_load`

In Section 3.5.1.2 we described Tcl's default handling of unknown commands. One of the steps described there was how the unknown command handler uses the `auto_load` command to locate definitions of commands that are not currently known to the Tcl interpreter. We now look at `auto_load` in more detail.

When the default handler for unknown commands is called, it uses the `auto_load` command to try and locate the definition of the command that was invoked.

```
auto_load COMMANDNAME
```

The command returns 1 if the command could be located and defined and 0 otherwise. It works by searching the list of directories stored in the global variable `auto_path` for files named `tclIndex`. Each `tclIndex` file contains an index that maps command names to the associated script for creating that command. Generally, this command creation script is simply a `source` command that executes a file in the directory containing the `tclIndex` file.

13.2.1. The `tclIndex` files

The `tclIndex` file is actually just a Tcl script that adds entries to an array `auto_index` that maps command names to the script to be executed to define that command. For example, here is a line from the `tclIndex` file for the Tcl system library.

```
set auto_index(history) [list source [file join $dir history.tcl]]
```

The very first time the `history` command is referenced, Tcl's unknown command handler will invoke auto_load which will

- First check the `auto_index` global array for an entry for the `history` command. If found, it executes the corresponding script which presumably will define the required command.
- If no entry is found in the `auto_index` array, `auto_load` will evaluate all `tclIndex` files found in the directories listed in the `auto_path` global variable.
- The first step is then repeated except that if there is still no matching entry in the `auto_index` array, `auto_load` returns 0 indicating the command was not found.

A `tclIndex` file may be written and maintained manually by hand but is usually generated using the `auto_mkindex` command.

```
auto_mkindex DIR ?GLOBPAT ...?
```

The command processes all files in the directory *DIR* that match any of the file name patterns *GLOBPAT*. These patterns are in the syntax used by the `glob` command.

If no *GLOBPAT* arguments are specified, the command defaults to `*.tcl`. A `tclIndex` file containing `auto_index` entries of the form we saw above is then written to the same directory.

> ⚠ If you have procedure names that contain special `glob` pattern characters such as `*`, the `auto_mkindex` command can get confused.

13.3. Packages

Tcl *packages* are one way of bundling a library of commands and procedures identified by a name and version. An application desiring to use the functionality provided can request the package to be loaded and even demand a specific version of the package.

13.3.1. Naming packages

Packages are identified by their name, for example http. Package names may contain arbitrary characters although it is advisable to avoid special characters. Packages may contain the :: namespace separator character sequence as well. Note however that these are **not** treated as namespace characters as packages have no direct correlation with namespaces.

Packages implemented as Tcl modules impose some additional restrictions on package names which we discuss in Section 13.5.

13.3.2. Package versioning

Over time, new releases of a library contain feature enhancements, bug fixes and so on. Version numbers are used to distinguish these releases. A Tcl installation may contain multiple versions of a single package and applications can choose which version they wish to use.

13.3.2.1. Package version syntax

Version identifiers take the form of a sequence of decimal numbers generally separated by a . character. For example, 8, 8.6 and 8.6.1000 are all valid version numbers. Version numbers in this form, using only . separators, are assigned to *stable* releases, i.e. releases that have been deemed ready for production use.

As a special case, a version number may contain the letters a or b in place of exactly one . separator; for example, 8.6a5, 8.6b7. These versions indicate *unstable* releases where functionality might change and which might not have undergone sufficient testing to be considered production ready. The a and b signify "alpha" and "beta" quality releases.

The leftmost number in a version identifier is the **major** version and the following number, if present, is the **minor** version. By convention, package releases are expected to follow certain norms with respect to changes in major and minor versions:

- Packages are expected to maintain **backward** compatibility within a major version. Thus the 1.2 release of a package is expected to maintain compatibility with applications that make use of versions 1.0 or 1.1 of the package. There is no expectation of **forward** compatibility. Applications that work with version 1.3 may not work with 1.2.
- Conversely, a change in major versions implies potentially incompatible changes in functionality. This is true in both the forward and backward directions. For example, applications that work with version 2.0 of a package will not necessarily work with either 1.0 or 3.0.

13.3.2.2. Comparing package versions: `package vcompare|vsatisfies`

Version numbers follow a sequence where higher version numbers indicate a later release of a package. When comparing version numbers, the leftmost version numbers have higher significance and any missing version fields are treated as 0. For example, 8.6.1 is a later version than 8.5.100 while 8.6 and 8.6.0 are equal.

If a version number includes a and b as separators, they are treated as an additional version component with values -2 and -1 respectively. For example, 8.6b22 is treated as version 8.6.-1.22 and therefore less than 8.6.0. Consequently, versions marked alpha or beta are naturally deemed earlier than stable versions having the same major and minor levels.

The package vcompare command can be used to compare two version numbers.

```
package vcompare VERA VERB
```

The command returns -1, 0 or 1 depending on whether *VERA* is less than, equal, or greater than *VERB*.

```
package vcompare 8.6 8.6b22 → 1
```

The `package vsatisfies` command offers a more flexible method to check if a package satisfies certain version requirements.

```
package vsatisfies VER REQ ?REQ ...?
```

The command returns 1 if *VER* meets at least one of the requirements stipulated by the *REQ* arguments. These arguments must be in one of the forms shown in Table 13.1.

Table 13.1. Package version requirements syntax

Requirement	Description

MIN-MAX	This requirement specifies a range within which the version must reside. If *MIN* and *MAX* are equal, the version must also be the same. Otherwise, the version must be at least *MIN* and strictly less than *MAX*.
	
	`package vsatisfies 8.6.6 8.5-8.7 → 1`
	`package vsatisfies 8.5 8.7      → 0`
	`package vsatisfies 8.7 8.7      → 1`

MIN-	The version must be at least *MIN*. There is no limit for the upper bound for the version.
	
	`package vsatisfies 8.6 8- → 1`
	`package vsatisfies 9 8-   → 1`

MIN	The version must be at least *MIN*. The upper bound of the permissible range is the next higher major version relative to *MIN*.
	
	`package vsatisfies 8.6 8 → 1`
	`package vsatisfies 9 8   → 0`

For example, Tcl version 8.6.1 had some bugs in its I/O implementation so to avoid this version while allowing any other Tcl with major version 8, you could write

```
if {![package vsatisfies [info patchlevel] 8-8.6.1 8.6.2]} {
    error "Unsupported version"
}
```

The above will error if the Tcl major version is not exactly 8 or if the version is 8.6.1 (remember the upper bound is not included in the permitted range).

13.3.3. Discovering packages

The `package names` command can be used to enumerate all the packages that are known (not necessarily loaded) to the Tcl interpreter. However, getting the complete list requires Tcl to have already searched all its library directories at least once. This can be done by attempting to load a non-existent package.

```
catch {package require nosuchpackage} → 1
```

This forces a search of all library directories. We can then enumerate the available packages.

```
% package names
→ rcs logger counter math::rationalfunctions fileutil::magic::mimetype Tcl00 Plotchart zi...
```

When a package has multiple versions installed, the package versions command will list the available versions for that package.

```
package versions http → 2.8.9 1.0
```

The command returns an empty list if no packages of that name are available.

13.3.4. Installing packages

Tcl does not have one single standardized method of installing packages.

If you are using a OS-provided distribution, other OS-supported Tcl packages can be installed in the same manner as Tcl, for example from within the Bash shell

```
sudo apt-get install PACKAGE
```

If you are using the ActiveState Tcl distribution, you can use the teacup program to both install and update packages from their remote distribution site. From within a Bash shell or Windows command prompt,

```
teacup install PACKAGE
```

The Windows installer based distributions do not have a remote update capability at the time of writing. However, they bundle many commonly used packages within the distributions. These can be individually installed or uninstalled through the standard Windows Control Panel Programs and Features dialog's Change menu option.

In cases where the distribution does not include the package or has a different version of the package than that desired, follow the package's installation instructions. In many cases, installation consists of simply extracting the contents of a compressed archive into a directory that is included in the package search path stored in the auto_path global variable. We describe the use of this variable in Section 13.3.5.

13.3.5. Searching for libraries

The auto_load and the package require commands search a list of directories, the *search path*, to locate tclIndex and pkgIndex.tcl files respectively. This directory list is given by the auto_path global variable.

When a Tcl interpreter is created, auto_path is initialized by concatenating the following in order:

- The value given by the TCLLIBPATH environment variable. This value is treated as the string representation of a Tcl list each element of which specifies a directory. Note that this implies that if the \ character is used as directory separator, it must be doubled as \\ to avoid Tcl interpreting as a backslash escape sequence.
- The directory given by the tcl_library global variable.
- The parent directory of the directory in tcl_library
- The directories listed in the tcl_pkgPath global variable if it exists.

Applications are free to add directories (or even remove them though this is generally not recommended) to the search path by modifying auto_path appropriately.

Tcl examines all directories listed in auto_path for pkgIndex.tcl files and evaluates them. These files register package names and versions into a package index database as described in Section 13.3.8. This package database is then checked at the time of loading as we describe next.

> This package search description does not apply to module-based packages. The procedure for those is described in Section 13.5.2.

13.3.6. Loading packages: `package require`

To use the commands implemented by a package, the package must first be loaded into the Tcl interpreter. This is accomplished by the `package require` command.

```
package require NAME ?REQ ...?_
package require -exact NAME VERSION
```

In the first form of the command, the optional *REQ* arguments indicate version requirements using the syntax described in Section 13.3.2.2. The second form requires the package to be the exact version specified and is equivalent to the first form written as

```
package require NAME VERSION-VERSION
```

The command then loads the package as follows:

- It checks if the package named *NAME* is already loaded into the interpreter and whether it meets the specified version requirements, if any. If so, it returns the actual version number of the loaded package. If the loaded package version does not meet version specifications, the command will raise an error as multiple versions of a package cannot be loaded into a single interpreter.

- If the package is not already loaded, the command checks the internal package index database. If a suitable version is found in there, it loads the package by evaluating the associated script.

- If not found in the package index, Tcl further searches for it as described in Section 13.3.5. If no matches are found, the command raises an error. If one or more matches are found, the command selects the latest version present (modulo the stable/unstable attribute described below), loads it into the interpreter and returns its version number.

Some examples:

```
package require http
package require http 2-
package require -exact 2.8
```

In a previous section, we illustrated use of the `package vsatisfies` command to check we are running Tcl with major version 8 except for 8.6.1. We could also do the check with the following command

```
package require Tcl 8-8.6.1 8.6.2 → 8.6.6
```

This is because the Tcl implementation itself presents a package interface as well.

> The `package require` command loads the package only into the interpreter invoking the command. This means you can actually load multiple versions of a package as long as they are loaded into different interpreters. For example, suppose the bulk of your application uses a newer version of the `http` package but one particular piece needs the functionality of V1 of the package for whatever reason. You can accomplish that as follows:
>
> ```
> package require http              → 2.8.9
> set ip [interp create]            → interp0
> $ip eval {package require http 1} → 1.0 ❶
> ```

```
interp alias {} geturlv1 $ip http::geturl → geturlv1
```

❶   Load V1 of the package

You can now call `geturlv1` to use the V1 version of the `http::geturl` command. Note this may not always be possible with packages that include binary shared library extensions.

Use of multiple interpreters is described in Chapter 20.

13.3.6.1. Choosing stable versus unstable packages

There is one other consideration when Tcl selects a package to load when multiple versions of the package are present. As discussed in Section 13.3.2.1, package versions are distinguished between stable and latest, possibly unstable, versions. The latter embed the a or b characters to mark them as alpha or beta versions.

When Tcl selects a package to load in response to a `package require` command, treatment of unstable packages is affected by the *selection mode*. This mode may have the values `stable` and `latest`. In `latest` mode, the highest version available of the package is chosen irrespective of whether it is stable or not. In `stable` mode, the highest *stable* version of the package is chosen unless no stable versions are available in which case it falls back to the highest unstable version.

The `package prefer` command allows setting and retrieval of this package selection mode.

```
package prefer ?stable|latest?
```

If no arguments are specified, the command returns the current mode.

```
package prefer → stable
```

If `latest` is specified as an argument, the mode is set accordingly. Passing `stable` as an argument is a no-op — the current mode is not changed irrespective of its value. In both cases, the command returns the mode.

```
package prefer stable → stable
package prefer latest → latest
package prefer stable → latest ❶
```

❶   Note that mode is **not** changed

At startup, the mode is set to `stable` unless the TCL_PKG_PREFER_LATEST environment variable is set in which case the mode is initialized to `latest`. The value of of TCL_PKG_PREFER_LATEST is immaterial.

13.3.7. Checking if a package is loaded: package present

We saw earlier that we can list all packages available to a Tcl interpreter using `package names`. This does not tell us if a package is already loaded. To do that we need to use the `package present` command.

```
package present ?-exact? NAME REQ ?REQ...?
```

The command works exactly like `package require` except that it will not load the package if it was not already present. If the package *NAME* is loaded, the command returns its version. Otherwise, an error is raised.

```
package present math Ø package math is not present
package require math → 1.2.5
package present math → 1.2.5
```

13.3.8. Creating packages

Up to this point, we have discussed what packages are and how they are used. We will now go into how to construct them.

A package consists of

- zero or more Tcl script files
- zero or more binary executables in the form of shared libraries
- zero or more data files such as images
- a pkgIndex.tcl file contains, amongst other things, commands that tell Tcl the package name and version and the script to be evaluate to load the package into the interpreter.

Commonly, the package contains a "main" script which is sourced into the interpreter and in turn reads any additional script files and loads the shared libraries if present.

We will illustrate the process of package creation through an example. Our package, named sequences, will provide procedures, arith_term and geom_term, for calculating the nth term of arithmetic and geometric sequences respectively. We will modularize this large package by breaking up the implementation into two files seq_geom.tcl

```
# seq_geom.tcl
namespace eval seq {
    proc geom_term {a r n} {
        return [expr {$a * $r**($n-1)}]
    }
}
```

and seq_arith.tcl respectively.

```
# seq_arith.tcl
namespace eval seq {
    proc arith_term {a i n} {
        return [expr {$a + ($n-1)*$i}]
    }
}

source [file join [file dirname [info script]] seq_geom.tcl]

package provide sequences 1.0
```

We will create these files in their own directory, also called sequences (though the directory name need not match the package name).

We also treat seq_arith.tcl as the "main" script for the package so we have added the package provide command at the end which informs Tcl when the file is sourced that the sequences package version 1.0 is now loaded into the interpreter.

The package provide command takes the form

```
package provide NAME ?VERSION?
```

where NAME is the name of the package. When used to define or create a package, the VERSION argument must be supplied and is taken as the version of the package being provided. If VERSION is not specified, the command returns the version number of the package if it has already been previously provided, and an empty string otherwise.

We saw earlier the use of the package present command to check if a package has already been loaded into the interpreter. The package provide command without the *VERSION* argument is an alternate means of doing this.

```
if {[package provide Tk] eq ""} {
    puts "Package Tk is not loaded."
}
```

What remains to be done is to create the pkgIndex.tcl file for the package. For our sample package, the contents of this file are shown below

```
package ifneeded sequences 1.0 [list source [file join $dir seq_arith.tcl]]
```

When Tcl searches for a package, it (among other actions) evaluates all pkgIndex.tcl files found in the package search path as described in Section 13.3.5. A pkgIndex.tcl file is just a normal Tcl script and may contain any Tcl commands but it's primary purpose is to inform Tcl about the packages it makes available and how they are to be loaded. It does this through the package ifneeded command.

```
package ifneeded NAME VER ?SCRIPT?
```

When this command is evaluated, Tcl makes a note that version *VER* of package *NAME* may be loaded by evaluating *SCRIPT*. This information is used when an application requests a package to be loaded as we described in Section 13.3.6. If the *SCRIPT* argument is not provided, the command returns the script that was previously registered for loading the package.

To find the location a package was loaded from, you can often use the package ifneeded command without the *SCRIPT* argument. For example,

```
% package ifneeded fileutil [package require fileutil]
→ source c:/tcl/lib/tcllib1.18/fileutil/fileutil.tcl
```

Before evaluating a pkgIndex.tcl file, Tcl sets the global variable dir to the path of the directory containing the pkgIndex.tcl file being evaluated. In our example we use this to load the main script for our package.

A slightly more capable example demonstrates the use of packages in conjunction with auto loading. Our pkgIndex.tcl file for our sequences package could have been written as follows:

```
namespace eval seq {
    proc setup_autoload dir {
        global auto_index
        foreach cmd {arith_term geom_term_geom} {
            set auto_index([namespace current]::$cmd) [list source [file join $dir \
                seq_arith.tcl]]
        }
    }
}
package ifneeded sequences 1.0 [list seq::setup_autoload $dir]
```

Now when an application does a package require sequences, the package scripts are not immediately read in. Instead they will be evaluated the first time either seq_arith or seq_geom is called in a similar manner as for the tclIndex based auto loading facility described in Section 13.2.

In a nutshell, the pkgIndex.tcl can be as sophisticated as you need it to be. It is advisable to keep it relatively short however, as it is read and evaluated during Tcl's package search even when it is not the package being requested.

Tcl includes a command, pkg_mkIndex, that creates pkgIndex.tcl files and a related command, pkg::create, that you can use instead of manually creating the file. However, manual creation is simple (as we saw above) for simple packages and for more complex cases these commands are sufficiently lacking that their use is discouraged. We therefore do not describe them further.

One final note about pkgIndex.tcl files. A single pkgIndex.tcl file may contain multiple package ifneeded commands each registering a different package or even a different version of each package. You will find this or similar methods used in "package bundles" like Tcllib[1] which are composed of multiple packages.

13.4. Shared library extensions: load

Tcl commands can be implemented natively in shared libraries referred to as Tcl extensions. These commands are made available in a Tcl interpreter by loading the extension. Usually the package author will load the extension by providing a suitable pkgIndex.tcl file so that the application loads it with the package require command.

When no pkgIndex.tcl file is provided or when you are authoring the package and have to create the pkgIndex.tcl file, the load command can be used to load the extension.

```
load ?-global? ?-lazy? ?--? PATH ?INITNAME? ?INTERP?
```

Here PATH specifies the file path of the shared library. The optional INITNAME and is used to construct the name of the function in the extension that is to be called to initialize it. The optional INTERP specifies the name of the interpreter into which the extension is to be loaded. By default, the extension is loaded in the interpreter that executes the load command. This is only relevant in applications using multiple Tcl interpreters. Multiple interpreters are discussed in Chapter 20.

After loading it, Tcl calls a specific function in the shared library to initialize it. In a normal interpreter, the name of this function is constructed by changing the first letter of INITNAME to upper case and the remaining to lower case, and appending _Init to the result. For example, if INITNAME was myext, the name of the initialization function would be Myext_Init.

In a safe interpreter, discussed in Section 20.6, the name of the initialization function is similar except that _SafeInit is appended instead of _Init as above.

If the shared library does not export a function of the constructed name, the load command will fail with an error.

If most cases, the initialization name need not be specified as it is by convention the same as the shared library extension base name. Thus an extension myext.so can be loaded simply as

```
load /path/to/myext.so
```

The -global and -lazy options are very rarely used and not discussed here. Refer to the Tcl documentation of load for details.

Another command useful in conjunction with load is info sharedlibextension which returns the file extension used for shared libraries on that platform.

```
info sharedlibextension → .dll
```

That allows us to write our above example as

```
load myext[info sharedlibextension]
```

---
[1] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

Note however that in most cases the file extension need not be specified as it will be automatically added. Moreover, the full path need not be specified if the shared library is present on the search path for shared libraries for that system. Nevertheless, it is good practice to specify the full path so as to prevent errors from multiple files of that name present on the search path.

13.4.1. Including shared libraries in packages

A shared library may be wrapped as a package either for the user's convenience or because it is only part of a package that includes other files, scripts or even other shared libaries.

In the simplest case, where the shared library is self-contained, the `pkgIndex.tcl` file could look as follows:

```
package ifneeded binpkg 1.0 [list load [file join $dir binpkg[info sharedlibextension]]]
```

In our example, the name of the package and the shared library are both `binext`. We make some allowance for platform differences in the file name extension used for shared libraries by calling `info sharedlibextension`.

13.5. Modules

Although the Tcl package mechanism is flexible, that flexibility has a cost associated with it. When locating packages, Tcl has to go through the package directory search path looking for `pkgIndex.tcl` files. These have to then be read and evaluated as Tcl scripts. In a large Tcl application that loads many packages, this can result in a noticeable delay at startup time, particularly when the application resides on a remote network location. Tcl modules provide an alternate scheme for libraries that mitigates these startup costs at the price of some flexibility.

Tcl modules incorporate two changes that reduces the time required for locating them:

- The module name and version are encoded into the file name itself and Tcl does not need to evaluate a file to retrieve this information as it does with `pkgIndex.tcl` files in the package case.
- The directory search path for modules is more limited.

Although implemented differently than the package form we discussed earlier, Tcl modules are used with many of the same commands. In particular,

- The `package names` command used for discovering available packages includes modules as well.
- Modules are loaded with `package require`. When searching for a module, Tcl will give preference for a module based implementation of a package before a traditional one (assuming both have the same version).
- The versioning syntax for modules is the same as discussed in Section 13.3.2.

For this reason, when we will any reference to "packages" henceforth will include what we will term as "traditional" packages as well as modules. The rest of this section only describes the areas where modules differ from traditional packages.

13.5.1. Module file names

A Tcl module is stored as a **single** file containing a Tcl script. The name of the file must be the package name, followed by a - character, followed by the package version and an extension of `.tm`. Specifically, it must match the regular expression

```
([_[:alpha:]][:_[:alnum:]]*)-([[:digit:]].*)\.tm
```

For example, the `http` package that ships with Tcl is implemented as a module in the file `http-2.8.9.tm`.

It is strongly suggested that module based packages not use upper case characters in the package name. Since the package names are mapped to file names, there is potential for confusion between file systems that distinguish character case and those that do not.

If the package name contains any : : character sequences, they are treated specially during the module search process. This is explained in the next section.

13.5.2. Searching for modules

One important respect in which module based packages differ from the traditional packages is in how they are located. Module searches do not follow the process described for auto-loading and traditional packages described in Section 13.3.5.

When searching for a module based package, Tcl first constructs a partial file path for the module. This is the same as the package name with one change — any : : character sequences in the package name are replaced by the directory separator character. So for example

```
package require math::calculus
```

would translate to a file base name of math/calculus-*. The * allows for different versions of the package. This partial path is appended to each directory present in the module search path (described below). If one or more files matching the constructed path exists, the equivalent of the following command is executed for each such file:

```
package ifneeded PACKAGENAME VERSION [list source MODULEPATH]
```

Here VERSION is extracted from the module file name and MODULEPATH is the path to the matching file.

As described in Section 13.3.8 for traditional packages, this effectively registers the module in the package database. Requiring the package will result in evaluation of the corresponding source command.

The module search path

The module search path is completely independent of the auto_path global variable used for traditional packages. It is given by evaluating the command tcl::tm::path list.

```
% tcl::tm::path list
→ C:/Tcl/866/x64/lib/tcl8/site-tcl C:/Tcl/866/x64/lib/tcl8/8.0 C:/Tcl/866/x64/lib/tcl8/8.1
    ↳ C:/Tcl/866/x64/lib/tcl8/8.2 C:/Tcl/866/x64/lib/tcl8/8.3 C:/Tcl/866/x64/lib/tcl8/8.4
    ↳ C:/Tcl/866/x64/lib/tcl8/8.5 C:/Tcl/866/x64/lib/tcl8/8.6
```

This is the list of directories that Tcl will examine when looking for a module.

Adding directories to the module search path

You can add individual directories to this search path with the tcl::tm::path add command.

```
tcl::tm::path add ?DIR ...?
```

Each argument to the command is added in turn to the front of the search path in order. If an argument already exists in the search path, it is ignored.

There is an important restriction enforced in the directories included in the search path. No directory in the search path may be an ancestor of another. For example, the following will raise an error:

```
% tcl::tm::path add /temp/foo
% tcl::tm::path list
→ /temp/foo C:/Tcl/866/x64/lib/tcl8/site-tcl C:/Tcl/866/x64/lib/tcl8/8.0
    ↳ C:/Tcl/866/x64/lib/tcl8/8.1 C:/Tcl/866/x64/lib/tcl8/8.2 C:/Tcl/866/x64/lib/tcl8/8.3
    ↳ C:/Tcl/866/x64/lib/tcl8/8.4 C:/Tcl/866/x64/lib/tcl8/8.5 C:/Tcl/866/x64/lib/tcl8/8.6
% tcl::tm::path add /temp
∅ /temp is ancestor of existing module path /temp/foo.
% tcl::tm::path add /temp/foo/bar
∅ /temp/foo/bar is subdirectory of existing module path /temp/foo.
```

An alternative means of adding directories to the module search path is the `tcl::tm::roots` command which adds zero or more "roots" to the module search path. This differs from `tcl::tm::add` in that the passed arguments are not directly added. Rather, for each argument *ROOT* passed to it, the command adds directories of the form:

- *ROOT*/`tcl`*MAJOR*/`site-tcl` where *MAJOR* is the major version of this Tcl interpreter
- One or more directories *ROOT*/`tcl`*MAJOR*/*MAJOR*.*MINOR* for every value of *MINOR* that is less than or equal to the minor version of this Tcl interpreter.

For example, after evaluating the command

```
% tcl::tm::roots ~/lib
```

in Tcl 8.6 interpreter, the module search path will look as follows:

```
% print_list [tcl::tm::path list]
→ C:/Users/ashok/Documents/lib/tcl8/site-tcl
  C:/Users/ashok/Documents/lib/tcl8/8.0
...Additional lines omitted...
```

Removing directories from the module search path

You can remove directories from the module search path with the `tcl::tm::path remove` command.

```
tcl::tm::path remove ?DIR …?
```

Arguments that do not exist in the search path are ignored.

Module search path initialization

The module search path is initialized adding directories in the following order:

- Subdirectories of the form `tcl`*MAJOR*/*MAJOR*.*MINOR* under the parent directory of the path returned by the `info library` command.
- Subdirectories of the form `tcl`*MAJOR*/*MAJOR*.*MINOR* under the parent directory of the current process executable path as returned by the `info nameofexecutable` command.
- The contents of the `TCL`*MAJOR*_*MINOR*_`TM_PATH` environment variables. These values are interpreted as directories separated by the `;` character on Windows and `:` on other platforms.
- The contents of the `TCL`*MAJOR*.*MINOR*_`TM_PATH` environment variables. These are interpreted as above but their use is discouraged as use of the `.` character in environment variables is not portable.

In all the above paths, *MAJOR* is the major Tcl version of the current interpreter and *MINOR* takes on the values less than equal to the minor version of the current interpreter. So for example, for Tcl 8.6, *MAJOR* would be 8 and *MINOR* would take on all values between 0 and 6.

13.5.3. Installing modules

Installation of packages implemented as modules is done in the same distribution-specific manner described for traditional packages in Section 13.3.4.

If the module is not included with the specific Tcl distribution, installation by hand is very simple as modules must by definition be implemented as a single file. This file simply needs to be copied to an appropriate directory on the module search path. Place the module in the site-specific directory or directory named after the lowest supported Tcl minor version.

For example, if the module requires a minimal Tcl version of 8.5, place it in the directory given by one of the following locations:

```
% file join [info library] .. tcl8 site-tcl
→ c:/tcl/866/x64/lib/tcl8.6/../tcl8/site-tcl
% file join [info library] .. tcl8 8.5
→ c:/tcl/866/x64/lib/tcl8.6/../tcl8/8.5
```

The latter will make it available to all versions of Tcl 8 later than 8.5 but not to earlier versions.

13.5.4. Creating modules

In the simplest case, where the package is implemented as a single Tcl script, creating a module just involves naming the file appropriately. When multiple scripts are involved, the files can be simply concatenated together. For example, to create a module for our example package we can execute the following at the Unix shell prompt:

```
cat seq_geom.tcl seq_arith.tcl > sequences-1.0.tm
```

The equivalent on Windows would be

```
copy seq_geom.tcl+seq_arith.tcl > sequences-1.0.tm
```

Note the module file name reflects the package name and version and has the .tm extension. Also note that the script must have an `package provide` command, as for traditional packages, that specifies the package name and version. This is contained in the `seq_arith.tcl` file in our example.

When multiple files are coaelsced into a module file, make sure to concatenate them in the correct order in cases where the scripts evaluate code at run time. In such cases, files defining the procedures or data must appear before scripts that invoke them during the loading itself.

13.5.5. Including binaries in modules

When a module-based package is requested by an application, Tcl loads the file implementing the module with the standard `source` command. This means the same technique described in Section 10.2 for including binary data at the end of Tcl script files can be used with modules. In particular, a shared library Tcl extension can be distributed as a Tcl module by appending it to the module script separated by a Ctrl-Z character as described there. This can then be copied to the file system and loaded from there as a shared library. The Tcler's Wiki [2] has several examples of this technique, one of which is http://wiki.tcl.tk/19801.

For a discussion of various means of including binary data in modules see TIP #190: Implementation Choices for Tcl Modules.

13.6. Packages versus modules

How does one make a choice between distributing script libraries as a traditional package versus a module? There are several considerations:

- Modules are easier to distribute as they are a single file. Packages need to be archived into zip, tar.gz or similar format and unarchived on the target.
- Modules are faster to load unless they embed shared libraries.
- Library scripts that have significant platform or version-specific components are easier shipped as packages as the `pkgIndex.tcl` file can load the appropriate pieces based on runtime information.
- Packages that include shared libraries are better implemented as packages. Although modules can support shared libraries as described in the previous section, there are several drawbacks to this. Copying the shared

---

[2] http://wiki.tcl.tk

library out to disk and reading it back incurs a load time performance hit. Some heuristic based virus scanners also flag this behaviour of writing code to disk and executing it as reflective of malware.

13.7. Multiplatform packaging: platform package

It is useful and convenient for the end user if the package can be installed in single directory, perhaps on the network, from where it can be loaded into Tcl interpreters running on differing architectures. For packages that are purely script-based, this is obviously not an issue. However, if your package includes shared libraries, support for multiple platforms from a single installation directory is a little trickier because the pkgIndex.tcl file for the package must load the appropriate shared library from the installation directory. The platform package addresses this requirement of a well-defined means of identifying the operating system and architecture of the host system. The package is part of the Tcl core distribution but must be explicitly loaded before its commands can be invoked.

```
package require platform → 1.0.14
```

The package implements three commands, identify, generic and patterns, all of which are placed under the platform namespace.

The platform::identify command returns an identifier for the platform that encodes the operating system, C runtime version and CPU architecture. For example, on an Linux Ubuntu system,

```
% platform::identify
→ linux-glibc2.19-x86_64
```

while on a Windows 32-bit system,

```
% platform::identify
→ win32-ix86
```

The platform::generic command is similar except it returns a less exact identifier that encodes the "family" of platforms. For example, on the same Ubuntu system,

```
% platform::generic
→ linux-x86_64
```

The third command in the package is platform::patterns. This command takes an argument that is a platform identifier as returned by platform::identify. It then returns a list of all platform identifiers that are compatible with the one passed.

Again, on our Ubuntu system,

```
% platform::patterns [platform::identify]
→ linux-glibc2.19-x86_64 linux-glibc2.18-x86_64 linux-glibc2.17-x86_64
    ↳ linux-glibc2.16-x86_64 linux-glibc2.15-x86_64 linux-glibc2.14-x86_64
    ↳ linux-glibc2.13-x86_64 linux-glibc2.12-x86_64 linux-glibc2.11-x86_64
    ↳ linux-glibc2.10-x86_64 linux-glibc2.9-x86_64 linux-glibc2.8-x86_64 linux-glibc2.7-x86_64
    ↳ linux-glibc2.6-x86_64 linux-glibc2.5-x86_64 linux-glibc2.4-x86_64 linux-glibc2.3-x86_64
    ↳ linux-glibc2.2-x86_64 linux-glibc2.1-x86_64 linux-glibc2.0-x86_64 tcl
```

Let us see an example of how these commands might be used to load the correct shared library image from a package that is installed for multiple architectures. Here is the pkgIndex.tcl file for our imaginary binpkg package that supports multiple platforms within a single installation.

```
apply {
    {package_name dir} {
        set filename $package_name[info sharedlibextension]
        set ident [platform::identity]
        set subdirs [list $ident \
                          [platform::generic] \
                          {*}[platform::patterns $ident]]
        foreach subdir $subdirs {
            set path [file join $dir $subdir $filename]
            if {[file exists $path]} {
                package ifneeded binpkg 1.0 [list load $path]
                return
            }
        }
    }
} binpkg $dir
```

We have placed our code inside an anonymous procedure so as to not clutter the global namespace with our temporary variables. We pass the name of the package, binpkg, and the directory path where the package is installed as arguments to this function. (As discussed earlier, this last is set by Tcl in the dir variable before the pkgIndex.tcl is sourced.)

The code checks for the existence of a shared library in the following subdirectories in order:

- The subdirectory named after the platform identifier as returned by platform::identify
- The subdirectory corresponding to the generic platform identifier for the current platform
- Finally, the list of subdirectories corresponding to platforms that are **compatible** to the current platform. This list is returned by the platform::patterns command

The first suitable shared library found is used. Note that if the search fails, no package ifneeded is invoked. Consequently, on such platforms even though the pkgIndex.tcl file may be evaluated, it will not register any packages with the interpreter and any attempt to load the package will fail.

13.7.1. The platform::shell package

The platform::shell package is similar to the platform package except that while the latter returns platform information for the currently executing Tcl shell, the former returns information about a different Tcl shell residing on the same machine. The shell of interest is identified by its path and must be executable on the same machine as the current shell.

The package provides platform::shell::identify and platform::shell::generic commands that are functionally similar to the platform::identify and platform::generic commands except that they return the corresponding information for the targeted shell.

As an example, both 32-bit and 64-bit shells may be installed on the same 64-bit Windows machine. Assuming we are currently running the 64-bit shell, we can retrieve

```
% package require platform
→ 1.0.14
% package require platform::shell
→ 1.1.4
% platform::identify
→ win32-x86_64
% platform::shell::identify c:/tcl/866/x86/bin/tclsh.exe
→ win32-ix86
```

The package also has an additional command platform::shell::platform which returns the contents of platform element of the target shell's tcl_platform array we described in Chapter 2.

```
% platform::shell::platform c:/tcl/866/x86/bin/tclsh.exe
→ windows
```

Where might one make use of the package::shell commands? The primary reason for the existence of these commands is for code repositories and installers where the Tcl shell running the installer script is not necessarily the same as the target shell for which a package is being installed. They allow the installer to detect the architecture of the target shell and copy the appropriate files there.

13.8. Introspecting package configuration

Packages (using the term generically to include modules and shared libraries) may wish to expose certain configuration information to applications such as implemented features, build information etc. They should do so by implementing a pkgconfig command within the package's namespace. This command should support at least the subcommands list and get. The pkgconfig list command should take no arguments and return a list of configuration keys. The pkgconfig get command should take a single argument which is a configuration key and return the corresponding value. Applications can then invoke this command to retrieve information about the package. Note that the key names and contents are entirely up to the package.

As an example, Tcl itself exposes its configuration through the pkgconfig command in tcl namespace.

```
% tcl::pkgconfig list ❶
→ debug threaded profiled 64bit optimized mem_debug compile_debug compile_stats
    ↳ libdir,runtime bindir,runtime scriptdir,runtime includedir,runtime docdir,runtime
    ↳ libdir,install bindir,install scriptdir,install includedir,install docdir,install
% tcl::pkgconfig get optimized ❷
→ 1
```

❶ Lists all available keys provided by the Tcl package
❷ Tells us whether compiled with optimization enabled

13.9. Chapter summary

A means for packaging and distribution of libraries is a requirement of any programming language environment. In this chapter we examined the core facilities Tcl provides for this purpose. Later in Chapter 19, we will see another means of packaging libraries and entire applications for distribution based on Tcl's virtual file system technology.

13.10. References

TIP189
> *TIP #189: Tcl Modules*, Kupries, Porter et al, Tcl Improvement Proposal #189, http://www.tcl.tk/cgi-bin/tct/tip/189

TIP190
> *TIP #190: Implementation Choices for Tcl Modules*, Kupries, Wippler, Hobbs, Tcl Improvement Proposal #190, http://www.tcl.tk/cgi-bin/tct/tip/190

14

Object-Oriented Programming

This chapter describes Tcl features that support object oriented programming. It does *not* go into detail about what constitutes object oriented programming, what its benefits are, or how your classes should be designed. The answers generally depend on who you ask and there have been enough words written on the topic.

Nevertheless, as we go along we will briefly describe some basic concepts for the benefit of the reader who really is completely unexposed to OO programming.

A bit of history

One of the knocks against Tcl in its early days was that it did not support object oriented programming. This criticism was both incorrect and unfair because Tcl did in fact support not one, but several, OO implementations. This misconception was at least partly due to the fact that these OO systems did not come as part of the core language, but rather were implemented as extensions or packages. In fact, writing an OO system in Tcl became a rite of passage for many Tcl programmers.

Some of these systems became fairly widely used and remain so today:

- *IncrTcl*'s name was a take-off on C++ and so is its design. It was intended to make programmers used to that language feel at home. It was one of the earliest Tcl-based OO extensions to be widely used.

- *Snit* (Snit's Not Incr Tcl) is a popular OO implementation which is particularly useful towards building Tk widgets.

- *XoTcl* and its successor *nx* are OO implementations designed for research into dynamic OO programming.

The experience gained from these system led to the implementation of a OO system in the Tcl core — TclOO. This became part of the Tcl 8.6 release and is also available as an extension for Tcl 8.5. TclOO can be used as a standalone OO system by itself. However, one of its goals was also to provide the base facilities required for layering other OO systems on top.

The Tcl based OO programming described in this book is based on TclOO.

Most of the example code in this chapter is based on a framework for modeling banks. Our bank has accounts of different types, such as savings and checking, which allow operations like deposits and withdrawals. Some of these are common to all account types while others are unique. We have certain privileged customers who get special treatment and we have to also follow certain directives from Big Brother.

No, Citibank cannot run its operations based on our framework but it suffices for our illustrative purposes.

14.1. Objects and classes

The core of OO programming involves, no surprise, *objects*. An object, often a representation of some real world entity, captures state (data) and behaviour which is the object's response to *messages* sent to it. In most languages, implementation of these messages involves calling *methods* which are just function calls with a context associated with the object. For example, the state contained in an object representing a bank account would include items

such as the current balance and account number. The object would respond to messages to deposit or withdraw funds from the account.

A *class* is (loosely speaking) a template that defines the data items and methods (collectively called members) encapsulated by objects of a specific type. More often than not, creating a object of the class, often known as instantiating an object, is one of the duties of the class.

Not every OO system has, or needs, the notion of a class. *Prototype* based systems instead create objects by "cloning" an existing object — the prototype — and defining or modifying members.

TclOO provides facilities to support both the classy and the classless [1] models.

> TclOO actually exposes sufficient internal functionality to allow you to develop your own vision of what object-oriented programming models should look like. The ITcl 4.0 implementation for instance is layered on the services provided by the TclOO. For most folks though who are not into experimentation with OO as a way of life, the base functionality provided by TclOO suffices for most purposes.

14.2. Classes

We will start off with first describing classes as most OO programming code you will encounter in Tcl is based on the use of classes.

14.2.1. Creating a class

Classes are created in TclOO using the oo::class create command. Let us create a class, Account, that models a banking account.

```
% oo::class create Account
→ ::Account
```

This creates a new class Account that can be used to create objects representing bank accounts.

> The class Account is actually just another Tcl command and could have been created in any namespace we choose, not necessarily the global one. For example, either
>
> ```
> oo::class create bank::Account
> namespace eval bank {oo::class create Account}
> ```
>
> would create a new class Account in the bank namespace, entirely unrelated to our Account class in the global namespace.

There is however no class definition associated with our Account class and therefore there is as yet no state or behaviour defined for objects of the class. That is done through one or more *class definition scripts*. We will look at the contents of these definition scripts throughout this chapter, but for now, simply note that class definitions can be built up in incremental fashion. A definition script can be passed as an additional argument to the oo::class create command, in the form

```
oo::class create CLASSNAME DEFINITIONSCRIPT
```

and also through oo::define commands which take the form

```
oo::define CLASSNAME DEFINITIONSCRIPT
```

---

[1] No value judgement intended.

Thus the statements

```
oo::class create CLASSNAME DEFINITIONSCRIPT
```

and

```
oo::class create CLASSNAME
oo::define CLASSNAME DEFINITIONSCRIPT
```

are equivalent. As is generally the case, Tcl has the flexibility to fit your programming style.

One advantage of the oo::define command is that it may be used multiple times for the same class to define it in incremental fashion. We will see this we work through this chapter. Moreover, the command also has a second syntactic form:

```
oo::define CLASSNAME SUBCOMMAND ARG ?ARG ...?
```

Here SUBCOMMAND is one of the commands that may be used in a class definition script. The following listing shows the two equivalent forms of oo::define.

```
oo::define Account {
    method foo {} {}
    method bar {} {}
}

oo::define Account method foo {} {}
oo::define Account method bar {} {}
```

This second form of oo::define can be useful when one of the arguments to the subcommand is to be taken from a variable, a situation that often arises in metaprogramming with objects.

14.2.2. Destroying classes

A class, as we shall see later, is also an object and like all objects can be destroyed by invoking its destroy method.

```
% Account destroy
```

Classes can also be destroyed by renaming the corresponding command to the empty string:

```
rename Account ""
```

The above will erase

- the definition of the Account class,
- any classes that inherit from (see Section 14.4), or mix-in (see Section 14.6), the Account class
- all objects belonging to all destroyed classes.

Not commonly used in operational code, this ability to destroy classes is sometimes useful during interactive development and debugging to reset to a known clean state.

We will be using the Account class throughout the chapter so let us recreate it before we move on.

```
% oo::class create Account
→ ::Account
```

14.2.3. Defining data members

In our simple example, the state for an account object includes an account number that uniquely identifies it and the current balance in the account.

We will need data members to hold this information and we define them through the variable command within a class definition script.

```
% oo::define Account {
    variable AccountNumber Balance ❶
}
```

❶ The author uses mixed case for data members to avoid conflicts with names of arguments and local variables.

This defines the data members for the class as per-object variables. AccountNumber and Balance are then visible within all methods of the class and can be referenced there without any qualifiers or declarations.

There can be multiple variable statements, each defining one or more data members. These append new variable definitions to those existing. However, if the -set option is specified for the command, the current set of variable definitions is **replaced** by the ones defined in the current command. You can also remove all existing variable definitions without creating new ones by specifying the -clear option.

Data members do not have to be declared using variable in a class definition script. They can also be declared within a method using the my variable command which we show later.

> Note the difference between the variable statement in the context of a class definition and the variable command used to define namespace variables. They both have very similar function but the former only defines data member names, not their values whereas the latter defines names of variables within a namespace as well as their initial values.

14.2.4. Defining methods

Having defined the data members, let us move on to defining the methods that comprise the behaviour of an Account object. An Account object responds to requests to get the current balance and to requests for depositing and withdrawing funds.

Methods are defined through the method command which, like variable, is executed as part of a class definition script.

```
oo::define Account {
    method UpdateBalance {change} {
        set Balance [+ $Balance $change]
        return $Balance
    }
    method balance {} { return $Balance }
    method withdraw {amount} {
        return [my UpdateBalance -$amount]
    }
    method deposit {amount} {
        return [my UpdateBalance $amount]
    }
}
```

As you see, a method is defined in exactly the same manner as proc defines a Tcl procedure. Just like in that case a method takes an arbitrary number of arguments including a variable number of trailing arguments collected as the args variable. The difference from a procedure lies in how it is invoked and the context in which the method executes.

14.2.4.1. Method visibility

Another point about method definitions concerns method visibility. An *exported* method is a method that can be invoked from outside the object's context. A *private* method on the other hand, can only be invoked from within another method in the object context. Methods that begin with a lower case letter are exported by default. Thus in our example, deposit and withdraw are exported methods while UpdateBalance is not. Method visibility can be changed by using the export and unexport commands inside a oo::define class definition script. Thus

```
oo::define Account export UpdateBalance
```

would result in the private UpdateBalance method being exported. Conversely,

```
oo::define Account unexport UpdateBalance
```

will remove it from the exported list.

> The export and unexport commands are not limited to the methods defined in that particular class. They can also be applied to methods inherited from superclasses or mix-ins. This is useful, for example, when a derived class wants to limit functionality supported by a base class.

14.2.4.2. Deleting methods

Method definitions can be deleted at any time with the deletemethod command inside a class definition script.

> The following code snippet will crash Tcl versions prior to 8.6.2 due to a bug in the Tcl implementation.

```
% oo::class create C {method print args {puts $args}}
→ ::C
% C create c
→ ::c
% c print some nonsense
→ some nonsense
% oo::define C {deletemethod print}
% c print more of the same
Ø unknown method "print": must be destroy
```

Deletion of methods from classes is rarely used. However, deletion of methods from objects is sometimes useful in object specialization (see Section 14.5).

14.2.4.3. Renaming methods

The renamemethod command is used within a class definition script to rename an existing method.

```
oo::class create C {method print args {puts $args}}
C create c
oo::define C renamemethod print output
```

Once renamed, the method must be invoked by its new name, even for existing objects.

```
% c print foo
Ø unknown method "print": must be destroy or output
% c output foo
→ foo
```

We have not discussed class inheritance as yet, but we will just note here that renaming a method in a class will not rename methods of the same name in any ancestors or descendents for that class.

14.2.5. Constructors and destructors

There is one final thing we need to do before we can start banking operations and that is to provide some means to initialize an Account object when it is created and perform any required clean up when it is destroyed.

These tasks are performed through the special methods named constructor and destructor. These differ from normal methods in only two respects:

- They are not explicitly invoked by name. Rather, the constructor method is automatically run when an object is created. Conversely, the destructor method is run when the object is destroyed.
- The destructor method definition differs from other methods in that it only has a single parameter — the script to be run. It does not have a parameter corresponding to arguments.

For our simple example, these methods are straightforward.

```
oo::define Account {
    constructor {account_no} {
        puts "Reading account data for $account_no from database"
        set AccountNumber $account_no
        set Balance 1000000
    }
    destructor {  ❶
        puts "[self] saving account data to database"
    }
}
```

❶   Note the syntax of the destructor definition

Both constructors and destructors are optional. They do not have to be defined in which case TclOO will simply generate empty methods for them.

14.2.6. The unknown method

Every object has a method named unknown which is run when no method of that name is found in the method chain (see Section 14.9) for that object.

The definition of the unknown method takes the form

```
oo::define CLASSNAME {
    method unknown {target_method args} {.. implementation ..}
}
```

The unknown method is passed the name of the invoked method as its first argument followed by the arguments from the invocation call.

The default implementation of this method, which is inherited by all objects from the root oo::object object, raises an error. Classes and objects can override the default implementation method to take some other action instead.

An example of its use is seen in the COM client implementation in TWAPI[2]. The properties and methods exported from a COM component are not always known beforehand and in fact can be dynamically modified. The TclOO based wrapper for COM objects defines an unknown method that looks up method names supported by a COM component the first time a method is invoked. If found, the lookup returns an index into a function table that can then be invoked through the ComCall method. The implementation of unknown looks like

---

[2] http://twapi.sf.net

```
oo::define COMWrapper {
    method unknown {method_name args} {
        set method_index [COMlookup $method_name]
        if {$method_index < 0} {
            error "Method $method_name not found."
        }
        return [my ComCall $method_index {*}$args]
    }
}
```

(This is a greatly simplified, not entirely accurate or correct, description for illustrative purposes.)

14.2.7. Modifying an existing class

As we have seen in previous sections, you can incrementally modify a class using oo::define. Practically nothing about a class is sacred — you can add or delete methods, data members, change superclasses or mix-ins, and so on.

The question then arises as to what happens to objects that have already been created if a class is modified. The answer is that existing objects automatically "see" the modified class definition so for example any new methods can be invoked on them. Or if you add a mix-in or a superclass, the method lookup sequence for the object will be appropriately modified.

However, some care should be taken when modifying a class since existing objects may not hold all state expected by the new class. For example, the new constructors are (obviously) not run for the existing objects and thus some data members may be uninitialized. The modified class code has to account for such cases.

14.3. Working with objects

Having defined our model, we can now begin operation of our bank to illustrate how objects are used.

14.3.1. Creating an object

An object of a class is created by invoking one of two built-in methods on the class itself. The create method creates an object with a specific name. The new method generates a name for the created object.

```
% set acct [Account new 3-14159265]
→ Reading account data for 3-14159265 from database
  ::oo::Obj217
% Account create smith_account 2-71828182
→ Reading account data for 2-71828182 from database
  ::smith_account
```

Creating an object also initializes the object by invoking its constructor.

The created objects are Tcl commands and as such can be created in any namespace.

```
% namespace eval my_ns {Account create my_account 1-11111111}
→ Reading account data for 1-11111111 from database
  ::my_ns::my_account
% Account create my_ns::another_account 2-22222222
→ Reading account data for 2-22222222 from database
  ::my_ns::another_account
```

Note that my_account and my_ns::my_account are two distinct objects.

14.3.2. Destroying objects

Objects in Tcl are **not** garbage collected as in some other languages and have to be explicitly destroyed by calling their built-in destroy method. This also runs the object's destructor method.

```
% my_ns::my_account destroy
→ ::my_ns::my_account saving account data to database
```

Any operation on a destroyed object will naturally result in an error.

```
% my_ns::my_account balance
∅ invalid command name "my_ns::my_account"
```

Objects are also destroyed when its class or containing namespace is destroyed. Thus

```
% namespace delete my_ns
→ ::my_ns::another_account saving account data to database
% my_ns::another_account balance
∅ invalid command name "my_ns::another_account"
```

generates an error as expected.

14.3.3. Invoking methods

An object in Tcl behaves like an ensemble command of the form

```
OBJECT METHODNAME args....
```

This is the form used to invoke a method on the object from code "outside" the object.

```
% $acct balance
→ 1000000
% $acct deposit 1000
→ 1001000
```

As discussed in Section 14.2.4, when calling a method from another method **in the same object context**, the alias my is used to refer to the current object. So the deposit method we saw earlier calls the UpdateBalance method as:

```
my UpdateBalance $amount
```

14.3.3.1. Method contexts

A method runs in the context of its object's namespace (see Section 14.11.2.1). This means the object data members such as Balance, defined through variable, are in the scope of the method and can directly be referenced without any qualifiers as seen in the method definition above.

The method context also makes available several commands — such as self, next and my — which can only be called from within a method. These cannot be invoked from outside the method body even by prefixing with the object's command name.

We will see various uses of these commands as we proceed but for now note the use of my to refer to a method of the object in whose context the current method is running.

14.3.4. Accessing data members

Data members are not directly accessible from outside the object. Methods, such as balance in our example, have to be defined to allow callers to read and modify their values. Many OO-purists, and even non-purists like the author, believe this to be desirable.

However, Tcl being Tcl, it is always possible to add a variable access capability using the fact that each object has a private namespace that can be retrieved through introspection with the info object namespace command. Thus,

```
% set [info object namespace $acct]::Balance 5000
→ 5000
% $acct balance
→ 5000
```

This practice breaks encapsulation and is **not** recommended. However, some OO systems layered on top of TclOO do offer this feature in a structured manner that does not explicitly expose internal object namespaces. These are however not discussed here.

A good alternative is to automatically define *accessor* methods for public variables without the programmer having to explicitly do so. One such implementation is described in the Lifecycle Object Generators paper.

14.4. Inheritance

The defining characteristic of OO systems is support for *inheritance*. Inheritance refers to the ability of a *derived* class (also refered to as a *subclass*) to specialize a class — called its *base* class or *superclass* — by extending or modifying its behaviour.

Thus in our banking example, we may define separate classes representing savings accounts and checking accounts, each inheriting from the base account and therefore having a balance and methods for deposits and withdrawal. Each may have additional functionality, for example check writing facilities for the checking account and interest payments for the savings account.

The intention behind inheritance is to model *is-a* relationships. Thus a checking account *is a* bank account and can be used at any place in the banking model where the behaviour associated with a bank account is expected. This *is-a* relation is key when deciding whether to use inheritance or some other facility such as mix-ins.

Let us define our SavingsAccount and CheckingAccount. Instead of using oo::define as before, we will provide the full class definition as part of the oo::class command itself.

```
oo::class create SavingsAccount {
    superclass Account
    variable MaxPerMonthWithdrawals WithdrawalsThisMonth
    constructor {account_no {max_withdrawals_per_month 3}} {
        next $account_no
        set MaxPerMonthWithdrawals $max_withdrawals_per_month
    }
    method monthly_update {} {
        my variable Balance
        my deposit [my MonthlyInterest]
        set WithdrawalsThisMonth 0
    }
    method withdraw {amount} {
        if {[incr WithdrawalsThisMonth] > $MaxPerMonthWithdrawals} {
            error "You are only allowed $MaxPerMonthWithdrawals withdrawals a month"
        }
        next $amount
    }
    method MonthlyInterest {} {
        my variable Balance
        return [format %.2f [* $Balance 0.005]]
    }
}
oo::class create CheckingAccount {
    superclass Account
    method cash_check {payee amount} {
```

```
        my withdraw $amount
        puts "Writing a check to $payee for $amount"
    }
}
→ ::CheckingAccount
```

The `superclass` command in the class definition establishes that `SavingsAccount` and `CheckingAccount` inherit from `Account`. This statement by itself means they will behave exactly like the `Account` class, with the same methods and variables defined. Further declarations will extend or modify the class behaviour.

14.4.1. Methods in derived classes

Methods available in the base class are available in derived classes as well. In addition, new methods can be defined, such as `cash_check` and `monthly_update` in our example, that are only present on objects of the derived class.

If the derived class defines a method of the same name as a method in the base class, it overrides the latter and will be called when the method is invoked on an object of the derived class. Thus the `withdraw` method of the `SavingsAccount` class overrides the `withdraw` method of the base `Account` class. However, we are just modifying the original method's functionality with an additional condition, not replacing it. Therefore, after making the check we want to just pass on the request to the base class method and not duplicate its code. This is done with the command `next` which invokes the superclass method with the same name as the current method. This *method chaining* is actually only an example of a broader mechanism we will explore in detail in Section 14.9.

Constructors and destructors are also chained. If a derived class does not define a constructor, as is true for the `CheckingAccount` class, the base class constructor is invoked when the object is created. If the derived class does define a constructor, that is invoked instead and it is up to that constructor to call the base class constructor using `next` as appropriate. Destructors behave in a similar fashion.

Note that `next` may be called at any point in the method, not necessarily in the beginning or the end.

14.4.2. Data members in derived classes

Derived classes can define new data members using either `variable` in the class definition or `my variable` within a method as in `withdraw`.

Because data members are **always** defined in the namespace of the object, you have to careful about conflicts between variables of the same name being defined in a base class and a derived class if they are intended to represent different values.

Data members defined in a parent (or ancestor) class are also accessible within a derived class **but** they have to be brought within the scope of the method through the `variable` declaration in the derived class definition or the `my variable` statement within a method as is done in the implementation of `MonthlyInterest`. Although we use a direct variable reference there for expository purposes, in the interest of data hiding and encapsulation, direct reference to variables defined in ancestors should be avoided if possible. It would have been better to write the statement as

```
my deposit [format %.2f [* [my balance] $rate]]
```

Let us try out our new accounts.

```
% SavingsAccount create savings S-12345678 2
→ Reading account data for S-12345678 from database
    ::savings
% CheckingAccount create checking C-12345678
→ Reading account data for C-12345678 from database
    ::checking
% savings withdraw 1000
```

```
→ 999000
% savings withdraw 1000
→ 998000
% savings withdraw 1000 ❶
Ø You are only allowed 2 withdrawals a month
% savings monthly_update
→ 0
% checking cash_check Payee 500 ❷
→ Writing a check to Payee for 500
% savings cash_check Payee 500 ❸
Ø unknown method "cash_check": must be balance, deposit, destroy, monthly_update or withdraw
```

❶   Overridden base class method
❷   Method defined in derived class
❸   Check facility not available for savings

14.4.3. Multiple inheritance

Imagine our bank also provides brokerage services. Accordingly we define the following class:

```
oo::class create BrokerageAccount {
    superclass Account
    method buy {ticker number_of_shares} {
        puts "Buying $number_of_shares shares of $ticker"
    }
    method sell {ticker number_of_shares} {
        puts "Selling $number_of_shares shares of $ticker"
    }
}
→ ::BrokerageAccount
```

The company now decides to make it even more convenient for customers to lose money in the stock market. So we come up with a new type of account, a Cash Management Account (CMA), which combines the features of the checking and brokerage accounts. We can model this in our system using *multiple inheritance*, where the corresponding class inherits from more than one parent class.

```
oo::class create CashManagementAccount {
    superclass CheckingAccount BrokerageAccount
}
```

> ⚠ Be careful when using multiple superclass statements as the earlier declarations are overwritten if the -append option is not specified. The above example written using multiple superclass commands would be written as:
>
> ```
> oo::class create CashManagementAccount {
>     superclass CheckingAccount
>     superclass -append BrokerageAccount
> }
> ```

Our CMA account can do it all.

```
% CashManagementAccount create cma CMA-00000001
→ Reading account data for CMA-00000001 from database
  ::cma
% cma cash_check Payee 500
→ Writing a check to Payee for 500
% cma buy GOOG 100
```

→ Buying 100 shares of GOOG

Use of multiple inheritance is a somewhat controversial topic in OO circles. Be as it may, TclOO offers the facility, and also an alternative using mix-ins (see Section 14.6), and leaves the design choices for programmers to make.

14.5. Specializing objects

The next thing we talk about, object specialization, may be new to readers who are more familiar with class-based OO languages such as C++ where methods associated with objects are exactly those that are defined for the class(es) to which the object belongs.

In TclOO on the other hand, we can further "specialize" an *individual* object by overriding, hiding, and deleting methods defined in the class or even adding new ones. In fact, the potential specialization includes features such as forwarding, filters and mix-ins but we leave them for now as we have not discussed them as yet. As we will see, we can even change an object's class.

Specialization is done through the oo::objdefine command which is analogous to the oo::define command for classes except that it takes an object as its argument instead of a class. With the obvious exception of the commands constructor, destructor and superclass, all commands, like method, variable, export etc., available within oo::define scripts can also be used inside the script passed to oo::objdefine. The difference is that they act on a specific object instead of a class.

Like oo::define, oo::objdefine also has two syntactic forms.

```
oo::objdefine OBJ DEFINITIONSCRIPT
oo::objdefine OBJ SUBCOMMAND ARG ?ARG ...?
```

14.5.1. Object-specific methods

Let us illustrate with our banking example. Imagine our banking system had the requirement that individual accounts can be frozen based on an order from the tax authorities. We need to define a procedure we can call to freeze an account so all transactions on the account will be denied. Correspondingly, we need a way to unfreeze an frozen account. The following code accomplishes this.

```
proc freeze {account_obj} {
    oo::objdefine $account_obj {
        method UpdateBalance {args} {
            error "Account is frozen. Don't mess with the IRS, dude!"
        }
        method unfreeze {} {
            oo::objdefine [self] { deletemethod UpdateBalance unfreeze }
        }
    }
}
```

When the freeze procedure is passed an Account object, it uses oo::objdefine to override the UpdateBalance method that was part of the object's class definition with a object specific UpdateBalance method that raises an error instead.

It then defines a new method unfreeze that can be called on the object at the appropriate time to restore things back to normal. We could have actually defined an unfreeze procedure instead of a unfreeze method as follows:

```
proc unfreeze {account_obj} {
    oo::objdefine $account_obj deletemethod UpdateBalance
}
```

This would have accomplished the same job in a clearer manner. We chose to implement an unfreeze method instead to illustrate that we can actually change an object's definition even from **within** the object.

There are a couple of points that need to be elaborated:

- The self command is only usable within a method and returns the name of the current object when called without parameters. Thus the oo::objdefine command is instructed to modify the object itself. We will see other uses of the self command later.

- Although not required in our example, it should be noted that variables defined in the class are not automatically visible in object-specific methods. They need to be brought into scope with the my variable command.

- When called from within a oo::objdefine script, the deletemethod erases the specified object-specific methods. It does **not** affect methods defined in the class so the original UpdateBalance will still be in place and will no longer be overridden.

Let us see how all this works. At present Mr. Smith can withdraw money freely from his account.

```
% smith_account withdraw 100
→ 999900
```

So far so good. Now we get a court order to freeze Mr. Smith's account.

```
% freeze smith_account
```

Mr. Smith tries to withdraw money and run away to the Bahamas.

```
% smith_account withdraw [smith_account balance]
0 Account is frozen. Don't mess with the IRS, dude!
```

Have we affected other customers?

```
% $acct withdraw 100
→ 4900
```

No, only the smith_account object was impacted.

Cornered Mr. Smith pays up to unfreeze the account.

```
% smith_account unfreeze
% smith_account withdraw 100
→ 999800
```

Notice that the class definition of UpdateBalance was not lost in the process of adding and deleting the object-specific method.

This ability to define object-specific methods can be very useful. Imagine writing a computer game where the characters are modeled as objects. Several characteristics of the objects, such as the physics determining movement, are common and can be encapsulated with a class definition. The special "powers" of each character cannot be part of this class and defining a separate class for each character is tedious overkill. The special power of a character can instead be added to the character's object as a object-specific method. Even modeling scenarios like temporary loss of a power without a whole lot of conditionals and bookkeeping becomes very simple using the object specialization mechanisms.

14.5.2. Changing an object's class

Being a true dynamic OO language, TclOO can even change the class of an object through oo::objdefine. For example, one might change a savings account to a checking account.

```
% set acct [SavingsAccount new C-12345678]
→ Reading account data for C-12345678 from database
  ::oo::Obj228
% $acct monthly_update
→ 0
```

So far so good. Let us attempt to cash a check.

```
% $acct cash_check Payee 100
∅ unknown method "cash_check": must be balance, deposit, destroy, monthly_update or withdraw
```

Naturally that fails because it is not a checking account. Not a problem, we can fix that by morphing the object to a CheckingAccount.

```
% oo::objdefine $acct class CheckingAccount
```

We can now cash checks successfully

```
% $acct cash_check Payee 100
→ Writing a check to Payee for 100
```

but monthly updates no longer work as the account is no longer a SavingsAccount.

```
% $acct monthly_update
∅ unknown method "monthly_update": must be balance, cash_check, deposit, destroy or withdraw
% $acct destroy
→ ::oo::Obj228 saving account data to database
```

Needless to say, you have to be careful when "morphing" objects in this fashion since data members may differ between the two classes.

Note the optional form of the oo::objdefine command that we have used in the above code fragment. When the script passed to oo::define or oo::objdefine contains only one command, it can be directly specified as additional arguments to oo::define or oo::objdefine.

Lifecycle Object Generators describes an example of when such morphing might be used. Consider a state machine where each state is represented by a class that implements the state's behaviour. When a state change occurs, the state machine object changes its class to the class corresponding to the target state. See the abovementioned reference for implementation details.

14.6. Using mix-ins

Earlier we looked at the use of inheritance to extend a class. We will now look at another mechanism to extend or change the behaviour of classes (and objects) — mix-ins.

The literature on the subject describes mix-ins in several different ways, often depending on language-specific capabilities. From this author's perspective, a mix-in is a way to package a bundle of related functionality such that it can be used to extend one or more classes or objects. In some languages, multiple inheritance is used for this purpose but we will postpone that discussion until after we have seen an example of a mix-in.

Let us go back to our banking model. Imagine we have an Electronic Fund Transfer (EFT) facility that provides for transferring funds to other accounts. We will not worry about how this is done but just assume some global procedures are available for the purpose. This facility is available to all savings accounts but only to selected

checking accounts. There are several ways we could implement this but our preference in this case is for mix-ins over the alternatives for reasons we discuss later.

In TclOO a mix-in is also defined as a class in exactly the same manner as we have seen earlier. In fact, in theory any class can be a mix-in. What sets a mix-in apart is the conceptual model and how the class is **used**. In our example, the EFT facility would be modeled as a class that implements two methods, transfer_in and transfer_out. Conceptually, the class does not represent an *object*, but rather a *capability* or, as is termed in some literature, a *role*. It adds functionality to a "real" object.

```
oo::class create EFT {
    method transfer_in {from_account amount} {
        puts "Pretending $amount received from $from_account"
        my deposit $amount
    }
    method transfer_out {to_account amount} {
        my withdraw $amount
        puts "Pretending $amount sent to $to_account"
    }
}
→ ::EFT
```

Since we want all checking accounts to have this facility, we will add EFT to the CheckingAccount class as a mix-in. This is accomplished with the mixin command within a class definition script.

```
% oo::define CheckingAccount {mixin EFT}
% checking transfer_out 0-12345678 100
→ Pretending 100 sent to 0-12345678
% checking balance
→ 999400
```

We are now able to do electronic transfers on all checking accounts.

> Note that modifying the class definition in **any** manner, in this case adding a mix-in, also impacts **existing** objects of that class. Thus the checking object automatically supports the new functionality.

In the case of savings accounts, we only want **select** accounts to have this facility. Assuming our savings object represents one of these privileged accounts, we can add the mix-in to just that object through oo::objdefine.

```
% oo::objdefine savings {mixin EFT}
% savings transfer_in 0-12345678 100
→ Pretending 100 received from 0-12345678
  1003090.0
% savings balance
→ 1003090.0
```

Notice that the EFT class does not really know anything about accounts. It encapsulates features that can be added to **any** class or object that defines the methods deposit and withdraw required to support the mix-in's functionality. So if we had a BrokerageAccount class or object, we could mix it in there as well.

14.6.1. Using multiple mix-ins

A class or object may have multiple classes mixed in. So for example if we had a facility for electronic bill presentment implemented as a mix-in class BillPay, we could have added it along with EFT as a mix-in in a single statement

```
oo::define CheckingAccount {mixin BillPay EFT}
```

or as multiple statements

```
oo::define CheckingAccount {
    mixin EFT
    mixin -append BillPay
}
```

By default, the `mixin` command overwrites existing mix-in configuration so without the `-append` option when using multiple `mixin` statments, only class `BillPay` would be mixed into `CheckingAccount`.

We could also clear out all mix-ins from the class by specifying the `-clear` option to the `mixin` command.

14.6.2. Mix-ins versus inheritance

Because one of its goal is to provide the required infrastructure for additional OO system to be built on top, TclOO offers a wide variety of capabilities that sometimes overlap in their effect. The question then arises as to how to choose the appropriate feature for a particular design requirement. One of these design choices involves mix-ins and inheritance.

We offer the author's thoughts on the matter. Luckily, these tend to be few and far between so a couple of paragraphs is sufficient for this purpose.

Instead of mixing our `EFT` class into `CheckingAccount`, we could have made it a superclass and used multiple inheritance instead. Or even modified or derived from the `CheckingAccount` class to add transfer methods. Why did we choose to go the mix-in route?

Not directly inheriting or modifying the `CheckingAccount` class was a no-brainer for obvious reasons. The functionality is something that could be used for other account types as well and it does not make sense to duplicate code and add it to every class that needs those features. That leaves the question of multiple inheritance.

There were several considerations:

- Inheritance implies an *is-a* relationship between classes. Saying a checking account is-a "account that has transfer features" sounds somewhat contrived.

- The above stems from the fact that EFT does not really reflect a real object. It is more like a set of features or capabilities that accounts have. In the real world, it would be a checkbox on a account opening form for a checking account. The general thinking is that such classes are better modeled as mix-ins.

- Perhaps most important, when implemented as a mix-in, we can provide the feature sets to individual accounts, for example to specific savings accounts. You cannot use multiple inheritance to specialize individual objects in this manner.

For these reasons, mix-ins seemed a better choice in our design (aside from the fact that we needed **some** example to illustrate mix-ins).

There is one practical aspect of TclOO design that may drive your decision. Methods implemented via mix-ins appear in the method chain **before** methods defined on the object whereas inherited methods appear **after**. This was not relevant to our example because the mix-in only added new methods. It did not override exising ones.

14.7. Method forwarding

A method can be *forwarded* to another command, its *target*, so that when the method is invoked on the object, the target command is invoked instead. Forwarded methods may be defined on the class or on an object.

```
forward METHOD TARGET ?ARG ...?
```

This command may appear in the definition script for either classes or objects. *METHOD* is the name of the method to be defined. *TARGET* is the command to be invoked when that method is invoked. There are no restrictions on what *TARGET* may be. It may be a Tcl procedure or command, another object, a coroutine etc. It is invoked within

the context of the object so that name resolution occurs in that context. So for example, if `TARGET` is my, the method is effectively forwarded to another method defined for the same object.

When `METHOD` is invoked, any optional `ARG` arguments specified in the `forward` declaration are prepended to the list of arguments provided by the caller before it is passed to the target command.

Method forwarding is used in many patterns in object-oriented programming, for example composition. Earlier we defined the cash management account using multiple inheritance, in effect treating a `CashManagementAccount` as being a `CheckingAccount` and a `BrokerageAccount`. We could instead have thought of it as a consolidated account that **contained** the two account types. The class would then be defined as

```
oo::class create ConsolidatedAccount {
    constructor {acct_no} {
        CheckingAccount create checking_account $acct_no
        BrokerageAccount create brokerage_account $acct_no
    }
}
→ ::ConsolidatedAccount
```

We want the same operations available for this new account type as before. We can do this by forwarding methods to the appropriate contained account. For example, cash withdrawals would happen from the checking account.

```
oo::define ConsolidatedAccount {
    forward buy brokerage_account buy
    forward sell brokerage_account sell
    forward cash_check checking_account cash_check
    forward withdraw checking_account withdraw ❶
}
```

❶ Note we want withdrawals to be from the checking account

When the methods are forwarded, the target commands are resolved within the object's context. Thus `brokerage_account` and `checking_account` refer to the account objects we created within the object's context in the constructor.

This now has behaviour very similar to our `CashManagementAccount` based solution.

```
% ConsolidatedAccount create consolidated CONS-0000001
→ Reading account data for CONS-0000001 from database
  Reading account data for CONS-0000001 from database
  ::consolidated
% consolidated cash_check Payee 500
→ Writing a check to Payee for 500
% consolidated buy GOOG 100
→ Buying 100 shares of GOOG
```

Forwarding may also supply arguments to the targeted command as shown below.

```
oo::objdefine consolidated {
    forward quick_cash my withdraw 100
}
consolidated quick_cash
→ 999400
```

For illustrative purposes, this last definition differs from previous ones in the following respects:

- The forwarded method is only defined for the object, not the class.
- It is forwarded to another method in the same object through the use of the my command.

- It supplies an argument within the forwarding definition.

14.8. Filter methods

Imagine Mr. Smith is suspected of being up to his old tricks again and we need to monitor his accounts and log all activity. How would we do this? We could specialize every method for his accounts via oo::objdefine and log the activity before invoking the original method. We would have to do this for every method available to the object — those defined in the object, its class (and superclasses), object mix-ins and class mix-ins. This would be tedious and error prone. Moreover, since Tcl is a dynamic language, we would have to make sure we do that any time new methods were defined for the object or any ancestor and mix-in.

Filter methods offer a easier solution. A filter method is defined in the same manner as any method in the class or object. It is marked as a filter method using the filter command. Any method invocation on the object will then result in the filter method being invoked first.

We can add a filter method to the account object whose activity we want to track.

```
oo::objdefine smith_account {
    method Log args {
        my variable AccountNumber
        puts "Log([info level]): $AccountNumber [self target]: $args"
        return [next {*}$args]
    }
    filter Log
}
```

Now all actions on the account will be logged.

```
% smith_account deposit 100
→ Log(1): 2-71828182 ::Account deposit: 100
  Log(2): 2-71828182 ::Account UpdateBalance: 100
  999900
```

Notice from the output that all method invocations, even those called internally from deposit are recursively logged. The filter method must be aware that it may be recursively entered. We use the info level command to show the stack level. When methods are chained, the filter is called for **every** method in the chain.

Some additional notes on filter methods:

- Many times, the filter method needs to know what method the caller is actually trying to invoke. The self target command is useful for this purpose.
- Multiple filters may be present and are chained like any other method.
- Because filter methods are called for **all** method invocations, they are generally defined with a variable number of arguments.
- Filter methods may be defined on an object, as in our example, or on a class, in which case they will affect all objects belonging to the class.
- Our filter method is not exported because it starts with an upper-case letter. This means it will not be called accidentally by clients of the object. However, there is no requirement that filter methods must be private.
- The filter method normally passes on the call to the target method via next which can be called at any point in the filter method. Moreover, the filter is not required to call the target method at all.
- The filter method may choose to pass on the target method result, or something else entirely.
- Filter methods are bypassed when invoking constructors, destructors or the unknown method of a class.

14.8.1. Defining a filter class

The filter declaration need not occur in the same class that defines the filter method. This means you can define a generic class for a filter which can be mixed into a "client" class or object which can install or remove the filter at appropriate times as desired.

Let us rework our previous example. To start with a clean slate, let us get rid of the Log method defined earlier.

```
% oo::objdefine smith_account {
    filter -clear ❶
    deletemethod Log
}
```

❶   Note -clear option to clear any currently defined filters

Then we define a class that does the logging.

```
% oo::class create Logger {
    method Log args {
        my variable AccountNumber
        puts "Log([info level]): $AccountNumber [self target]: $args"
        return [next {*}$args]
    }
}
→ ::Logger
```

Since we only want transactions for that account to be logged, we mix it into the object and add the filter.

```
% oo::objdefine smith_account {
    mixin Logger
    filter Log
}
% smith_account withdraw 500
→ Log(1): 2-71828182 ::Account withdraw: 500
  Log(2): 2-71828182 ::Account UpdateBalance: -500
  999400
```

As you can see, we have the same behaviour as before. The advantage of course is that defining a class allows a collection of additional behaviours to be abstracted and easily added to any class or object without repeating the code.

In our example above, we used the -clear option to filter to first remove any filters from the object before adding the Log filter later. This is because filter by default will append to any filters already installed. Alternatively, we could have specified the -set option to filter which would **replace** any existing filters.

14.8.2. When to use filters

Filters could be replaced by other techniques such as overriding and then chaining methods. Conversely, method overrides, such as in our account freeze example, could be replaced by filters. Usually though it is clear which one makes the most sense. Some general rules are

- If we need to hook into multiple methods, it is easiest to use a filter method rather than override individual methods. If necessary, self target can be used within the filter to selectively hook specific methods as illustrated in Section 14.11.3.4.
- When a method behaves more as an "observer" on an object as opposed to being a core part of the object's function, a filter method is a better fit.
- Filter methods are always placed at the front of the method chain so that can be a factor as well in deciding to use a filter.

14.9. Method chains

Throughout this chapter we have seen that when a method is invoked on an object, the code implementing the method for that object may come from several different places — the object, its class or an ancestor, a mix-in, forwarded methods, filters or even unknown method handlers. TclOO locates the code to be run by searching the potential implementations in a specific order. It then runs the first implementation in this list. That implementation **may** choose to chain to the next implementation in the list using the next command and so on through the list.

14.9.1. Method chain order

For the exact search order and construction of this method chain, see the reference documentation of the next command. Here we will simply illustrate with an example where we define a class hierarchy with multiple inheritance, mix-ins, filters and object-specific methods. Note our method definitions are empty because we are not actually going to call them.

```
oo::class create ClassMixin { method m {} {} }
oo::class create ObjectMixin { method m {} {} }
oo::class create Base {
    mixin ClassMixin
    method m {} {}
    method classfilter {} {}
    filter classfilter
    method unknown args {}
}
oo::class create SecondBase { method m {} {} }
oo::class create Derived {
    superclass Base SecondBase
    method m {} {}
}
→ ::Derived
```

Having defined our classes, let us create an object and add in some object-specific methods and mix-ins.

```
Derived create o
oo::objdefine o {
    mixin ObjectMixin
    method m {} {}
    method objectfilter {} {}
    filter objectfilter
}
```

We have created an object of class Derived that inherits from two parent classes, all of which define a method m. Further we have mix-ins for both a class and directly into the object. To confuse matters further, we have filters defined at both the class and object levels.

What will the method chain for method m look like? Luckily, we do not have to work it out while reading the manpage. We can do it through introspection via the info object call command.

```
% print_list [info object call o m]
→ filter objectfilter object method
  filter classfilter ::Base method
  method m ::ObjectMixin method
  method m ::ClassMixin method
  method m object method
  method m ::Derived method
  method m ::Base method
```

```
method m ::SecondBase method
```

The output shows the method chain so we can see for example that the filter methods are first in line.

The info object call command returns a list that contains the method chain for a particular method invocation for a particular object. Each element of the list is a sublist with four items:

- the type which may be method for normal methods, filter for filter methods or unknown if the method was invoked through the unknown facility
- the name of the method which, as noted from the output, may not be the same as the name used in the invocation
- the source of the method, for example, a class name where the method is defined
- the *implementation* type of the method which may be method or forward

We reiterate that not every method in the chain is automatically invoked. Whether a method occuring in the list is actually called or not will depend on preceding methods passing on the invocation via the next command.

14.9.2. Method chain for unknown methods

What does the method chain look like for a method that is not defined for the object? We can find out the same way.

```
% print_list [info object call o nosuchmethod]
→ filter objectfilter object method
  filter classfilter ::Base method
  unknown unknown ::Base method
  unknown unknown ::oo::object {core method: "unknown"}
```

As expected, the unknown method, where defined, is called. Note the root oo::object object which is the ancestor of **all** TclOO objects, has a predefined unknown method.

14.9.3. Retrieving the method chain for a class

The above example showed the method chain for an object. There is also a info class call command that works with classes instead of objects.

```
% print_list [info class call Derived m]
→ filter classfilter ::Base method
  method m ::ClassMixin method
  method m ::Derived method
  method m ::Base method
  method m ::SecondBase method
```

14.9.4. Inspecting method chains within method contexts

Within a method context, the command self call returns more or less the same information for the current object as info object call.

In addition, you can use self call from within a method context to locate the current method in the method chain. This command returns a pair, the first element of which is the same as the method chain list as returned by info class call command. The second element is the index of the current method in that list.

An example will make this clearer.

```
% catch {Base destroy} ❶
→ 0
% oo::class create Base {
    constructor {} {puts [self call]}
    method m {} {puts [self call]}
```

385

```
}
→ ::Base
% oo::class create Derived {
    superclass Base
    constructor {} {puts [self call]; next}
    method m {} {
        puts [self call]; next
    }
}
→ ::Derived
% Derived create o
→ {{method <constructor> ::Derived method} {method <constructor> ::Base method}} 0
  {{method <constructor> ::Derived method} {method <constructor> ::Base method}} 1
  ::o
% o m
→ {{method m ::Derived method} {method m ::Base method}} 0
  {{method m ::Derived method} {method m ::Base method}} 1
```

❶   Clean up any previous definitions

Note the special form `<constructor>` for constructors. Destructors similarly have the form `<destructor>`.

> Constructor and destructor method chains are only available through `self call`, not through `info class call`.

14.9.5. Looking up the next method in a chain

At times a method implementation may wish to know if it is the last method in a method chain and if not, what method implementation will be invoked next. This information can be obtained with the `self next` command from within a method context.

We illustrate by modifying the m method of the Derived class that we just defined.

```
% oo::define Derived {
    method m {} { puts "Next method in chain is [self next]" }
}
% o m
→ Next method in chain is ::Base m
```

As seen, `self next` returns a pair containing the class or object implementing the next method in the method chain and the name of the method (which may be `<constructor>` and `<destructor>`). In the case the current method is the last in the chain, an empty list is returned.

Notice that although the next method in the method chain is printed out, it does not actually get invoked because the m method in Derived no longer calls `next`.

> Do not confuse `self next` with `next`. The latter invokes the next method in the method chain while the former only tell you what the next method is.

There is one important issue solved by `self next` that we will illustrate with an example. Imagine we want to package some functionality as a mix-in class. The actual functionality is immaterial but it is intended to be fairly general purpose (for example, logging or tracing) and mixable into any class.

```
% oo::class create GeneralPurposeMixin {
    constructor args {
        puts "Initializing GeneralPurposeMixin";
```

```
        next {*}$args
    }
}
→ ::GeneralPurposeMixin
% oo::class create MixerA {
    mixin GeneralPurposeMixin
    constructor {} {puts "Initializing MixerA"}
}
→ ::MixerA
% MixerA create mixa
→ Initializing GeneralPurposeMixin
  Initializing MixerA
  ::mixa
```

So far so good. Now let us define another class that also uses the mix-in.

```
% oo::class create MixerB {mixin GeneralPurposeMixin}
→ ::MixerB
% MixerB create mixb
Ø Initializing GeneralPurposeMixin
  no next constructor implementation
```

Oops. What happened? If it is not clear from the error message, the issue is that the GeneralPurposeMixin class naturally calls next so that class that mixes it in can get initialized through its constructor. The error is raised because class MixerB does not have constructor so there is no "next" method (constructor) to call.

This is where self next can help. Let us redefine the constructor for GeneralPurposeMixin.

```
% oo::define GeneralPurposeMixin {
    constructor args {
        puts "Initialize GeneralPurposeMixin";
        if {[llength [self next]]} {
            next {*}$args
        }
    }
}
% MixerB create mixb
→ Initialize GeneralPurposeMixin
  ::mixb
```

It all works now because we only call next if there is in fact a next method to call.

14.9.6. Controlling invocation order of methods

As we have seen in our examples, a method can use the next command to invoke its successor in the method chain. With multiple inheritance, mix-ins, filters involved, it may sometimes be necessary to control the order in which inherited methods are called. The next command, which goes strictly by the order in the method chain, is not suitable in this case.

The nextto command allows this control. It is similar to next except that it takes an argument that specifies the name of the class that implements the next method to be called.

```
nextto CLASSNAME ?args?
```

Here CLASSNAME must be the name of a class that implements a method appearing later in the method chain.

When might you use this ? Well, imagine you define a class that inherits from two classes whose constructors take different arguments. How do you call the base constructors from the derived class? Using next would not work because the parent class constructors do not take the same arguments.

That's where nextto rides to the rescue as illustrated below.

```
oo::class create ClassWithOneArg {
    constructor {onearg} {puts "Constructing [self class] with $onearg"}
}
oo::class create ClassWithNoArgs {
    constructor {} {puts "Constructing [self class]"}
}
oo::class create DemoNextto {
    superclass ClassWithNoArgs ClassWithOneArg
    constructor {onearg} {
        nextto ClassWithOneArg $onearg
        nextto ClassWithNoArgs
        puts "[self class] successfully constructed"
    }
}
→ ::DemoNextto
```

We can now call it without conflicts.

```
% [DemoNextto new "a single argument"] destroy
→ Constructing ::ClassWithOneArg with a single argument
  Constructing ::ClassWithNoArgs
  ::DemoNextto successfully constructed
```

14.10. Programming without classes

Our exposition of object-oriented programming in Tcl has so far been centered around classes. Behaviours, object construction, inheritance and other relationships are all expressed in terms of classes.

However, not all object-oriented programming models involve classes. Another style, refered to in various forms as *classless* or *prototype-based programming*, dispenses with classes completely. Instead, objects are *cloned* from other objects, called prototypes, with the same methods and configuration. The inheritance features in class-based systems are replaced by the ability to add, remove or otherwise modify methods and delegate others.

As we stated in our introductory chapter, one of the defining features of TclOO is its flexibility in being adaptable to different programming models. Here we present classless object-based programming as one example of this.

We have already seen one of the requirements for such a system — the ability to define methods at an individual object level. We now describe how TclOO fulfils two additional ones — creating objects outside of classes, and cloning of objects.

The first of these is very straightforward, given our earlier discussion of oo::object. Given its dual nature as a class as well as an object, we can use it for creating a classless object.

```
oo::object create oa → ::oa
```

Strictly speaking, this object does have a class as seen below.

```
info object class oa → ::oo::object
```

For practical purposes though we can still treat this as classless as we did not have to go through an explicit class definition.

We now have what is essentially a shell of an object. The only method available for the object is destroy which it inherits from oo::object. There are no object-specific methods defined on it yet.

```
info object methods oa → (empty)
```

We now need to fill the object with methods and data. We know how to do that with `oo::objdefine`.

```
oo::objdefine oa {
    variable x
    method setx {val} {set x $val}
    method getx {} {set x}
}
oa setx 100
→ 100
```

We now have a functioning object. The last requirement is the ability to clone the object. The `oo::copy` command does this for us.

```
oo::copy OBJ ?NEWOBJ?
```

The command creates a new object named *NEWOBJ* that is a copy of *OBJ*. If *NEWOBJ* is not specified, the new object is created with an automatically generated name. The new object is of the same class as the source object, includes any object-specific methods that were defined on it and contains the same variables with values as of the time of copying.

```
oo::copy oa ob → ::ob
ob getx       → 100
```

As seen, we now have a new object ob that is the copy of oa with the same methods and data.

We are now free to extend (or not) this new object as we wish.

```
oo::objdefine ob method doublex {} {incr x $x} → (empty)
ob doublex                                      → 200
```

Notice that we now have similar functionality to method inheritance and overriding in class-based systems. A full-blown prototype-based object system would need more machinery than described here but that is all implementable with the mechanisms we have described so far and the introspection capabilities we will describe in the next section.

One final point about `oo::copy` that is worth noting. In some cases, copying across methods and variable definitions may not suffice for cloning an object. For example, the source object may have a variable that is being traced or a file that is open. To deal with such cases, the object may define a `<cloned>` method. This will be invoked at the time the object is cloned and is passed the source object as its sole argument. This method can then do any additional work required to create a full clone of the original.

14.11. OO introspection

Introspection of classes and objects from any context is primarily accomplished through the `info class` and `info object` ensemble commands. These have subcommands that return different pieces of information about a class or an object. In addition, the `self` command can be used for introspection of an object from inside a method context for that object.

The Tcl object browser[3] is a useful tool for peering inside classes, objects and namespaces. It displays the class hierarchy, objects, namespaces and method definitions in an easily navigable interface.

14.11.1. Introspecting classes

14.11.1.1. Enumerating classes

Classes are also objects in TclOO and therefore the same command `info class instances` used to enumerate objects can be used to enumerate classes.

```
% info class instances oo::class
→ ::oo::object ::oo::class ::oo::Slot ::Account ::SavingsAccount ::CheckingAccount
  ↳ ::BrokerageAccount ::CashManagementAccount ::EFT ::ConsolidatedAccount ::Logger
  ↳ ::ClassMixin ::ObjectMixin ::SecondBase ::Base ::Derived ::GeneralPurposeMixin ::MixerA
  ↳ ::MixerB ::ClassWithOneArg ::ClassWithNoArgs ::DemoNextto
```

We pass `oo::class` to the command because that is the class that all classes (or class objects, if you prefer) belong to. The returned list contains two interesting elements:

- `oo::class` is returned because as we said it is a class itself (the class that all class objects belong to) and is therefore an instance of itself.
- If that were not confusing enough, `oo::object` is also returned. This is the root class of the object hierarchy and hence is an ancestor of `oo::class`. At the same time it is a class and hence must be an instance of `oo::class` as well.

This circular and self-referential relationship between `oo::object` and `oo::class` seems strange but it is what allows all programming constructs in TclOO to be work in consistent fashion. It is also a common characteristic of many OO systems.

As before, we can also restrict the classes returned by specifying a pattern that the name must match.

```
% info class instances oo::class *Mixin
→ ::ClassMixin ::ObjectMixin ::GeneralPurposeMixin
```

14.11.1.2. Checking if an object is a class

The `info object isa class` command returns 1 if its argument references a class and 0 otherwise. A class is an instance of `oo::class` or one of its subclasses.

```
info object isa class SavingsAccount → 1
info object isa class savings          → 0 ❶
info object isa class clock            → 0 ❷
```

❶ An object but not a class
❷ A command and not a class

A related command is `info object isa metaclass` which returns 1 if the passed argument is a class that can create classes.

```
info object isa metaclass oo::class → 1
info object isa metaclass Account    → 0 ❶
```

❶ A class but not one that can create classes

---

[3] https://chiselapp.com/user/eugene.mindrov/repository/tcl-class-browser/home

14.11.1.3. Inspecting class relationships

The `info class superclasses` command returns the **direct** superclasses of a class.

```
% info class superclasses CashManagementAccount
→ ::CheckingAccount ::BrokerageAccount
% info class superclasses ::oo::class
→ ::oo::object
```

Notice that `oo::object` is a superclass of `oo::class`.

Conversely, the `info class subclasses` will return the classes directly inheriting from the specified class.

```
% info class subclasses Account
→ ::SavingsAccount ::CheckingAccount ::BrokerageAccount
```

As one might expect, there is also a command, `info class mixins` for listing mix-ins.

```
% info class mixin CheckingAccount
→ ::EFT
```

14.11.2. Introspecting objects

As for classes, there are several commands that let us introspect on TclOO objects.

14.11.2.1. Object identity

Under some circumstances, an object needs to discover its own identity from within its own method context. A method may need to know the command name of the object

- when an object method has to be passed to a command callback
- when an object is redefined "on the fly" from within a method, its name must be passed to `oo::objdefine`. See Section 14.5.1 for an example.

This can be accomplished with the `self object` command, which can also be called as simply `self`. We have seen this used in several instances in this chapter.

In addition to the command used to access it, an object may also be identified by the unique namespace in which the object state is stored. This is obtained through the `self namespace` command within a method context or with the `info object namespace` command elsewhere.

```
% oo::define Account {method get_ns {} {return [self namespace]}}
% savings get_ns
→ ::oo::Obj223
% set acct [Account new 0-0000000]
→ Reading account data for 0-0000000 from database
  ::oo::Obj256
% $acct get_ns
→ ::oo::Obj256
% info object namespace $acct
→ ::oo::Obj256
```

Notice that when we create an object using new, the namespace matches the object command name. This is an artifact of the implementation and this should not be relied on. In fact, like any other Tcl command, the object command can be renamed.

```
% rename $acct temp_account
```

```
% temp_account get_ns
→ ::oo::Obj256
```

As you can see, the object command and its namespace name no longer match. Also note that the namespace does not change when the object is renamed.

14.11.2.2. Checking if a command is an object

The info object isa object command can be used to check if a command is actually an object.

```
info object isa object savings       → 1 ❶
info object isa object clock         → 0 ❷
info object isa object nosuchcommand → 0 ❸
```

❶   A OO object
❷   A command but not an object
❸   No such command

14.11.2.3. Enumerating objects

The info class instances command returns a list of objects belonging to the specified class.

```
% info class instances Account
→ ::oo::Obj217 ::smith_account ::temp_account
% info class instances SavingsAccount
→ ::savings
```

As seen above, this command will only return objects that directly belong to the specified class, not if the class membership is inherited.

You can optionally specify a pattern argument in which case only objects whose names match the pattern using the rules of the string match command are returned. This can be useful for example when namespaces are used to segregate objects.

```
% info class instances Account ::oo::*
→ ::oo::Obj217
```

14.11.2.4. Inspecting class membership

You can get the class an object belongs to with the info object class command.

```
info object class savings → ::SavingsAccount
```

The same command will also let you check whether the object belongs to a class, taking inheritance into account.

```
info object class savings SavingsAccount  → 1
info object class savings Account         → 1
info object class savings CheckingAccount → 0
```

The info object isa typeof is an alternate means of getting at the same information.

```
info object isa typeof savings Account → 1
```

For enumerating the classes mixed-in with an object, use `info object mixins`, analogous to `info class mixins`.

```
info object mixins savings → ::EFT
```

Conversely, to check if a class is **directly** mixed into an object, use `info object isa mixin`.

```
info object isa mixin savings EFT → 1
```

From within the method context of an object, the command `self class` command returns the class defining the currently executing method. **Note this is not the same as the class the object belongs to as the example below shows.**

```
% catch {Base destroy} ❶
→ 0
% oo::class create Base {
    method m {} {
        puts "Object class: [info object class [self object]]"
        puts "Method class: [self class]"
    }
}
→ ::Base
% oo::class create Derived { superclass Base }
→ ::Derived
% Derived create o
→ ::o
% o m
→ Object class: ::Derived
  Method class: ::Base
```

❶   Clean up any previous definitions

> ⚠ The `self class` command will fail when called from a method defined directly on an object since there is no class associated with the method in that case.

14.11.3. Introspecting methods

14.11.3.1. Enumerating methods

The list of methods implemented by a class or object can be retrieved through `info class` methods and `info object` methods respectively. Options can be specified to control whether the list includes inherited and private methods. The `-private` option controls whether non-exported methods are also included in the returned list.

```
% info class methods CheckingAccount ❶
→ cash_check
% info class methods CheckingAccount -private ❷
→ cash_check
% info class methods CheckingAccount -all ❸
→ balance cash_check deposit destroy get_ns transfer_in transfer_out withdraw
% info class methods CheckingAccount -all -private ❹
→ <cloned> UpdateBalance balance cash_check deposit destroy eval get_ns transfer_in
    ↳ transfer_out unknown variable varname withdraw
% info object methods smith_account -private ❺
```

❶ Lists all methods defined and exported by CheckingAccount itself
❷ Lists both exported and private methods defined by CheckingAccount
❸ Lists all methods defined and exported by CheckingAccount, its ancestors, or mix-ins
❹ Lists both exported and private methods defined by CheckingAccount, its ancestors, or mix-ins
❺ Lists exported and non-exported methods defined in the object itself

The list of methods returned includes forwarded methods as well as shown by our example class for method forwarding.

```
% info class methods ConsolidatedAccount ❶
→ cash_check sell buy withdraw
% info object methods consolidated ❷
→ quick_cash
```

❶ Methods forwarded in the class
❷ Methods forwarded in the object

You can distinguish between "normal" methods and forwarded methods in the returned list by querying its type with the info class methodtype and info object methodtype commands. The commands will return method for the former and forward for the latter.

```
info class methodtype Account withdraw            → method
info class methodtype ConsolidatedAccount withdraw → forward
info object methodtype consolidated quick_cash     → forward
```

14.11.3.2. Retrieving method definitions

To retrieve the definition of a specific method that is not forwarded, use info class definition. This returns a pair consisting of the method's arguments and its body.

```
% info class definition Account UpdateBalance
→ change {
        set Balance [+ $Balance $change]
        return $Balance
    }
```

The method whose definition is being retrieved has to be defined in the specified class, not in an ancestor or a class that is mixed into the specified class.

Similarly, info object definition will return the definition of a method **directly** defined on an object. It will raise an error if passed a method name that is defined on the object's class.

Constructors and destructors are retrieved differently via info class constructor and info class destructor respectively.

```
% info class constructor Account
→ account_no {
        puts "Reading account data for $account_no from database"
        set AccountNumber $account_no
        set Balance 1000000
    }
```

For methods that are forwards, the info class forward and info object forward return information about the forward definition in a class and object respectively.

```
% info class forward ConsolidatedAccount buy
→ brokerage_account buy
```

```
% info object forward consolidated quick_cash
→ my withdraw 100
```

14.11.3.3. Inspecting method chains and contexts

The info class call command retrieves the method chain for a method. From a method context, the self call command returns similar information while self next identifies the next method implementation in the chain. We have already discussed these in detail in Section 14.9. There are two additional related commands that provide further information within a method context.

- The self method command returns the name of the current method being executed.
- The self caller command returns information of the caller of the method **when called from another method**. This is in the form of a list of three elements — the class, the object and the calling method.

```
oo::class create C {
    method m {} {
        lassign [self caller] cls obj meth
        puts "In [self method], called from method $meth in object $obj of class $cls"
    }
    constructor {} { my m }
}
C create c
→ ::c
  In m, called from method <constructor> in object ::c of class ::C
```

14.11.3.4. Inspecting filters

The list of methods that are set as filters can similarly be obtained with info class filters or info object filters.

```
% info object filters smith_account
→ Log
```

Note that info object filters will return a list of filter methods **directly** defined on the object. It will not include filters defined on the object's class.

When a method is run as a filter, it is often useful for it to know the real target method being invoked. This information is returned by self target which can only be used from within a filter context. Its return value is a pair containing the declarer of the method and the target method name.

For example, suppose instead of logging every transaction as in our earlier example, we only wanted to log withdrawals. In that case we could have defined the Log command as follows:

```
oo::define Logger {
    method Log args {
        if {[lindex [self target] 1] eq "withdraw"} {
            my variable AccountNumber
            puts "Log([info level]): $AccountNumber [self target]: $args"
        }
        return [next {*}$args]
    }
}
```

We would now expect only withdrawals to be logged.

```
% smith_account deposit 100
→ 999500
% smith_account withdraw 100
```

```
→ Log(1): 2-71828182 ::Account withdraw: 100
  999400
```

The other piece of information that is provided inside a filter method is about the filter itself and is available through the self filter command.

Let us redefine our Log filter yet again to see this.

```
% oo::define Logger {
    method Log args {
        puts [self filter]
        return [next {*}$args]
    }
}
% smith_account withdraw 1000
→ ::smith_account object Log
  ::smith_account object Log
  998400
```

As seen above, the self filter command returns a list of three items:

- the name of the class or object where the filter is declared. Note this is **not** necessarily the same as the class in which the filter method is defined. Thus above, the filter was **defined** in the Logger class but **declared** in the smith_account object.

- either object or class depending on whether the filter was declared inside an object or a class.

- the name of the filter.

📋 You will see two output lines in the above example. Remember the filter is called at *every* method invocation. Thus the Log method is invoked twice, once before the withdraw method, and then again when that method in turn calls UpdateBalance.

14.11.4. Enumerating data members

The command info class variables returns the list of variables that have been declared with the variable statement inside a class definition and are therefore automatically brought within the scope of the class's methods.

```
% info class variables SavingsAccount
→ MaxPerMonthWithdrawals WithdrawalsThisMonth
```

The listed variables are only those defined through the variable statement **for that specified class**. Thus the above command will not show the variable Balance as that was defined in the base class Account, not in SavingsAccount.

For enumerating variables for an object as opposed to a class, there are two commands:

- info object variables behaves like info class variables but returns variables declared with variable inside object definitions created with oo::objdefine. These may not even exist yet if they have not been initialized.

- info object vars returns variables currently **existing** in the object's namespace and without any consideration as to how they were defined.

Hence the difference between the output of the following two commands.

```
% info object variables smith_account
% info object vars smith_account
```

```
→ Balance AccountNumber
```

The first command returns an empty list because no variables were declared through variable for **that object**. The second command returns the variables in the object's namespace. The fact that they were defined through a class-level declaration is irrelevant.

14.12. Chapter summary

This chapter described the core Tcl facilities for object-oriented programming. What distinguishes these from many other languages is their dynamic nature, which permits classes and objects to be changed on the fly, and the extensive customizability that permits many different styles of object-oriented programming. The references at the end of the chapter explore these aspects in more detail.

14.13. References

DKF2005
> *TIP #257: Object Orientation for Tcl*, Fellows et al, http://tip.tcl.tk/257. The Tcl Implementation Proposal describing TclOO.

WOODS2012
> *Lifecycle Object Generators*, Woods, 19th Annual Tcl Developer's Conference, Nov 12-14, 2012. Describes a number of useful OO patterns built on top of TclOO.

DKF2013
> *Adventures in TclOO*, Fellows, www.tclcommunityassociation.org/wub/proceedings/.../Adventures-in-TclOO.pdf. Discussion of some advanced TclOO techniques and applications.

15

The Event Loop

Great minds discuss ideas; average minds discuss events; small minds discuss people.

— Eleanor Roosevelt

Not possessing a great mind, and unwilling to accept the possibility of a small one, I'm left with no choice but to discuss events.

Most of the programming we have discussed so far involves a sequential style of program flow where the Tcl interpreter executes a script one command at a time and then terminates when the last command in the script is finished. For applications that are primarily command line utilities, like file search, this is adequate. For other "long running" applications whose function is to react to different types of external events, this is not suitable model. Common examples include GUI applications which react to events like mouse clicks and key presses and network services that respond to requests from multiple clients. These applications sit idle waiting for specific events to occur upon the occurence of which they execute *event handlers*. These event handlers may update the display in response to a key press, send back a web page to the requesting client and so on. These applications are said to be written in *event driven* style.

15.1. Event sources and types

Events may come from a variety of sources, such as input devices, network connections and so on. Correspondingly there are a number of different event types. Out of the box, Tcl supports the following event types:

- Timer events that are generated on the expiry of some time interval. The `after` command is one way for scheduling these.
- Channel events that indicate I/O activity on the channel. These events may be related to arrival of data on a serial port, a network connection request, the user entering a line on standard input and so on.
- Idle events, which are "pseudo-events" that are generated when the system has no other events to process. These are used by applications to schedule background tasks that are to be run when the system is otherwise idle. We sometimes refer to these as *idle tasks*.

In addition, extensions can add both new event sources and event types. The most commonly encountered ones are those added by the Tk graphical interface toolkit. These events include display events such windows becoming visible, mouse movement and clicks and keyboard input.

How you register event handlers for the different events is dependent on the event source generating the events. For example, use the `chan event` or `fileevent` commands to register for channel events, the `after` command for scheduling timer events, the Tk `bind` command for Tk user interface events and so on.

We will look at the `after` command in Section 15.3 and the channel related commands in Chapter 17 but first let us look at the event loop.

15.2. The Tcl event loop

Tcl provides built-in support for event-driven applications through its *event loop*. The basic flow of event driven applications in Tcl is as follows:

1. At startup the application registers event handlers to be invoked on events of interest.

2. The event loop is entered. This sits in wait for events to occur and on the occurence of such an event, invokes the associated handler(s).

3. Each handler takes whatever actions are necessary in response to the event. As part of its operation, a handler may register itself or other handlers for additional events or unregister existing handlers.

4. After the handler completes, control returns to the event loop and the cycle is repeated.

The second step above, entering the event loop, may be done by the application's native code outside of the Tcl interpreter, or from within a Tcl script through any of several commands like `update` or `vwait`. Notice that the latter implies that event loops can be nested, something that we will look at in more detail later.

> As we will see in later chapters, a Tcl application may consist of multiple threads each running multiple Tcl interpreters. In this case, each thread runs an independent event loop which is shared among all interpreters running in that thread.

15.2.1. The event and idle task queues

Internally, the event loop implementation runs off of two queues:

- The event queue: on the occurence of a physical event, such as a mouse click or arrival of data on a channel, the driver or module responsible will place an entry in this queue with details of the event and the handler to be called for the event.

- The idle task queue: this second queue holds tasks to be executed when the system is idle, i.e. has no events to process. At the script level, tasks can be placed on this queue through the `after idle` command.

15.2.2. Event loop operation

When the event loop runs,

1. It first checks the event queue for any handlers that have been queued up to be run and executes them if any. Note the handlers may themselves add new entries to the event queue.

2. If the event queue is empty, it checks with various registered event sources if new events have occurred. Any new events are placed on the event queue and the event loop goes back to the first step.

3. If there are no entries to process on the event queue, the event loop runs **all** the tasks present on the idle task queue.

4. If the idle task queue is empty, the event loop sits in wait for the next event to occur.

15.2.3. Entering the event loop

An application may itself enter the event loop from the C level. For example, the `wish` application evaluates the contents of any script file supplied as an argument and then enters the event loop processing user interface events. Thus in a `wish` based application, the event loop is effectively always running.

Alternatively, the Tcl script itself can initiate the event loop from script. This is commonly done from long running programs like network servers that are not GUI programs and therefore use `tclsh` to host the interpreter rather than `wish`.

Since we do not delve into C programming in this book, we will only concern ourselves with the latter method — running the event loop from a script.

15.2.3.1. Waiting on a variable: `vwait`

The most common way of entering the event loop from a script is the `vwait` command.

```
vwait VARNAME
```

The *VARNAME* variable name is resolved in the **global** context, and not in the context of the caller of vwait. You can also use a fully qualified namespace variable instead.

The command enters the Tcl event loop calling the handlers for events as they occur until the variable *VARNAME* is written at which point the command returns.

Unlike wish, the tclsh application does **not** automatically enter the event loop. You will therefore often find the following command at the end of a script that implements a tclsh based event-driven application such as a network server.

```
vwait forever
```

When the above command is executed, tclsh will enter the event loop processing network connection requests until one of the event handlers either calls exit to terminate the application or sets the variable forever at which point the command returns, the end of the script is reached and tclsh exits. We will see an example of such a server in Chapter 17.

Note that there is nothing special about the name forever above. It is commonly used as a hint that the command is supposed to loop forever and not return.

Here is a short example illustrating vwait that you can try out in the tclsh shell. The example uses the after command that we examine later to schedule timer events that trigger after a specified interval.

```
% after 1000 [list puts "1 second elapsed"]
→ after#0
% after 2000 [list puts "2 second elapsed"]
→ after#1
% after 3000 [list set ::done 1]
→ after#2
% vwait done
→ 1 second elapsed
  2 second elapsed
% puts $done
→ 1
```

Running the above sequence, you will see the tclsh prompt disappear for 3 seconds. Because the event loop is running, the timer events are triggered at one second intervals. After 3 seconds the variable done is set causing the vwait command (and event loop) to return.

15.2.3.1.1. Avoiding deadlocks with vwait

The vwait command should be used with some care from within code that runs within an event handler else deadlock can occur. Consider the following script.

```
proc handler {} {
    puts "handler enter"
    set ::varA 1
    vwait ::varB
    puts "handler exit"
}
proc demo {} {
    after 1000 handler
    vwait ::varA
    set ::varB 1
}
demo
```

If you run this fragment in a tclsh shell, you will find that the prompt does not return after printing handler enter. The expectation was that setting varA within the handlerA would permit the demo procedure to continue

which in turn would set varB allowing handler to also complete. However, what actually happens is that control will not return to demo until **after** handler completes. But handler cannot complete until varB is set. Hence deadlock.

This example is contrived but similar situations will arise in real life if you are not careful. The fundamental point to be made is that **multiple calls to** vwait **do not execute "in parallel", they are nested**. The outermost call will not return until inner calls have returned.

These situations generally arise as a result of trying to make asynchronous execution appear synchronous. See the vwait documentation in the Tcl reference pages for additional information and workarounds. In many cases coroutines, which we discuss in Chapter 21, are a better option.

As an aside, the above example also illustrates that it is possible to enter the event loop recursively.

15.2.3.2. Single invocation: update

The vwait command runs the event loop continuously until a variable is set, waiting for events if there are none to be processed currently. In contrast, the update command invokes the event loop once and returns when there are no pending events to be processed.

```
update ?idletasks?
```

If idletasks is not specified, the command enters the event loop and runs all event handlers that are pending including new ones that may be added while the event loop is running. The command returns when no more events remain in the event queue. The following short example illustrates its behaviour.

```
proc handler {} {
    puts "Event 0"
    after 0 [list puts "Event 1"]
}
after 0 handler
after 1000 [list puts "Event 2"]
update
→ Event 0
  Event 1
```

The after command schedules a timer to expire after a specified time. In this case 0 means the timer expires immediately while 1000 indicates expiry after one second. When a timer expires, a timer event is triggered and the corresponding handler becomes pending. When update is called, all **pending** handlers are run. This causes handler to run which again schedules another timer to expire immediately. Thus the event loop invocation runs that as well. On the other hand, the one second timer has not expired as yet so no more event handlers are pending and the event loop terminates and the update command returns. Thus we see Event 0 and Event 1 being printed and not Event 2. If you do run update (or the event loop by any other means) after a second has elapsed, you will see the one second timer handler being run as well.

If the idletasks argument is specified to the update command, only background tasks scheduled to run when the event loop is idle are processed.

```
after 0 [list puts "After 0"]
after idle [list puts "After idle"]
update idletasks
→ After idle
```

The after idle command registers a script to be run when the event loop is idle. Notice from the output that only the idle event handler was run. The timer event handler was not run even though it would have been pending. Moreover, the example shows that update idletasks forces any idle handler to be invoked **even if other events were pending** and the event loop was not actually idle.

⚠ Update considered harmful

The update command is often used to keep an application's user interface responsive during the course of a long computation. This is not considered good practice due to reentrancy and data consistency issues. The Tcler's Wiki[1] page http://wiki.tcl.tk/1255 discusses this in some detail.

In the author's experience, the only time an update has been mandated is for forcing window geometry calculation and propagation in the Tk GUI extension on some platforms that use native OS widgets.

15.2.4. Event handlers and the call stack

It is worthwhile taking a look at how the call stack (see Section 10.5) and the C stack (see Section 10.5.6) appear when an event handler is running. Figure 15.1 shows one such snapshot.

Figure 15.1. Call stack in an event handler

The above figure reflects the state of the two stacks when the handler procedure is running in the following code snippet.

```
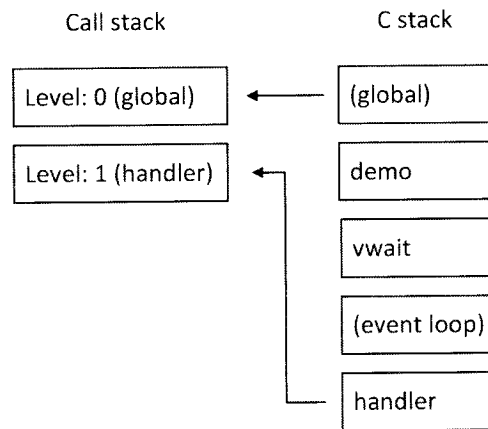proc handler {} {
    puts "handler level: [info level]"
    set ::done 1
}
proc demo {} {
    puts "demo level: [info level]"
    after 0 handler
    vwait ::done
}
demo
→ demo level: 1
  handler level: 1
```

---

[1] http://wiki.tcl.tk

Note the following points from the figure:

- From the perspective of the internal C stack, the call to vwait which in turn runs the event loop and executes the handler, all add a level. Further calls to vwait (or update which would behave similarly) from within the event handler would add more levels to the C stack.

- On the other hand, the call stack that maintains execution and variable contexts is reset to the global level. The output of our little script which showed both demo and handler executing at level 1 corroborates this. This illustrates that event handlers are run in the global context. You cannot expect to use upvar or uplevel to reach into the context of the demo procedure. Note that those contexts are not lost, they will be restored once the vwait command returns.

15.3. Scheduling execution of code: after

We now turn our attention to how we can add entries to the event and idle queues to be invoked via the event loop. There are many mechanisms through which this can happen as we will describe in later chapters. Here we only look at the simplest of these:

- Timer expiry events which can be used to run code after a specified interval
- Registering tasks to be run when the system is idle and no events are pending processing.

The after command is used for both purposes, and more.

> ⚠️ As of Tcl 8.6, one limitation of the after command to keep in mind is that it depends on the system clock for time-keeping. If the system clock is inaccurate or jumps for whatever reason, after will not correctly measure intervals. There is work in progress to fix this behaviour which will likely show up in future releases of Tcl.

15.3.1. Suspending execution

Most systems have a sleep call which effectively suspends the caller for a specified interval. The same can be done in Tcl with the after command.

```
after MILLISECS
```

This call will halt the execution of all code, including the event loop, in the current thread for the specified number of milliseconds. For example,

```
after 250
```

will put the current thread to sleep for a quarter of a second.

15.3.2. Scheduling code

The next form of the after command schedules code to run after a specified time interval has elapsed.

```
after MILLISECS SCRIPT ?SCRIPT …?
```

The command sets up a timer that will expire after *MILLISECS* milliseconds and returns right away. The result of the command is an identifier for the timer that can be passed to after cancel to cancel it if desired.

When the timer expires, an entry is added to the event queue with a handler that will invoke the script formed by concatenating the *SCRIPT* arguments separated by a space in the same manner as the concat or eval commands.

We have already seen examples of this command for setting up timers. Here is a slightly more realistic example that uses the http package to retrieve a Web page in conjunction with a timeout within which the transaction must complete.

```
package require http
proc http_data_sink {token} {
    set ::status done
}
proc geturl_with_timeout {url ms} {
    after $ms {set ::status timeout}
    set http_token [http::geturl $url -command http_data_sink]
    vwait ::status
    if {$::status eq "timeout"} {
        http::cleanup $http_token
        error "Operation timed out."
    }
    set data [http::data $http_token]
    http::cleanup $http_token
    return $data
}
```

Let us try to retrieve a Web page.

```
% geturl_with_timeout http://www.example.com 10000
→ <!doctype html>
  <html>
  <head>
...Additional lines omitted...
```

That works fine, but if we give it a short time to do its work,

```
% geturl_with_timeout http://www.example.com 10
0 Operation timed out.
```

the timer event fires first and the operation is timed out.

A special case of scheduling an task is when 0 is specified as the value for the *MILLISECS* argument. In this case the timer expires immediately and the handler script is appended to the event queue. This idiom is often used when the programmer wants some piece of code to run only after the current handler completes execution. Another use is to break up a long computation into smaller pieces while allowing other handlers to run. We will say more about this in Section 15.3.3.1.

The cron package from Tcllib[2] provides a layered interface to the after command that may be more convenient. For example, it permits absolute times to be specified for executing code and can schedule recurring events. It is also potentially more efficient when a large number of timer events are to be scheduled as it collapses multiple timers that expire at the same timer.

15.3.3. Running on idle: after idle

The after command can also be used to add background tasks to the idle task queue.

```
after idle SCRIPT ?SCRIPT …?
```

This command behaves almost identically to the after 0 form of the command except that the script formed by concatenating the *SCRIPT* arguments is added to the idle task queue instead of the event queue. Here is a short example that illustrates the relation between the two queues.

---

[2] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

```
% after idle [list puts "Idle task executed"]
→ after#11
% after 0 [list puts "Event handled"]
→ after#12
% update ❶
→ Event handled
  Idle task executed
```

❶   Run all pending event and idle tasks

Notice that the idle handler runs after the event handler even though it was queued first.

As before, the command returns an identifier that can be used to remove the task from the queue with the `after cancel` command.

15.3.3.1. Avoiding event queue starvation

Running a long computation prevents the event loop from running and responding to user input, network activity and so on. A common technique used to avoid this is to break up the computation into pieces and run each part in turn from the event loop.

We will illustrate with our prior example of summing the first N natural numbers. Our procedure will do one addition operation and then queue itself back on the event queue to continue with the next stage of computation.

```
proc background_sum {n {sum 0}} {
    if {$n <= 0} {
        puts "Sum is $sum"
    } else {
        puts "Calculating..."
        incr sum $n
        after 0 [list background_sum [incr n -1] $sum]
    }
}
```

Notice when we call the command nothing is printed because the computation happens via the event loop (this assumes you are running in `tclsh` and not `wish` where the event loop is already running):

```
% after 0 background_sum 2
→ after#13
```

We can run the event loop to finish the computation.

```
% update
→ Calculating...
  Calculating...
  Sum is 3
```

The `update` command keeps executing handlers as long as the event queue is not empty. Since we keep adding a timer (with a 0 expiration) the event loop will keep running until the computation is completed. Morever, while this computation is going on, other events that may arrive in the meanwhile will also be queued and executed in the order of arrival. The user interface will stay responsive, network connections will be accepted and so forth.

However, there is one flaw in the above. As long as there are entries in the event queue, their handlers will be executed and the **event loop will never move on to the idle task queue**. The idle task queue will be starved of any execution cycles and activities like updating of windows which happen at idle time will not happen.

To demonstrate this, let us write a script that will keep running in the background on the idle queue.

```
proc idler {{n 2}} {
    puts Idle!
    if {$n > 0} {
        after idle [list idler [incr n -1]]
    }
}
```

Now we fire up the two scripts.

```
% after idle idler
→ after#16
% after 0 background_sum 2
→ after#17
% update
→ Calculating...
  Calculating...
  Sum is 3
  Idle!
  Idle!
  Idle!
```

As shown by the output, the idle task does not get to run until the computation was done.

Queuing our computation on the idle task queue would not solve the problem either because once the event loop starts processing the idle task queue, it will continue to do so until it is empty. We will therefore starve the event queue instead.

The solution to this is to modify our procedures as follows

```
proc idler {{n 2}} {
    puts Idle!
    if {$n > 0} {
        after 0 [list after idle [list idler [incr n -1]]] ❶
    }
}
proc background_sum {n {sum 0}} {
    if {$n <= 0} {
        puts "Sum is $sum"
    } else {
        puts "Calculating..."
        incr sum $n
        after idle [list after 0 [list background_sum [incr n -1] $sum]] ❷
    }
}
```

❶ Modified line
❷ Modified line

Now if we run our previous code, the script output is interleaved indicating neither queue is starved.

```
% after idle idler
→ after#22
% after 0 background_sum 2
→ after#23
% update
→ Calculating...
  Idle!
  Calculating...
  Idle!
```

```
Sum is 3
Idle!
```

In essence, instead of rescheduling itself directly, the code "bounces" itself off the other queue ensuring entries on that queue get to run as well.

> This technique applies to computations that directly or indirectly reschedule themselves continuously. If **new** independent events are arriving at a rate faster than the rate at which the queues can be emptied, this would obviously not help.

15.3.4. Cancelling tasks: `after cancel`

Any timer events and idle tasks that have been scheduled with the `after` command can be cancelled with `after cancel` which can take one of two forms:

```
after cancel ID
after cancel SCRIPT ?SCRIPT …?
```

In the first form, the command's argument is an identifier returned with the `after` or `after idle` commands. The corresponding event is cancelled.

```
set id1 [after 0 puts Timer1]  → after#32
set id2 [after 0 puts Timer2]  → after#33
after cancel $id1              → (empty)
update                         → Timer2
```

Only the second timer fires as the first one has been cancelled.

In the second form, instead of specifying the timer identifier, the caller can specify the actual script that was scheduled. Repeating the above example but using the idle task queue and the second form of the command,

```
after idle puts Timer1    → after#34
after idle puts Timer2    → after#35
after cancel puts Timer1  → (empty)
update                    → Timer2
```

Again, only the second timer fires as the first was cancelled.

Note that the command does not raise an error if no matching timer is found in either case.

15.3.5. Querying `after` handlers

The `after info` command can be used to query the current event handlers registered with the `after` commands.

```
after info ?ID?
```

If no `ID` argument is specified, the command returns a list of identifiers of the currently active handlers.

```
after 1000 {puts "Timer"} → after#36
after idle {puts "Idle"}  → after#37
after info                → after#37 after#36
```

If `ID` is specified, it returns a pair whose first element is the associated handler script and second is either `timer` or `idle` depending its type.

```
% foreach id [after info] {
    lassign [after info $id] script type
    puts "$id ($type): $script"
    after cancel $id
}
→ after#37 (idle): puts "Idle"
  after#36 (timer): puts "Timer"
```

Note that timers that have already been triggered or that have been canceled do not show up in the results returned by `after info`.

15.4. Event loop error handling

When an error exception is raised it is propagated up the call stack as described in Section 11.2.2 until it is handled at some call level. If it reaches all the way to the global level in a Tcl application that is not event-driven, the result is application dependent. The `tclsh` shell in interactive mode will trap the error and display it to the user. In non-interactive mode, `tclsh` will by print the error to standard output and exit.

Things work differently with event driven applications. If an error is raised during the execution of an event (or idle task) handler and propagates up to the event loop, it is reported through a background exception handler. The event loop invokes this handler with two additional arguments: the interpreter result and a dictionary of return options. These are exactly the same values that are captured by a `catch` command and are described in Section 11.4.1 and Section 11.4.2.

The `tclsh` and `wish` shells provide their own default background exception handlers which display the error message on the `stderr` channel and through a window dialog respectively.

Here is a demonstration of the `tclsh` default background error handler.

```
% proc demo {arg} {}
% after 0 demo ❶
→ after#2
% catch update
→ wrong # args: should be "demo arg"
      while executing
  "demo"
      ("after" script)
  0
%
```

❶   Intentionally call demo with wrong number of arguments

Notice that `catch` command returned a 0 indicating the `update` command ran without errors. The generated error exception was handled by the event loop and not propagated to `update`. The default background exception handler invoked by the event loop printed the error stack to the standard error channel.

15.4.1. Custom background error handling: `interp bgerror`

An application can choose to customise default handling of background exceptions by calling `interp bgerror`.

```
interp bgerror INTERPRETER CMDPREFIX
```

The handling of background errors can be customized on a per-interpreter basis and the *INTERPRETER* argument specifies the path of the interpreter to be customized. We will take a deeper look at interpreter paths in Chapter 20 but for the moment we just mention that the empty string refers to the current interpreter.

The *CMDPREFIX* argument is a command prefix that will be called with two additional arguments, the error result and return options dictionary as earlier described.

To illustrate, let us define our own background exception handler for our above example.

```
% proc bghandler {message ropts} { puts stderr "MyApp error: $message" }
% interp bgerror {} bghandler
→ bghandler
% after 0 demo
→ after#2
% catch update
→ MyApp error: wrong # args: should be "demo arg"
  0
%
```

Our error handler does essentially the same thing as the default one except adding an application name and only printing the error message instead of the whole stack. In a real application, you might choose to log the error to a file or some other more sophisticated action.

> In old versions of Tcl, the global bgerror command was used for customizing background error handling. This is now deprecated in favour of interp bgerror.

15.5. Chapter summary

In this chapter we have introduced the event loop which forms the basis of asynchronous programming facilities required for almost any long running application. We also looked at the basic use of the event loop to run scheduled and background tasks and described how error exceptions in these are managed. In subsequent chapters, we will examine the use of the event loop for advanced I/O, networking and interprocess communications.

16

Processes and Pipelines

The most basic form of integration with other programs and applications is the exchange of data by coupling the standard inputs and outputs of processes. Tcl provides two commands for this purpose.

- The exec command, starts one or more child processes and returns any content written to their standard output as the result of the command.
- The open command on the other hand provides more flexibility by returning a channel that can be used to communicate with child process(es).

Both commands also support *process pipelines* wherein multiple processes are chained via their standard input and output.

16.1. Executing child processes: exec

The exec command starts a *pipeline* of one or more processes.

```
exec ?-keepnewline? ?-ignorestderr? ?--? ARG ?ARG …? ?&?
```

Here the ARG arguments specify one or more programs to run along with their parameters or special character sequences that separate the programs and indicate redirection of input and output. If the last argument is not the & character, the command result is the data written to standard output by the last process in the pipeline. Any trailing newline character in the output is discarded unless the -keepnewline option is specified. The significance of the & character and the -ignorestderr option are discussed later. As always, -- can be used to indicate the end of options.

In the simplest case, the command starts a single process and returns the data written to its standard output as the result of the command. For example, here we run the netstat program and collect its output.

```
% set connections [exec netstat -n]
→
  Active Connections

    Proto  Local Address          Foreign Address        State
    TCP    192.168.1.128:53143    40.100.136.18:443      ESTABLISHED
...Additional lines omitted...
```

In the general form, the arguments can specify multiple programs comprising a process pipeline where each program and its parameters is separated from the other programs by a | or |& character sequence. In the former case, the standard output of the preceding process in the pipeline is fed into the standard input of the next process. In the latter case, both standard output and standard error of the preceding process are piped into the standard input of the next.

In the absence of any I/O redirection or errors, the output of the last process in the pipeline is returned as the result of the exec command.

Here is a pipeline where we filter the output of netstat through the findstr program on Windows to only retrieve UDP connections.

```
% set udp_connections [exec netstat -an | findstr UDP]
→   UDP     0.0.0.0:3544          *:*
    UDP     0.0.0.0:3702          *:*
    UDP     0.0.0.0:3702          *:*
...Additional lines omitted...
```

16.1.1. Passing program arguments

When passing argument values to the executed programs, keep in mind that the arguments first undergo substitutions as per Tcl's quoting rules (see Section 3.2). Depending on the program being executed, they may then be subject to that program's quoting rules which in all likelihood differ from those of Tcl. Thus care must be taken to appropriately escape program arguments when special characters are part of the passed arguments. Unfortunately, different programs follow different conventions, particularly on Windows, so the escaping of special characters is necessarily program-specific.

The same also applies to file paths that may be passed to the child processes. On Windows for example, many programs do not accept / as a path separator. Thus attempting to produce a directory listing of the Tcl installation binary directory with Windows command shell's dir internal command will fail.

```
% exec cmd /c dir [file dirname [info nameof]]
0 Parameter format not correct - "l".
```

The passed file path must be transformed into native form with the file nativename command.

```
% exec cmd /c dir [file nativename [file dirname [info nameof]]]
→   Volume in drive C is OS
    Volume Serial Number is E8C6-0D60

    Directory of c:\tcl\866\x64\bin

  03/30/2017  07:28 PM    <DIR>          .
  03/30/2017  07:28 PM    <DIR>          ..
...Additional lines omitted...
```

> This transformation of file paths is only required for program arguments. If a directory path is specified for the program name itself, Tcl will automatically convert it to the native form. No explicit conversion is necessary.

Also to be noted is that Tcl's exec behaviour should not to be confused with that of Unix shells. The latter implicitly do glob-style expansion of wildcard patterns in arguments to programs whereas Tcl's exec command does not. Thus the following command executed in a Unix shell:

```
ls *.c
```

would not be

```
exec ls *.c
```

in Tcl but rather

```
exec ls {*}[glob *.c]
```

16.1.2. Locating programs

The program to be executed for each stage of the exec pipeline may be an executable image or a shell script (on Unix) or a batch file (on Windows). It may be specified as an absolute path, a relative path or just a file name with no directory component. The path will undergo tilde substitution (see Section 9.1.1.3) as appropriate.

If the program is specified purely by name (i.e. no directory components are present), exec will look for the file in an operating system dependent fashion. On Unix, it looks for the file in directories specified in the PATH environment variable. On Windows, the command will look in the directory containing the Tcl application, the current directory, the Windows system directories and finally the directories in the PATH environment variable.

Additionally, on Windows platforms if the search fails and no extension was specified in the program path, the command will repeat the search by appending .com, .exe, .bat and .cmd to the file name.

16.1.2.1. Locating internal commands: auto_execok

Some "programs" that are executed in command shells are not separate executables at all but are actually implemented internal to the shell. For example, the DIR command for listing directories at the Windows command prompt is internal to the Windows cmd.exe shell. Attempting to run this directly from Tcl will raise an error.

```
% exec dir *.*
→ couldn't execute "dir": no such file or directory
```

Tcl provides a command, auto_execok, for dealing with such commonly used commands that are built into the operating system command shells.

```
auto_execok PROGRAMNAME
```

The command returns a list of words to be passed to exec to run that program.

```
% auto_execok notepad
→ C:/WINDOWS/system32/notepad.EXE
% auto_execok dir
→ C:/Windows/System32/cmd.exe /c dir
```

Notice the difference in the two outputs. Since dir is an internal command of cmd.exe, auto_exec returns the command prefix to be used to invoke it. We can then run it as

```
% exec {*}[auto_execok dir] *.*
→   Volume in drive C is OS
    Volume Serial Number is E8C6-0D60

    Directory of C:\temp\book

...Additional lines omitted...
```

16.1.3. Redirecting I/O

By default, the output of each process in the pipeline is supplied as the input to the next process. This behaviour can be changed for each process in the pipeline for both input and output through special character sequences in the arguments to exec. This I/O redirection takes a form very similar to that used in Unix or Windows command shells.

16.1.3.1. Redirecting input

By default, the **first** process in an exec pipeline reads its standard input from the standard output of the parent Tcl application that invoked the exec command. This behaviour can be changed so that the process gets its input

- from a file
- from an open channel in the Tcl application
- from a value in the Tcl application

We will demonstrate all three techniques by recursively invoking tclsh as a separate process and printing its PID.

On Windows platforms, the examples will not work with the wish shell because GUI programs on Windows do not have a real operating system provided standard input or output.

Redirecting input from a file

To have the first process read its standard input from a file, prefix the file path with a < character. Let us first write out our sample file that we use as input.

```
set chan [file tempfile temppath] ❶
puts $chan { puts "My PID is [pid]" }
close $chan
```

❶   See Section 9.2.10

Now to recursively invoke ourselves with standard input for the child process redirected to this file, pass the file path prefixed with < to the exec command.

```
% exec [info nameofexecutable] <$temppath
→ My PID is 10328
```

Note that you can optionally separate the file path from the < character with whitespace so we could also have written the above as

```
% exec [info nameofexecutable] < $temppath
→ My PID is 4056
```

(Note the space character before the temppath reference.)

Redirecting input from a channel

As an alternative to redirecting input from a file, you can redirect input from a channel that is already open by prefixing the channel with the <@ character sequence. For example, a variation of the above example:

```
% set chan [open $temppath r]
→ file3787000
% exec [info nameofexecutable] <@$chan
→ My PID is 1732
% close $chan
```

As shown in the example, the channel must have been opened for reading for channel based redirection to work. As before, the child process will exit when it encounters an EOF on the input channel.

Not all channel types are supported for use with input redirection. In particular, on Windows platforms network socket based channels cannot be used unlike on Unix. Moreover, on all platforms reflected channels (see Section 17.3) also do not work with input redirection.

You have to keep two factors in mind when using the channel based input redirection. The first is that the file access pointer is shared between the parent and the child so the following sequence of commands will not yield the desired result.

```
set fd [open foo.txt w+]           → file3037340
puts $fd {puts "My PID is [pid]"}  → (empty)
exec [info nameofexecutable] <@$fd → (empty)
close $fd                          → (empty)
```

The reason why this does not work is that after the write to the channel, the file access pointer is positioned at the end. Consequently, when the child process reads from the channel it only sees the end-of-file marker and exits. This problem can be fixed by inserting a seek to reposition the access pointer to the front of the file.

```
set fd [open foo.txt w+]           → file3787000
puts $fd {puts "My PID is [pid]"}  → (empty)
chan seek $fd 0 start              → (empty)
exec [info nameofexecutable] <@$fd → My PID is 3220
close $fd                          → (empty)
```

Now the child process reads the file as desired.

The other point to remember is the potential for race conditions. If you change the order of operations to invoke the exec before the puts, it is possible that the child process will run before the write to the file, find it empty and exit.

For these reasons, the channel redirection mechanism is not recommended as a means for continuous communication between the parent and the child. We will see alternative means described in Section 16.2 and Section 16.3 that are more suitable for such scenarios.

Redirecting input from a Tcl value

The final option available for redirecting the standard input of the spawned process is through the << redirection operator. Instead of a file path or a channel, this redirects input from the specified value. Our example could be written as

```
% exec [info nameofexecutable] << {puts "My PID is [pid]"}
→ My PID is 4620
```

Tcl arranges for the argument following the << to be passed in to the child process in its standard input. We have specified the value in our example as a braced string literal. Of course it could also have been the result of a command or a variable reference as well.

16.1.3.2. Redirecting output

Just as for the input side, standard output and error of processes in an exec pipeline can also be redirected. There are more combinations possible here which can be confusing so we first lay out elements that are common to all.

- Only the standard output of the **last** process in the pipeline can be redirected. Other processes always send their output to the next process.
- On the other hand, the redirection of standard error output applies to **all** processes. It is not possible to only redirect it for any single process (not even the last in the pipeline).
- The single > character sequence always indicates writing to the channel at its current file access position. The double >> sequence on the other hand always appends.
- A prefix of 2 before any of the above character sequences indicates the redirection only applies to standard error and not to standard output.
- A suffix of & indicates the redirection applies to both standard output and standard error.
- The @ character specifies redirection to an open channel in the Tcl application calling the exec. The channel must have been opened for writing.

- If output is redirected, the result of the exec command is the empty string.

Sending standard output to a file

The > character followed by a file path will write the standard output of the last process in the pipeline to that file, overwriting it if it already exists.

```
exec netstat -an | findstr ESTABLISHED > connections.log
```

Note that the result of the exec command above is the empty string due to the redirection.

The << redirection works similarly except that it will append to the file instead of overwriting it.

Sending standard error to a file

The 2> character sequence followed by a file path will write the standard error of all processes in the pipeline to that file, overwriting it if it already exists. The standard outputs are unaffected. In the example below we spawn off another Tcl interpreter and have it print messages to standard output and error.

```
% exec [info nameofexecutable] 2> error.log << {
    puts stdout "This is standard output"
    puts stderr "This is standard error"
}
→ This is standard output
% print_file error.log
→ This is standard error
```

As seen above, only the standard error is redirected to the file while the standard output appears as the result of the exec command.

The standard error output can be appended to the file instead of overwriting it by using the 2>> redirection instead.

> Redirecting the standard error modifies the error handling behaviour of exec. This is discussed in Section 16.1.4.

Sending output to a channel

Instead of sending standard output and error to a file, it can be sent to an open channel in the current Tcl interpreter by using >@ and 2>@ respectively.

```
% set chan [file tempfile temppath]
→ file420a2d0
% exec {*}[auto_execok date] /t >@ $chan
% close $chan
% print_file $temppath
→ Tue 07/04/2017
```

Note that there are no equivalent redirections that append to channels.

Conflating standard output and error

The redirections discussed so far have dealt with independently redirecting standard output and error. For example, one might write

```
exec grep Tcl_.*Init {*}[glob *.c] > matches.txt 2> errors.txt
```

In cases where you want both standard output and error to go to the **same** destination, append a & character to the appropriate operator. Thus >& and >>& will overwrite or append the standard output of the **last** process and the standard error of **all** processes in the pipeline to the specified file.

```
exec [info nameofexecutable] >& output.log << {
    puts stdout "This is standard output"
    puts stderr "This is standard error"
}
print_file output.log
→ This is standard output
  This is standard error
```

The >&@ works similarly except it writes to an open channel.

> ⚠ The use of the >& as above is **not** the same as separately redirecting the standard output and error to the same file as below.
>
> ```
> exec [info nameofexecutable] > output.log 2> output.log << {
>     puts stdout "This is standard output"
>     puts stderr "This is standard error"
> }
> print_file output.log
> → This is standard error
> ```
>
> As you see from the above, the output from one can overwrite or mangle output from the other.

One final form, 2>@1, redirects standard error to be included as part of the command result. Since the standard output is already included in the result, the following will return both as the result of the exec command.

```
set result [exec [info nameofexecutable] << {
    puts stdout "This is standard output"
    puts stderr "This is standard error"
} 2>@1]
puts $result
→ This is standard output
  This is standard error
```

16.1.4. Error handling in exec

Error handling for exec invocation is complicated by the fact that the error may come from different sources:

- The exec command itself may raise an error if one of the programs in the pipeline is not found or does not have execute permission for the user and so on.
- The program(s) run but terminate with some error condition.

Moreover, the executed programs may signal error conditions or abnormal termination in different ways:

- The process may exit with a non-0 exit code.
- The process may write to its standard error output.

The application needs to be able to recognize and distinguish all these different forms. A further complication is that not all programs follow the above conventions. For example, a search application may use the exit code as a result value returning the number of matches. Others may use standard error to record progress messages, not necessarily errors. For these reasons, error detection is very much dependent on the program(s) being executed. The discussion below is focused on distinguishing the various cases. Interpretation as errors or normal behaviour is up to the application.

We can interactively explore various scenarios by spawning a Tcl process that imitates application behaviour. For starters we assume that standard error has **not** been redirected as that affects error handling.

First, consider an attempt to execute a program that does not exist.

```
% catch {exec nosuchprogram} result ropts
→ 1
% dict get $ropts -errorcode
→ POSIX ENOENT {no such file or directory}
```

As expected, the catch command result indicates an error. The -errorcode element of the return options dictionary gives the details of the failure. Other error codes are also possible, such as insufficient permissions. These are generally returned as POSIX error codes.

Another possibility is that the program starts up but suffers abnormal termination via a segment violation, signal etc. In this case the error code will be of the form

```
CHILDKILLED PID SIGNALNAME MESSAGE
```

where *PID* is the process identifier of the terminated child process, *SIGNALNAME* indicates the signal (SIGTERM, SIGSEGV etc.) that forced the termination and *MESSAGE* is the human readable description of the reason for termination. For example, a null pointer access in the child process would result in an error code of

```
CHILDKILLED 12408 SIGSEGV {segmentation violation}
```

Finally, there is the possibility of the program itself signalling an error. Tcl considers a child process exiting with a non-0 exit code **or** writing to its standard error output to be an error. We can simulate both these conditions by recursively invoking the Tcl shell.

```
catch {
    exec [info nameofexecutable] << {
        puts "This is standard output"
        exit 3
    }
} result ropts
→ 1
```

The catch command returns an error because child exited with a non-0 exit code (3 in this case). The error code from the return options dictionary also includes this exit code.

```
puts [dict get $ropts -errorcode] → CHILDSTATUS 10092 3
```

Just as for the normal completion of an executed program, the result includes its standard output. In addition, the error message is also appended to this result.

```
% puts $result
→ This is standard output
  child process exited abnormally
```

The other situation considered as an error is if the child writes to its standard error. This is simulated by the following snippet.

```
catch {
    exec [info nameofexecutable] << {
        puts "This is standard output"
        puts stderr "This is standard error"
```

```
        puts "This is standard output again"
        exit 0
    }
} result ropts
→ 1
```

Again, the `catch` command result indicates an error exception even though the child process exited with an exit code of 0. This is because it wrote to its standard error. The error code however shows up as NONE.

```
puts [dict get $ropts -errorcode] → NONE
```

The result of the exception includes the standard output of the child followed by its standard error content. Note the latter always appears at the end no matter the order in which `puts` statements were executed.

```
% puts $result
→ This is standard output
  This is standard output again
  This is standard error
```

If the `-ignorestderr` option is specified, `exec` does not treat any output to standard error as an error condition.

```
% catch {
    exec -ignorestderr -- [info nameofexecutable] << {
        puts "This is standard output"
        puts stderr "This is standard error"
    }
} result ropts
→ 0
% puts $result
→ This is standard output
```

As before, the result includes both standard output and standard error but now the command does not raise an error exception as evinced by the 0 result of the `catch` command above.

Another way to accomplish the same thing is to redirect the standard error using any of the error redirectors like 2> etc.

The following pseudocode template summarizes handling of all these various cases using the `try` command described in Section 11.4.3. Depending on the program being executed, the application can take appropriate action depending on whether the condition signifies a real error or not. Errors for which trap clauses are not specified will be automatically propagated.

```
try {
    set result [exec command parameters ...]
} trap NONE output {
    # Application exited with a 0 exit code but wrote to standard error.
    # The variable output will contain the standard output content
    # followed by the standard error content
    ...do whatever...
} trap CHILDSTATUS {- ropts} {
    # Child exited with a non-0 exit code
    # Retrieve the PID and exit code
    lassign [dict get $ropts -errorcode] -> pid exit_code
    ...do whatever...
} trap CHILDKILLED {- ropts} {
    # Child terminated abnormally
    # Retrieve the PID, signal and message
    lassign [dict get $ropts -errorcode] -> pid signal reason
    ...do whatever...
```

```
} trap CHILDSUSP {_ ropts} {
    # Child suspended
    # Retrieve the PID, signal and message
    lassign [dict get $ropts -errorcode] -> pid signal reason
    ...do whatever...
} trap POSIX {- ropts} {
    # Other errors like permissions, file not existing etc.
    # Retrieve POSIX error mnemonic and reason
    lassign [dict get $ropts -errorcode] -> posix_code reason
    ...do whatever...
}
```

16.1.5. Running background processes

Normally the exec command waits for all processes in the pipeline to terminate and returns as its result the standard output of the last process in the pipeline. However, if the last argument to exec is &, the command returns immediately running the processes in the background. The return value is a list containing the process identifier (PID) of each process in the created pipeline.

In the following example, the exec command returns immediately without waiting for the netstat and findstr processes to finish executing.

```
% exec netstat -an | findstr ESTABLISHED > connections.log &
→ 9300 11952
```

The list of PID's returned is in the order of processes in the pipeline.

Tcl does not provide any built-in commands dealing with process monitoring and management. However, the cross-platform processman module in Tcllib[1] as well as the Windows-specific process module in TWAPI[2] provide commands for checking for process existence, terminating processes and so on. Our example above could be modified as follows to let us know when the background processes were done without blocking the exec command itself.

```
package require processman
set pids [exec netstat -an | findstr ESTABLISHED > connections.log &]
::processman::onexit [lindex $pids 1] { puts "Background processing done" }
```

Note that the above assumes the Tcl event loop is running.

If no I/O redirection is in effect, the standard output and standard error of the last process in the pipeline will go to the application's standard output and error.

16.1.6. Limitations in exec

There are certain limitations in exec for some scenarios:

- It does not provide for an "interactive" bidirectional data exchange with the child process. For example, we cannot use it fire up another copy of our Tcl applications and feed it a command, get back the result and then repeat that sequence.
- There is no control over encodings, line translations etc. with respect to data written to and read from the child process.
- Its syntax makes it not just difficult but impossible to pass arguments that match certain character sequences such as those used for redirection.

---

[1] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html
[2] http://twapi.sf.net

- It has certain platform-specific limitations such as not being able to execute child process with elevated privileges on Windows.

The first two of these are addressed in the next section and a proposal and accompanying implementation to fix the third in the next release of Tcl is already in place as Tcl Improvement Proposal #424 available at http://www.tcl.tk/cgi-bin/tct/tip/424.html.

To overcome the last limitation, you will need the help of third party extensions. For example, the TWAPI [3] extension has commands that offer these capabilities for the Windows platform.

16.2. Channels for process pipelines: open

We introduced the open command in Chapter 9 where we used it to create an I/O channel to a file on disk. The command is in fact more general in that it can be used to create channels of different types. Here we examine its use for creating channels to process pipelines.

Use of open to create channels for process pipelines has several advantages over exec at the cost of some slight complexity:

- It allows for arbitrary sequences of bidirectional data exchange with the child process.
- The data exchange can be asynchronous, a capability we will describe in Section 17.1.
- The encodings used for the data exchange can be controlled by appropriately configuring the channel.
- Being a channel, we are are able to use all capabilities of Tcl channels including applying channel transforms as described in Section 17.2. For example, we could transparently compress the data we are piping into the process pipeline.

A channel to a process pipeline is opened using the same syntax we saw in Section 9.3.2.

```
open PATH ?ACCESS? ?PERMISSIONS?
```

The *ACCESS* and *PERMISSIONS* arguments are the same as described there for file channels. The *PATH* argument on the other hand must begin with the | character. The rest of the *PATH* argument is treated in the same fashion as the arguments to the exec command.

Used in this form, open returns a channel that may be used to write to the standard input of the first process in the pipeline or read from the standard output of the last process (assuming no redirections are in effect). The returned channel must as always be released with the close command when done.

If the channel to a process pipeline is in blocking mode, the close will not return until all processes in the pipeline have ended.

The operations permitted on the returned channel depend on the *ACCESS* argument as shown in Table 16.1. The descriptions in the table assume that no redirections are in effect. For example, if the output of the pipeline is redirected to a file, no data will be read from the channel. Some illustrative examples follow the table.

---

[3] http://twapi.sf.net

Table 16.1. Access mode for pipelines using open

Mode	Description
r, rb	The channel is opened as read-only in text and binary modes respectively. The standard input of the first process in the pipeline is taken from the standard input of the current process. The standard output of the last process in the pipeline can be read from the created channel.
r+, rb+, r+b, w+, wb +, w+b	The channel is opened for reading and writing in text and binary mode respectively. Any writes to the channel will be fed into the standard input of the first process in the pipeline. The standard output of the last process in the pipeline can be read from the created channel.
w, wb	The channel is opened only for writing in text and binary mode respectively. Any writes to the channel will be fed into the standard input of the first process in the pipeline. The channel cannot be read and any output from the last process in the pipeline will go to the current standard output unless any redirection is in place.

A read-only pipe

Our first example is a variation of one of the exec based ones we saw earlier. We list network connections using the netstat program and pipe the output to findstr to filter them. Since we never need to pass any data **to** the process pipeline, we can open the channel in read-only mode.

```
% set chan [open "|netstat -an | findstr ESTABLISHED" r]
→ file3a9b030
% while {[gets $chan line] >= 0} {
    puts $line
}
→    TCP      192.168.1.128:53143     40.100.136.18:443       ESTABLISHED
     TCP      192.168.1.128:53226     40.100.138.18:443       ESTABLISHED
...Additional lines omitted...
% close $chan
```

Having a channel in hand, we can process the output data a line at a time should we choose unlike for exec where we got all the data in one lump. For our example, this may not matter much since the output data is limited in size and easy enough to split into lines. But in the general case, where the data is either very large or an continuous stream (we will see an example of this later), exec is not a viable option.

A write-only pipe

Our second example involves a write-only pipe where we will write to the gzip program to compress data that we will generate incrementally.

```
% set chan [open "|gzip - > foo.zip" w] ❶
→ file7
% chan configure $chan -encoding utf-8 -translation lf
% puts $chan "Line 1"
% puts $chan "Line 2"
% close $chan
```

❶ Specifying - as the input file causes gzip to read data from its standard input

Several additional points are illustrated by this example:

- Redirection operators like > above can be used with open in the same fashion as with exec.
- We can call chan configure to set various options on the channel. In this case, because most compression programs expect binary data, we configure the channel to transform our Unicode strings to binary data as

described in Section 9.3.8. Without this, the channel would use system encoding which may or may not be able to handle all characters.

- We do not need to collect all data and feed it to gzip in one step as for exec. We can write it piecemeal as it is generated.

- We need to close the channel when done. Otherwise, not only will we have a resource leak with the channel handle, it will also cause gzip will hang around waiting for more data.

16.2.1. Running tclsh in a pipeline

It is sometimes useful in real world applications to "drive" tclsh in a pipeline for executing ancillary tasks, parallelizing computation and so on. We demonstrate such usage here.

Additionally, this will also serve as an example of

- Using a bi-directional pipe where the application both writes to and reads from the child process.
- Using the list command to construct the program and argument parameters.
- Additional channel configuration that must be set up for some applications.

> If on Windows, the code below must be run from tclsh or some other Tcl console application, not from wish or a GUI based one as the latter do not have standard input/output.

Our opening command itself looks different from what we have seen earlier.

```
% set chan [open |[list [info nameofexecutable] -encoding utf-8] r+]
→ file3ab1570
```

The r+ argument opens the pipe for both reading and writing. The -encoding option to tclsh informs that it should expect UTF-8 encoded data in its standard input. The list command is used to correctly form the arguments to the open command.

Consider if we had written the command as

```
set chan [open "|[info nameofexecutable] -encoding utf-8" r+]
```

Now, if our tclsh was installed in a directory with spaces in its path, say under Program Files on Windows, Tcl will attempt to execute the following after command substitution.

```
set chan [open "|C:/Program Files/Tcl/bin/tclsh.exe -encoding utf-8" r+]
```

This will fail as space in the path will cause the open command to treat C:/Program as the name of the program to run. Although some combination of quoting and escapes would also work, it is generally simpler and less error prone to use list to correctly form the arguments when command or variable substitution are involved as in our example.

As an aside, we could have placed the first argument in quotes

```
set chan [open "|[list [info nameofexecutable] -encoding utf-8]" r+]
```

but that is not necessary as long there is no whitespace between the | and [characters.

The next thing you need to be aware of relates to buffering in the channel (see Section 9.3.5.1). By default, the created channel is fully buffered as we can verify:

```
chan configure $chan -buffering → full
```

Fully buffered channels offer highest performance but are not convenient in scenarios like ours where commands we write to our child tclsh process need to be immediately sent across. We could do this by explicitly calling chan flush after every write but it is easier to just set the channel to be line buffered instead. At the same time we also set our channel to use UTF-8 encoding to match the encoding our child tclsh is expecting.

```
% chan configure $chan -buffering line -encoding utf-8
```

The child tclsh process is now running and since it was not passed a script file on the command line, it will loop reading commands from its standard input, i.e. the pipe, and executing them.

First, let us configure its buffering for the same reason listed above for our side of pipe. We write the appropriate command to the pipe.

```
% puts $chan {chan configure stdout -buffering line}
```

The child tclsh will then read the command and set its stdout channel configuration appropriately.

At this point, we list two important distinctions between running tclsh in a pipeline versus running it interactively.

- In interactive mode, tclsh will write a prompt to the standard output when it is ready for the next command. It does not do this when reading from a pipe.
- Secondly, unlike in interactive mode, the result of the evaluated command is **not** written to standard output.

We can check whether the child tclsh thinks it is in interactive mode by asking it to print the value of the tcl_interactive global variable in the child process. This value can then be read back from our end of the pipe.

```
% puts $chan {puts $tcl_interactive}
% gets $chan
→ 0
```

We now know that the child is non-interactive mode. We do not therefore have to worry about dealing with the tclsh prompt characters being read from the pipe and having to separate them from the actual data.

At the same time, we need to be aware that in non-interactive mode tclsh will not print the result of evaluated commands to its standard output. So if, instead of forcing an explicit write using puts as above, we had invoked the following commands

```
puts $chan {set tcl_interactive}
gets $chan
```

our shell would have appeared to hang. The child tclsh does not write the result of the set to its standard output. Consequently, our gets command would sit there waiting forever (since we have not discussed non-blocking I/O as yet).

If for some reason, you want the child tclsh to output prompts and display the result of evaluated commands, you can set the value of the tcl_interactive variable to 1 (obviously in the **child** tclsh, not in our parent shell).

When we are done with the child tclsh, we can either send it an exit command or simply close our end of the pipe which will cause it to exit. Here we will explicitly ask it to exit.

```
% puts $chan {exit}
```

Now our attempt to read from the pipe returns an empty string and eof indicates an end of file on the channel at which point we can close it.

```
gets $chan   → (empty)
eof $chan    → 1
close $chan → (empty)
```

16.2.2. Pipeline process ids: pid

The process identifiers of all processes present in a pipeline associated with a channel can be retrieved with the pid command.

```
pid CHANNEL
```

The command returns the list of PIDs for the processes in the pipeline and an empty list if the channel specified is not a process pipeline.

```
% set chan [open "|netstat -an | findstr ESTABLISHED" r]
→ file4154cd0
% pid $chan
→ 7896 9392
```

16.2.3. Error handling in pipelines

If any of the processes running in the pipeline signal an error either with the exit status or by writing to their standard error as described in Section 16.1.4, the close on the pipeline channel will throw an exception.

```
% set chan [open |[list [info nameofexecutable] -encoding utf-8] r+]
→ file4154cd0
% chan configure $chan -buffering line -encoding utf-8
% puts $chan {exit 2} ❶
% close $chan
∅ child process exited abnormally
% puts $::errorCode
→ CHILDSTATUS 11792 2
```

❶   Force child to exit with an error code

As displayed above, the error code corresponds to the child exiting with a non-0 status.

16.3. Standalone pipes: chan pipe

In the previous sections, we have seen various ways of creating and communicating with child processes through pipes and redirections. However there are some scenarios that are at best awkward to program for using these means.

One of these involves reading the standard output and standard error of a child process as separate data streams. This is not possible with the redirection forms we have seen where we can at most redirect the standard error into standard output and then somehow separate out the two, which may or may not be possible. Alternatively, standard error may be redirected to a file but the semantics of a file are very different when it comes to EOF and other aspects.

The chan pipe command provides a solution.

```
chan pipe
```

The command creates an operating system pipe and returns a list containing two channels, the first for reading from the pipe, and the second for writing to it. Any data written to the second channel will be read from the first.

Let us interactively observe how the channels work. First we create the pipe and assign the read and write channels to rchan and wchan respectively.

```
% lassign [chan pipe] rchan wchan
```

We will be sending simple text across so we configure them to be line buffered so writing a line will immediately flush the pipe making the line available on the read side.

```
% chan configure $rchan -buffering line
% chan configure $wchan -buffering line
```

Now we write data into the pipe via the write-side channel. As expected, it can be read from the read-side channel.

```
% puts $wchan "Testing 1 2 3..."
% gets $rchan
→ Testing 1 2 3...
```

Trying to write to the read-side channel is an error as is reading from the write-side channel.

```
% puts $rchan "Fail"
Ø channel "file41545d0" wasn't opened for writing
% gets $wchan
Ø channel "file3ba88d0" wasn't opened for reading
```

Closing the write side channel will be detected as end-of-file on the read side.

```
% close $wchan
% gets $rchan ❶
% chan eof $rchan
→ 1
% close $rchan
```

❶   Empty string returned because of EOF

Separating standard output and error

Having seen the basic working of a pipe, let us see how it might serve our purpose. As in previous examples, we will spawn a second copy of our Tcl shell as the child process.

Again, we start by creating a pipe.

```
% lassign [chan pipe] rchan wchan
```

Then as before, we use redirection into a channel to have the child process write its standard error to this pipe (see Section 16.1.3.2).

```
% set chan [open |[list [info nameofexecutable] 2>@ $wchan << {
    puts "This is standard output"
    puts stderr "This is standard error"
}]]
→ file41563d0
% close $wchan
```

The channel returned by open will be attached to the child's standard output. The write side of our pipe will be attached to the child's standard error through the 2>@ redirection.

Notice we then immediately close the write-side in our application. This is important because otherwise the write-side pipe file descriptor will have two references - one held by us and the other by the child process. When the child closes its end or exits, we will not see EOF on our read-side channel because the write-side will still be open due to the reference we hold. By closing it, we ensure that we see the EOF when the child closes the write side of the pipe.

Now we can independently read the child standard output and error.

```
gets $chan    → This is standard output
gets $rchan   → This is standard error
close $chan   → (empty)
close $rchan  → (empty)
```

Filter programs

Another use of channel pipes arises in conjunction with programs that act as "filters". They read in their standard input, apply some transform to it and write the result to their standard output. We saw examples earlier in this chapter that used findstr as a filter.

However, the technique demonstrated there would not work with some filter programs because they do not write their output until they see an EOF on their standard input. For such programs, we are in something of a Catch-22. It will not return any data until sees an EOF. For that to happen, we have to close the the channel returned by open. But then once we close the channel, we cannot read back the data the program writes!

Standalone pipes provide a solution for this as well as demonstrated in the script below. We use the filter_upcase.tcl script to simulate these filters. It will keep reading standard input until EOF and then write the upper case form of the input data to its standard output.

```
# filter_upcase.tcl
fconfigure stdin -buffering line
fconfigure stdout -buffering line
set result ""
while {[gets stdin line] >= 0} {
    append result [string toupper $line]\n
}
puts stdout $result
exit 0
```

Now we need to call this script as a child process, feed it some data and read back the results. Although we could use open for our demo as in our previous example, we will instead use exec just to illustrate its use with standalone pipes. Unlike for open which returns a channel, here we will need to create **two** pipes; one to send data to the child as its standard input, and one to read data from the child as its standard output.

```
% lassign [chan pipe] childread mywrite  ❶
% lassign [chan pipe] myread childwrite  ❷
```

❶ Child's standard input
❷ Child's standard output

We then fire up the child with appropriate redirections. We also close the child's end of the channels for reasons explained in the previous example.

```
% exec [info nameofexecutable] scripts/filter_upcase.tcl <@ $childread >@ $childwrite &
→ 1640
% close $childread
% close $childwrite
```

We now write to the channel connected to the child's standard input.

```
% puts $mywrite "This is a test"
% puts $mywrite "This is only a test"
```

Then we close our write-side pipe so that the child will see EOF on its standard input and know it can stop reading and write out the transformed data.

```
% close $mywrite
```

The transformed data now appears on our side of the child's standard output channel.

```
% read $myread
→ THIS IS A TEST
   THIS IS ONLY A TEST

% close $myread
```

Now the dirty little secret is that we only used this example as a means of demonstrating pipe usage. Our simple example could have been done more easily via the method discussed in the next section.

16.4. Half-closing of channels

We described in the previous section one method of running filter programs that wait for EOF on standard input before writing to standard output. The close command provides an alternate, possibly simpler, means for working with such program.

We described the basic operation of the close command in Section 9.3.3. As described there, the command can take an optional second argument, read or write, that specifies that the channel is to be closed only for that direction of data transfer.

```
close CHANNEL ?DIRECTION?
chan close CHANNEL ?DIRECTION?
```

We can make use of this capability to close the standard input the child process. The code below makes use of this technique to run our example child process from the previous section.

We open a pipe to the child process for read and write.

```
% set chan [open |[list [info nameofexecutable] scripts/filter_upcase.tcl] r+]
→ file41545d0
```

We then write our data to the pipe and then close **only the write side** of the pipe which then results in the standard input of the child seeing an EOF.

```
% puts $chan "This is a test"
% puts $chan "This is only a test"
% chan close $chan write
```

The child will then write to its standard output and since the read side of the channel is still open, we can read its output.

```
% read $chan
→ THIS IS A TEST
  THIS IS ONLY A TEST

% close $chan
```

This technique is clearly simpler, but less general, than the one described in Section 16.3.

Half-closing of channels is not limited to pipes. You can also use it for network sockets where closing the client socket for writes would indicate to the server that no more data is forthcoming from the client while leaving the connection open in the reverse direction.

16.5. Passing environment to child processes

The process environment variables in a child process are inherited from its parent. If you want to pass a different environment to the child, you need to save the env array, modify it as per what is desired for the child, start the child process and then restore env from the saved copy.

This is further complicated in a multi-threaded environment since the env global values are reflected across all threads in a process. You would need to employ one of the synchronization mechanisms described in Section 22.11 to coordinate modification of env.

One possible work-around is to run the child process through an intermediary that allows setting of the environment such as /usr/bin/env on Unix or the command shell on Windows.

16.6. Interprocess communications

Exchanging data between processes through pipes is the simplest and most convenient mechanism available. However, for more structured communication, more sophisticated alternatives are needed, like COM on Windows or D-Bus on Unix platforms. Although not built into Tcl and described here, these are available as extensions. See Appendix A for a listing.

16.7. Chapter summary

In this chapter, we described Tcl's support for locating and running other applications, and exchanging data through the standard input and output mechanisms. One of the main features that gave Unix its flexibility and power was the ability to transform data through pipelines of processes that perform specialized tasks. The exec and open commands bring the same flexibility to Tcl.

We have now covered basic input and output, both with respect to files in Chapter 9 and to processes in this chapter. We will now move on in the next chapter to the more advanced I/O facilities available in Tcl.

17

Advanced I/O

In Chapter 9 we introduced the channel abstraction and basic operations in Tcl in the context of reading and writing files. We also saw the use of channels for I/O with child process pipelines in Section 16.2. We now expand on this topic and delve into more advanced topics related to I/O in Tcl:

- Asynchronous input and output operations
- Transforming data during I/O
- Defining new channel types using Tcl's reflected channel abstraction

17.1. Asynchronous I/O

All the I/O operations we have seen so far with files as well as process pipelines have involved *blocking* I/O where the invoked command, get s, read etc., will not return until the I/O operation is completed or the channel end-of-file is reached. For channels backed by files, this behaviour is acceptable for the most part. For channels where the incoming data is intermittent and not always immediately available, this mode of operation is undesirable since the process may be blocked from doing any useful work for long intervals.

For example,

- The application may fire off a process pipeline to carry out a long computation. If it is blocked while reading from the pipeline, it cannot respond to the user or do any other tasks until the computation completes.
- A network server waiting for data from a client will be blocked from communicating with other clients and effectively can only service one at a time.
- Serial port communication is slow to begin with and if there is a human at a terminal on the other end, there is a lifetime between character arrivals. Unless the application is dedicated to responding to that device, it is not feasible to block while waiting for data to show up on the port.

These are the types of situations for which *non-blocking* I/O is designed. When a channel is in non-blocking mode, the command will return immediately even if the I/O operation cannot be completed. The application can then attempt to try the operation at a later point.

Using threads for blocking I/O

Another solution, as in the original releases of the Java language, involves using threads but that not only adds unnecessary complexity but also has scalability issues. Although Tcl also provides threading capabilities at the script level as we will see in Chapter 22, threads are best used for computations that mostly independent or in cases where the underlying API itself does not support a non-blocking mode of operation, as in some database drivers.

There is one issue that arises with non-blocking I/O and that is with regard to when the application retries the I/O operation. Polling continuously is no better than blocking and polling at intervals is neither efficient nor responsive. It would be nice if there were a mechanism whereby the application is notified when a channel is ready for the required operation. As always, Tcl doesn't disappoint! Channels can deliver such notification events through the same machinery we described in Chapter 15.

Non-blocking channels and channel event notifications are almost always used in combination to perform *asynchronous I/O*. Channels are set to non-blocking mode and any I/O operations take place only in response to channel events.

For ease of exposition however, we will start off with a description of non-blocking I/O without involving channel events.

17.1.1. Non-blocking I/O

A channel's blocking mode is controlled with the `-blocking` option to the same `fconfigure` or `chan configure` commands that we saw in Section 9.3.4 for setting other channel configuration options.

```
chan configure CHAN -blocking BOOLEAN
fconfigure CHAN -blocking BOOLEAN
```

The channel *CHAN* is set to blocking mode if *BOOLEAN* is a boolean true value and to non-blocking otherwise. Note that setting the blocking mode affects both read and write operations. It is not possible to set them independently.

17.1.1.1. Non-blocking reads

A read operation on a channel may block because the input data buffers and device are empty or contain less data than what was requested. The effect of this condition on non-blocking read operations depends on the specific command invoking the operation.

17.1.1.1.1. Reading lines in non-blocking mode: `chan gets`, `gets`

The `chan gets` and equivalent `gets` commands retrieve a single line from a channel. If no complete line is available and blocking mode is in effect, the commands will wait unless end of file is reached.

```
chan gets CHANNEL ?VARNAME?
gets CHANNEL ?VARNAME?
```

In non-blocking mode, **if a complete line is available**, the command behaves the same as in blocking mode:

- If *VARNAME* is specified, the line is stored in the variable of that name and the command returns the number of characters in the line. Remember that end of line characters are neither stored nor included in the character count. Thus an empty input line — consecutive newlines with no other intervening characters — will result in the empty string being stored in *VARNAME* and the command returning 0.
- If *VARNAME* is not specified, the command returns the line (possibly as the empty string) as its result.

In the case where **a complete line is not available**, the command differs from blocking mode operation:

- If *VARNAME* is specified, the command returns -1 as its result. The variable is not affected.
- If *VARNAME* is not specified, the command returns an empty string as its result.

When is a complete line not available? It could be for one of two reasons:

- The channel is at end of file.
- The channel is not at end of file but the received data does not (yet) contain a line terminator that would form a complete line.

The following commands can be used to distinguish the two cases:

- The `chan eof`, or equivalent `eof`, command returns 1 if the channel is at end of file and 0 otherwise.
- The `chan blocked`, or equivalent `fblocked`, command returns 1 if the channel is not at end of file but is blocked because the incoming data does not form a complete line. Otherwise it returns 0.

When an attempted read fails because a complete line is not available, it can be useful to see how many characters are present in the channel input buffer. That way, if it exceeds some maximum permitted line length, in a network

protocol for instance, we can take appropriate action such as terminating the connection. The chan pending command will return the number of buffered bytes (**not** characters) in a channel.

```
chan pending DIRECTION CHANNEL
```

Here *DIRECTION* may be input or output depending on whether we are interested in the input or output side of the channel. We'll see an example use below.

OK, enough of the theory. Let us experiment with non-blocking behaviour using a pipe channel as described in Section 16.3. You will recall that data written to the output end of the pipe can be read from the input end. We create the pipe and assign the input and output channels.

```
lassign [chan pipe] in out → (empty)
```

We put the output side into line buffering mode so it will flush automatically any time it sees a newline character.

```
chan configure $out -buffering line → (empty)
```

We are now ready to communicate over the pipe channel. First we will put the input end into non-blocking mode and attempt to read a line. Since we have not written to the pipe as yet, the input buffer will be empty.

```
chan configure $in -blocking 0 → (empty)
gets $in line                   → -1
```

The returned character count is -1 as expected. Attempting a read using the second form of gets gets back an empty string.

```
chan gets $in → (empty) ❶
```

❶ Remember that gets and chan gets are equivalent. We can use either.

We can confirm that the channel is not at end of file and that there is no line in the input buffer.

```
chan eof $in     → 0
chan blocked $in → 1
```

Now let us write two **empty** lines to the pipe.

```
chan puts $out "" → (empty)
chan puts $out "" → (empty)
```

We use the first form of gets to attempt to read a line.

```
chan gets $in line → 0
```

The return value of 0 indicates that an empty line was read. Let us read the second empty line without passing the variable argument.

```
set line [gets $in] → (empty)
string length $line → 0
```

Again we get back an empty line. How do we know whether it is a "real" empty line, or an incomplete one, or end of file? We use chan eof / eof and chan blocked / fblocked to find out.

```
eof $in      → 0
fblocked $in → 0
```

Both return 0 so we know it was indeed a empty line sent by the "remote" end of the channel.

Let us then write data to the channel without an end of line character. Notice we need to do an explicit flush to make sure the data is sent to read side of the channel.

```
chan puts -nonewline $out "A few words" → (empty)
chan flush $out                         → (empty)
chan gets $in line                      → -1
chan gets $in                           → (empty)
chan eof $in                            → 0
chan blocked $in                        → 1
```

We see that even though there is data in the input buffer the two forms of the chan gets command returned -1 and an empty string respectively, as the buffered data did not form a complete line. The chan eof and chan blocked calls confirmed as much.

> Do not confuse the buffering mode we set with -buffering line with line completion handling. The -buffering option only controls when data is flushed from the output buffers.

We can in fact confirm that there are characters in the input buffer with chan pending.

```
chan pending input $in → 11
```

The command tells us there are 11 bytes pending in the input.

Let us now examine the end of file condition. We close the output side of the pipe and attempt another read.

```
close $out → (empty)
gets $in   → A few words
```

Notice that on end of file, the buffered input content is delivered even though no newline characters were present.

Subsequent reads then indicate end of file which we can check with eof.

```
chan gets $in line → -1
gets $in           → (empty)
eof $in            → 1
fblocked $in       → 0
close $in          → (empty)
```

Before moving on to its sibling — the read command — you may want to play further with the various scenarios and how they affect the results returned by gets, eof and fblocked.

17.1.1.1.2. Reading characters in non-blocking mode: chan read, read

We described the basic operation of the chan read, and the equivalent read, commands in Section 9.3.6.2. The commands have two forms, one where the number of characters to read is not specified.

```
chan read ?-nonewline? CHANNEL
read ?-nonewline? CHANNEL
```

In the other, the caller explicitly asks for a specific number of characters.

```
chan read CHANNEL NUMCHARS
read CHANNEL NUMCHARS
```

In blocking mode, the commands read a specified number of characters from a channel or all characters till end of file if no character count is specified. If the specified number of characters is not available, or end of file is not reached in the case of no count being specified, the read command will wait.

In non-blocking mode, both forms alter their behaviour to return whatever data is available even if less than requested. We will again use a pipe for demonstration. This time since we are writing a character stream, we will set the buffering mode to none to ensure the data is passed along right away.

```
lassign [chan pipe] in out          → (empty)
chan configure $out -buffering none → (empty)
chan configure $in -blocking 0      → (empty)
```

An attempt to read at this point returns empty strings. As before we can use eof and friends to check the cause.

```
chan read $in     → (empty)
chan read $in 1   → (empty)
chan eof $in      → 0
chan blocked $in  → 1
```

Let us write a single character and attempt to read two.

```
puts -nonewline $out A → (empty)
chan read $in 2        → A
```

As you can see, the read returns a single character which is all that was available in the input buffer. The same holds true if we tried to read to end of file.

```
puts -nonewline $out B → (empty)
chan read $in          → B
eof $in                → 0
```

When closing the pipe, we take the opportunity to reiterate another point about end of file conditions. When we check for end of file after the pipe is closed, eof returns 0, **not** 1.

```
close $out → (empty)
eof $in    → 0
```

This is because chan eof / eof **only detect an end of file after a input command** (gets **or** read) **fails** because of an end of file condition. To prove that point,

```
read $in  → (empty) ❶
eof $in   → 1
close $in → (empty)
```

❶   Empty string returned due to EOF

17.1.1.2. Non-blocking writes: chan puts, puts

The non-blocking behaviour on the output side using chan puts or puts is a lot simpler than on the input side. In blocking mode when a write is made to a channel, if its internal buffers (if in use) are full, an attempt is made to flush them to the underlying device. If the device cannot accept the data, the puts command will block until such a time that the device is ready to accept more data.

In non-blocking mode, Tcl will accept the data and store it in its internal buffer, growing it as necessary. When the device is ready to accept more data, this internal buffer is flushed to the device behind the scenes. However, **this requires that the Tcl event loop be running**. This is not generally a problem because non-blocking I/O is almost always used in conjunction with channel events to implement asynchronous I/O.

In non-blocking mode, even the flush / chan flush commands do not (can not) flush the internal buffers to the device if it is not ready. They will be flushed in the background, potentially after the command returns. To force data to be written out immediately, the channel must be placed in blocking mode, flushed and then reverted to non-blocking mode. Of course, this means the application is blocked until the flush completes.

Although Tcl will accept any amount of data on output in non-blocking mode, it is advisable to use channel event notifications to only write when the channel is ready to accept data. Otherwise, there is a danger of the output buffers growing unacceptably large causing memory pressure.

And that leads us to a discussion of event-driven I/O.

17.1.2. Event driven I/O: chan event, filevent

As we discussed earlier, efficient asynchronous I/O requires some means for an application to be notified when a channel is ready to receive data on output or has data available for reading. The channel subsystem in Tcl provides these notifications by generating events when channels are ready for input or output. An application can then register callbacks to be invoked on the occurence of these events.

Channel events are tied into the same eventing infrastructure we described in Chapter 15 and therefore require the event loop to be running. On every iteration of the event loop, each channel driver is given an opportunity to add events to the event queue. If any channel event handlers are registered for a channel and it has notifications pending, the channel driver will enqueue an entry to event queue with the event details and handler information. When the event loop processes the event queue, it will invoke any queued handlers just as it does for timer events.

The handlers for channel events are registered with the chan event, or equivalent fileevent command.

```
chan event CHANNEL EVENT ?HANDLER?
fileevent CHANNEL EVENT ?HANDLER?
```

The HANDLER argument is the callback script that should be invoked in reaction to the notification event specified by EVENT. If there is an existing handler already registered, it is replaced. If HANDLER is not specified, the command returns the script currently registered for that channel for the EVENT event or an empty string if none is registered. If HANDLER is passed as the empty string, the currently registered handler, if any, is unregistered.

The EVENT argument indicates the type of event and must be either readable or writable.

A readable event is generated under either of two conditions:

- Data is available to be read from the channel.
- The channel is at end of file.

As we will see in our examples, the handler script generally uses the methods described in the previous sections to read data or detect end of file on the channel.

A `writeable` event is generated when the channel is ready to accept data from the application. A special case of this condition occurs on asynchronous network socket connections when the connection set up is completed and is open for transmitting data. We will look at this case further in Chapter 18.

As a first example of event driven I/O, let us revisit our pipe examples from the previous section except that instead of blindly reading from the pipe, we will only attempt to read when we are notified that data is available. The example also highlights some points you need to be aware of when programming asynchronous I/O so we will go through it slowly.

```
lassign [chan pipe] in out                      → (empty)
chan configure $out -buffering none -blocking 0 → (empty)
chan configure $in -blocking 0                  → (empty)
```

We have turned off buffering for the output channel so that data is immediately written to the pipe for reasons we will see later.

Next, we write a procedure that implements the event handler.

```
proc read_handler {chan} {
    set status [catch {gets $chan line} nchars]
    if {$status == 0 && $nchars >= 0} {
        puts "Received: $line"
        return
    }
    if {$status || [chan eof $chan]} {
        puts "All done!"
        chan event $chan readable {}
        set ::exit_flag 1
        return
    }
    puts "Incomplete line"
}
```

Our procedure will be invoked for every `readable` event on the channel of interest. It starts off with attempting to read a line from the channel. We use the `catch` command to trap any possible errors that the `gets` command might raise. This might happen, for example, if the remote end aborts a network connection.

Then, if there were no errors and the command read a line successfully (`nchars` is not negative), we print the line and return.

Otherwise, if either an error occured on the channel read or if the channel is at end of file, we remove the read handler from the channel, set an exit flag and return. Note that use of `exit_flag` is because we will be using `vwait` to run the event loop for our little example. In a real application which is already running the event loop, this line would not be necessary.

! When an end of file is seen on a channel, it is crucial to either remove the read handler from the channel as we have done, or to close the channel in the handler itself before returning. Otherwise, the channel will continuously raise `readable` events because the channel is at end of file.

When none of the previous conditions are satisfied, the procedure falls through to the end to print the `No data` message. We will see in a bit the conditions under which this can happen.

Now we register our read handler for the input channel. Notice that we have to explicitly pass in the channel as an argument to the handler since the channel subsystem does not itself pass in this information. Just for kicks, we also confirm that it is registered.

```
chan event $in readable [list read_handler $in] → (empty)
chan event $in readable                          → read_handler file3b4bef0
```

Having set up the read handlers on the input channel, it is time to write to the pipe and see what happens. To simulate data arriving intermittently as from a remote network, we will split up the writes into partial writes with intervening delays, all scheduled through the event loop with the `after` command. (This is also the reason why we turned off buffering for the output channel earlier.)

```
after 50 [list puts $out "Hello World!"]        → after#38
after 100 [list puts -nonewline $out "Goodbye "] → after#39 ❶
after 150 [list puts $out "World!"]             → after#40
after 200 [list close $out]                     → after#41
```

❶ Note incomplete line

Finally, we get the event loop rolling with the `vwait` command. In a real application that expected to do asynchronous I/O the event loop would already be running but that is not true for our interactive shell. So we have to explicitly start it.

```
% vwait ::exit_flag
→ Received: Hello World!
  Incomplete line
  Received: Goodbye World!
  All done!
% close $in
```

Let us now examine the resulting output.

- When the first string written to the channel is received on the input side, our read handler is invoked as there is data available on the channel. Since an entire line is available, the `gets` call succeeds and the line is printed out.
- The second string written to the channel is an incomplete line. When the data arrives on the input side, our read handler is again invoked. This time the `gets` returns -1 because a complete line is not available in the input buffer. The first `if` condition fails. Moreover, since the end of file is not reached, the second `if` condition also fails, and control passed to the bottom of the procedure resulting in `Incomplete line` being printed.
- The third write results in another invocation of the read handler. This time a complete line is available and printed.
- Finally the sending side closes its end of the pipe. The resulting end of file on the input side also triggers the read handler. This time the `gets` returns -1 and `eof` indicates end of file. Consequently, the second `if` block is executed. **Here we are careful to remove the read handler from the channel otherwise we would be continuously invoked.** We could have closed the channel instead but we leave that for the main line code.
- Because the read handler set the `exit_flag`, the `vwait` command terminates the event loop and returns. We go on to close the input channel.

In our example, we used `gets` to read the channel a line at a time. We could also have used the `chan read`/`read` commands as well keeping in mind the differences with respect to `gets` that we described in Section 17.1.1.

The above example dealt with read handlers. We will not show an example of a write handler here as we will discuss further examples in Chapter 18.

One final note about channel event handlers. If the event handler raises an uncaught exception, Tcl will unregister it from the channel.

17.1.3. Closing non-blocking channels

There are a couple of considerations to keep in mind with regards to closing channels that are in non-blocking mode.

- The `close` command on a non-blocking channel returns immediately before any buffered data is written out. The data is then flushed in the background. This requires the event loop to continue to be active. Moreover, the

application should not assume the data is will be available on the underlying device when the close command returns.

- Any open non-blocking channels must be switched to blocking mode before exiting the process. Otherwise any buffered data may not be written out before the process exits.

17.1.4. An interactive command line

When you run tclsh without any arguments, it enters an interactive read-eval-print-loop (REPL) where it executes Tcl entered on the command line. This default REPL uses blocking I/O and does not have the event loop active. In order to have any event loop based functionality, such as asynchronous I/O, you need to enter the event loop through a command such as vwait. However, you then lose the ability to enter commands at the command line.

Another common situation that arises is in an event loop based application such as a network server. The script implementing the server usually has a vwait at the end that activates the event loop. Although these are generally "background" applications, it is nevertheless useful for them to be able to expose an interactive command line interface for purposes such as configuration, troubleshooting etc.

An event-driven REPL interface is useful in both the above scenarios. It collects interactive command input using non-blocking I/O permitting other event processing to proceed without interruption. A basic implementation is shown below.

```
namespace eval repl {}
proc repl::prompt {prompt} {
    puts -nonewline stderr $prompt
    flush stderr
}

proc repl::repl {} {
    variable command
    variable done

    set command ""
    prompt "% "
    fileevent stdin readable [namespace current]::repl_handler
    vwait [namespace current]::done
}
```

The repl::repl procedure is intended to be called from the main application to enter the event loop while also displaying a REPL for command input. It does some initialization, sets up a read handler on standard input and then enters the event loop.

The read handler repl_handler, shown below, is where the hard work is all done. It is invoked when a full line is available on the standard input or if the input channel is closed. In the latter case, it simply terminates the vwait loop after removing itself as the input handler. Otherwise, it appends the new line to any previously collected input. If the command is syntactically complete, it is executed in the global scope and the result printed. If the command is not complete, the secondary prompt is displayed.

```
proc repl::repl_handler {} {
    variable command
    if {[gets stdin line] < 0} {
        fileevent stdin readable {}
        set [namespace current]::done 1
        return
    }
    append command $line
    if {[info complete $command]} {
        fileevent stdin readable {} ❶
        set status [catch {uplevel #0 $command} result]
        fileevent stdin readable [namespace current]::repl_handler
```

```
        if {$result ne ""} {
            if {$status == 0} {
                puts $result
            } else {
                puts stderr $result
            }
        }
        set command ""
        prompt "% " ❷
    } else {
        append command \n
        prompt "> " ❸
    }
}
```

❶ Avoid nested calls
❷ Primary command prompt
❸ Secondary command prompt

One point to be noted above is that the read handler disables itself before calling uplevel and then restores itself afterward. This is a precautionary measure in case the script executed in the uplevel call itself recursively enters the event loop. In that case we do not want our read handler to be called if more input is available until the currently executing uplevel has finished execution.

The above implementation leaves out some details for pedagogic purposes. More complete implentations are available in the Tcler's Wiki[1], for example the one at http://wiki.tcl.tk/1968.

17.2. Channel transforms

In Section 9.3.8 we described the use of the -encoding option to transparently encode data written to a channel. This removes the burden from applications to remember to explicitly encode strings when writing to a file. This is very convenient when the output commands may be spread over multiple locations in the application.

Now consider an application writing to a log file where the data must be compressed to save disk space. Or perhaps encrypted in some form to hide passwords or other private details. Or both. Clearly the same kind of transparency afforded by the -encoding option would be very useful here as well.

Obviously Tcl cannot have built-in facilities for all the infinite ways that data might be transformed. So it provides something almost as good — a way for applications to implement their own *channel transforms* that can alter the data stream flowing through a channel. Moreover, multiple channel transforms can be applied to a channel in a stacked fashion so the output of one transform is fed into the next. Thus we can write a compression transform and an encryption transform to meet our needs and even combine them to implement a compress and encrypt "combo" channel.

17.2.1. Channel transform basic operation

Figure 17.1 shows data flow in a channel when the command puts is invoked with encoding set to UTF-8 and line endings to CRLF. The flow on the left side has no channel transforms applied. The Tcl channel top layer then encodes the input data, translating newlines to CR-LF pairs and emits a byte stream to the underlying device which may be a file, a network socket etc.

The right side of the figure shows the flow with two transforms pushed onto the channel, first the base64 transform that we assume converts binary data into base64 encoding and a second transform which compresses[2] the data using the DEFLATE algorithm we discussed in Section 4.16. This is not necessarily a sensible combination of transforms but ... whatever. It serves our illustrative purpose.

---

[1] http://wiki.tcl.tk
[2] Short strings like in our example will actually land up being longer

```
        puts $chan Olá                          puts $chan Olá

    String  │  Olá                          String  │  Olá
            ▼                                        ▼

    Tcl channel subsystem                    Tcl channel subsystem
     -encoding utf-8                          -encoding utf-8
     -translation crlf                        -translation crlf

         │ Buffer │                               │ Buffer │
            ▼                                        ▼

   Byte stream │ 4f 6c c3 a1 0d 0a        Byte stream │ 4f 6c c3 a1 0d 0a
            ▼                                        ▼

         Device                              deflate transform

                                     Byte stream │ f3 cf 39 bc 90
                                                 │ 97 0b 00
                                                 ▼

                                          base64 transform

                                     Byte stream │ 38 38 38 35 76 4a
                                                 │ 43 58 43 77 41 3d
                                                 ▼

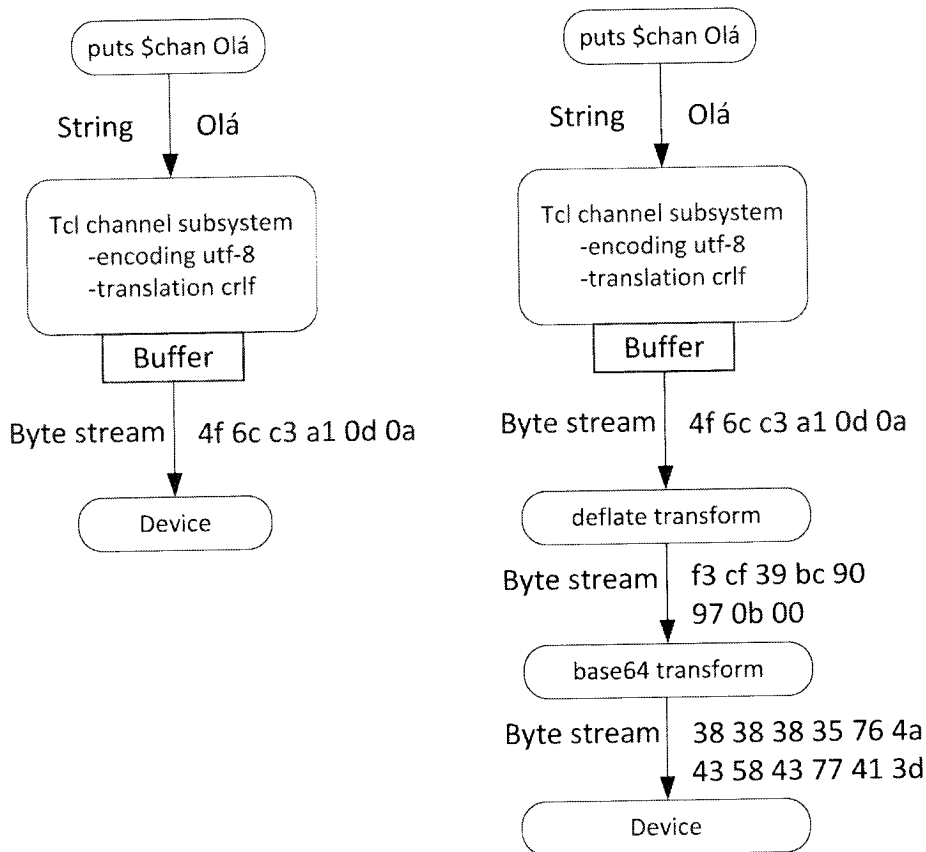                                               Device
```

Figure 17.1. Basic channel operation

As shown in the diagram,

- The top layer of the Tcl channel subsystem produces a byte stream (or a binary string in Tcl parlance) and this is what the channel transforms see. There are no "characters" at this level so if the transform wants to do character based operations like upper-casing all text, it gets trickier than you might think. We will revisit this issue later.

- When multiple channels are pushed on to a channel, the later ones are placed on top of earlier ones. Hence the term "stacked" transforms.

- The channel subsystem calls each transform in turn, passing it the data returned by the previous transform and using the data returned by the transform as the input to the next. The data produced by the bottom most transform is written to the device.

- Although not depicted in the figure, a transform is free to return an empty value in which case the transform below is not called. An example of this behaviour would be exhibited by block oriented encryption algorithms which operate on fixed length sequences of bytes. In general, a channel transform may convert and pass on all, some or none of the data passed to it. The rest can be buffered internally for later processing.

The above discussion described operations on the channel output path. For the input path, the operations are very similar, except in reverse order, and we will not describe them separately.

17.2.2. Implementing channel transforms

The Tcl channel subsystem expects channel transforms to be implemented in the form of a command prefix that will be invoked with arguments such as `read` that are subcommands indicating the operation to carry out. Channel transforms may be implemented as namespace ensembles, TclOO object or even as a simple procedure whose first argument is treated as the operation to perform.

The subcommands that a channel transform must implement fall into three categories:

- Those that must be implemented by all transforms
- Those that must be implemented for transforms that affect read operations
- Those that must be implemented for transforms that affect write operations

The complete list of subcommands is shown in Table 17.1.

Table 17.1. Channel transform subcommands

Subcommand	Direction	Description
initialize *HANDLE MODE*	Both	Called to initialize the transform.
finalize *HANDLE*	Both	The last call on the transformer to permit it to clean up any allocated resources.
clear *HANDLE*	Both	Called to have the transformer clear internal buffers or state. This command is optional.
drain *HANDLE*	Read channels	Called to force the transform to return any internally buffered data to the channel subsystem so it can be passed to the higher layers. This subcommand is optional.
limit? *HANDLE*	Read channels	This subcommand is a way for the transform to tell the Tcl channel subsystem to limit the number of "read-ahead" bytes. This command is optional.
read *HANDLE BUFFER*	Read channels	Called to pass data from the device or another transformer below this transform.
flush *HANDLE*	Write channels	Called when internally buffered data in the transformer must be passed to the device or transformer below. This command is optional.
write *HANDLE BUFFER*	Write channels	Called to pass data into the transformer from the layer above.

Note that a channel transform need not be applicable to both input and output sides of a channel. For input-only and output-only transforms, the `write` and `read` subcommands respectively, need not be implemented.

Let us examine the implementation of these subcommands in detail by working through an example. We will implement a channel transformation for encrypting data using the RC4 cipher.

> ✖ The use of RC4 in practice is not recommended because of vulnerabilities in the algorithm itself. We use it here because of the simplicity of the RC4 interface allows us to focus on the channel abstraction itself.

The RC4 implementation we use is from the `rc4` package of Tcllib[3].

```
package require rc4
```

---
[3] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

We will use Tcl's object oriented features instead of a namespace for implementing our transform as it makes it slightly easier to keep a context for keying material. We start by defining a class that implements the transform wherein the subcommands shown in Table 17.1 are methods of the class. For ease of explanation, we will construct the class piecemeal.

```
oo::class create RC4Transform {
    variable RC4Read
    variable RC4Write
    constructor {key} {
        set RC4Read [rc4::RC4Init $key]
        set RC4Write [rc4::RC4Init $key]
    }
}
```

Since encryption requires keying material, we pass it as an argument to the constructor. The constructor passes this to the RC4 package to get back handles that can be used for encryption operations on output and decryption on input. We store these away into member variables.

17.2.2.1. Initializing channel transforms

Next we define the mandated initialize subcommand.

```
oo::define RC4Transform {
    method initialize {transform_handle mode} {
        return {initialize read write finalize}
    }
}
```

The first parameter of all subcommands of a channel transformation is a handle to the channel transform (**not** the owning channel). Implementation based on namespaces or procedure calls generally make use of this handle, called transform_handle in our sample code, to distinguish between different channel instances applied to different channels. With a TclOO based implementation like ours, the context is implicit in the object instance because a different instance is pushed onto each channel. Thus this handle is not of much use to our implementation and it is ignored in all methods.

The second parameter to initialize is a list of one or two elements, from the strings read and write, indicating whether the attached channel is readable and writable. Our transformation does not care so we will ignore this parameter as well in our method.

The return value from initialize should be the list of subcommands supported by the transformation.

17.2.2.2. Finalizing channel transforms

The other mandated command is finalize which is called by the Tcl I/O system when the transformation is popped from the channel.

```
oo::define RC4Transform {
    method finalize {transform_handle} {
        rc4::RC4Final $RC4Read
        rc4::RC4Final $RC4Write
        [self] destroy
    }
}
```

The transformation handle is the only parameter passed and we ignore it for the reasons explained above. The purpose of the call is to allow the transformation to release any resources. In our case, we release the resources associated with the RC4 handles. No further calls are made to the transformation by the I/O system once this call returns.

We also choose to have the transformation commit suicide by calling its destroy method. By doing so, we free the application from having to keep track of the transformation object itself and destroying it at some suitable point. We could have also chosen to leave it to the application to destroy the object after the channel transformation is popped from the channel. This is useful when the channel transformation may hold some additional computed value based on the data passing through the channel. For example, a channel transformation may compute a checksum on data passing through a channel and allow it to be retrieved after the channel is closed.

17.2.2.3. Transforming data

Finally, we come to the actual data transfer and implementation of the read and write subcommands.

```
oo::define RC4Transform {
    method read {transform_handle bytes} {
        return [rc4::RC4 $RC4Read $bytes]
    }
    method write {transform_handle bytes} {
        return [rc4::RC4 $RC4Write $bytes]
    }
}
```

The read and write commands both take two argments, the channel transformation handle which we ignore as before, and the binary data to be transformed. Our implementation is very simple as we leave it up to the RC4 package to do the hard work.

Here then is the complete channel transformation in a condensed form.

```
package require rc4
oo::class create RC4Transform {
    variable RC4Read
    variable RC4Write
    constructor {key} {
        set RC4Read  [rc4::RC4Init $key]
        set RC4Write [rc4::RC4Init $key]
    }
    method initialize {transform_handle mode} {
        return {initialize read write finalize}
    }
    method finalize {transform_handle} {
        rc4::RC4Final $RC4Read
        rc4::RC4Final $RC4Write
        [self] destroy
    }
    method read {transform_handle bytes} {
        return [rc4::RC4 $RC4Read $bytes]
    }
    method write {transform_handle bytes} {
        return [rc4::RC4 $RC4Write $bytes]
    }
}
→ ::RC4Transform
```

We will look at how we could use our RC4 transform in a bit but first we take a short detour to discuss some buffering issues that are not relevant to our example because of its simplicity but need to be managed in more complex transforms.

17.2.2.4. Buffering in channel transforms

Our example was greatly simplified by the fact that we had no need for internal buffering. In particular, data coming in to the transform in either direction was immediately passed on.

We now look at issues that arise when this is not possible due to the characteristics of the transform. Examples include

- Transforms that are block oriented such as the base64 encoding scheme or the DES encryption algorithm. These require the data to be transformed in multiples of fixed size blocks. Since there is in general no control over how the Tcl I/O system and other transforms pass in data, partial blocks have to be buffered internally and then transformed when further calls complete the block.
- Some transforms such as zlib compression are stream oriented but perform better when the data is compressed in larger chunks. These may choose to internally buffer data for performance reasons.
- Some transforms may deal with variable length blocks. These are the most complex to deal with. A common example of this kind of transform is anything dealing with characters as opposed to bytes. Consider a transform that changes the character case of all data passing through to upper case. Depending on the character encoding in effect on the channel, each character may be encoded as a variable number of bytes. The transform then has to parse the byte stream to recover and transform individual characters. To further complicate matters, a multibyte character may be split across multiple read (or write) calls.

At some point, such as when the channel is closed or the channel transform is popped from the channel, Tcl has to ask the transformation to flush its internal buffers and return their content. It does this by calling the flush and drain subcommands. The former is called for the output side of the channel and the latter for the input. Both commands take the transformation handle as their only parameter and are expected to complete transforming the data in their internal buffers and return the result.

The other situation concerning internal buffers involves the application calling chan seek on the channel. Since this changes the access pointer for the file (or any channel that supports seeks), internal buffers must be cleared. In this situation, the Tcl I/O subsystem calls the clear subcommand. In response, the channel transformation is expected to clear its internal buffers, **both** input and output, throwing away all stored data in the process.

Channel transformations that do internal buffering generally need to implement these three subcommands.

Applications should avoid calling chan seek or seek on channels which have transforms applied. By their very nature, transforms like compression are not amenable to these operations.

We will not present any examples of channel transformations that do internal buffering. You can look at the source code for the tcl::transform::base64 package in Tcllib[4] for a real world example.

17.2.2.5. Limiting read-ahead

For reasons of performance, in normal operation the Tcl I/O system will read more bytes from a device than are actually requested by the application. This can be a problem for transforms that expect data streams in some bounded format. Consider using a transform on network channel to decompress a compressed chunk of an HTTP stream. If the Tcl I/O system reads ahead and passes data beyond the compressed portion, the transform will process the compressed part but has no way of returning the remaining portion back to the I/O system if the transform is then removed from the channel. (The application in this case knows how many bytes are compressed and pops the transform after reading that number.)

The channel transformation can implement the limit? command to control the amount of read-ahead done by Tcl on input. It takes the channel transformation handle as argument and should return the upper limit for the number of bytes that Tcl is allowed to read ahead, returning a negative integer to indicate no upper limit. It is called by Tcl before every read subcommand invocation. The subsequent read subcommand invocation will never pass in data of length more than the (non-negative) value returned by the limit? call.

17.2.3. Using channel transforms

Let us now look at how an application utilizes a channel transform using chan push and chan pop.

---

[4] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

```
chan push CHANNEL CMDPREFIX
chan pop CHANNEL
```

The chan push command places a channel transform at the top of the channel transform stack, i.e. just below the Tcl I/O buffering layer as shown in Figure 17.1. Successive chan push commands result in a stack of transformations with the last one pushed on top.

The chan pop command removes the topmost channel transform from the specified channel. It is not possible to remove any transform except the topmost.

The CMDPREFIX is a command prefix callback that in effect identifies the channel transform. When carrying out various I/O operations, Tcl invokes CMDPREFIX with the subcommand arguments as described in the previous section.

We will use our RC4Transform channel transformation across a pipe for demonstrative purposes. First we create an instance of the transformation and the pipe.

```
set xform [RC4Transform new secret] → ::oo::Obj296 ❶
lassign [chan pipe] in out          → (empty)
```

❶   Warning: it is a terrible idea to use an ASCII string for keying but this is just a demo

We will apply this transform only on the **output** side of a pipe. On the other end of the pipe, we will simply read the channel in binary mode without the transform applied. In effect, this will tell us what bytes are being written to the underlying operating system pipe.

```
chan push $out $xform                  → file4102620
chan configure $in -translation binary → (empty)
puts $out "Just a demo"                → (empty)
close $out                             → (empty)
```

Reading the other end of the pipe tells us the raw bytes written from the output side. We will dump the data in hex using our bin2hex helper procedure.

```
bin2hex [set encrypted [read $in]] → a7 43 a1 68 a2 c5 f6 c2 57 a6 d4 d1 f9
close $in                          → (empty)
```

As you can see the data was encrypted, something we can verify by decrypting with the rc4::rc4 command.

```
rc4::rc4 -key "secret" $encrypted → Just a demo
```

Of course, in a real application both ends of the communication would have a transform applied. Obviously, the same shared secret has to be used for both.

```
set in_xform [RC4Transform new secret]  → ::oo::Obj297
set out_xform [RC4Transform new secret] → ::oo::Obj298
lassign [chan pipe] in out              → (empty)
chan push $out $out_xform               → file4134d50
chan push $out $in_xform                → file4134d50
puts $out "Just a demo"                 → (empty)
close $out                              → (empty)
read $in                                → Just a demo
close $in                               → (empty)
```

17.2.4. Applications of channel transforms

Our demonstration of channel transforms involved transforming the data passing through a channel. A number of such transforms are implemented by the `tcl::transform` package in Tcllib[5]. These include

- `base64` and `hex` for encoding binary data
- `otp` (one time pad) and `rot` encryption
- `zlib` for compression (although the built-in capabilities described in Section 17.2.5 make this superfluous).

However, transformation of data is not the only use for channel transforms. Several transformations provided Tcllib[6] do not modify the data but do computations on them instead.

- `crc32`, `adler32` compute checksums on the data passing through
- `counter` simply counts the bytes flowing through a channel

A short example illustrating their use in stacked fashion:

```
package require tcl::transform::crc32
package require tcl::transform::counter
lassign [chan pipe] in out
::tcl::transform::crc32 $out -write-variable crc
::tcl::transform::counter $out -write-variable counter
puts $out "Just a demo"
close $out
puts "Wrote $counter bytes with a CRC of $crc"
→ Wrote 13 bytes with a CRC of 3760966656
```

The count of bytes is the number of bytes, **not** characters, passing through the channel after any encoding, line ending translation etc.

As an aside, note from the example that there is no explicit `chan push` call. The Tcllib[7] transforms make this call internally on the channel passed to them. This is different from our implementation of `RC4Transform` and is a stylistic choice.

Yet another form of channel transform, again available in Tcllib[8] just observes the data passing through a channel. Since channel transforms can be pushed and popped dynamically, this can be very useful for troubleshooting and logging purposes. The `tcl::transform::observer` transformation implements this functionality.

17.2.5. Zlib channel transforms

In Section 4.16, we introduced the `zlib` command for compressing binary strings. In addition to the features described there, the command also directly supports use as a channel transform with the `zlib push` command.

```
zlib push ALGORITHM CHANNEL ?OPTIONS?
```

The command pushes a zlib transformation on to the specified channel. The transformation may be popped with the normal `chan pop` command. The *ALGORITHM* argument specifies the type of compression or decompression to be applied and must be one of the values `deflate`, `inflate`, `compress`, `decompress`, `gzip`, `gunzip`.

The command takes the options shown in Table 17.2.

---

[5] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html
[6] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html
[7] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html
[8] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

Table 17.2. `zlib push` command options

Option	Description
`-dictionary` `BINDATA`	See Table 4.20.
`-header` `HEADER`	See Table 4.20.
`-level` `LEVEL`	See Table 4.20.
`-limit` `LIMIT`	Specifies a limit for read-ahead of data. See Section 17.2.2.5. Defaults to 1.

Once the transform is pushed on a channel, the `-dictionary`, `-header` and `-limit` options can be read with the `fconfigure` or `chan configure` commands.

In addition, `chan configure` / `fconfigure` take two additional options. The first, `-checksum`, is a read-only option that returns the checksum for the uncompressed data seen on the channel up to that point. The second option, `-flush`, is a write-only option that takes values `sync` or `full`. See Section 4.16.2.9 for their effect.

A simple example of using the transform follows. We create a channel to a file, push the compression transform onto the channel, and write to it.

```
set chan [file tempfile path]                → file3b33750
zlib push compress $chan                     → file3b33750
puts $chan [string repeat "abcdefghij" 10]   → (empty)
```

Let us also retrieve the checksum before we close the channel.

```
flush $chan                         → (empty)
chan configure $chan -checksum      → 525608894
close $chan                         → (empty)
```

!
Notice that we flush the channel before retrieving the checksum. Otherwise the transformation would not have seen all the data yet. If the buffering on the channel is turned off, this is not necessary.

Let us see if the file is compressed.

```
file size $path → 23
```

Indeed it is. But let us read it back to ensure it is compressed and not truncated. Along the way, we retrieve the checksum and see that it is correct.

```
set chan [open $path r]             → file4108640
zlib push decompress $chan          → file4108640
read $chan                          →
 abcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghij
chan configure $chan -checksum → 525608894
close $chan                         → (empty)
```

One point to note is that even though the compressed file is a binary file, not text, we do not open it in binary mode. The mode for open is the same mode used to write the file (defaulting to text in our example). The transformation sits **below** the text / binary translation layer of the Tcl I/O subsystem so it will always see the raw bytes from the file irrespective of how the channel is configured.

17.3. Reflected channels

Whereas channel transforms allow us to process data as it is being passed through channels, Tcl's reflected channel facility presents a way to define entire new channel types at the script level. With reference to Figure 17.1, reflected channels occupy the "device" box in the I/O stack.

Our discussion is mostly focused towards **implementation** of reflected channels as **using** them is for the most part no different from the built-in channels. However, there are some notable limitations that we describe in Section 17.3.3.

17.3.1. Implementing reflected channels

Just like channel transforms, reflected channels are implemented as command prefixes that are invoked by Tcl's I/O system in response to various I/O operations. These command prefixes may be in the form of namespace ensembles, TclOO objects or anything else that will accept subcommands that specify the operation.

The complete list of these subcommands is shown in Table 17.3. As for channel transforms, implementation of some of these subcommands is optional depending on whether the channel supports read or write operation.

Table 17.3. Reflected channel subcommands

Subcommand	Direction	Description
initialize *HANDLE MODE*	Both	Called to initialize the channel.
finalize *HANDLE*	Both	The last call on the channel to permit it to clean up any allocated resources.
watch *HANDLE EVENTS*	Both	Called to inform the channel as to the types of I/O events, read and/or write, that should be reported.
read *HANDLE COUNT*	Read channels	Called to read *COUNT* bytes from the channel.
write *HANDLE BYTES*	Write channels	Called to write the *BYTES* binary string to the channel.
seek *HANDLE OFFSET BASE*	Both, optional	This subcommand should move the file access pointer as specified.
configure *HANDLE OPTNAME OPTVAL*	Both, optional	Called to configure a channel configuration option.
cget *HANDLE OPTNAME*	Both, optional	Called to retrieve the value of a channel configuration option.
cgetall *HANDLE*	Both, optional	Called to retrieve the values of all channel configuration options.
blocking *HANDLE MODE*	Both, optional	Called to set the blocking mode of a channel

As before, we will look at these subcommands in detail as we go through the implementation of a simple reflected channel. Our illustrative sample implements a channel that will simply store data written to it and allow it to be read out. The `tcl::chan::fifo` module in Tcllib[9] already provides such a reflected channel but ours will differ in a couple of respects for pedagogic purposes.

Whereas we demonstrated channel transformations using TclOO, for reflected channels our implementation will use namespaces instead so they don't feel left out. All our subcommands will be implemented as procedures within this namespace which we will name `bureaucrat`. Why, you ask? Because

- It introduces unnecessary delays in data transfer without doing any useful work in the process. In our case, this is to better demonstrate event-driven I/O.
- It will sit on one task and refuse to accept another until the first one is done. This is for demonstrating flow control and how blocking is signaled to the Tcl I/O system.

---

[9] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

For reasons that will become clear later, our reflected channel requires the event loop to be running.

17.3.1.1. Initializing a reflected channel

On channel creation, the Tcl I/O system will call the `initialize` subcommand for the reflected channel.

```
namespace eval ::bureaucrat {
    variable channels
    array set channels {}
}
proc ::bureaucrat::initialize {chan mode} {
    variable channels
    set channels($chan) {
        State    OPEN
        Data     {}
        InFlight 0
        Delay    100
        Blocking 1
        Watch    {}
    }
    return {initialize finalize watch read write configure cget cgetall blocking}
}
```

The first parameter of `initialize` is the channel name itself. While a TclOO based implementation like the one we demonstrated for channel transforms, a namespace based one does not maintain an implicit per-channel context. We therefore store the per-channel state as a dictionary in an array `channels` indexed by the channel name. The semantics of the keys in the dictionary will be clarified as we go along.

The second parameter to `initialize` is a list of one or two elements `read` and/or `write` indicating whether the attached channel is readable and writable. We ignore this parameter as we do not care.

The return value from `initialize` should be the list of supported subcommands. We will not support the seek operation and therefore notice it is not included in the returned list.

17.3.1.2. Closing a reflected channel

When an application closes a reflected channel, Tcl invokes the `finalize` subcommand to inform the reflected channel that it is now history and clean up any allocated resources.

```
proc ::bureaucrat::finalize {chan} {
    variable channels
    unset -nocomplain channels($chan)
}
```

Our toy reflected channel is self contained so all we need to do is get rid of the state variable. In channels like network connections, the remote end would have to be notified and so on.

17.3.1.3. Configuring a reflected channel

A reflected channel may **optionally** define its own configuration options in addition to the standard ones defined by Tcl for all channels. The `configure` subcommand is called when an application wants to set the value of an option.

> The standard configuration options on channels are handled by the Tcl I/O system itself and never passed down to the reflected channel.

In our example, we will allow the channel delay to be configured by an application. We only do minimal error checking for brevity.

```
proc ::bureacrat::configure {chan optname optval} {
    variable channels
    if {$optname ne "-delay"} {
            error "Unknown option \"$optname\"."
    }
    dict set channels($chan) Delay $optval
    return
}
```

The configure subcommand is optional and a reflected channel is not obliged to provide one.

Correspondingly, the channel may implement two subcommands cget and cgetall for retrieving configuration values. Again, these are optional. However, if either one is provided, the other one must be be provided as well. The cget command should return the value of the specified option. The cgetall command should return a dictionary of all options and their values.

```
proc ::bureacrat::cget {chan optname} {
    variable channels
    if {$optname ne "-delay"} {
        error "Unknown option \"$optname\"."
    }
    return [dict get $channels $chan Delay]
}

proc ::bureacrat::cgetall {chan} {
    variable channels
    return [list -delay [dict get $channels $chan Delay]]
}
```

17.3.1.4. Non-blocking mode and event driven I/O

The next two subcommands we discuss, blocking and watch pertain to non-blocking and event-driven I/O operations.

Channels always start out in blocking mode. When an application sets or resets the blocking mode of a channel with a call of the form

```
chan configure $chan -blocking 0
```

the Tcl I/O system makes a call down to the channel's blocking subcommand to inform it of the expected blocking mode of operation. In our case, we simply save the mode in the channel's state. We will make use of its setting when we implement the actual reads and writes.

```
proc ::bureaucrat::blocking {chan mode} {
    variable channels
    dict set channels($chan) Blocking $mode
}
```

While the blocking command is optional, the watch command is mandatory. It is called by the Tcl I/O system in response to the application indicating an interest in read and/or write events on the channel (see Section 17.1.2).

```
proc ::bureaucrat::watch {chan events} {
    variable channels
    set watched [dict get $channels($chan) Watch]
    dict set channels($chan) Watch $events
    if {"read" in $events && "read" ni $watched} {
        notify $chan read
    }
```

```
    if {"write" in $events && "write" ni $watched} {
        notify $chan write
    }
}
```

The events parameter to watch is a list, possibly empty, containing some combination of the values read and write. It indicates what kind of channel events the application has registered an interest in with the chan configure or fconfigure commands. The Watch element of our state dictionary keeps track of this information. We check if we are **adding** either event type as an event of interest. If so, the notify procedure, that we define next, is called to send an appropriate event to the application. We do this check to avoid sending extraneous events in case we were already notifying that event type.

The notify procedure is internal to our implementation and not called directly by the Tcl I/O system. Its purpose in life is to generate an event notification to the application if the conditions warrant it.

```
proc ::bureaucrat::notify {chan event} {
    variable channels
    dict with channels($chan) {
        if {$event ni $Watch} {
            return
        }
        if {$event eq "read"} {
            if {[string length $Data] == 0 && $State ne "EOF"} {
                return
            }
        } else {
            if {$State ne "OPEN"} return
        }
    }
    after idle [list after 0 [list chan postevent $chan $event]]
}
```

The notify procedure is short but there are several important points to be noted therein so we will go over it line by line.

- First, if the application has indicated no interest in the event we return.
- Next we have to check if the conditions implied by the event are present.
- For read events, the channel must have data available to read **or** it must be at end-of-file. If neither of these conditions exist, we will not send a read notification.
- For write events, we impose no restrictions except that the channel must be in the open state for the application to be able to write.
- If the proper conditions are met, we generate an appropriate event with the chan postevent command.

We will discuss this last point in a bit but first we need to touch on the channel state conditions. In our simple example, the channel is always in only one state — that we designate as OPEN — from the time it is created to the time it is closed. Channels in the real world tend to be more complex. An network connection for example may transition from a "connecting" state to an "open" state, to an "end of file" state if the remote end closes the connection. Our watch procedure assumes such state transitions for pedagogic purposes even though the channel itself does not go through these states.

Regarding the sending of events to the application, Tcl provides the chan postevent command to do the needful.

```
chan postevent CHANNEL EVENTLIST
```

Here CHANNEL is the channel for which the event is to be generated and and EVENTLIST is a list containing one or both of read or write denoting the events. In response, the Tcl I/O system will invoke any channel event handlers that were registered for that channel.

One final explanation is warranted for the manner in which chan postevent is called. Basically, chan postevent is scheduled through the event loop and not called directly. The reason for this is that chan postevent implementation will directly invoke the application event handler. That may in turn call back into any of the channel I/O API's and thereby back into our implementation even before we have completed the notify procedure. This reentrancy can get very tricky to deal with and we avoid it by scheduling through the event loop. That ensures that the current call is unwound completely before any application code calls into our code.

As to the strange after idle after 0 idiom, it prevents starvation of the event queue. See Section 15.3.3.1 for a full explanation.

17.3.1.5. Implementing data output

We finally come to implementing the main functions of a reflected channel, viz. some form of data transfer. We will begin with the output side of data transfer.

The Tcl I/O system will call the write subcommand to write data to the device. Although not directly relevant to our implementation, remember from our discussion of channel transforms that this is binary data after all the channel encoding and translation have taken place. This subcommand need not be implemented for a read-only channel.

Our implementation of write has a couple of artifacts. First, it introduces an artificial delay in the data transfer so that data is not immediately available on the input side. Second, it restricts data flow such that while the data is "on the way" to the input side, additional data will not be accepted. These are for the purposes of illustrating non-blocking and event driven operation.

Here is the code itself.

```
proc ::bureaucrat::write {chan bytes} {
    variable channels
    if {[string length $bytes] == 0} {
        return 0
    }
    dict with channels($chan) {
        if {$InFlight} {
            if {$Blocking} {
                return -code error EAGAIN
            } else {
                return -code error EAGAIN
            }
        }
        set InFlight 1
        after $Delay [list [namespace current]::delayed_receive $chan $bytes]
    }
    return [string length $bytes]
}
```

We implement the data transfer delay by scheduling a callback to the delayed_receive procedure with the after command. The InFlight element in the channel's state dictionary is used as a flag to remember that data is "in flight" via the event loop. Before the actual scheduling of the callback, we check this flag and if set, we return the EAGAIN error code to the caller.

The Tcl I/O system treats an error code of EAGAIN in special fashion. It is taken as an indicator that the channel cannot accept more data at this time, because its output buffers are full, or the remote end has flow controlled the connection and so on. The chan blocked command would then return 1 to the application. Tcl itself will continue to retry sending any buffered data at a later point via the event loop.

The error code used for indicating the blocking condition **must** be the string EAGAIN. You cannot use the integer value of this POSIX error instead as the value differs from system to system.

You may noticed something strange in the bodies of the if clause that checks the Blocked variable. Both bodies are the same! Normally, when Blocked is true the channel implementation should block and not return until either more data is available or an end of file condition occurs. However, we have no means of blocking and therefore are forced to deal with the condition in the same way as the non-blocking case. For a full explanation, see Section 17.3.3. **In a reflected channel that was backed by a OS descriptor, the implementation should block on that descriptor when** Blocked **is true instead of returning a** EAGAIN **error.**

The return value from the write should be the number of bytes that the channel accepted. In our implementation, we accept the whole data and thus return its length. However, this need not be the case. A channel may choose to accept only part of the data for whatever reason and return just the length of the accepted part. Tcl will then hold on to the remaining data and include it in a future call.

We now come to the "second half" of the output implementation. The delayed_receive procedure can be thought of as the equivalent of the procedure that will receive data from the remote end of a network connection.

```
proc ::bureaucrat::delayed_receive {chan bytes} {
    variable channels
    if {! [info exists channels($chan)]} {
        return;
    }
    dict with channels($chan) {
        if {$State ne "OPEN"} {
            return
        }
        append Data $bytes
        set InFlight 0
    }
    notify $chan read
    notify $chan write
}
```

The first thing delayed_receive does is check that the channel has not been closed between the time the procedure was queued and the time it was invoked. It also checks if the channel is still in the OPEN state (as we said before, this is not really required for our simple implementation). If both conditions are met, it appends the new data to the incoming data buffer. The InFlight flag is reset since there is no data queued in transit. It then calls the notify procedure we defined earlier to generate a channel read event so any waiting event handlers will be invoked. Since we were limiting writes while the data was in transit, we also need to generate a write event in case any handlers were waiting on the output side as well.

The need to generate a write event is really a peculiarity of our example because data is being input on the same channel as it is output. In most real world cases, there would be no need to generate a write event when **receiving** data.

Before moving on to implementation of the input side, one final point to reiterate is that the data written with the write subcommand should be treated as a byte stream with no direct correspondence with puts commands at the application level. Due to encoding, translation and buffering by Tcl, not only might the number of bytes differ, but a single puts may be split into multiple write invocations, multiple puts may be combined into a single write and so on.

17.3.1.6. Implementing data input

The read subcommand is invoked by the Tcl I/O system when it needs to fetch more data from the channel. As for the output side, there is no immediate correspondence between gets or read command invocations at the application level and the read subcommand of the channel.

The read subcommand is passed the channel name and the number of input bytes to be read. If there are fewer bytes available than are requested, it is permitted to return a smaller number **but not more** than requested. Moreover, returning zero bytes will be treated as an end-of-file condition by Tcl. If there are no bytes currently available but the channel is **not** at EOF, the procedure must signal this by raising EAGAIN error just as for the output side as we described earlier.

Given the above, our read implementation should be mostly self-explanatory.

```
proc ::bureaucrat::read {chan count} {
    variable channels
    dict with channels($chan) {
        if {[string length $Data] == 0} {
            if {$Blocking} {
                return -code error EAGAIN
            } else {
                return -code error EAGAIN
            }
            return -code error EAGAIN
        }
        set bytes [string range $Data 0 $count-1]
        set Data [string range $Data $count+1 end]
    }
    notify $chan read
    return $bytes
}
```

We do call notify at the end so that if there is still more data pending, the application read event handlers will be notified appropriately.

With regards to both bodies of the if statement checking the Blocking condition being identical, see the explanation above for the write subcommand.

17.3.1.7. Reflected channel creation

We are almost done with our reflected channel implementation. Just two things remain:

- We have defined a bunch of procedures to implement the various operations that are expected of the reflected channel. However, the Tcl I/O system expects to invoke a single command prefix to which it appends the subcommand and arguments.
- We need to provide a way for the application to create our new channel type.

For the first task, we can simply create an ensemble command (see Section 12.6).

```
namespace eval ::bureaucrat {
    namespace export *
    namespace ensemble create
}
```

This will allow our procedures to be called as

```
bureaucrat initialize ...
bureaucrat read ...
```

and so on. Note our ensemble also makes visible our ancillary procedures like notify but there is no harm done. If this offends your sensibilities, you can be more specific about the exported subcommands.

The second task, a means of creating a bureaucrat channel, is already provided for us by Tcl with the chan create command.

```
chan create MODE CMDPREFIX
```

Here MODE is a list of one or two elements with values read or write that determine whether the channel is read-only, write-only or read-write. The CMDPREFIX parameter is the command prefix the Tcl I/O system should call to implement the various operations. Thus to create a bureaucrat channel, the application would call

```
set chan [chan create {read write} ::bureaucrat]
```

17.3.1.8. Seeking in a reflected channel

By its nature, just like the built-in sockets our reflected channel type is not amenable to seeking and thus will not actually implement the optional . However, we describe it here just so as to not get an incomplete grade on the book.

The signature for the seek subcommand is

```
seek CHANNEL OFFSET BASE
```

The command is called to position the access pointer from where the next read or write will take place. The arguments are exactly those we described for the application level chan seek command so we will not describe them again.

The subcommand return value should be the absolute position of the access pointer after it has been moved. The application level chan tell also maps to this seek subcommand. Tcl will call it with OFFSET set to 0 and BASE set to current, in effect getting the value of the access pointer without actually moving it.

17.3.1.9. The complete channel implementation

We can now put together our complete implementation (with some minor stylistic changes to put procedures inside the namespace block).

```
namespace eval ::bureaucrat {
    variable channels
    array set channels {}

    proc initialize {chan mode} {
        variable channels
        set channels($chan) {
            State    OPEN
            Data     {}
            InFlight 0
            Delay    100
            Blocking 1
            Watch    {}
        }
        return {initialize finalize watch read write configure cget cgetall blocking}
    }

    proc finalize {chan} {
        variable channels
```

```
    unset -nocomplain channels($chan)
}

proc configure {chan optname optval} {
    variable channels
    if {$optname ne "-delay"} {
        error "Unknown option \"$optname\"."
    }
    dict set channels($chan) Delay $optval
    return
}

proc cget {chan optname} {
    variable channels
    if {$optname ne "-delay"} {
        error "Unknown option \"$optname\"."
    }
    return [dict get $channels $chan Delay]
}

proc cgetall {chan} {
    variable channels
    return [list -delay [dict get $channels $chan Delay]]
}

proc blocking {chan mode} {
    variable channels
    dict set channels($chan) Blocking $mode
}

proc watch {chan events} {
    variable channels
    set watched [dict get $channels($chan) Watch]
    dict set channels($chan) Watch $events
    if {"read" in $events && "read" ni $watched} {
        notify $chan read
    }
    if {"write" in $events && "write" ni $watched} {
        notify $chan write
    }
}

proc notify {chan event} {
    variable channels
    dict with channels($chan) {
        if {$event ni $Watch} {
            return
        }
        if {$event eq "read"} {
            if {[string length $Data] == 0 && $State ne "EOF"} {
                return
            }
        } else {
            if {$State ne "OPEN"} return
        }
    }
    after idle [list after 0 [list chan postevent $chan $event]]
}

proc write {chan bytes} {
    variable channels
    if {[string length $bytes] == 0} {
```

```
            return 0
        }
        dict with channels($chan) {
            if {$InFlight} {
                if {$Blocking} {
                    return -code error EAGAIN
                } else {
                    return -code error EAGAIN
                }
            }
            set InFlight 1
            after $Delay [list [namespace current]::delayed_receive $chan $bytes] ❶
        }
        return [string length $bytes]
    }

    proc delayed_receive {chan bytes} {
        variable channels
        if {! [info exists channels($chan)]} {
            return;
        }
        dict with channels($chan) {
            if {$State ne "OPEN"} {
                return
            }
            append Data $bytes
            set InFlight 0
        }
        notify $chan read
        notify $chan write
    }

    proc read {chan count} {
        variable channels
        dict with channels($chan) {
            if {[string length $Data] == 0} {
                if {$Blocking} {
                    return -code error EAGAIN ❷
                } else {
                    return -code error EAGAIN
                }
                return -code error EAGAIN
            }
            set bytes [string range $Data 0 $count-1]
            set Data [string range $Data $count+1 end]
        }
        notify $chan read
        return $bytes
    }

    namespace export *
    namespace ensemble create
}
→ ::bureaucrat
```

❶ Postpone the actual write
❷ Same code as the false clause! See the text

17.3.2. Using reflected channels

Use of a reflected channel is along the same lines as a built-in Tcl channel. We demonstrate our bureaucrat channel here, more so to prove it works than to illustrate any new commands or techniques.

We start off creating the channel and configuring it to our liking. Except for channels backed by terminal devices, Tcl defaults to full buffering whereas we would like line buffering. Also, we want to test event driven I/O and further verify our custom option -delay is effective.

```
% set chan [chan create {read write} ::bureaucrat]
→ rc18
% chan configure $chan -buffering line -blocking 0 -delay 1000
```

We set up a handler to be called when the channel is writable and implement it using an anonymous procedure that we define using our lambda helper. Because of the artificial flow control we implemented, we expect it will be triggered every second or so based on the value we set for -delay. It then writes out the current time to the channel.

```
% set counter 0
→ 0
% chan event $chan writable [lambda {chan} {
    if {[incr ::counter] > 2} {
        close $chan
        set ::done 1
        return
    }
    puts $chan [clock format [clock seconds] -format %T]
} $chan]
```

Similarly, our read handler prints out the time the message is received. Strictly speaking we do not need the end-of-file check since we are writing and reading into the same channel. When the writer closes the channel, no handlers will be called anyways once it is closed. But we include the check for completeness.

```
% chan event $chan readable [lambda {chan} {
    set message [gets $chan]
    if {$message eq "" && [chan eof $chan]} {
        close $chan
        return
    }
    puts "Received \"$message\" at [clock format [clock seconds] -format %T]"
    return
} $chan]
```

Finally, assuming we are running in tclsh without the event loop active, we fire it up.

```
% vwait ::done
→ Received "11:45:52" at 11:45:53
  Received "11:45:53" at 11:45:54
```

As we hoped, messages are sent at one second intervals and received after our configured delay.

17.3.3. Reflected channel limitations

There are some limitations in the functionality of reflected channels that we now outline.

The first and most important is that reflected channels are not associated with operating system level descriptors or handles and are invisible to other processes or even other non-Tcl threads. Therefore they cannot be passed to child processes, used for exec redirection and so on.

Secondly, reflected channels cannot support the half-closing of read-write channels as can socket channels. This limitation is actually not specific to reflected channels as even file based channels do not support half-close operations.

The third limitation is more subtle and involve blocking operation. Moreover, it is only specific to reflected channel implementations in which the same channel was used for both reading and writing within a single Tcl thread. Since this is the the type of reflected channel we used for illustrative purposes, we demonstrate with the following short example code.

```
set chan [chan create {read write} ::bureaucrat] → rc19
```

Note the channel is a blocking channel. Given it has no content, we expect the following read to permanently block. What happens instead is that we get an empty string back and if we call chan block, it indeed indicates no data is available.

```
read $chan 1        → (empty)
chan blocked $chan → 1
chan eof $chan      → 0
```

As per the Tcl reference page for read, for blocking channels our read should have only returned the empty string if the channel was at end-of-file. If no data was available, the command should have blocked.

This behaviour could be indeed be seen as a bug in our implementation. However, it is also exhibited by similar reflected channel implementations like fifo in Tcllib [10]. The issue, and reason why it cannot be fixed, is as follows. When our channel implementation's read subcommand is invoked and no data is available, there is no way for it to block **while still allowing the same thread to run and write to the channel to unblock it**. Use of commands like vwait for this purpose would violate blocking semantics. We are thus stuck between a rock and a hard place resulting in the seen behaviour.

It is worth reiterating that this flaw is only present in specific types of reflected channels. Those backed by a real operating system device, or one in which data is always available, such as the random channel in Tcllib that sources random byte sequences, will not have this blocking issue. Moreover, even for those that do, an application can work around is by checking chan blocked on empty results and not assuming end-of-file.

Nevertheless, these channels are best utilized in non-blocking event-driven mode.

17.3.4. Applications of reflected channels

There are a number of useful applications of reflected channels available in Tcllib. We list some of these here.

- tcl::chan::null implements a write-only channel that simply discards anything written to it.
- tcl::chan::zero implements a read-only channel that returns a continuous stream of null characters.
- tcl::chan::string implements a read-only channel that returns the contents of a string. This is useful when a command only operates on channel content and you want it to process the data in a string.
- tcl::chan::fifo is similar to our bureaucrat without its delay and flow control artifacts.
- tcl::chan::cat implements a read-only channel that allows reading of multiple channels via a single channel. The following code will slurp all content of files with an extension of txt in the current directory.

```
package require tcl::chan::cat
set chan [tcl::chan::cat {*}[lmap fn [glob *.txt] {open $fn}]]
set content [read $chan]
close $chan
```

---

[10] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

- `tcl::chan::random` implements a read-only channel that returns a continuous stream of random bytes.

```
package require tcl::chan::random              → 1
package require tcl::randomseed                → 1
set chan [tcl::chan::random [tcl::randomseed]] → rc20
bin2hex [read $chan 4]                         → 08 04 6e 1f
bin2hex [read $chan 6]                         → 22 2a 50 32 45 f1
close $chan                                    → (empty)
```

- `tcl::chan::variable` is similar to `tcl::chan::string` above except that the channel is writable as well and reflects the content of a variable.

```
package require tcl::chan::variable → 1.0.3
set var "abc"                       → abc
set chan [tcl::chan::variable var]  → rc21
puts -nonewline $chan "def"         → (empty)
flush $chan                         → (empty)
set var                             → def
close $chan                         → (empty)
```

- `tcl::chan::textwindow` implements a write-only channel that appends any content written to it to a Tk text window.

Tcllib [11] provides some other channels not listed above as well. In addition, it provides a pair of TclOO classes, `tcl::chan::core` reflected channel class and `tcl::chan::events` reflected channel class, which form the basis of the above implementations and can be used for your reflected channels as well.

17.4. Chapter summary

We introduced Tcl I/O in Chapter 9 and delved into the more advanced aspects of asynchronous I/O, channel transforms and reflected channels in this chapter. What remains with respect to I/O is a discussion of communication facilities in Tcl and that is where we will go next.

17.5. References

TIP219
> *Tcl Channel Reflection API*, Andreas Kupries, Tcl Improvement Proposal #219, http://www.tcl.tk/cgi-bin/tct/tip/219

TIP230
> *Tcl Channel Transformation Reflection API*, Andreas Kupries, Tcl Improvement Proposal #230, http://www.tcl.tk/cgi-bin/tct/tip/230

---

[11] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

18

Networking and Communications

> Communication is everyone's panacea for everything.
>
> — Tom Peters

And what's true for humans is true for software. Communication with other systems, whether local or over the Internet, is part of almost every modern application. Tcl provides support for communications at multiple levels and protocols:

- Raw TCP and UDP
- Application level protocols like HTTP, FTP
- Communication over serial ports

We will cover all of these in this chapter.

18.1. Network communications

Tcl's support for networking is for the most part limited to the Internet (IP) suite of protocols though other protocols suites such as IPX are available through platform-specific packages. Even for IP, the core Tcl command set only supports communication over raw TCP. Support for UDP and application level protocols like HTTP, FTP etc. is provided through additional packages.

18.1.1. IP addresses

The socket command used to create a TCP communication channel will accept both IPv4 and IPv6 format addresses .

Applications that need to explicitly parse or otherwise manipulate IP addresses can use the ip package from Tcllib[1]. For example,

```
% package require ip
→ 1.3
% ip::normalize 2404:6800:4007:807::2004 ❶
→ 2404:6800:4007:0807:0000:0000:0000:2004
% ip::contract 2404:6800:4007:0807:0000:0000:0000:2004 ❷
→ 2404:6800:4007:807::2004
% ip::version 2404:6800:4007:807::2004 ❸
→ 6
% ip::version 216.58.197.36
→ 4
```

❶ Canonical form for an IPv6 address
❷ Compact form for an IPv6 address
❸ IP version

---

[1] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

The package provides a number of other commands as well that are particularly useful for applications focused on network or system administration.

18.1.2. DNS names

Instead of IP addresses, DNS names may also be used with the socket commands. Tcl will internally resolve them to addresses as required.

The info hostname command can be used to retrieve the name of the local system.

```
info hostname → ares
```

Note however that the name returned may not be fully qualified.

Tcl does not itself provide a means to map a DNS name to an IP address but you can use the dns package from Tcllib[2] for this purpose.

```
% package require dns
→ 1.3.5
% dns::nameservers ❶
→ 8.8.8.8 8.8.4.4 192.168.1.1
```

❶ Retrieve the list of name servers configured for the system

To retrieved information for a DNS name you have to first obtain a handle for it using dns::resolve and check for success with dns::status.

```
% set tok [dns::resolve www.microsoft.com]
→ ::dns::4
% dns::status $tok
→ ok
```

You can then retrieve all records received in the DNS response by passing the received handle to dns::result.

```
% dns::result $tok
→ {name www.microsoft.com type CNAME class IN ttl 849 rdlength 35 rdata
   ↳ www.microsoft.com-c-2.edgekey.net} {name www.microsoft.com-c-2.edgekey.net type CNAME
   ↳ class IN ttl 14534 rdlength 55 rdata
   ↳ www.microsoft.com-c-2.edgekey.net.globalredir.akadns.net} {name
   ↳ www.microsoft.com-c-2.edgekey.net.globalredir.akadns.net type CNAME class IN ttl 897
   ↳ rdlength 24 rdata e1863.dspb.akamaiedge.net} {name e1863.dspb.akamaiedge.net type A class
   ↳ IN ttl 17 rdlength 4 rdata 23.66.237.138}
```

Alternatively, you can just query it for specific fields.

```
% dns::address $tok ❶
→ 23.66.237.138
```

❶ Note return value is a **list**, possibly with a single element

The dns::cleanup command must be called afterwards to release resources.

---

[2] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

```
→ dns::cleanup $tok
```

The dns package will use UDP to communicate with the DNS server if the `tcludp` package is available and TCP otherwise. This can be controlled via the `-protocol` option to the `dns::resolve` command.

> The dns package **only** looks up names via the DNS system. This may not be the same as how the host system resolves names. For example, a Windows system may use WINS in addition to DNS.

The package has a number of other commands and configuration options. See the reference documentation for details.

18.1.3. Text and binary protocols

It is particularly important to ensure the channel configuration settings are correct for the application protocol used over a network connection. Obviously, both the client and server ends must use the same settings.

Binary protocols

For binary application protocols that treat data as a sequence of bytes, the channel's `-translation` option must be set to `binary`. This will also have the desired side effect of setting the channel encoding to `binary` as well and turning off special handling of end-of-line and end-of-file characters. Usually

```
chan configure $chan -translation binary
```

suffices to set up the channel appropriately. Needless to say, the strings written to the channel must also be binary strings (see Section 4.13).

Text protocols

For text based protocols, the channel options to consider are `-translation`, `-encoding` and, rarely, `-eofchar`.

- The application protocol defines the character sequence for line endings and accordingly the `-translation` option must be set to `lf`, `cr` or as is most common, `crlf`.
- The default `-encoding` on the created channel is the system encoding. This is rarely correct since the systems at the two ends could very well be using different system encodings. Most modern protocols use "utf-8" while older ones like older ones like SMTP may use `iso8859-1`.
- For text protocols that are **line** oriented, it can be convenient to set the `-buffering` option to `line` mode. Otherwise, you need to remember to do a `chan flush` on the channel at appropriate times.
- Application protocols do not generally use character sequences to indicate end of file so the `socket` command's default of disabling this feature is usually appropriate.

A suitable configuration for HTTP might be

```
chan configure $chan -translation crlf -encoding utf-8
```

Note however that protocols like HTTP can require the encoding to be changed midstream, for example to transfer binary data.

18.1.4. Communicating over TCP

The *Transmission Control Protocol* (TCP) is a stream oriented protocol where the data being transferred is seen as sequence of bytes without any message boundaries. This is a natural fit for Tcl's I/O model and allows the standard channel commands like `puts`, `read` etc. to be used for communicating over TCP as well.

18.1.4.1. Writing TCP clients: `socket`

We will start off with the client end of TCP connections, describing the basic operation, asynchronous connects and various channel options specific to sockets.

The `socket` command is used by client applications to establish a TCP connection to a server.

```
socket ?-myaddr LOCALADDR? ?-myport LOCALPORT? ?-async? SERVER SERVERPORT
```

The command returns a read-write channel which can be used in the same fashion as was described for files and process pipes.

The *SERVER* parameter is the DNS name or IP address (either IPv4 or IPv6) of the server to which the connection is to be established and *SERVERPORT* is the port number on which the server is listening.

The address and port used for the local end of the connection will be automatically picked by the system. However, if required the application can specify the `-myaddr` and `-myport` options to bind the socket to a specific local address and port.

Like all channels, socket based channels must also be closed with the `close` / `chan close` command once the application is done with them. Moreover, as was described for process pipelines in Section 16.4, it is also possible to "half-close" a channel by shutting down only one direction of the two-way connection. For example,

```
close $so write
```

18.1.4.1.1. Connecting synchronously

If the `-async` option is not specified, the `socket` command will connect synchronously to the server and block until the connection is established.

Here is a simple example to retrieve a Web page.

```
% set http_req "GET http://www.example.com HTTP/1.1\n"  ❶
→ GET http://www.example.com HTTP/1.1

% set so [socket www.example.com 80]
→ sock0000000003CE1EE0
% chan configure $so -encoding utf-8 -translation crlf -buffering line  ❷
% puts $so $http_req
% close $so write  ❸
% read $so
→ HTTP/1.1 200 OK
  Cache-Control: max-age=604800
  Content-Type: text/html
  Date: Mon, 01 May 2017 06:34:57 GMT
  Etag: "359670651+gzip+ident"
  ...Additional lines omitted...
```

❶ Note the extra newline to indicate end of HTTP headers
❷ HTTP is line-oriented with CR-LF line endings
❸ Half close the channel so server knows we are done

One point to be noted above is the half-closing of the channel. We described half-closing in Section 16.4. Its purpose here is the essentially the same as we described there. For HTTP 1.1, the server allows the client to send multiple requests. Our half-close indicates to the server that no more requests will be arriving, allowing it to then close its end of the connection. That in turn will result in an EOF on our end of the channel causing our `read` to complete. Without the half-close, our unconstrained `read` would block until the server timed out and closed its connection.

18.1.4.1.2. Connecting asynchronously

By default, the socket command will block until the connection to the server is fully established. To have the connection be established asynchronously, the application can specify the -async option. The command will then initiate the connection but return right away before the connection is fully established.

> As of the time of writing, if SERVER specifies a host name as opposed to an IP address, the socket command is not completely asynchronous as the name lookup is done in synchronous fashion.

Note that even with -async specified, the channel is still in blocking mode so any attempt to do read from the socket will block until the connection is established **and** data is received from the other end.

Most commonly, an asynchronous connect is followed by a call to put the channel into non-blocking mode and registering read and write event handlers. When the connection is established, the write event is fired indicating that the channel may now be written. Conversely, the read handler will be invoked upon the arrival of data over the connection.

Let us rewrite our previous script in asynchronous style using non-blocking sockets. Like all our asynchronous and event driven I/O examples, this script also relies on the event loop and hence the vwait on the variable done. In a real application which already had the event loop active, this would not be required.

```
proc on_read so {
    set s [read $so]
    if {[string length $s] == 0} {
        if {[eof $so]} {
            close $so
            set ::done 1
        }
        return
    }
    puts $s
}

proc on_write so {
    puts $so "GET http://www.example.com HTTP/1.1\n"
    chan event $so writable {}
    chan close $so write
}

set so [socket -async www.example.com 80]
fconfigure $so -blocking 0 -encoding utf-8 -translation crlf -buffering line
chan event $so readable [list on_read $so]
chan event $so writable [list on_write $so]
vwait ::done
```

We have already seen similar code in Chapter 17 so we just point out the differences. The on_write handler will be called when the socket channel is ready to be written to, the first instance of which is when the connection to the remote server is established. It writes the GET request to the channel and then removes the write handler on the channel as we will not have anything more to write to it. We also close the socket for writes so the server knows we have nothing more to send.

When data arrives, the on_read handler is called which reads all data received so far and prints it, closing the socket on an end of file which indicates the connection is closed.

18.1.4.2. Writing TCP servers: `socket -server`

We now turn our attention to the other end of a network connection — the server side. A server starts listening on a port using a different form of the socket command.

```
socket -server COMMANDPREFIX ?-myaddr LOCALADDR? SERVERPORT
```

The command returns a channel for a socket that is listening on port *SERVERPORT*. If *SERVERPORT* is 0, the system will pick a random unused port whose value can be retrieved with the -sockname option to chan configure.

By default, incoming requests to any of the local system's addresses, both IPv4 and IPv6 (if supported by the system) will be accepted. The -myaddr option may be used to restrict accepted connections to those targeting a specific DNS name or IP address on the local system.

When a client's connection request is received, Tcl creates a **new** channel attached to the network connection for that client. It then invokes *COMMANDPREFIX* with three additional arguments: the new channel which is to be used to communicate with this client, the client's IP address and client port number.

Note the distinctions between the original channel returned by the socket command and the new channel:

- All communications occurs over the client channels created in response to received connection requests. The channel returned by a socket -server command cannot be used for any data transfer.
- Closing a client channel does not impact the listening channel. New connections will continue to be accepted. Likewise, closing the listening channel will prevent new connections from being accepted but will not impact existing connections in any way.
- The channel options, such as -buffering, -encoding, etc. have to be set individually for each client channel. They are **not** "inherited" from the listening socket.

Here is an example server that services multiple clients. It assumes a line oriented protocol and responds to each line from a client with the client's port number followed by the received line with the characters reversed.

We start off by defining the command that will invoked in response to every connection request. We want to be able to serve multiple clients so we make the accepted client channel non-blocking and attach a read handler.

```
proc on_accept {so client_ip client_port} {
    chan configure $so -buffering line -encoding utf-8 -blocking 0 -translation crlf
    chan event $so readable [list on_read $so $client_port]
}
```

Our read handler reverses the line and sends it back to the client. As a special case, an empty line from the client will end our server.

```
proc on_read {so client_port} {
    set n [gets $so line]
    if {$n > 0} {
        puts $so "$client_port: [string reverse $line]"
        return
    } elseif {$n == 0} {
        exit 0 ❶
    } elseif {[chan eof $so]} {
        close $so
    }
}
```

❶ Empty line → treat as exit server command (**any** client can shut down server)

Now we create the listening socket on a port. Instead of picking a port ourselves, we will let the system pick one for us by specifying 0 as the port number. Also we will only listen on our local loopback address. After creating the listening socket the server will sit waiting in the event loop.

```
set listener [socket -server on_accept -myaddr 127.0.0.1 10042]
vwait forever
```

Here is the entire server script.

```
# server.tcl
proc on_accept {so client_ip client_port} {
    chan configure $so -buffering line -encoding utf-8 -blocking 0 -translation crlf
    chan event $so readable [list on_read $so $client_port]
}

proc on_read {so client_port} {
    set n [gets $so line]
    if {$n > 0} {
        puts $so "$client_port: [string reverse $line]"
        return
    } elseif {$n == 0} {
        exit 0
    } elseif {[chan eof $so]} {
        close $so
    }
}

set listener [socket -server on_accept -myaddr 127.0.0.1 10042]
vwait forever
```

Let us run the server **in a separate process** from an interactive client.

```
% exec [info nameofexecutable] scripts/server.tcl &
→ 9072
```

When connecting, we take care to configure the client channels with the same options as the server end.

```
% set so [socket 127.0.0.1 10042]
→ sock0000000003F071B0
% chan configure $so -buffering line -encoding utf-8 -translation crlf
% puts $so abc
% gets $so
→ 53463: cba
```

And just to prove we can have multiple connections without the server getting blocked, start a second client connection.

```
% set so2 [socket 127.0.0.1 10042]
→ sock00000000041CB930
% chan configure $so2 -buffering line -encoding utf-8 -translation crlf
% puts $so2 def
% gets $so2
→ 53464: fed
%
% puts $so ""      ❶
% close $so
% close $so2
```

❶   Shut down the server

18.1.4.3. TCP connection state

The `chan configure` / `fconfigure` commands can return additional information for socket based channels. These are read-only options and shown in Table 18.1.

Table 18.1. Socket-specific configuration options

Option	Description
`-connecting`	Returns 1 if the socket is still in the process of connecting and 0 otherwise. This option is only supported for client side sockets and is intended to be used for asynchronous connects that specify the `-async` option. ```% set so [socket -async 127.0.0.3 9999] → sock00000000041CB930 % chan configure $so -connecting → 1```
`-error`	Returns any error status currently associated with a socket. ```% set so [socket -async 127.0.0.3 9999] → sock00000000041CB930 % wait 1000 ❶ % chan configure $so -error → connection refused``` ❶ Hang around a bit for the connect to fail
`-peername`	Returns a list of three elements containing the address, hostname, and port of the remote end of a connection. This option is not supported for a listening socket as it is not connected to a remote peer. ```% set so [socket www.microsoft.com 80] → sock000000000310BAC0 % chan configure $so -peername → 23.15.107.229 a23-15-107-229.deploy.static.akamaitechnologies.com 80 % close $so```
`-sockname`	For connected sockets this returns a list of three elements containing the address, hostname and port of the local end of a connection. For listening sockets, the return value is a flat (i.e. not nested) list that may contain more than one such triple corresponding to each address and port that is being listened on. ```% set so [socket -server accept 9999] → sock000000000431E2F0 % chan configure $so -sockname → 0.0.0.0 0.0.0.0 9999 :: :: 9999```

18.1.5. Communicating over UDP

As implied by its name, the *User Datagram Protocol* (UDP) is not a stream protocol like TCP. Data transfer in UDP takes place in the form of individual messages or *datagrams*. Aside from the fact that message delivery and sequencing is not guaranteed, UDP is not a good fit for Tcl's channel-based I/O model which has no concept of message boundaries or means to demarcate them. This must be kept in mind when using channels to communicate over UDP.

Because Tcl has no built-in support for it, one of several available extensions must be used for communicating over UDP. The one we will describe is the `tcludp` package available from https://sourceforge.net/projects/tcludp/ as it is the one most widely used and available on both Windows and Linux/Unix platforms.

```
package require udp → 1.0.11
```

18.1.5.1. Creating a UDP socket

A UDP socket is created with the udp_open command.

```
udp_open ?PORT? ?reuse? ?ipv6?
```

The command creates a UDP network socket and returns a channel tied to it. Unlike TCP, there is no notion of a "connection" in UDP so from a UDP protocol perspective there is no command variation to create a "client" or "server" side socket per se. Of course, at the application protocol level, a DNS "client" may make a request to a DNS "server" but at the UDP level there is no real distinction.

The optional *PORT* parameter specifies the port on which the socket will receive packets. If 0 or unspecified, the system will select an unused port for the application. This can then be retrieved with the custom -myport option to the fconfigure or chan configure commands.

By default, the system will not permit the same address and port to be used on multiple sockets. Specifying the reuse keyword as an argument sets the SO_REUSEADDR flag on the socket permitting such sharing.

Finally the ipv6 keyword will create an IPv6 socket instead of the default IPv4 one.

Let us create a few UDP sockets for us to play with.

```
set so1 [udp_open]                                      → sock788
set so2 [udp_open]                                      → sock820
set so3 [udp_open]                                      → sock816
chan configure $so1 -buffering none -translation binary → (empty)
chan configure $so2 -buffering none -translation binary → (empty)
chan configure $so3 -buffering none -translation binary → (empty)
```

For reasons we will delve into in Section 18.1.5.6, we turn off buffering on all channels and set the channels to binary mode.

Since we have not specified the port numbers, the system would allocate them for us. We can find the allocated ports via the -myport option.

```
set so1_port [chan configure $so1 -myport] → 57012
set so2_port [chan configure $so2 -myport] → 57013
set so3_port [chan configure $so3 -myport] → 57014
```

18.1.5.2. Sending UDP datagrams

Given that a UDP socket is not connected to a particular remote end point, it can be used to send to **any** remote UDP end point. This does mean that the receiving address and port have to be explicitly specified. Since Tcl's channel puts command does not have a means to specify this, it must be configured on the channel before doing the send. This is done by the -remote option to fconfigure / chan configure.

```
% chan configure $so2 -remote [list localhost $so1_port]
→ localhost 57012
```

The option takes a pair consisting of the remote hostname or IP address and port number. All outgoing datagrams on that channel will be sent to that end point until it is changed with another call to set the -remote option.

You can then send datagrams over the channel with the `puts` command. Each `puts` command results in a single datagram **as long as you follow the guidelines** in Section 18.1.5.6.

```
puts -nonewline $so2 "so2->so1 (0)"                    → (empty) ❶
puts -nonewline $so2 "so2->so1 (1)"                    → (empty)
fconfigure $so2 -remote [list localhost $so3_port] → localhost 57014
puts -nonewline $so2 "so2->so3 (0)"                    → (empty)
```

❶   Note we are writing strings to a channel configured to be binary. That is OK as long as strings are all ASCII.

Just as a UDP socket may send to multiple end points, a socket may also receive from multiple end points. Thus `so1` may receive data from `so3` as well.

```
fconfigure $so3 -remote [list localhost $so1_port] → localhost 57012
puts -nonewline $so3 "so3->so1 (0)"                    → (empty)
```

Note the use of `-nonewline` with all calls to `puts`. Like the buffering options, this is discussed in Section 18.1.5.6.

18.1.5.3. Receiving UDP datagrams

Datagrams received on a socket are received with the `read` command. The `gets` command is not recommended for reasons outlined in Section 18.1.5.6.

```
% puts "Received <[read $so1]> on so1"
→ Received <so2->so1 (0)> on so1
```

Each `read` returns a single datagram. Since a UDP socket may receive datagrams from multiple remote end points, we need to know the sender. This can be obtained with the `-peer` option to `fconfigure`. This will return a pair containing the remote address and port corresponding to the last read datagram.

```
chan configure $so1 -peer            → 127.0.0.1 57013
puts "Received <[read $so1]> on so1" → Received <so2->so1 (1)> on so1
chan configure $so1 -peer            → 127.0.0.1 57013
puts "Received <[read $so1]> on so1" → Received <so3->so1 (0)> on so1
chan configure $so1 -peer            → 127.0.0.1 57014
```

Our snippets above used blocking mode operation. As we described for some types of reflected channels in Section 17.3.3, UDP channels (as implemented by the `tcludp` package) also do not block if no data is available. They return an empty string instead. Calling `fblock` / `chan blocked` will then return 1.

```
read $so2          → (empty)
chan blocked $so2 → 1
```

As for reflected channels, it is then best to use UDP channels in event driven mode in the same fashion we described elsewhere for other channels.

18.1.5.4. Receiving broadcast datagrams

In order to receive datagrams that are broadcast on the network, the `-broadcast` option must be set to 1 on the UDP channel.

```
chan configure $so1 -broadcast 1 → 1
```

The above command will result in broadcast datagrams being received on that channel. Conversely, setting the option to 0 will stop reception of broadcasts on that channel.

18.1.5.5. Multicast operation

The `tcludp` package supports multicast operations for both IPv4 and IPv6 networks. To join and exit a multicast group, specify the `-mcastadd` and `-mcastdrop` configuration options respectively.

```
chan configure CHANNEL -mcastadd GROUPINFO
chan configure CHANNEL -mcastdrop GROUPINFO
```

Here `GROUPINFO` is a list of one or two elements, the first of which is the group address and the second optional element is a platform-specific network interface identifier.

The multicast groups that the channel is registered for can be obtained with the `-mcastgroups` option.

All commands return the multicast membership after execution of the command.

```
chan configure $so1 -mcastadd 224.1.1.1   → 224.1.1.1
chan configure $so1 -mcastadd 224.1.1.2   → 224.1.1.1 224.1.1.2
chan configure $so1 -mcastgroups          → 224.1.1.1 224.1.1.2
chan configure $so1 -mcastdrop 224.1.1.1  → 224.1.1.2
chan configure $so1 -mcastgroups          → 224.1.1.2
```

UDP channels have two additional options that are specifically useful for multicast operation. The `-mcastloop` option controls whether multicast datagrams **sent** on the socket are also placed on the channel's input. This option is `true` by default. The other option, `-ttl`, allows the *time to live* value to be specified. This controls the number of router hops the datagram may cross before it is dropped.

18.1.5.6. Best practices for UDP

Although not mentioned as such in the `tcludp` reference documentation, the author recommends you follow certain guidelines when using UDP channels as implemented by that package.

- Always configure the channel with `-buffering` set to none and `-translation` set to `binary`. Correspondingly, write only binary strings to the channel with any encoding etc. done explicitly via the `encoding` command.
- Do not register any channel transforms on a UDP channel.
- Use the `-nonewline` option when writing to the channel with `puts`. Each `puts` call will correspond to a single datagram.
- Receive datagrams from the channel with `read` command with no length argument specified. Each such `read` will return a single datagram.

The rationale for the above recommendations is summarized below.

UDP data transfer takes place through datagrams with are conceptually indepedent messages with clear boundaries. Tcl channels on the other hand are stream oriented where data is interpreted purely as a sequence of bytes. Channels therefore do not have, neither at the command level nor internally, any mechanisms to demarcate boundaries within the byte stream. Data written by the application is transformed, encoded, buffered and then sent on to the driver (`tcludp` in our case) as a logical stream of bytes through multiple "write" calls to the driver. There is no correspondence between the `puts` at the application level and the writes as seen by the driver. They may be merged or split in any fashion.

This then leads to the question of how are the datagram boundaries to be discerned by the `tcludp` driver. Having no other choice, it **assumes** that each write it receives from the Tcl I/O system is a single datagram. Since the application has no direct control over writes at that level, it has to somehow arrange for a single `puts` to map to a single `write`. Disabling buffering, encodings and line ending translation is the first step towards that. Even then, a `puts` call actually results in two writes from the Tcl I/O system to the channel driver with the first containing the data provided by the application and the second containing the implicit newline character appended by the `puts` command. This results in a spurious datagram containing the newline character by itself. The `-nonewline` option to `puts` avoids this problem.

On the input side, read is preferable to gets because the latter hides the boundaries between datagrams, something to which most datagram based protocols are not amenable.

18.1.6. Layered protocols

Protocols layered on TCP and UDP are supported by Tcl through additional packages and extensions. These include security protocols such as SSL/TLS as well as higher level application protocols, such as HTTP, FTP, SMTP etc. We will not cover them in this book but they are listed in Appendix A. There is also an entire book devoted to their use with Tcl — see Tcl 8.5 Network Programming.

Here is a short example of the most pervasive of these application protocols — HTTP. The supporting package, http, actually comes as part of Tcl itself though you do have to explicitly load it.

```
package require http → 2.8.9
```

The package only supports the client side of the HTTP protocol. The http::geturl command fetches a URL and returns a token to be used for further operations. It stores the results in an internal array whose elements can be retrieved with various commands.

```
% set tok [http::geturl http://www.example.com]
→ ::http::1
% http::status $tok ❶
→ ok
% http::code $tok ❷
→ HTTP/1.1 200 OK
% http::data $tok ❸
→ <!doctype html>
  <html>
  <head>
      <title>Example Domain</title>

  ...Additional lines omitted...

% http::cleanup $tok
```

❶   Whether the HTTP request completed
❷   The code returned by the HTTP server
❸   The body of the HTTP response

The package offers many other options for asynchronous operations, formatting queries, custom headers, proxy configuration and so forth.

18.2. Communication over serial ports

Before the Internet and Gigabit desktop connections, there were serial ports. In the time of dinosaurs, my son would say but the reality is that serial ports are still used in a wide variety of devices ranging from the humble RS-232 console ports still available on some servers to industrial control equipment. Tcl's channel-based I/O system includes support for communicating over these interfaces.

A channel to a serial port is opened with the same open command we described for opening disk based files.

```
open PATH ?ACCESS? ?PERMISSIONS?
```

Here PATH specifies the serial port. The ACCESS and PERMISSIONS parameters are exactly as we described for files in Section 9.3.2 so we will not describe them here. Similarly use of puts for output, gets and read for input and event driven I/O follows the same patterns we have described for the various channel types. We will therefore

stick only to those aspects like configuration options that are specific to serial ports. The discussion below assumes familiarity with serial port terminology and operation.

On Unix/Linux systems, serial port channels are opened with the by specifying a file system path that maps to a serial port. This is usually, but not necessarily of the form /dev/tty*N* where *N* is the serial port number. On Windows systems, serial ports are specified either in the form COM*N*: where *N* is a serial port number in the range 1-9 or in the form //./com*N* where *N* is **any** serial port number. Platforms other than Windows and Unix may use some other syntax to identify serial ports.

18.2.1. Serial port speed, parity, and bit lengths

The -mode option to chan configure is used to modify or retrieve the speed (baud rate), parity bits, data bits and stop bits settings for a serial port. The argument to the option consists of 4 values separated by commas in the form *SPEED*, *PARITY*, *DATABITS*, *STOPBITS* corresponding to those four settings. The *PARITY* value should be one of n, o, e, m or s signifying none, odd, even, mark or space respectively. The *DATABITS* value should be a number in the range 5-8. The *STOPBITS* value should be either 1 or 2.

18.2.2. Serial port flow control

Serial ports may use one of several handshaking mechanisms for implementing flow control to ensure input buffers are not overrun. The mechanism used can be configured with the -handshake option to chan configure. The configured value may be one of those shown in Table 18.2.

Table 18.2. Values for handshake configuration

Value	Description
none	Turns off any form of flow control.
rtscts	Enables hardware based flow control using the RTS and CTS control signals.
dtrdsr	*Windows only*. Enables hardware based flow control using the DTR and DSR control signals.
xonxoff	Specifies software based handshake. The pair of characters used for flow control can be then be specified with the -xchar option. The value should be a pair of characters used to enable and disable data transfer. The default value is the ASCII standard XON/XOFF pair.

The handshake can also be implemented by directly setting the state of the RTS and DTR control lines and querying the status of the CTS, DSR, RING and DCD lines.

The state of the outgoing control lines, RTS and DTR, is controlled with the -ttycontrol option. It is recommended that you not directly control these with the -handshake option set to rtscts or dtrdsr. The value for the option is a dictionary keyed by the signal names RTS and DTR with the corresponding element value set to 0 or 1 to turn the signal off and on.

The -ttycontrol option also permits setting or resetting the BREAK condition on the serial port by setting the value of the BREAK key in the dictionary passed as the option value.

Conversely, detection of the input control signals is done with the -ttystatus option. This returns a dictionary with the keys CTS, DSR, RING and DCD with the corresponding values being 0 or 1 reflecting the signal states.

18.2.3. Serial port buffer and queue sizes

On Windows and Unix, the current number of bytes in the input and output queues can be obtained with the read-only option -queue. This returns a pair containing the number of characters currently present in the input and output queues.

Further, on Windows systems the maximum size of the input queue buffer can be set with the -sysbuffer option. The value a list of one or two integers. The first specifies the size of the input buffer and the second, if present, the size of the output buffer.

18.2.4. Timers related to serial ports

There are two timers that can be configured for serial port channels. The first of these is available for both Windows and Unix and set through the -timeout option to chan configure. This sets the interval after which a blocking read operation will time out. The option value is specified in milliseconds. The granularity of the timer value is platform dependent.

The other timer is only applicable to Windows systems. It is set with the -pollinterval option and controls the maximum time between polling for file events. **Setting this value lower than the default 10ms Tcl uses to check for all types of events will also increase the frequency of the latter.**

18.2.5. Checking for serial port errors

The read-only chan configure option -lasterror returns more detailed error information for serial ports than the standard error codes used by Tcl's file I/O commands. Examples include receive buffer overruns, framing errors, etc. See the reference documentation for possible values, their causes and potential remedies.

18.3. Chapter summary

We started with the basics of file input/output in Chapter 9, described interprocess I/O in Chapter 16, delved into advanced operations involving asynchronous I/O, channel transforms and reflected channels in Chapter 17. With the discussion of communications in this chapter, we have now concluded our discussion of Tcl's extensive I/O facilities.

18.4. References

KOC2010

> *Tcl 8.5 Network Programming*, Kocjan, Beltowski, PACKT Publishing, 2010. Contains extensive discussion of building network-aware applications in Tcl including use of a wide variety of network related packages and libraries.

19

Virtual File Systems and Tclkits

The Virtual File System, or VFS, abstraction allows applications to view structured data as a hierarchy of directories and files even though it may not stored as such in a real file system. The data can then be operated on with the standard Tcl channel I/O commands. An example would be a VFS for a remote FTP site that would permit the application to open and perform I/O on a remote file in the same manner as a file on the local system.

Tcl's VFS framework also forms the basis of *tclkits*, which is a technology for deploying entire Tcl applications or packages as a single file without needing any installation steps or additional support files. This chapter describes the use of tclkits as well.

19.1. Using VFS

Although VFS itself is implemented within the Tcl core, accessing it at the script level requires the `tclvfs` extension. It permits both implementation of new VFS types in pure Tcl scripts as well as providing a number of VFS implementations for FTP, WebDAV, ZIP archives and others.

We first demonstrate the use of a VFS through the FTP based VFS available as part of the `tclvfs` extension.

```
% package require vfs::ftp
→ 1.0
```

The next step is to *mount* the VFS at a *mount point* which can be any file system path. By convention, `tclvfs` based packages provide a Mount command for this purpose.

```
Mount VFSPATH LOCALPATH
```

Here *VFSPATH* identifies the resource of interest within the VFS and *LOCALPATH* identifies a local file system path where the VFS resource will be made accessible. *LOCALPATH* need not exist and if it does, the file or directory corresponding to the path will not be accessible until the VFS is unmounted. The return value from the Mount command is a handle that is later required for unmounting the VFS.

In our FTP example, *VFSPATH* will identify a remote FTP directory.

```
% set mount_handle [vfs::ftp::Mount ftp://ftp.vim.org /ftp-vim]
→ 9
```

We can now treat the remote FTP directory like a local directory modulo some caveats that we list later. For example, we can even change the current directory to that location and list files glob or use file system inspection commands.

```
% cd /ftp-vim
% glob *
→ ftp mirror pub site vol
% file size pub/documents/published/books/stevens.netprog.errata.gz
→ 5885
```

We can open channels to files in the VFS and perform I/O on them, **even using channel transforms if desired.**

```
% set fd [open pub/documents/published/books/stevens.netprog.errata.gz rb]
→ rc6
% zlib push gunzip $fd
→ rc6
% gets $fd
→ ----------------------------------------------------
% gets $fd
→ Typos and errors found in "UNIX Network Programming"
% gets $fd
→ (Last updated March 19, 1995)
% close $fd
```

When a VFS is no longer required it should be unmounted. Again by convention each VFS supplies a command named Unmount to do the needful.

```
Unmount MOUNTHANDLE LOCALPATH
```

The *MOUNTHANDLE* argument passed to Unmount is the return value from the Mount command and *LOCALPATH* is the local mount point at which VFS resides. We can unmount our remote FTP directory as follows. Before we do that though, we will switch back to our original directory.

```
% cd ..
% vfs::ftp::Unmount $mount_handle /ftp-vim
→ 1
```

We could also have unmounted the VFS by calling the generic vfs::unmount command.

```
vfs::unmount /ftp-vim
```

In that case the mount point suffices to unmount and we do not really require the mount handle.

> In general, using cd to change the current directory to one located in a VFS is not a good idea. It creates a "split" view of the current directory since the operating system itself is unaware of the existence of the VFS. In any case, changing directories at any time, except possibly during application startup, is a bad idea even outside a VFS since it has process-wide effect.

At this point, we really need to step back and give the Tcl I/O system a big round of applause. Consider what we just did. The "application" code treated a **compressed, remote FTP** resource as though it were a plain old local uncompressed file. That is pretty, well, cool!

Before moving on, there are some additional finer points to be noted about VFS.

* A VFS is mounted **process-wide**, meaning all Tcl interpreters within the process see the mount.
* All C code that uses the Tcl file system or channel API's will also see VFS mounts. This means that extensions that use these API's (and not the C runtime routines) will also benefit from being able to treat VFS resources as normal files without any additional coding. So for example, Tk could display images from a VFS exactly as it would from a local file.

By the same token, there are also some caveats,

* A VFS is invisible to the operating system, other processes and any code even within the same process that does not use the Tcl file system API's. For example, you cannot pass an executable stored in a VFS to the exec command to run it since the operating system is unaware of the VFS. Note however, that Tcl's load command is VFS-aware so you can load a shared library from a VFS. Tcl will copy the shared library to a temporary location on the local file system and pass that path to the operating system loader.

- There are unsurmountable differences between different VFS types that you have to be aware of. For example, not all may support links or may differ in terms of their case-sensitivity (consider a remote FTP VFS on Unix accessed from a Windows client).

- VFS implementations may have some performance-related aspects you need to be aware of. For example, reading even a single byte from a FTP based VFS will cause the entire remote file to be retrieved and loaded into memory.

Even with these caveats, for the most part VFS provides a very generalized way of accessing different resources with the standard Tcl file system and channel commands.

19.1.1. URL mounts

An alternative way of mounting a VFS is based on a URL type. Naturally, this is only possible for virtual file systems based on URL's like FTP or HTTP. The vfs::urltype package implements this functionality.

```
% package require vfs::urltype
→ 1.0
```

As before, the Mount command is used to mount a specific URL type. It takes a single parameter, the URL type.

```
vfs::urltype::Mount URLTYPE
```

So running the following command

```
% vfs::urltype::Mount ftp
→ Mounted at "ftp://"
```

will result in Tcl treating ftp:// as an additional volume. We can verify this as below.

```
% file volumes
→ ftp:// C:/ D:/ E:/ F:/
% file split ftp://foo/bar
→ ftp:// foo bar
```

Now **any** FTP based URL can be treated as just another file path. So we could have written our previous example using this method as well.

```
→ set fd [open ftp://ftp.vim.org/pub/documents/published/books/stevens.netprog.errata.gz rb]
  rc8
% zlib push gunzip $fd
→ rc8
% gets $fd
→ --------------------------------------------------
% gets $fd
→ Typos and errors found in "UNIX Network Programming"
% close $fd
```

When unmounting a urltype VFS, simply specify the URL type which serves as the mount handle as well.

```
% vfs::urltype::Unmount ftp
% file volumes
→ C:/ D:/ E:/ F:/
```

19.2. Implementing a VFS

We now turn our attention to implementing a new type of VFS. We will create a VFS that acts as an in-memory file system. Since our purpose is to describe the `tclvfs` interfaces, our virtual file system is very simplistic.

As for channel transforms and reflected channels, implementing a VFS involves writing a handler that implements a set of subcommands for the operations defined by the Tcl file system component. All subcommands take the same first three arguments, shown as *ROOT*, *RELPATH* and *ORIGPATH* in the table below. Here *ROOT* is VFS mount path. In our example in the previous section, this would be `/ftp-vim`. *RELPATH* is the rest of the path (relative to *ROOT*) so that *ROOT/RELPATH* is the full absolute path. *ORIGPATH* is the path as originally specified in the invocation of the Tcl I/O command. This is translated to *ROOT/RELPATH* through normalization.

The subcommands that need to be implemented are shown in Table 19.1.

Table 19.1. VFS driver subcommands

Subcommand	Description
access *ROOT RELATIVE ORIGPATH MODE*	Called to check if the specified access mode is compatible with permissions on the given path.
createdirectory *ROOT RELATIVE ORIGPATH*	Called to create a directory.
deletefile *ROOT RELATIVE ORIGPATH*	Called to delete the specified file.
fileattributes *ROOT RELATIVE ORIGPATH* ?*INDEX*? ?*VALUE*?	Called to retrieve file attribute names or set their values.
matchindirectory *ROOT RELATIVE ORIGPATH PATTERN TYPES*	Called to retrieve files matching the specified pattern and type.
open *ROOT RELATIVE ORIGPATH*	Called to open a channel to the specified file.
removedirectory *ROOT RELATIVE ORIGPATH RECURSIVE*	Called to delete the specified directory.
stat *ROOT RELATIVE ORIGPATH*	Called to retrieve file information.
utime *ROOT RELATIVE ORIGPATH ATIME MTIME*	Called to set the access and modification time of a file.

We will now provide an example implementation of a virtual file system. We will implement the commands shown in the table above and make use of the `vfs::filesystem mount` and `vfs::filesystem unmount` commands for mounting and unmounting our file system.

Our `memfs` package will implement an in-memory virtual file system. Mounting a `memfs` will create a new VFS instance with no content. Any content written to this file system will be erased when the VFS instance is unmounted. An application may mount multiple `memfs` VFS instances, each independent of the others.

We require the `vfs` package which exposes Tcl's VFS features at the script level. We will also need the `tcl::chan::variable` package which we will use to implement channels targeting our VFS.

```
package require vfs
package require tcl::chan::variable
```

We will choose to implement our VFS using namespaces as we did for our virtual channel example and define an ensemble corresponding to the VFS driver subcommands.

```
namespace eval memfs {
    namespace ensemble create -parameters {fs_id} -subcommands {
        access createdirectory deletefile fileattributes
        matchindirectory open removedirectory stat utime
    }
}
```

> A minor point is worth noting about our namespace ensemble definition. Since we support multiple instances of our VFS, we need to pass an instance identifier to these subcommand procedures in addition to the arguments passed by the VFS core. The VFS core takes a single command prefix and appends its own arguments to it. Our instance identifier will be part of the command prefix and hence will appear **before** the subcommand. We define the ensemble accordingly with the -parameters option to indicate the subcommand position. See Section 12.6 for details.

19.2.1. Signalling VFS errors

Let us first deal with signalling of errors as the VFS subsystem expects implementations to use a specific call for this purpose as opposed to directly raising exceptions using the standard Tcl error or throw commands. This is to ensure a consistent set of error messages and error codes irrespective of the underlying file system.

Errors within the VFS driver should be signalled by calling the vfs::filesystem posixerror command and passing it a numeric code representing a POSIX error. This command will then raise a standard Tcl exception with an appropriately formatted error code. Since numeric codes are hard to remember, we will define a wrapper, posix_error, that will also accept the mnemonic equivalent of numeric error codes.

```
proc memfs::posix_error {err} {
    if {![string is integer -strict $err]} {
        set err [::vfs::posixError $err]
    }
    vfs::filesystem posixerror $err
}
```

So for example, we can call

```
posix_error ENOENT
```

to report an error that a file or directory does not exist.

19.2.2. Mounting and unmounting

Following vfs package conventions, we will name our command for mounting a memfs VFS as Mount.

```
proc memfs::Mount {mount_path} {
    set id [init_fs]
    vfs::filesystem mount $mount_path [list [namespace current] $id]
    vfs::RegisterMount $mount_path [list [namespace current]::Unmount $id]
    return $id
}
```

It takes a single argument which is the mount point where our file system will be located. It calls an internal command init_fs to create a new file system. The vfs::filesystem mount command then mounts the file system at the specified location. It has the general syntax

```
vfs::filesystem mount ?-volume? PATH CMDPREFIX
```

Here *PATH* is the mount point and *CMDPREFIX* is the command prefix that implements the VFS driver. In our example, this is the ensemble command we defined earlier which has the same name as our implementation namespace. Notice we also pass in the memfs instance identifier returned by init_fs command. This allows our implementation to distinguish between multiple memfs instances.

The -volume prefix specifies that the file system is also a new volume as seen by the file volumes command. We saw an example of such a VFS with the vfs::urltype::ftp example earlier. This option must not be specified for file systems, like memfs that will mount within an existing native file system path.

The call to vfs::RegisterMount is strictly not necessary. However, it allows the application to unmount out VFS by passing the mount point to the generic vfs::unmount command instead of having to call our VFS-specific Unmount command.

Next we implement the corresponding command for unmounting a memfs VFS instance.

```
proc memfs::Unmount {fs_id mount_path} {
    variable file_systems
    if {![info exists file_systems($fs_id)]} {
        return
    }
    vfs::filesystem unmount $mount_path
    unset file_systems($fs_id)
    namespace delete $fs_id
    return
}
```

The main point to note here is the call to vfs::filesystem unmount. The rest of the code is specific to our implementation and essentially deletes all data associated with that VFS instance. Instead of directly calling our Unmount command, it is recommended applications should call vfs::unmount so that it can update its database of mounted file systems.

> In any case, applications should not call the vfs::filesystem unmount command themselves. That command will remove the file system from Tcl's view but will **not** notify the VFS driver that the file system has been unmounted.

19.2.3. VFS operations

We now move on to implementation of the driver commands called by the Tcl VFS subsystem for operating on a file within the VFS. The first four arguments to all these commands are identical. The first is the memfs VFS instance id (which we passed as part of the command prefix), the mount point path, the relative path within the VFS, and the original path as specified in the command that invoked the file operation. For example, if our VFS was mounted at /tmp/mem and the current working directory was /tmp, then a command like

```
file exists mem/foo.txt
```

would result in the arguments after the memfs instance id being /tmp/mem, foo.txt and mem/foo.txt respectively.

19.2.3.1. Checking access: access

We start off with implementation of the access command. This is used by the VFS subsystem to check if a specific file or directory can be accessed with the specified mode. The mode is passed as an additional argument in the form of a bitmask that indicates the type of desired access. If 0, only existence of the file is to be checked. The low three bits signify execute (least significan bit), write and read access respectively. Our file system does not implement access permissions and so for directories we will allow all access types. For files however, we will disallow execute access since the operating system has no knowledge of our file system and cannot execute files residing in it. (At the Tcl level, exec will not work.)

```
proc memfs::access {fs_id root relpath origpath mode} {
    switch -exact -- [node_type $fs_id $relpath] {
        "" { posix_error ENOENT }
        file { if {$mode & 1} { posix_error EACCES } }
        dir { }
    }
    return
}
```

The implementation uses our internal node_type command to check for whether the path is a regular file or directory. If the specified access is allowed the command returns normally with the return value being immaterial. If access is **not** allowed, the command must raise a POSIX error as discussed previously. In our case, we return ENOENT when the path does not exist and EACCES when it does not have the requested execute access permission.

19.2.3.2. Creating directories: `createdirectory`

Next we implement the `createdirectory` call which is straightforward. The command is expected to create a new directory at the specified path within our file system if it does already exist. If the path corresponds to an existing regular file, it should raise a POSIX error.

```
proc memfs::createdirectory {fs_id root relpath origpath} {
    node_add_dir $fs_id [node_find $fs_id $relpath dir]
}
```

The implementation uses two internal commands. The first of these, node_find, maps a file path to the location key for the corresponding node in our internal file system structures. The node itself need not exist but if it does, the optional third argument mandates that it must be of the specified type. Thus in the above call, node_find would raise a POSIX error if the node existed and was not a directory. We will see node_find used throughout our implementation.

The other internal command, node_add_dir simply creates an empty directory and associated structures at a specified node location.

19.2.3.3. Deleting directories: `removedirectory`

The removedirectory command deletes a directory. It takes an additional argument which must be a boolean value. If true then the directory and its contents should be recursively deleted; otherwise the command should raise a POSIX error if the directory is not empty. It is not an error if the directory does not exist.

```
proc memfs::removedirectory {fs_id root relpath origpath recursive} {
    node_del_dir $fs_id [node_find $fs_id $relpath dir] $recursive
}
```

The implementation is pretty much identical to that of createdirectory above except we call node_del_dir which does the hard work of deleting the directory structure and its contents.

19.2.3.4. Creating and opening files: open

The open VFS driver subcommand essentially has to implement the file system level operations of the Tcl open and chan open commands which applications use to create files as well as open them for I/O. The command takes two additional arguments that specify the access mode and permissions in the case of creating a new file. These are equivalent to the ones passed to the Tcl open command.

```
proc memfs::open {fs_id root relpath origpath mode perms} {
    variable file_systems

    set node_key [node_find $fs_id $relpath file]
    set exists [expr {[node_type $fs_id $relpath] ne ""}]
    set truncate 0
    switch -glob -- $mode {
        "" -
        r* {
            if {! $exists} {
                posix_error ENOENT ❶
            }
        }
        a* -
        w* {
```

```
            if {$exists} {
                if {[string index $mode 0] eq "w"} {
                    set truncate 1
                }
            } else {
                node_add_file $fs_id $node_key
            }
        }
        default {
            error "Unsupported mode \"$mode\""
        }
    }
    set chan [node_add_channel $fs_id $node_key $truncate]
    set close_callback [list [namespace current]::node_close_handler  \
                            $fs_id $node_key $chan]
    return [list $chan $close_callback]
}
```

❶   File must exist

The implementation of open is a little longer than others but should be straightforward to follow at this stage. We have already described the node_find and node_type internal commands. The mode parameter, as for the Tcl open, specifies the open mode in the form of r, r+, w etc. and the switch statement takes appropriate action depending on the mode. The node_add_file command creates a new node of type file in our file system structure.

The return value of the command should be a list of one or two elements. The first element must be the handle to an open channel to use for performing I/O on the file. The second element is optional. If present it should be command prefix to be invoked when the channel is closed. Our internal node_add_channel command creates a new channel to a specified file (node). It also adds the channel to the list of channels internally associated with the file since we want to prevent files from being deleted while they have channels open to them. For the same reason, we also need to know when a channel is closed so that we can remove it from this list. Thus we make use of the second optional element in the return value to declare a callback to be invoked on channel close. This callback node_close_handler will remove the channel from the channel list and also update the access and modification time stored for the node (file).

19.2.3.5. Deleting files: deletefile

The deletefile command is almost identical to the removedirectory command we implemented above except it deals with removal of files, not directories, and hence has no need for recursion control.

```
proc memfs::deletefile {fs_id root relpath origpath} {
    node_del_file $fs_id [node_find $fs_id $relpath file]
}
```

A design decision we have made is that files with open channels to them cannot be deleted. This follows the Windows model (as opposed to Unix). Our node_del_file implementation will report an error via posix_error on an attempt to delete a file which is open. Naturally, this decision affects removedirectory as well.

19.2.3.6. Setting file timestamps: utime

The utime command is called to set the last access and modification timestamps for a file or directory. It takes two additional arguments corresponding to the access and modification time respectively. These are specified in terms of the number of seconds since the epoch, January 1, 1970.

```
proc memfs::utime {fs_id root relpath origpath atime mtime} {
    node_set_times $fs_id [node_find $fs_id $relpath] $atime $mtime
}
```

The internal commands in our implementation set these timestamps for various operations. For example, creation of a file will also update the modification time for its parent directory. The utime command is specifically called by Tcl in response to the file atime and file mtime commands.

19.2.3.7. File statistics: stat

The stat command is returns information about a file or directory. The return value from the command should be a dictionary with the following keys: dev, ino, mode, nlink, uid, gid, size, atime, mtime, ctime and type. These keys have exactly the same semantics we described for the file stat command.

```
proc memfs::stat {fs_id root relpath origpath} {
    return [node_stat $fs_id [node_find $fs_id $relpath]]
}
```

Our node implementation maintains the relevant information internally and our driver API can again just delegate to it.

19.2.3.8. File attributes: fileattributes

As discussed in the Files and Basic I/O chapter, files can be associated with certain attributes that are specific to the file system. These attributes are managed with the file attributes commmand. VFS drivers need to implement a corresponding fileattributes command to allow Tcl to access attributes for files within the VFS. The command is invoked in three forms:

- If no additional arguments (other than the standard ones) are specified, the command should return a list of attribute names supported by the file system. **Every such call must return the same names in the same order.**

- If a single additional argument is specified, it will be an integer index into the list of attribute names. The command should return the value of this attribute.

- If two additional arguments are specified, the first is the integer index into the attribute list as above. The second is the value to assign to the attribute.

Our implementation is shown below.

```
proc memfs::fileattributes {fs_id root relpath origpath args} {
    set attr_names [lsort [node_attr_names]]
    if {[llength $args] == 0} {
        return $attr_names
    }
    set node_key [node_find $fs_id $relpath]
    set attr_name [lindex $attr_names [lindex $args 0]]
    if {[llength $args] == 1} {
        return [node_attr $fs_id [node_find $fs_id $relpath] $attr_name]
    } else {
        return [node_attr $fs_id [node_find $fs_id $relpath] $attr_name [lindex $args 1]]
    }
}
```

As always, it relies on the underlying internal functions to to the actual work. The node_attr_names command returns a list of attribute names supported by our VFS. We ensure we always pass them back in the same order by sorting them before returning. We then use the node_attr command to get or set the attribute on the node corresponding to the specified path.

Although not shown above, our VFS supports two attributes: -contenttype and -encoding. Applications can assign any values they want to these attributes as VFS has no interest or control over their semantics. The **intended** use is for applications to store the encoding used for the file content in the -encoding attribute and the content type (similar to the Content-Type HTTP header) which specifies the format of the content like text/html

in the -contenttype attribute. Note these are advisory attributes and have to be explicitly set by the application There is no way for the VFS to detect the encoding in use or the type of content stored in the file.

19.2.3.9. Matching files: matchindirectory

One final VFS driver command remains to be implemented. The matchindirectory command is what Tcl's file system calls to retrieve directory contents. The return value of the command is expected to be a (possibly empty) list containing file names. Commands like glob also support matching based on patterns and file types. Correspondingly, the matchindirectory command takes two additional arguments that specify a glob pattern and a file type specifier.

If the glob pattern is the empty string, the relpath argument is the relative path of a file or directory. The command should then only check if the path exists **and** is of the specified type. If so, it should return a list containing the corresponding **original path** that was specified on the command line, not the relative path. If the path does not exist or is not of the specified type, an empty list is returned.

If the glob pattern is not the empty string, the relpath is always a path to an existing directory whose contents are to be matched against the pattern. The command should then return the names that match the pattern and the specified type.

```
proc memfs::matchindirectory {fs_id root relpath origpath pat type} {
    variable file_systems
    if {[string length $pat] == 0} {
        set file_type [node_type $fs_id $relpath]
        if {($file_type eq "dir" && [::vfs::matchDirectories $type]) ||
            ($file_type eq "file" && [::vfs::matchFiles $type])} {
            return [list $origpath]
        } else {
            return {}
        }
    }

    if {[node_type $fs_id $relpath] ne "dir"} {
        return {}
    }

    set node_key [node_find $fs_id $relpath]
    set matches {}
    if {[::vfs::matchDirectories $type]} {
        foreach name [node_subdirs $fs_id $node_key] {
            if {[string match $pat $name]} {
                lappend matches [file join $origpath $name]
            }
        }
    }
    if {[::vfs::matchFiles $type]} {
        foreach name [node_files $fs_id $node_key] {
            if {[string match $pat $name]} {
                lappend matches [file join $origpath $name]
            }
        }
    }
    return $matches
}
```

Our implementations makes use of two internal commands node_files and node_subdirs that return the names of files and subdirectories under the specified node. These are then matched against the pattern to construct the returned list.

The type argument specifies whether the returned list should include files, directories or both. We treat it as opaque and use the utility commands vfs::matchDirectories and vfs::matchFiles to ascertain whether

entries of a specific type are to be included or not. Note that the type argument can indicate that **both** files and directories are to be included.

The Tcl glob command allows the type argument to limit matching files based on other criteria as well such as permissions. These criteria are not exposed for VFS systems.

The vfs package provides another utility command vfs::matchCorrectTypes that serves as an alternative to matchDirectories and matchFiles.

```
vfs::matchCorrectTypes TYPES FILELIST ?DIR?
```

The command returns a list of names from *FILELIST* that fulfil the type requirements specified by *TYPES*. If *DIR* is not specified, *FILELIST* must contain absolute paths. If *DIR* is specified, it must be a directory and *FILELIST* should be the list of file and directory names within that directory. Depending on your internal file system implementation, you may find matchCorrectTypes more convenient to use.

19.2.3.10. memfs internals

Due to space limitations, we will not go into detail regarding our node_* commands that implement our VFS internals. You can download the memfs.tcl file from the book's web site to see the implementation.

Time to see if our VFS actually works.

```
% package require memfs
→ 1.0
% memfs::Mount /mem
→ 1
% file mkdir /mem/dir
% set fd [open /mem/dir/foo.txt w]
→ rc24
% puts -nonewline $fd "It lives!"
% set enc [fconfigure $fd -encoding]
→ cp1252
% close $fd
% glob /mem/dir/*
→ C:/mem/dir/foo.txt
% file attribute /mem/dir/foo.txt -contenttype text -encoding $enc
```

It appears the file was created. Let us read it back. We stored the encoding used to write it as a file attribute. So we will configure the channel accordingly while reading.

```
% set fd [open /mem/dir/foo.txt]
→ rc25
% fconfigure $fd -encoding [file attribute /mem/dir/foo.txt -encoding]
% read $fd
→ It lives!
```

An attempt to delete the file should fail as we have the file open. Retrying after closing the file should allow the delete operation to succeed.

```
file delete /mem/dir/foo.txt Ø error deleting "/mem/dir/foo.txt": permission denied
file exists /mem/dir/foo.txt → 1
close $fd                     → (empty)
file delete /mem/dir/foo.txt → (empty)
file exists /mem/dir/foo.txt → 0
```

Finally we verify we can unmount our file system.

```
% vfs::unmount /mem
% file exists /mem
→ 0
```

It's all good! Ship it!

19.3. VFS introspection

The set of current VFS mounts can be retrieved with the `vfs::filesystem info` command. With no arguments, the command returns the list of mount points.

```
% memfs::Mount /mem
→ 1
% vfs::ftp::Mount ftp://ftp.vim.org /ftp-vim
→ 0
% vfs::filesystem info
→ C:/ftp-vim C:/mem
```

If the optional argument is specified, it must be a mount point path. In this case the command returns the VFS driver command prefix that will be invoked to handle requests to that file system.

```
% vfs::filesystem info /mem
→ ::memfs 1
% vfs::filesystem info /ftp-vim
→ vfs::ftp::handler 0 {}
```

19.4. Single file deployment: Tclkit, Starkit, Starpack

Except in the simplest cases, deploying an application or even a large library package involves distribution of

- The Tcl interpreter itself, e.g. `tclsh` or a custom built executable
- The Tcl script files comprising the application
- Any packages, modules and binary extensions required
- Support files such as icons and other resources

The presence of multiple files in a directory hierarchy means deployment cannot be a simple copy and run operation. The files have to combined into some archive or installation format for distribution and then unpacked or installed on the target system. Although this can be an inconvenience, there is another more irksome issue for the user. If multiple unrelated Tcl applications are installed, there is potential for interference between the applications. Each may require different versions of Tcl and libraries, expect different settings in environment variables like `TCLLIBPATH` and so on.

The Tclkit technology is a solution that makes deployment as simple as copying a single file and running it with no additional steps required. Moreover, the application is completely self-contained and will have no interference with other Tcl installations including other Tclkit based applications.

Single file deployment alternatives

There are several alternative solutions for single file deployment similar to that of Tclkit. Here we describe Tclkit because it is probably the most widely used and also the one with which the author is most familiar. However, other solutions may have features not present in Tclkit based applications. For example, Freewrap can optionally encrypt the contents of the deployed file.

19.4.1. Tclkits, starkits, starpacks

A *starkit* (**St**andalone **R**untime) is a way to package an entire directory hierarchy and its contents into a single file. Conceptually starkits are similar to file archival formats such as `tar` or `zip` and in fact there are starkit variations based on those archive formats as well. What sets the starkit apart is some additional internal structure that allows a starkit to be mounted as a VFS.

The internal format of starkit archives was originally based on a database called Metakit. There are now also variations that use other formats like the above-mentioned zip format. For our purposes, the internal formats are immaterial for the most part and we will refer to them collectively as starkits.

The `tclkitsh` and `tclkit` applications are specially enhanced versions of `tclsh` and `wish` respectively that understand the format of Metakit-based starkits. The other formats of starkits have their own corresponding applications. For example, the applications that understand the Vlerq format are `tclkit-cli` and `tclkit-gui`. All these applications, which we will collectively refer to as *tclkits*, mount the starkit file as a VFS.

> The tclkit programs can also be used in place of the `tclsh` and `wish` shells. See Section 19.4.3.

Starkits and tclkits together comprise a two-file solution for distributing Tcl applications. So, for example, an application packaged as a starkit `myapp.kit` can be run as

```
tclkitsh myapp.kit ARG …
```

However, we can go a step further. The tclkit and starkit can be further combined into a **single** executable file, termed a *starpack*. This is a self contained executable holding the Tcl interpreter and libraries as well as as the application code with its supporting files. A starpack `myapp` (`myapp.exe` on Windows) constructed from the `tclkitsh` and `myapp.kit` above can be executed simply as

```
myapp ARG …
```

19.4.2. Obtaining a tclkit

Tclkits are available from multiple sources on the Internet. We list only some of the more popular ones below. Although originally based the same technology, they have some differences in their internal structure and build systems.

19.4.2.1. Downloading prebuilt tclkits

The easiest way to obtain a prebuilt tclkit is to download the appropriate version for your operating system platform from one of the locations below.

- The KitCreator[1] project
- The Kitgen Build System[2] (KBS) project
- The ActiveState[3] distribution includes tclkit binaries. They are termed *basekits* and available in the `bin` directory of an ActiveTcl installation.

19.4.2.2. Building tclkits

Both KitCreator and KBS also provide build scripts that allow you build your own tclkit executables in case the prebuilt binaries do not support your platform or you want a custom version with a different internal format or

---

[1] http://tclkits.rkeene.org
[2] https://sourceforge.net/projects/kbskit/files/kbs
[3] http://tcl.activestate.com

additional packages. Both download the required source files from their repositories. Building a tclkit then simply involves invoking the appropriate script, `kitcreator` in the case of KitCreator and `kbs.tcl` in the case of KBS.

KitCreator has two additional features:

- It supports cross-compiling a tclkit for a different target platform than the one on which the build system is running.
- It has an online build system[4] where you can specify through a Web interface one of almost two dozen target platforms and select any additional extensions desired. It will then build a customized tclkit for download.

Both KitCreator and KBS are based on the GNU toolchain. For building on Windows you will need to install MinGW[5] or MinGW-w64[6]. Alternatively, for Microsoft Visual C based builds, you may prefer to instead use the original kitgen[7] system. In addition to the GNU toolchain, this also provides `nmake` based makefiles suitable for Visual C builds. See the README file at the toplevel for instructions.

Our examples below assume we are running a downloaded tclkit that we have renamed as `tclkit-cli.exe`.

19.4.3. Using tclkits as Tcl shells

Tclkit binaries can be for the most part used in place of the standard Tcl shells `tclsh` and `wish`. In particular,

- when run without arguments, they will display an interactive prompt and execute any commands entered
- you can pass a Tcl script file to run and additional arguments on the command line just as for the standard shells

These characteristics make it very convenient to use tclkit interpreters on systems where Tcl is not installed. You can copy a single executable and gain full use of a Tcl command shell.

There are however some differences.

- In addition to Tcl script files, tclkit programs will also execute starkits as we explore in a bit. As of Tcl 8.6, the standard shells do not have the requisite VFS drivers built-in and will not recognize the starkit formats.
- Because tclkits are self contained, they do not examine environment variables like the `TCLLIBPATH` environment variable when setting up the `auto_path` variable for locating packages.
- Some tclkit variations do not include a full set of time zone and character encoding data. If this matters to your application, it is simply a matter of downloading or building one that does include this data.

19.4.4. The `sdx` tool

As we described previously, a starkit is a directory tree and its content packaged as a single file. Although a starkit can be constructed using Tcl's base VFS facilities, most commonly the `sdx` tool is used for this purpose. This wraps all the initialization and low level operations required to build a starkit into a set of high level commands callable from a command line. You can download it from several Tcl sites, for example https://chiselapp.com/user/aspect/repository/sdx/index.

Because `sdx` is itself packaged as a starkit, you can only run it using using a tclkit, not with `tclsh`.

The `sdx` tool comes with a built in help system. The general syntax for running the tool is

```
tclkit-cli sdx.kit ?SDX COMMAND? ?COMMAND ARGS?
```

If no arguments are provided, it will print a summary of available commands.

---

[4] http://kitcreator.rkeene.org/kitcreator
[5] http://www.mingw.org
[6] https://mingw-w64.org
[7] https://github.com/patthoyts/kitgen

```
c:\temp\tclkit-demo> tclkit-cli sdx.kit
→ Specify one of the following commands:
  addtoc eval fetch ftpd httpd httpdist ls lsk md5sum mkinfo mkpack mkshow mksplit mkzipkit
  ↳ qwrap ratarx rexecd starsync sync tgz2kit treetime unwrap update version wrap
  For more information, type:  sdx.kit help ?command?
```

The help command offers more detailed information for each command.

```
c:\temp\tclkit-demo> tclkit-cli sdx.kit help qwrap
→
    Quick-wrap the specified source file into a starkit

          Usage: qwrap file ?name? ?options?

          -runtime    file    Take starkit runtime prefix from file

          Generates a temporary .vfs structure and calls wrap to create
          a starkit for you.  The resulting starkit is placed into file.kit
          (or name.kit if name is specified). If the -runtime option is
          specified a starpack will be created using the specified runtime
          file instead of a starkit.

          Note that file may be a local file, or URL (http or ftp).
```

You will notice the tool comes with wide-ranging functionality, even including a basic FTP and Web server. Our discussion will be limited to the functionality related to the topic at hand — working with starkits and starpacks.

19.4.5. Building a single script starkit: sdx qwrap

Let us start with the simplest possible example — building a starkit from a single script. We will create a starkit containing the ubiquitous Hello World! program. We first create the file containing our script.

```
% write_file hello.tcl {puts "Hello World!"}
```

We now convert this to a starkit using the sdx qwrap command. At the Windows command prompt,

```
c:\temp\tclkit-demo> tclkit-cli sdx.kit qwrap hello.tcl
→ 5 updates applied
```

We see that a starkit hello.kit has been created.

```
c:\temp\tclkit-demo> dir /b *.kit
→ hello.kit
  sdx.kit
```

We can run the created starkit with the tclkit-cli application.

```
c:\temp\tclkit-demo> tclkit-cli hello.kit
→ Hello World!
```

Now, this obviously not very different from running

```
c:\temp\tclkit-demo> tclsh hello.tcl
→ Hello World!
```

But hold on before you thumb your nose at this. We will see next how this can then be used to build a fully self-contained executable.

We will also see later that the starkit is not constrained to contain a single file. An entire directory structure with multiple packages, extensions, and auxiliary files comprising complete application can be included within the starkit.

19.4.6. Building a single script executable: `sdx qwrap -runtime`

Distributing a starkit implies also distributing a tclkit executable or having the end user obtain it from somewhere. We can do better by combining the tclkit executable and the starkit into a single executable file — a *starpack*.

First, we need to make a copy of our tclkit executable. We will go into the reasons for this when we discuss the structure of tclkits.

```
c:\temp\tclkit-demo> copy tclkit-cli.exe tclkit-cli-runtime
→          1 file(s) copied.
```

We then run the `sdx qwrap` program again but this time with the `-runtime` option.

```
c:\temp\tclkit-demo> tclkit-cli sdx.kit qwrap hello.tcl hello.exe -runtime tclkit-cli-runtime
→ 872 updates applied
  4 updates applied
```

And voila, we now have a `hello` program, our very first starpack!

```
c:\temp\tclkit-demo> dir /b hello.*
→ hello
  hello.kit
  hello.tcl
```

Notice a quirk of the `sdx qwrap` command. **Even on Windows the executable file is created without an** exe **extension.** So if you are on that platform, it needs to renamed appropriately.

```
c:\temp\tclkit-demo> rename hello hello.exe
```

We now have a fully functional single file executable that comprises our entire application.

```
c:\temp\tclkit-demo> hello
→ Hello World!
```

The convenience of this cannot be overstated. Deployment consists of copying a file to the target system and installation is a no-op. And as we will see as we proceed through the chapter, these benefits are not limited to toy applications implemented in a single file.

19.4.7. Internal structure of a starkit

The `sdx` utility's `lsk` command allows us to inspect the internal structure of a starkit.

```
c:\temp\tclkit-demo> tclkit-cli sdx.kit lsk hello.kit
→
  hello.kit:
                        dir  lib/
        80  2017/07/04 11:45:57  main.tcl

  hello.kit/lib:
```

```
                              dir  app-hello/

 hello.kit/lib/app-hello:
         54  2017/07/04 11:45:57  hello.tcl
         79  2017/07/04 11:45:57  pkgIndex.tcl
```

Moreover, the lsk can even list the contents of a starkit embedded in a starpack.

```
c:\temp\tclkit-demo> tclkit-cli sdx.kit lsk hello.exe
→
  hello.exe:
       5114  2013/02/16 22:53:30  boot.tcl
         37  2013/02/16 22:53:30  config.tcl
                              dir  lib/
         80  2017/07/04 11:45:57  main.tcl
      57022  2013/02/16 22:53:30  tclkit.ico

  hello.exe/lib:
                              dir  app-hello/
...Additional lines omitted...
```

Notice the contents of the starpack are much larger since it includes not just the hello application but also the Tcl runtime. The format of the output should give you a hint that starkits are structured just like file systems and in fact are implemented as a VFS.

We can proceed to examine the internal contents in one of two ways. The first is through the usual VFS and Tcl I/O commands. The second is by extracting the contents of the startkit with the sdx unwrap command.

Let us first demonstrate the former, primarily to prove that the starkit is accessible as a VFS. We need to use the tclkit-cli application, and not tclsh for this as the latter does not by default have the requisite VFS drivers. From within tclkit-cli we first load the Metakit VFS driver and then mount the starkit.

```
% package require vfs::mk4
→ 1.10.1
% vfs::mk4::Mount hello.kit /hello
→ mkclvfs1
% glob /hello/*
→ C:/hello/main.tcl C:/hello/lib
```

We see that top level directory of the VFS has two entries, main.tcl and lib. We can access any file within the VFS with the standard Tcl I/O commands.

The second way to examine the content of a starkit (or starpack) is by extracting its contents with the sdx unwrap command.

```
c:\temp\tclkit-demo> tclkit-cli sdx.kit unwrap hello.kit
→ 5 updates applied
```

This will extract the contents of the starkit into a local directory hello.vfs.

```
c:\temp\tclkit-demo> dir /s /b hello.vfs
→ C:\temp\tclkit-demo\hello.vfs\lib
  C:\temp\tclkit-demo\hello.vfs\main.tcl
  C:\temp\tclkit-demo\hello.vfs\lib\app-hello
  C:\temp\tclkit-demo\hello.vfs\lib\app-hello\hello.tcl
  C:\temp\tclkit-demo\hello.vfs\lib\app-hello\pkgIndex.tcl
```

We can take a peek at the content of the main.tcl file.

```
c:\temp\tclkit-demo> type hello.vfs\main.tcl
→
    package require starkit
    starkit::startup
    package require app-hello
```

We will leave the details about the commands in `main.tcl` for the next section where we build a more complete example of a starkit. For the moment we will comment on some general points points about the tclkit structure.

The root of the VFS contains a file `main.tcl`. This file will be sourced by the tclkit application when the starkit is loaded. The `main.tcl` file above was automatically created by the `sdx qwrap` command. There is no requirement that this file have exactly the contents shown. It could contain any sequence of commands or even an entire application.

The only mandated requirement for starkits is the existence of this `main.tcl` file at the root of the starkit VFS. The rest of the starkit VFS may be structured in any fashion you choose.

The structure of our `hello` starkit is the boilerplate used by `sdx qwrap` for its automatically generated starkits. It has a `lib` directory beneath which it expects all packages to be placed. The `starkit::startup` command in `main.tcl` will add the directories below this to the `auto_path` variable. In the case of `sdx qwrap`, there is no option for including additional packages so this directory contains only a single subdirectory `app-hello`.

This directory contains our `hello.tcl` script **converted to a package form** and loaded with the `package require` command in `main.tcl`. To convert our script to a package, `sdx qwrap` adds the `pkgIndex.tcl` file

```
c:\temp\tclkit-demo> type hello.vfs\lib\app-hello\pkgIndex.tcl
→
    package ifneeded app-hello 1.0 [list source [file join $dir hello.tcl]]
```

and modifies our script to include `package provide` command.

```
c:\temp\tclkit-demo> type hello.vfs\lib\app-hello\hello.tcl
→ package provide app-hello 1.0

    puts "Hello World!"
```

We reiterate at this point that this structure is completely optional. If we were to manually structure the VFS, something we illustrate in the next section, we could have just put our script file in the VFS root directory and directly sourced it instead of converting it to a package. Or we could have included our script within `main.tcl` itself. The structure used by `sdx qwrap` is reflective of the conventions used when a `starkit` is used to deploy multiple packages and more complex applications.

19.4.8. A more complete starkit example: `sdx wrap`

Having seen the creation of a starkit from a single script, let us now create a more complete example. Our demo starkit will be multifunctional: it will include the `sequences` package from Section 13.3.8 as well as the standard `Hello World!` functionality. We foresee great demand for this combination.

Our demo has certain operational requirements:

• The starkit must be usable as a library where it can be loaded in the main application.

• It should also be usable as a standalone application itself when it is passed as the script argument to a tclkit application. In that case it should run the command specified by the user.

• It should be deployable as a single file executable.

• For ease of development, we should be able to run it in normal fashion even when it is not wrapped into a starkit. That way we can edit the files during development and re-test without having to build a starkit after every change.

We will follow the same basic structure as was created by `sdx qwrap`. At the top level, we have the `demo.vfs` directory which will be the root of our starkit's virtual file system. The directories and files below this are shown below.

```
c:\demo-dir> ls -R demo.vfs
demo.vfs:
hello.tcl  lib  main.tcl

demo.vfs/lib:
app-demo  sequences

demo.vfs/lib/app-demo:
demo.tcl  pkgIndex.tcl

demo.vfs/lib/sequences:
pkgIndex.tcl  seq_arith.tcl  seq_geom.tcl
```

We will start with the mandatory `main.tcl` file in the root of the starkit VFS.

```
# main.tcl
namespace eval demo {}
package require starkit
set demo::run_mode [starkit::startup]
set demo::vfs_root [file dirname [file normalize [info script]]]

package require sequences
source [file join $demo::vfs_root hello.tcl]

puts "run_mode: $demo::run_mode"

switch -exact -- $demo::run_mode {
    sourced { }
    unwrapped -
    starkit -
    starpack {
        package require app-demo
    }
    default {
        error "Unknown run mode $demo::run_mode"
    }
}
```

The scripts creates a namespace demo for its own use and then loads the `starkit` package that implements the required VFS drivers built into a tclkit application. The next command is a call to `starkit::startup` which has two effects that are directly relevant to us:

- It initializes the `auto_path` variable used for loading packages to the `lib` directory in the VFS. You are of course free to further modify `auto_path` as appropriate for your application. For example, you might have another directory as a sibling of `lib` that you want to add to the package path.

- It returns a value that indicates how the starkit is being used. We use this to determine whether our starkit should behave as a bundle of packages or a standalone application. The possible return values are shown in Table 19.2.

Specific tclkit variations may return values other than those shown in the table; for example, `service` indicating the starkit is running as a Windows service. See the documentation for your `tclkit` for these additions.

Table 19.2. Starkit start-up modes

Mode	Description
unwrapped	Indicates that the `main.tcl` is **not** part of a starkit and is being read in as a regular Tcl script. As we will see below, it is convenient during development for the application to be in "unwrapped" form as a set of Tcl scripts instead of a monolithic single-file starkit.
sourced	Indicates that the starkit was loaded with the `source` command either from the application or another starkit. This would generally indicate that the starkit is not itself the main application.
starkit	The starkit is the main file being run by the tclkit application from the command line and as such should provide the application functionality.
starpack	The starkit is bound to the tclkit executable thereby comprising a single-file application.

We will see all these modes in our example scenarios below.

The script then preloads our starkit functionality. It loads the `sequences` package which is placed under the `lib` directory and therefore found via `auto_path`. We chose not to implement `hello` as a package so the script simply sources it. Note that instead of preloading these, we could have chosen to omit these lines and thereby leaving it up to the application to explicitly load them.

For the purposes of our demonstration, we then print out the mode that the starkit is running in.

Finally, the `main` script checks the mode. A value of `sourced` indicates that the starkit is being loaded as a library. Nothing more needs to be done in this case as we have already loaded the contained functionality. All other values of the mode indicate the the starkit should run as an application. It then loads our application code which is implemented, following starkit conventions, as a package.

Our `Hello World!` functionality is our simple script from the previous section written as a procedure.

```
# hello.tcl
proc hello {} {
    puts "Hello World!"
}
```

Finally, we come to our application code. It is simple enough that we just present it here without any explanation. Our example usage later will clarify working if needed.

```
# demo.tcl
package provide app-demo 1.0
package require sequences

if {[catch {
    switch -exact -- [lindex $argv 0] {
        hello { hello }
        arith { seq::arith_term {*}[lrange $argv 1 end] }
        geom { seq::geom_term {*}[lrange $argv 1 end] }
        default {
            error "Unknown or missing command: must be hello, arith, geom"
        }
    }
} result]} {
    puts stderr $result
    exit 1
} else {
    if {$result ne ""} {
        puts stdout $result
    }
}
```

We are now ready to actually build a starkit from our demo application. The `sdx` utility's `wrap` command will create a starkit from a specified directory tree.

```
tclkit-cli sdx.kit wrap NAME ?options?
```

The `sdx wrap` command takes several options but we only describe basic usage here. It creates a starkit named *NAME* whose contents are created from a directory of the same name but with a `.vfs` file extension.

```
c:\demo-dir> tclkit-cli sdx.kit wrap demo -interp tclkit-cli
10 updates applied

c:\demo-dir> ls
demo  demo.bat  demo.vfs  sdx.kit  tclkit-cli-runtime  tclkit-cli.exe
```

The command creates two files, the starkit demo and a Windows batch file `demo.bat`. The latter is simply a Windows batch file that invokes the tclkit application passing it the starkit that was created.

```
@tclkit-cli demo %1 %2 %3 %4 %5 %6 %7 %8 %9
```

On Windows we can then run the starkit through the demo batch file. On Unix systems, the starkit can be directly run because it begins with the header

```
exec tclkit-cli "$0" ${1+"$@"}
```

causing Unix shells to run the starkit by passing it to `tclkit-cli`.

The purpose of the `-interp` option when creating the starkit is to specify which tclkit application should be used to run the starkit when the starkit is directly invoked in the Windows or Unix command shell. We specified it as `tclkit-cli` as that is the tclkit application we are using. If unspecified, it would default to `tclkit`.

The starkit can be run using **any** tclkit application by explicitly passing it as the command line argument. The use of the `-interp` option only applies to the case where the starkit is specified as the program name in the shell command line.

We are now ready to try out our application in various modes. First, we will use it as a package bundle in interactive mode.

```
c:\demo-dir> tclkit-cli
% source demo
run_mode: sourced
% hello
Hello World!
% seq::arith_term 2 3 4
11
```

Notice the run mode is printed as `sourced`.

Next, we will try running as an unwrapped application.

```
c:\demo-dir> tclkit-cli demo.vfs/main.tcl hello
run_mode: unwrapped
Hello World!

c:\demo-dir> tclkit-cli demo.vfs/main.tcl arith 2 3 4
run_mode: unwrapped
11
```

This allows us to edit and modify the files in the demo.vfs tree and test it without having to rebuild the application in test mode.

In production, the starkit will be directly invoked as the application either via passing it to the tclkit application,

```
c:\demo-dir> tclkit-cli demo hello
run_mode: starkit
Hello World!
```

or alternatively via the batch file (on Windows) or directly invoking the starkit (on Unix)

```
c:\demo-dir> demo arith 2 3 4
run_mode: starkit
11
```

The final variation for running our application is as a single-file executable. Like sdx qwrap, sdx wrap takes a -runtime option that specifies a tclkit application to use as the runtime for our starkit. The two are then bound together in a single executable.

We will use a slight variation of sdx wrap that uses the -vfs option.

```
c:\demo-dir> tclkit-cli sdx.kit wrap myapp.exe -vfs demo.vfs -runtime tclkit-cli-runtime
872 updates applied
9 updates applied

c:\demo-dir> ls
demo  demo.bat  demo.vfs  myapp.exe  sdx.kit  tclkit-cli-runtime  tclkit-cli.exe
```

This creates an starpack myapp.exe. The -vfs option specifies the VFS directory as demo.vfs and not myapp.vfs which would be the default based on the starpack name.

We now have single file application that can be copied to any Windows system and run without any additional steps.

```
c:\demo-dir> myapp hello
run_mode: starpack
Hello World!

c:\demo-dir> myapp arith 2 3 4
run_mode: starpack
11
```

19.4.9. Considerations for multiplatform starkits

Let us take a moment to talk about issues related to multi-platform support. Starkits which are purely script based and have no binary extensions in their content, can be loaded by any tclkit application on any platform. This is true irrespective of the platform on which the starkit was built. Thus the demo starkit we built on our Windows system would work just as well on OS X or Linux.

Starkits that contain binary extensions are also portable across platforms as long as they follow the appropriate structure for packages that we describe in Section 13.7. The load command for loading binary extensions will recognize that the extension is in starkit and copy it out to the local file system from where it can be loaded by the OS loader.

Starpacks are platform-specific executables and, by their very nature, are not portable. You need to build a separate starpack for every platform for which you want to deploy a single-file executable.

Even though starpacks are platform-specific, they do not need to be built on their native platform. All you need to do is specify the appropriate tclkit for the target platform as the value of the -runtime option to sdx wrap. For example, we could build a x86 Linux-specific version of our myapp application by providing a x86 Linux tclkit.

```
c:\demo-dir> tclkit-cli sdx.kit wrap myapp.exe -vfs demo.vfs -runtime \
    tclkit-cli-runtime-linux
```

Here we assume tclkit-cli-runtime-linux is a tclkit application that has been built for our target Linux platform.

19.4.10. Starkit mount points

We need to touch upon one issue that you need to be aware of when using starkits and starpacks. As we discussed, starkits are seen by the Tcl I/O system as virtual file systems and as such have to be mounted. The mount point used for the VFS in starkits and starpacks is the local file system path for the starkit or starpack itself.

We can use our demo starkit for illustration purposes.

```
c:\demo-dir> tclkit-cli
% set kit [file normalize demo]
C:/demo-dir/demo
% file type $kit
file
```

As expected, file type returns the type of our starkit file as file. This is **before** we load it. However, **after** loading the starkit, we get a different result.

```
% source $kit
run_mode: sourced
% file type $kit
directory
```

It now show up as a directory! This is because the path C:\demo-dir\demo is now the mount point for the starkit and the root directory of the VFS.

We can confirm this with the vfs::filesystem info command.

```
% vfs::filesystem info
C:/demo-dir/demo C:/demo-dir/tclkit-cli.exe
```

Notice how C:/demo-dir/demo is listed as a file system. Furthermore, because tclkit applications themselves are structured as starpacks, our tclkit-cli executable file also shows up as a file system which leads to the following quirk.

```
% file type [info nameofexecutable]
directory
```

This is a consequence of the fact that tclkits and starpacks mount their contents as a file system at the mount point corresponding to their executable path. (At the risk of belaboring the point, we need to stress that this only applies to tclkit and starpack executables, not to the standard Tcl shells or applications.)

This implementation quirk is something you need to keep in mind when working with a starkit or starpack. For example, the following command

```
file copy [info nameofexecutable] target
```

will result in very different results in `tclsh` versus `tclkit-cli`. In the former case, `target` will be a copy of the `tclsh` executable. In the latter case, the `tclkit-cli` path is seen as a mounted directory and `target` will contain the entire directory structure contained within the tclkit VFS. You are encouraged to try the command and examine the difference.

19.4.11. Writable starkits

Our discussion so far has only involved read operations on starkits once they are constructed. Depending on the underlying technology used to implement a starkit, it is also possible to modify a starkit by writing to it. Among the three popular starkit technologies, Metakit, Vlerq and zip, only Metakit based starkits support writing.

To create a writable starkit, pass the `-writable` option to `sdx wrap`. For example,

```
c:\temp\demo>tclkit sdx.kit wrap demo.kit -writable
10 updates applied
```

You can then create, delete or write to files within the starkit using standard Tcl I/O commands **provided you are using a tclkit application based on Metakit technology**.

19.5. Chapter summary

In this chapter, we introduced virtual file systems and tclkits. Virtual file systems permit arbitrary structured data to be presented to the application as a local file system. It can then be accessed and worked on with the commonly used file and channel based commands. We saw examples of the utility of this for accessing remote files over FTP and for accessing process information as a file system.

We also saw the use of VFS for the purpose of creating single file Tcl applications. This has great benefits in terms of the ease with which applications can be deployed without the need for installers or additional packaging.

19.6. References

LAND2002
*Beyond TclKit - Starkits, Starpacks and other *stuff*, Landers, Proceedings of the 2002 Tcl Conference, http://www.digital-smarties.com/Tcl2002/tclkit.pdf

20

Interpreters

A *Tcl interpreter* runs Tcl programs within an execution context that includes a namespace hierarchy, command and variable definitions, and a call stack. When an application starts up, as part of its initialization it creates a new Tcl interpreter through a native call to the Tcl library. Some applications may even create multiple such interpreters from native code, either in the same thread or from different threads. These interpreters all run independently, oblivious to the existence of the others.

All our discussion so far has pertained to running Tcl within a single such interpreter created by the application. In fact, an interpreter may **itself** create additional *child* or *slave* interpreters. Unlike the interpreters created natively by the application, these slave interpreters are **not** independent. They run under the control of the creating interpreter, their *master*, in terms of the commands they may execute, how long they may run for and so on. Each slave may in turn create additional interpreters, resulting in an hierarchy of interpreters.

Multiple interpreters are useful in many situations. For example, a network server implemented in Tcl may use a different interpreter for every client, thereby avoiding even accidental interference between the contexts for each client. Another example is the use of multiple interpreters to implement domain-specific languages as dialects of Tcl. We will see examples of both in this chapter.

A special case of a slave interpreter is a *safe* interpreter which is created with all commands deemed to be dangerous from a security point of view removed. This is useful to allow execution of code from unknown and untrusted sources. An example is the execution of application plug-ins written by the user or third parties in a **safe** interpreter so as to avoid both inadvertent as well as malicious modification of the application code.

This chapter is devoted to the creation and use of multiple interpreters within a Tcl application.

20.1. Interpreter basics

We will start off with a discussion of interpreters are created and destroyed and the hierarchy in which they are arranged.

20.1.1. Creating interpreters: `interp create`

An interpreter is created with the `interp create` command.

```
interp create ?-safe? ?--? ?SLAVEPATH?
```

Specifying the optional `-safe` option results in the creation of a safe interpreter. Safe intepreters are discussed in Section 20.6. The `--` sequence indicates the end of options in case `SLAVEPATH` itself begins with a `-` character.

If `SLAVEPATH` is not specified, the command creates an slave interpreter as a direct child of the current interpreter (i.e. the interpreter invoking the `interp create` command). The name of the slave is automatically generated.

```
set child1 [interp create] → interp0
```

The return value from the command is the name of the newly created interpreter. In addition to creating the interpreter, a new command of the same name is also created in the **current** interpreter in the global namespace. This proxy command can be used to in various ways to evaluate code or otherwise control the slave.

20.1.2. Identifying interpreters

An interpreter is identified through a list of interpreter names that define a path through the interpreter hierarchy relative to the current interpreter. The *SLAVEPATH* argument may be specified to the interp create command to create a new interpreter at any level in the hierarchy **below** the current interpreter. The last element of this path is the name of the new interpreter. For example, we can create another direct child of the current interpreter and have it named level1slave.

```
set child2 [interp create secondchild] → secondchild
```

Since the list passed as the *SLAVEPATH* argument contains only one element, it becomes the name of the created child interpreter. We can also create interpreters deeper in the hierarchy.

```
set grandchild1 [interp create [list $child1 grandchild]] → interp0 grandchild
set grandchild2 [interp create {secondchild grandchild}]  → secondchild grandchild
```

Both grandchildren have the same name but are distinguished because their paths are different. The new interpreters are created in their **parent** which is then their "master", **not** in the interpreter that invoked the interp create command.

Note that all interpreters except the last in the path must already exist.

```
% interp create {nosuchinterp grandchild}
Ø could not find interpreter "nosuchinterp"
```

> Interpreter paths are **always** relative to the current interpreter. There is no concept of an "absolute" path and no way to reference an interpreter that is not a descendant of the current interpreter.

As a special case, commands that take an interpreter path as an argument will treat an empty list as referring to the current interpreter.

As is the case when *SLAVEPATH* is unspecified, a new command of the same name as the created interpreter is also created in its master. If a command of that name already exists in the master, it is overwritten. If the name contains one or more namespace separator sequences, the command is created within that namespace which is created if it did not already exist.

```
% interp create ns::ip  ❶
→ ns::ip
% interp slaves
→ secondchild ns::ip interp0
% ns::ip eval {set x 1}  ❷
→ 1
```

❶   Creates a slave ns::ip

❷   The command ns::ip is created in the master and can be used to evaluate scripts within the slave ns::ip

In such cases, deleting the namespace will also delete the interpreter contained in that namespace.

```
% namespace delete ns
% interp slaves
→ secondchild interp0
```

20.1.3. Inspecting the interpreter hierarchy

An interpreter can retrieve its slaves with the `interp slaves` command.

```
interp slaves ?INTERPPATH?
```

The command returns a list of the names of the interpreters that are direct children of the interpreter identified by `INTERPPATH`. If `INTERPPATH` is not specified, it defaults to the current interpreter.

```
interp slaves                          → secondchild interp0
interp slaves {}                       → secondchild interp0 ❶
interp slaves secondchild              → grandchild
interp slaves {secondchild grandchild} → (empty) ❷
```

❶ Empty path refers to current interpreter
❷ No fourth generation

To check for the existence of a specific descendent with a known path, use the `interp exists` command. This returns 1 if a specified interpreter exists and 0 otherwise.

```
interp exists {secondchild grandchild}                  → 1
interp exists {secondchild grandchild greatgrandchild} → 0
```

20.1.4. Destroying interpreters

An interpreter can destroy one or more of its descendents with the `interp delete` command.

```
interp delete ?INTERPPATH ...?
```

Each `INTERPPATH` argument identifies a descendent of the current interpreter. Deleting an interpreter will also destroy that interpreter's descendents if any. If any specified interpreter does not exist, an error is raised and any interpreters listed after that will not be destroyed.

```
interp slaves             → secondchild interp0
interp delete secondchild → (empty)
interp slaves             → interp0
```

Deleting an interpreter also deletes the command of that name.

20.2. Evaluating scripts in an interpreter

We saw in Chapter 10 the use of the `eval` command to execute scripts. The `interp eval` command is similar except that it allows execution of the script in **any** accessible interpreter by specifying its path.

```
interp eval INTERPPATH ARG ?ARG ...?
```

The command concatenates all ARG arguments in the same fashion as the eval command and then executes the script in the interpreter identified by INTERPPATH. The command

```
interp eval $child1 { set foo 1 } → 1
```

will set the value of the foo variable in the specified interpreter.

An alternative means of achieving the same result is to use the eval subcommand of the target interpreter's proxy command. The following is equivalent to the above.

```
$child1 eval {set foo 1} → 1
```

In both cases, the result is the result of the script evaluation in the specified interpreter.

Like the eval command, the arguments undergo double substitution (see Section 10.1.1). However, in this case, the two rounds of substitution take place in separate interpreters. The first round of substitution happens place in the current interpreter and the second in the specified target interpreter. The following snippet illustrates this.

```
set bar foo            → foo
set foo 2              → 2
$child1 eval set foo 1 → 1
eval set $bar          → 2
$child1 eval set $bar  → 1
```

20.3. Command aliases

A command *alias* maps a command in an interpreter to an implementation in another interpreter. Invoking the command in the first will execute the mapped command in the context of the second.

20.3.1. Defining aliases

An alias is defined with the interp alias command.

```
interp alias SRCINTERP SRCCMD TARGETINTERP TARGETCMD ?ARG ...?
```

The command creates a new alias named SRCCMD in the interpreter whose path is given by SRCINTERP. When SRCCMD is invoked within SRCINTERP, Tcl will execute TARGETCMD in interpreter TARGETINTERP. The arguments passed to TARGETCMD will consist of the ARG arguments, if any, specified in the interp alias command, followed by any arguments specified in the invocation of SRCCMD.

Another way to create aliases is with the alias subcommand of the interpreter's proxy command.

```
INTERPCMD alias SRCCMD TARGETCMD ?ARG ...?
```

This works similarly except that the target interpreter, i.e. the interpreter in which TARGETCMD will be run, is the current interpreter. In other words, the above is equivalent to

```
interp alias SRCINTERP SRCCMD {} TARGETCMD ?ARG ...?
```

The result of the command in both forms is an alias token which can later be used to introspect or delete the alias.

The primary motivation for using aliases is to execute commands in controlled fashion on behalf of a "less privileged" interpreter. Section 20.6 describes this in detail; here we stick to a couple of basic examples.

One of the simpler uses of aliases is as a short hand for lazy programmers who hate typing.

```
% interp alias {} substr {} string range
→ substr
% substr "abcd" 0 2
→ abc
%
% interp alias {} xmlify {} string map {< &lt; > &gt; & &amp; \" &quot; ' &apos; }
→ xmlify
% xmlify "2 is < 3"
→ 2 is &lt; 3
```

Note from the above examples that

- An empty path passed as the source or target interpreters refers to the current interpreter.
- You can create an alias within the same interpreter, i.e. where the target and source interpreters are the same.

Of course, you could use procedures for the above as well so here is an example where you do need aliases. Imagine our server application dedicates an interpreter to each client and also has a common logging system shared by all interpreters. We define an interpreter, logger, to collect these messages.

```
interp create logger
interp eval logger {
    set ::log_level 1
    proc log {level message} {
        if {$level >= $::log_level} {
            lappend ::messages $message
        }
    }
    proc messages {} {       # Returns collected messages and resets.
        return $::messages[set ::messages ""]
    }
}
```

Then when we create an interpreter for a client, we add to it aliases that map to log messages at each level.

```
interp create client0
interp alias client0 debug logger log 0
interp alias client0 log logger log 1
interp alias client0 err logger log 2
client0 eval {
    debug "This is a debug message."
    log "This is an informational message."
    err "This is an error message."
}
logger eval messages
→ {This is an informational message.} {This is an error message.}
```

20.3.2. Introspecting aliases

The list of aliases defined for an interpreter can be retrieved with the interp aliases command or the aliases subcommand of its proxy command.

```
interp aliases client0 → err log debug
client0 aliases        → err log debug
```

The target interpreter for an alias can then be determined with the `interp target` command or its proxy form.

```
interp target client0 debug → logger
```

Note that an empty result from the command means the current interpreter is the target.

The specific command prefix in the target interpreter that an alias resolves to can be retrieved with another syntactic form of `interp alias` or its proxy equivalent.

```
interp alias client0 debug → log 0
client0 alias debug        → log 0
```

20.4. Execution context in slaves

When a script is executed via `interp eval`, it is evaluated in the **current** context of the target interpreter, **not** its global context. The following snippet illustrates this.

```
set ip [interp create]
proc get_slave_context {ip} {
    return [$ip eval {namespace current}]
}
$ip alias context_demo get_slave_context $ip
$ip eval {
    namespace eval ns {context_demo}
}
→ ::ns
```

Note how the `namespace current` invocation from `get_slave_context` shows `ns` as the namespace context and not the `::` global context. The `context_demo` alias is invoked in the `ns` namespace context within the slave. This results in the `get_slave_context` procedure being run in the master. When that evaluates `namespace current` in the slave, that command is then invoked within the slave's current context, which is `ns`.

As another example, consider the invocation of an alias from within a procedure.

```
proc get_slave_var {ip} {
    return [$ip eval {set myvar}]
}
$ip alias var_demo get_slave_var $ip
$ip eval {
    set myvar "global"
    proc demo {} {
        set myvar "local"
        var_demo
    }
}
$ip eval demo
→ local
```

Again, note how the value returned is `local` indicating that the `set myvar` was executed in the **current** context of the slave, i.e. within the demo procedure context, and not the global context.

20.5. Cancelling script evaluation

The `interp cancel` command allows cancellation of a script running in an interpreter.

```
interp cancel ?-unwind? ?--? INTERP ?MESSAGE?
```

The effect is similar to that of throwing an error from within a script executing in the interpreter identified by the interpreter path *INTERP*. The difference is that

- first, the cancellation can be initiated from outside the script itself, and
- second, unlike error exceptions, the cancellation can be made to be untrappable with `catch` and `try`.

Let us examine the effect of this command by experimenting with the current interpreter itself. First we define three commands that will raise an exception, or call `interp cancel` on the current interpreter with and without the `-unwind` option.

```
proc raise_error {} { error "Error raised!" }
proc cancel {} { interp cancel {} "Interpreter execution cancelled!" }
proc cancel_unwind {} {interp cancel -unwind -- {} "Interpreter execution unwound!"}
```

Then we define a demo procedure that will call one of these procedures based on the argument passed. We will add a variable trace to further distinguish the cases.

```
proc demo {procedure} {
    puts "Entering demo"
    set x 1
    trace add variable x unset print_args
    catch $procedure message
    puts "Caught error message: $message"
    puts "Exiting demo"
}
```

Now let us try calling demo with each of the procedures we defined earlier.

```
% demo raise_error
→ Entering demo
  Caught error message: Error raised!
  Exiting demo
  Args: x, , unset
```

We do not have much to say for the above case as we have already discussed raising of errors extensively in Section 11.5.

The `interp cancel` command without the `-unwind` option behaves very similarly.

```
% demo cancel
→ Entering demo
  Caught error message: Interpreter execution cancelled!
  Exiting demo
  Args: x, , unset
```

Again, the error is caught and the variable trace fires as the procedure is exited. The real difference with the use of `interp cancel` compared to `error` is that the exception can be initiated from a **different** interpreter than the one in which the script is running.

Our final example uses the `interp cancel` command with the `-unwind` option.

```
% demo cancel_unwind
Ø Entering demo
  Interpreter execution unwound!
```

When the `-unwind` option is specified, the call stack is completely unwound in the target interpreter **without any chance of trap or trace handlers to run** as seen in the example.

507

> ⚠ Because the -unwind option does not give any error trap or trace handlers a chance to run, the common use of these handlers to release resources will be bypassed. Thus the mechanism should be used with extreme care and under limited circumstances.

Note that `interp cancel` aborts the execution of scripts in the target interpreter, the target interpreter itself is **not** deleted.

Our interpreter demonstrated the workings of `interp cancel` using the current interpreter. More commonly it is used in two scenarios:

- cancel scripts in a slave interpreter from within an aliased command. Applying the above discussion to this case should be straightforward.
- cancel scripts running in a interpreter within another thread (see Section 22.6).

20.6. Safe interpreters

> I didn't know I was a slave until I found out I couldn't do the things I wanted.
>
> — Frederick Douglas

There are several application use cases where the application needs to run source code from untrusted sources, the most common example being browsers running Javascript code downloaded from some random web site on your personal desktop. Browsers limit, not eliminate, security vulnerabilities arising from running malicious downloaded code by placing restrictions on the capabilities of the Javascript interpreter itself. This restrictions include access to the local file system and other operating system resources.

Similarly, an application may allow plug-ins from third parties that offer additional functionality. In such cases, the application needs to protect itself, and the user, from being compromised by any untrusted plug-ins that the user might have installed.

In another scenario, the application runs in a privileged mode on behalf of the user but still needs to ensure the user is prevented from accessing unauthorized resources or elevating his privileges.

Yet another example is a distributed system where computationally intensive tasks are assigned to multiple systems. For an additional layer of security, it is good practice to execute such remotely dispensed tasks in restricted envionment.

Tcl's solution for dealing with these situations is the *safe interpreter* which is a Tcl interpreter from which certain commands, which are considered dangerous when invoked from untrusted scripts, have been hidden.

A safe interpreter has the following restrictions placed on it:

- Certain commands like exec, open, socket etc. that can access system resources or affect the state of a process are hidden so they cannot be directly invoked by scripts running in a safe slave.
- Any slave interpreters created by safe interpreters will also be safe even if the -safe option is not specified in their creation.
- The global variable env is not available in safe interpreters as environment variables can contain sensitive information.
- Tcl's default auto-loading facilities described in Section 3.5.1.2 are not implemented.
- Binary extensions cannot be loaded into safe interpreters unless they have a specific entry point different from that used for normal interpreters. When initialized via this entry point, the extension must ensure it does not implement commands that are vulnerable to be misused by malicious scripts.
- Finally, a safe interpreter cannot change the recursion limit on itself or any other interpreter it might create. Recursion limits are discussed in Section 20.8.

A completely restricted safe interpreter can be too limited in functionality to be useful in many circumstances. It can essentially do operations on data and little else. Tcl therefore provides mechanisms for permitting controlled execution of hidden commands in interpreters:

- The aliases mechanism we saw earlier in this chapter which we will discuss further in the context of safe interpreters
- The ability to invoke hidden commands in the safe interpreter under the control of its master

Let us start by looking at how a safe interpreter is created.

20.6.1. Creating a safe interpreter

A safe interpreter is created with the same `interp create` command used to create a normal interpreter but with the added option `-safe` specified.

```
interp create -safe safe0 → safe0
```

We can verify that certain commands are **hidden** and therefore not directly available in the safe interpreter. Attempting to invoke them will result in an error being raised.

```
% safe0 eval {socket www.example.com 80}
0 invalid command name "socket"
```

See the reference documentation for `interp` for a complete list of the commands that are hidden in safe interpreters by default. Since commands can also be hidden dynamically at runtime, you can also programmatically retrieve the list of currently hidden commands.

Hidden commands are not actually removed from the interpreter and can be made available to the interpreter under the control of its master.

20.6.2. Aliasing in safe interpreters

As we mentioned in the introduction, removing **all** access to resources such as the file system would make safe interpreters usable in very few scenarios. For example, consider the use of safe interpreters in a web server to handle client connections. To be useful for this purpose, the safe interpreters would need to be able to read Web content, write to log files and so forth. At the same time, we would want to prevent it from being able to read files from outside the Web content directories or to overwrite the Web content and so on.

Let us look at the first mechanism available to achieve this goal of providing controlled access to resources — aliases. We already introduced this feature in Section 20.3. Our short illustration below demonstrates how a safe interpreter might write log messages to a file. Since the safe interpreter cannot directly do I/O, the trusted master interpreter must do it on its behalf. This is accomplished by defining a `log_message` command in the **slave** that maps to a command in the **master** that writes to the appropriate log file.

```
set safe_ip [interp create -safe]
set log_chan [open [file join /var/log $safe_ip.log] a]
$safe_ip alias log_message puts $log_chan
```

Note this uses the second form of alias definition. We could have also written it using the first form as

```
interp alias $safe_ip log_message {} puts $log_chan
```

with the same result.

Now scripts running in the slave interpreter can safely log messages to disk while still being restricted in terms of its ability to do I/O.

```
$safe_ip eval {
    log_message "Something noteworthy happened."
}
```

The above example enabled some very specific functionality within the safe interpreter. More often you want the safe interpreter to have a little more generalized capabilities. For example, the safe interpreter serving Web clients has to be allowed to read any file in the Web content directory based on the URL but no other. It may also permit writing of files uploaded by the user but again only to a specific directory. We might thus choose to expose a restricted form of the open command to the safe interpreter. In **pseudocode** form,

```
$safe_ip alias open SafeOpen $slave
proc SafeOpen {slave path access} {
    if {![path_is_allowed $path $access]} {
        error "Access denied!"
    }
    tailcall $slave invokehidden -- open $path $access
}
```

The code maps the open command in the safe slave interpreter to the SafeOpen command in the master passing the slave interpreter command as an additional parameter. When the

```
open index.html r
```

command is invoked in the **slave**, the **master** runs the command

```
SafeOpen interp0 open index.html r
```

The SafeOpen command first checks whether the desired access to the path is permitted (we will elaborate on this aspect in a later section). If not permitted, it raises an error. Otherwise, it opens the specified file in the slave interpreter with the invokehidden command that we describe a little later. The slave can then read and / or write that file as appropriate.

20.6.2.1. Precautions for aliased commands

Care must be exercised when executing code in the master on behalf of an untrusted safe slave.

Commands like SafeOpen that are aliased into safe interpreters must be very careful in how they handle any arguments passed by the slaves. In particular, these arguments must **never** be treated as scripts or parts thereof, for example by passing them to commands like eval, subst, expr etc. Otherwise a malicious script running in the untrusted safe interpreter can trick the master into executing arbitrary code.

Also keep in mind that any exceptions raised in the master interpreter will be propagated back to the slave where they can be caught and error stack examined. Therefore if you want to prevent inadvertent information leakage from the master to the slave, all errors must be caught and passed on to safe interpreter in some sanitized generic form.

20.6.3. Hidden commands

Having looked at the use of aliases with safe interpreters, let us now look at the other mechanism for using safe slaves under the control of a trusted master — hidden commands.

20.6.3.1. Invoking hidden commands

The commands that are hidden in an interpreter are **not** deleted or removed; they just cannot be directly invoked from within the safe interpreter. They can however be invoked by the master through the interp invokehidden command, or equivalently the invokehidden subcommand of the slave interpreter command.

```
interp invokehidden SLAVE ?-global? ?-namespace NAMESPACE? ?--? HIDDEN ?ARG ...?
SLAVE invokehidden ?-global? ?-namespace NAMESPACE? ?--? HIDDEN ?ARG ...?
```

Here the *HIDDEN* argument is the name of the command that is hidden in the slave interpreter and *ARG...* are any arguments to be passed to it. The -global and -namespace options permit the master to control the namespace context within which the hidden command is executed. By default, the command is executed in the current context of the slave. The -global and -namespace option force the execution to take place in the global or specified namespace instead.

We saw an example using invokehidden in the previous section where the open command was invoked in the slave interpreter to create a channel to a file after the master interpreter checked that the file path was one that the slave was permitted to access.

```
tailcall $slave invokehidden -- open $path $access
```

This results in the open command being run in the **slave** interpreter and returning a channel to the script in the slave.

> We could have also written the above invokehidden command as
>
> ```
> $slave invokehidden -- open $path $access
> ```
>
> The purpose behind using tailcall is that in case of errors, the error stack looks cleaner as the SafeOpen call will not be present on the stack at all as the tailcall replaces it with the open command invocation.

At this point the front benchers in the class will be objecting vociferously thereby waking up the back bench. Since the open command in the safe slave is aliased to SafeOpen in the master, won't the invokehidden call to open in the slave result in a recursive call back into SafeOpen ad infinitum? The answer is that a call to invokehidden will always invoke the **original** command that was hidden and not any alias or redefinition. Here is an illustrative example.

```
% safe0 eval { proc demo {} { return "The original demo" } }
% safe0 eval demo
→ The original demo
% safe0 hide demo
% safe0 eval demo ❶
Ø invalid command name "demo"
% safe0 eval { proc demo {} { return "The redefined demo" } } ❷
% safe0 eval demo ❸
→ The redefined demo
% safe0 invokehidden demo ❹
→ The original demo
```

❶ Errors because the command is now hidden
❷ Redefine the procedure
❸ The redefinition is executed.
❹ The original hidden procedure is executed

Notice how invoking demo within the safe0 interpreter runs the new definition of demo whereas invoking demo from the master via invokehidden runs the original hidden procedure. This behaviour ensures that the slave interpreter (remember it is potentially running untrusted code) cannot trick the master into running the wrong code by simply redefinining commands.

Like `interp eval`, `interp invokehidden` also runs code in the slave interpreter. One obvious difference is that `interp eval` runs arbitrary scripts whereas `interp invokehidden` runs a single command which must furthermore be a hidden command. A less obvious but important difference is that while the arguments supplied to `interp eval` undergo two rounds of substitutions, once in the master and once in the slave, arguments to `interp invokehidden` only undergo substitution in the master interpreter. This gives the master precise control over what arguments are passed to the hidden command.

20.6.3.2. Hiding and exposing commands

We noted earlier that certain commands are hidden at the time a safe interpreter is created. However, as our previous example showed, we can hide any command at any time with any of the following equivalent forms.

```
interp hide SLAVE CMDNAME ?HIDDENNAME?
SLAVE hide CMDNAME ?HIDDENNAME?
```

Both forms hide *CMDNAME* in the specified slave interpreter. The command can be invoked by the master via `invokehidden` with the name *HIDDENNAME* which defaults to *CMDNAME* if unspecified.

There is an important restriction that needs to be kept in mind with respect to the hiding of commands. Neither *CMDNAME*, nor *HIDDENNAME* if specified, can contain namespace qualifiers. The commands to be hidden are always looked up in the global namespace of the slave interpreter. This restriction prevents the slave from tricking the master interpreter into hiding the wrong command by executing in a namespace other than the global one.

Hidden commands may be made available to the slave interpreter with the `expose` subcommand.

```
interp expose SLAVE HIDDENNAME ?CMDNAME?
SLAVE expose HIDDENNAME ?CMDNAME?
```

Exposing a command makes it available to be called directly by the slave either under its original name or if *CMDNAME* is specified, under a new name. This technique is an alternative to renaming the original command when redefining it.

```
% safe0 expose demo original_demo
% safe0 eval demo
→ The redefined demo
% safe0 eval original_demo
→ The original demo
```

One final note about hiding and exposing commands is that although our discussion has been focused on safe interpreters, commands can be hidden or exposed in **any** slave interpreter, not just safe ones, though their main use is with the latter.

Here is an example of how the combination of aliases and hidden commands may be put to good use in contexts other than safe interpreters. In previous chapters, we saw several ways of "wrapping" existing commands to modify their behaviour. For example, we might want to log the every outgoing network connection in an application. We could create a new `socket` procedure that overrides the `socket` command and calls it. Here is a simple alternative.

```
interp hide {} socket
proc socket args {
    puts "New connection: $args"
    tailcall interp invokehidden {} socket {*}$args
}
close [socket www.example.com 80]
→ New connection: www.example.com 80
```

Note the current interpreter is invoking a hidden command within itself. This is possible only because the interpreter is not a safe interpreter.

20.6.3.3. Introspecting hidden commands

The list of commands that are currently hidden can be retrieved with the `interp hidden` command or the `hidden` subcommand of the interpreter command.

```
% interp hidden safe0
→ file tcl:file:isdirectory tcl:file:writable tcl:file:type tcl:file:tail tcl:file:readli...
% safe0 hidden
→ file tcl:file:isdirectory tcl:file:writable tcl:file:type tcl:file:tail tcl:file:readli...
```

In case it is not obvious, the above commands are executed from the safe interpreter's master, not from within the safe interpreter. Somewhat surprisingly, the safe interpreter can also find out what commands are hidden from it.

```
% safe0 eval {interp hidden {}}
→ file tcl:file:isdirectory tcl:file:writable tcl:file:type tcl:file:tail tcl:file:readlink
  ↳ tcl:file:executable socket tcl:file:size tcl:file:delete tcl:file:copy open
  ↳ tcl:file:isfile pwd unload tcl:file:rootname tcl:file:owned glob exec tcl:file:attributes
  ↳ tcl:file:dirname encoding tcl:file:normalize fconfigure load source tcl:file:volumes exit
  ↳ tcl:file:stat tcl:file:mtime tcl:file:extension tcl:file:tempfile tcl:file:link
  ↳ tcl:file:mkdir tcl:file:rename tcl:file:readable tcl:file:nativename tcl:file:lstat
  ↳ tcl:file:exists tcl:file:atime cd
```

20.6.4. Trusting safe interpreters

A master can mark a safe interpreter as trusted with the `interp marktrusted` command.

```
interp marktrusted SLAVE
SLAVE marktrusted
```

Marking an interpreter as trusted removes implicit restrictions such as automatically making its children as safe. However, it does **not** immediately expose any hidden commands. They still have to be explicitly made visible in the slave.

```
interp create -safe ip0                    → ip0
interp create {ip0 ip1}                     → ip0 ip1
interp issafe {ip0 ip1}                     → 1 ❶
ip0 marktrusted                             → (empty)
interp create {ip0 ip2}                     → ip0 ip2
interp issafe {ip0 ip2}                     → 0 ❷
ip0 eval {close [open c:/temp/foo.txt w]} Ø invalid command name "open" ❸
```

❶ Any further descendants are automatically safe
❷ After being marked as trusted, descendents are trusted by default
❸ However commands in the interpreter marked trusted are still hidden

20.6.5. Utilities for safe interpreters

As discussed, a completely restricted safe interpreter only has limited use. We therefore described mechanisms that allow execution of potentially dangerous commands within the safe interpreter under the control of the master. Nevertheless, security is a very tricky business and correct use of these mechanisms without introducing security holes when executing untrusted scripts in a safe interpreter is still fraught with potential pitfalls.

As an example, consider the path_is_allowed pseudo command we used earlier to check if a particular file was allowed to be accessed by a safe interpreter. An implementation of this command is not just a matter of checking against a database of permitted files and directories. It would have to ensure correct operation for both absolute and relative paths, presence of .. parent directory tokens, links and so forth. As a matter of principle, the real file paths should also be preferably hidden from the safe interpreter. Without careful coding and review, subtle errors will creep in.

Several libraries provide functionality that reduce the burden of extending safe interpreters. We briefly describe two such libraries and strongly encourage their use instead of rolling your own implementation.

20.6.5.1. Safe Tcl

Safe Tcl is a set of commands in the safe namespace that extend safe interpreters in a controlled manner.

> *Safe Tcl* needs to be distinguished from the *safe interpreters* we have discussed above. The former is layered on top of the latter to provide some additional conveniences. The terminology is slightly confusing but follows the Tcl reference documentation.

Safe Tcl defines aliases for the normally hidden commands source, file, encoding, load and exit. These aliases expose these commands in the slave interpreter but limit their permitted operations as shown in Table 20.1.

Table 20.1. Safe Tcl predefined aliases

Alias	Description
encoding	The encoding command is normally hidden because in addition to converting strings into various character encodings it allows setting of the system encoding. The alias permits all operations except this potentially unsafe change in the system encoding.
exit	The aliased version deletes the safe slave interpreter in which it is invoked instead of terminating the entire process.
file	The alias will only permit subcommands that operate purely on a syntactic basis and do not require access to the file system. The allowed subcommands are join, split, dirname, extension, root, tail and pathtype.
load	Only allows binary extensions to be loaded under certain restrictions described later.
source	Only allows Tcl scripts to be sourced under certain restrictions that are described later.

Creating a Safe Tcl interpreter

An application can make use of the library either by using the safe::interpCreate command to create a safe interpreter with the above extensions, or by using the safe::interpInit command to enable the extensions in an existing safe interpreter.

```
safe::interpCreate ?SLAVENAME? ?OPTIONS?
safe::interpInit EXISTINGSLAVE ?OPTIONS?
```

The difference with respect to the interp create -safe command is that the created interpreters are initialized with the predefined aliases. Let us try out a few commands to illustrate. An interpreter created using safe::interpCreate can invoke certain file commands that work purely on a syntactic basis. On the other hand, file subcommands that access the file system are still not permitted.

```
safe::interpCreate safeA              → safeA
safeA eval {file dirname /var/log}    → /var
safeA eval {file isdirectory /var/log} Ø not allowed to invoke subcommand isdirectory of file
```

In contrast, safe interpreters created using interp create cannot invoke the file command at all.

```
% interp create -safe safeB
→ safeB
% safeB eval {file dirname /var/log}
∅ invalid command name "file"
```

We can however add the aliases from the Safe Tcl library with `safe::interpInit`.

```
safe::interpInit safeB                 → safeB
safeB eval {file dirname /var/log} → /var
```

Deleting a Safe Tcl interpreter

Interpreters created with `safe::interpCreate` or initialized with `safe::interpInit` must be destroyed with the `safe::interpDelete` command, not with `interp delete` as internal structures related to the added safe extensions need to be cleaned up.

```
safe::interpDelete safeB → (empty)
```

The slave interpreter may also commit suicide by calling the `exit` command. Unlike in a normal interpreter, this does not terminate the entire process as it is aliased to the `safe::interpDelete` command.

Safe Tcl interpreter configuration

Both `safe::interpCreate` and `safe::interpInit` accept several options, shown in Table 20.2, that control certain aspects of behaviour. These options can also be set at any later time with the `safe::interpConfigure` command.

Table 20.2. Safe Tcl configuration options

Option	Description
`-accessPath` *DIRECTORIES*	Specifies the list of directories from which the slave interpreter is allowed to source or load files. This defaults to the list of directories that the master uses for auto-loading packages. Any attempt to source or load files from a directory not listed in *DIRECTORIES* will raise an error.
`-deleteHook` *SCRIPT*	Specifies a callback script to be run in the **master** interpreter when the slave interpreter is deleted. This script is run **before** the actual deletion and is passed an additional argument which is the name of the slave interpreter. An empty string for *SCRIPT* removes any existing callback.
`-nested` *BOOLEAN*	If specified as a boolean `true` value, the slave interpreter is permitted to load packages into any interpreters that it creates itself. Default is `false`.
`-statics` *BOOLEAN*	A Tcl application may include binary extensions that are statically linked into the application itself and thus are not associated with any file or directory paths. If this option is `true` (default), the slave interpreter can load such statically linked extensions; if `false`, it cannot.

File paths in Safe Tcl interpreters

Even when a safe interpreter needs to be provided access to (some portion of) the file system, it is still considered good practice to not expose the real file system paths to the untrusted scripts. Safe Tcl helps with this by using virtual tokens to represent the physical directories. You can see this by examining an `auto_path` element in the master and slave interpreters.

```
lindex $auto_path 0                     → c:/tcl/lib
safeA eval {lindex $auto_path 0} → $p(:0:)
```

Thus a script running inside the safe interpreter can only see the "virtual" directory tokens and not the real file system paths. When passing paths into the Safe Tcl interpreter, the master needs to translate real paths to the corresponding token. It can use the `safe::interpFindInAccessPath` command for this purpose. For example,

```
safe::interpFindInAccessPath safeA [lindex $auto_path 0] → $p(:30:)
```

The master may also add new paths that are permitted to be accessed by the `source` or `load` commands. The `safe::interpAddToAccessPath` command does the needful.

```
safe::interpAddToAccessPath safeA C:/temp → $p(:265:)
```

The command returns the token that will represent the path in the slave interpreter.

Troubleshooting Safe Tcl interpreters

Debugging scripts in safe interpreters can sometimes be troublesome because certain Tcl features like full stack traces on errors are not made available to avoid information being leaked to the untrusted scripts in the slave. Safe Tcl provides the `safe::setLogCmd` command to help with this problem. It permits specification of a script to be run in the master interpreter when "interesting" events happen in the slave. This script is invoked with an additional message describing the event.

```
% safe::setLogCmd puts
% safe::interpCreate safeB
→ NOTICE for slave safeB : Created
  NOTICE for slave safeB : tcl_libray was not in first in auto_path, moved it to front of...
  NOTICE for slave safeB : Setting accessPath=(c:/tcl/866/x64/lib/tcl8.6 c:/tcl/866/x64/l...
  NOTICE for slave safeB : auto_path in safeB has been set to {$p(:0:)} {$p(:1:)} {$p(:2:...
  safeB
% safeB eval {source foo.tcl}
∅ ERROR for slave safeB : "foo.tcl": not in access_path
  permission denied
```

To turn off logging, pass an empty string to the `safe::setLogCmd` command.

20.6.5.2. The `island` package

The Safe Tcl commands in the previous section only provide control of the directories as related to the `source` and `load` commands. They do not help with general purpose file operations. This functionality is provided by the `island` package available from http://wiki.tcl.tk/44380.

The package provides a single command `island::add` which specifies a directory that is allowed to be accessed by a safe interpreter. Using the package is straightforward and illustrated below.

```
% package require island
→ 0.2
% set ip [interp create -safe]
→ interp0
% $ip eval {close [open c:/temp/demo.txt w]} ❶
→ invalid command name "open"
% island::add $ip c:/temp ❷
→ C:/temp
% $ip eval {close [open c:/temp/demo.txt w]}
```

❶ Fails because open is hidden in a safe interpreter.
❷ Permit `C:/temp` to be accessed by the interpreter.

Note that the island::add command may be used multiple times to permit more than one directory to be accessed.

20.7. I/O in slave interps

Channel I/O in slave interpreters merits some special consideration. We have already seen that commands that **create** channels, such as open, socket etc., are hidden in safe interpreters (though notice the actual I/O command like puts, gets, read are not) and how channels can be opened in safe slaves through the use of controlled aliases in the master. Here we address another issue which applies to safe and normal interpreters alike. Channels created in an interpreter are not automatically available in other interpreters except for the standard input, output and error channels which are made available to all interpreters.

> On Windows platforms, applications like wish which are GUI applications and not console-based do not have real standard I/O file descriptors provided by the operating system. In this case, standard I/O is simulated to a pseudo-console window provided by the Tk GUI extension and not implemented as real channels. In this environment, the standard channels are only available to the main Tcl interpreter and not to any slaves.

We now look at how to make channels available in interpreters other than the ones that created them. There are two ways to accomplish this. The interp share and interp transfer commands both make a channel in an interpreter available in another interpreter. The difference is that with interp share, the channel remains available in the original interpreter as well whereas with interp transfer it does not.

```
interp share FROMINTERP CHANNEL TOINTERP
interp transfer FROMINTERP CHANNEL TOINTERP
```

Here *FROMINTERP* is the interpreter currently holding the channel *CHANNEL*, and *TOINTERP* is the interpreter where the channel is to be made available. Note that the channel name remains the same in all interpreters where the channel is accessible. In the case of shared channels, the channel seek pointer is also shared across all interpreters so if multiple interpreters perform I/O on the shared channel the data will be interleaved. A shared channel is not closed until all interpreters where it is accessible close the channel.

Here is a simple example illustrating use.

```
% set ip [interp create -safe]
→ interp2
% set fd [file tempfile]
→ file437faa0
% interp share {} $fd $ip
% $ip eval "set fd $fd" ❶
→ file437faa0
% $ip eval {puts $fd "Message from the slave."; close $fd}
% chan seek $fd 0 start
% read $fd
→ Message from the slave.
% close $fd
```

❶ Need to pass in the channel name to the slave

Note how the channel stays open in the parent after the slave closes it and that the parent needs to reset the channel pointer with a seek before reading the file.

20.8. Setting resource limits

Tcl offers a limited form of protection against runaway scripts in interpreters that result in excessive consumption of computing resources. These limits take the form of

- a maximum recursion depth for execution of Tcl scripts
- an upper limit on the number of commands that may be executed within an interpreter
- an absolute time by which an interpreter must finish executing a script

We describe these capabilities in this section.

> Besides these explicit limits, a master interpreter may also impose limits using mechanisms already discussed. For example, aliasing puts in a slave interpreter can count and limit the number of bytes written to a file by the slave.

20.8.1. The recursion limit

To protect against runaway scripts overflowing the space allocated for the C stack of the process, Tcl places a limit on the depth of the call stack of an interpreter. This limit can be retrieved or set with the interp recursionlimit command.

```
interp recursionlimit INTERP ?LIMIT?
SLAVE recursionlimit ?LIMIT?
```

If LIMIT is not specified, the command returns the current recursion limit for the specified interpreter (which may be the current one).

```
interp recursionlimit {} → 1000
```

We can test the effect of this limit.

```
% proc recurse {} { incr ::depth ; recurse }
% set depth 0
→ 0
% recurse
Ø too many nested evaluations (infinite loop?)
% set depth
→ 1000
```

As you see, Tcl will raise an error exception when the call stack depth exceeds the recursion limit.

We can also change the recursion limit if we wish. Ususally this change is made for slave interpreters.

```
set ip [interp create]
$ip recursionlimit 10
$ip eval {
    proc recurse {} {incr ::depth; recurse}
    set depth 0
    catch {recurse} message
    return "Error: failed at depth $depth: $message"
}
→ Error: failed at depth 9: too many nested evaluations (infinite loop?)
```

20.8.2. Limiting interpreter lifetimes

Tcl can limit the lifetime of an interpreter either by restricting the number of commands the interpreter can execute or by specifying an absolute time by which the interpreter must have finished execution. Both these limits are retrieved and set with the interp limit command.

```
interp limit INTERP command|time ?OPTIONS?
SLAVE limit command|time ?OPTIONS?
```

Depending on whether command or time is specified, the command pertains to the limit on command execution or the time by which the interpreter must complete.

20.8.2.1. Limiting number of commands executed

We will start with the command limits. If no options are specified, the command will return the current settings for the specified limit type.

```
% interp limit {} command ❶
∅ limits on current interpreter inaccessible
% interp limit $ip command
→ -command {} -granularity 1 -value {}
% $ip limit command
→ -command {} -granularity 1 -value {}
```

❶   Error because an interpreter may not read its own limits

The -granularity option controls how often Tcl will check the limit. For example, when set to 1, Tcl will check that the permitted command execution count is not exceeded before **every** command invocation. If set to 5, it will only check on every fifth command.

The -command option specifies a callback command that will be invoked in the global context of the interpreter that invoked the interp limit command when the command count limit is exceeded in the target interpreter. If an empty string, no callback is registered.

The -value option specifies the permitted command count. If empty, no limit is placed on the number of commands that may be executed by an interpreter.

Let us play around with changing the limits on an interpreter. We will define a procedure that will be called when the limit is exceeded. A point to note about the call back procedure is that it can **change the limit to permit the interpreter to continue execution**.

```
proc watchdog {ip max} {
    if {[interp limit $ip command -value] < $max} {
        puts "Raising command count limit further to $max"
        interp limit $ip command -value $max
    }
}
```

Now we create a slave interpreter and set limits on it. An important point to be noted is that the creation of the interpreter itself executes initialization commands in the slave. So when we set limits we may need to take this into account using the info cmdcount command to find how many commands have been executed so far.

```
set ip [interp create]
set nexecuted [$ip eval {info cmdcount}]
interp limit $ip command \
    -command [list watchdog $ip [+ $nexecuted 4]] \
    -value [expr {$nexecuted+2}]
```

Now we try executing commands in the slave.

```
$ip eval {info cmdcount} → 315
$ip eval {info cmdcount} → 316
$ip eval {info cmdcount} → Raising command count limit further to 318
  317
$ip eval {info cmdcount} → 318
$ip eval {info cmdcount} ∅ command count limit exceeded
$ip eval {info cmdcount} ∅ command count limit exceeded
```

A few words of explanation may be in order.

- We retrieved the command count as of interpreter creation (nexecuted).
- We want at most 4 more commands to be permitted to be run and this is the number we pass to watchdog as the second argument.
- However, for illustrative purposes we initially set the interpreter command count limit to be only 2 more, not 4.
- Now when we execute commands in the slave, the first two execute with not much fanfare. The third one however causes our initial limit to be exceeded and Tcl runs our watchdog procedure.
- The watchdog procedure **increases** the limit since it has not reached the maximum it was passed.
- After watchdog returns, Tcl checks the limit again. Since the **new** limit is greater than the current command count, it permits execution to continue for this command and the next.
- When the new limit is also exceeded, watchdog is called again. However, since the limit is now the maximum specified by the max argument, it is not increased further.
- Tcl therefore aborts execution of any further commands with an error message.

Note that the interpreter is **not** destroyed when the limit is exceeded. Only errors are generated. The master interpreter could choose to remove or extend the limits and continue to use the interpreter.

```
% $ip limit command -value {} ❶
% $ip eval {info cmdcount}
→ 320
% $ip eval {info cmdcount} ❷
→ 321
```

❶ Remove the limit
❷ Happy days are here again

20.8.2.2. Limiting interpreter duration

An alternative way to control the lifetime of an interpreter is by specifying a time by which the interpreter must finish execution. We can retrieve the current settings for an interpreter as for command count limits except that we specify time as an argument to interp limit instead of command.

```
% interp limit $ip time
→ -command {} -granularity 10 -milliseconds {} -seconds {}
```

The -command and -granularity settings have the same meaning as was discussed in the previous section. The former specifies a callback and the latter controls how frequently limits are checked.

The -seconds value is the absolute time, expressed as the number of seconds after the epoch (see Section 8.1), beyond which the interpreter must not be allowed to execute.

The -milliseconds value is an additional interval (specified in milliseconds) beyond the time specified by the -seconds value. This is of use in cases where units of seconds is not fine grained enough for the application's purpose.

Since the general working of time limits is similar to command limits in the previous section (including the ability to change limits in the callback), we only present a simple example.

```
% set now [clock seconds]
→ 1499148960
% $ip limit time -seconds [expr {$now + 2}] -granularity 1
% after 1000 ❶
% $ip eval {set foo 1} ❷
→ 1
% after 1000
```

```
% $ip eval {set foo 2} ❸
0 time limit exceeded
```

❶ Hang around for a second
❷ Succeeds because time limit not crossed
❸ Fails because time limit exceeded

The attempt to set the value of foo to 2 fails because the specified absolute time for the interpreter to complete has been exceeded.

> We set -granularity to 1 above for illustrative purposes as otherwise we would have had to execute more commands to demonstrate the effect. The cost of checking of absolute time limits can be significant so in general the granularity should not be lowered for time limits.

20.9. Using multiple interpreters

There are many ways applications might make use of multiple interpreters. We illustrate two common ones here.

20.9.1. A safe network server

A very common and obvious use of multiple interpreters is to handle client connections in a server by associating each client with a private slave interpreter. This has two benefits

- The client contexts are isolated from each other "for free". All context for a client is contained within a slave and no special care need be taken to ensure there is no leakage of data or other state between clients.
- Running safe slaves adds another layer of safety for the server. At the same time, depending on authentication and established authorization roles, different slaves may be configured with different privileges in terms of what they can execute and access.

Here is a variation of the simple network server we saw in Section 18.1.4.2. The difference here is that while that server simply reversed the line sent by the client, this server will execute it as an expression, essentially acting as a remote calculator. This is obviously dangerous since the expr command can evaluate arbitrary Tcl scripts. We will therefore have to do the evaluation within a safe interpreter.

As in our original network server, we start off by defining the command that will invoked in response to every connection request. For each new connection we create a safe interpreter that will be dedicated to that connection. Furthermore, we place limits on the number of commands that may be executed and the maximum time the interpreter is allowed to live. Then as before we make the accepted client channel non-blocking and attach a read handler passing it our slave interpreter.

```
proc on_accept {so client_ip client_port} {
    set slave [interp create -safe]
    set cmdlimit [$slave eval {info cmdcount}] ❶
    $slave limit command -value [incr cmdlimit 3]
    $slave limit time -seconds [expr {[clock seconds] + 60}]

    chan configure $so -buffering line -encoding utf-8 -blocking 0 -translation crlf
    chan event $so readable [list on_read $slave $so $client_port]
}
```

❶ Get number of commands executed so far in slave

Notice how we set the command limit. The slave internally evaluates commands as part of its internal initialization. So the limit we set has to be some increment to the number of commands already executed by the slave.

Our read handler reads each line from the client and evaluates it as an expression in the slave. For ease of illustration we do not bother dealing with expressions across multiple lines.

```
proc on_read {slave so client_port} {
    set n [gets $so line]
    if {$n >= 0} {
        set status [catch {$slave eval [list expr $line]} result]
        puts $so $result
        if {$status &&
            [lindex $::errorCode 0] eq "TCL" &&
            [lindex $::errorCode 1] eq "LIMIT"} {
            close $so
            interp delete $slave
        }
    } elseif {[chan eof $so]} {
        close $so
        interp delete $slave
    }
}
```

Make a note of the error handling. If the slave evaluation results in an error because of limits being exceeded, we close down the connection. Other errors like invalid input are simply reported back to the client.

Now we start up the server on our local address.

```
set listener [socket -server on_accept -myaddr 127.0.0.1 10042]
vwait forever
```

There is a design choice we made in our implementation to have the master interpreter handle the I/O and invoke the slave only to evaluate the expression. An alternative would have been to hand off the channel to the slave and have it handle the I/O itself. Our choice was based on simpler clean up of the channel when the slave interpreter crossed its limits.

Let us try out our server by connecting to it with telnet.

```
C:\temp\book> start tclsh safe_server.tcl
C:\temp\book> telnet 127.0.0.1 10042
Trying 127.0.0.1...
Connected to 127.0.0.1.
2+2
4
2 ** 4
16
[exit]
invalid command name "exit"
5+5
command count limit exceeded
Connection closed by foreign host.
```

As expected, because of the use of the safe slave, the client is not allowed to execute exit which would cause the entire server to exit. And once the client has used up its quota of commands, it is kicked out.

You can see that the use of multiple safe interpreters did not add much complexity to the code.

20.9.2. Implementing domain specific languages

Our second example illustrating the use of slave interpreters involves their use of implementing domain specific languages. A *domain specific language* (DSL) is a programming language that is targeted towards a specific

problem domain. The characteristics that distinguish DSL's from general purpose languages like Tcl arise from the fact that users of the language are often not programmers by profession. Rather, they are experts in their respective domains. Consequently, DSL's are generally simple in syntax and programming constructs, often closer in appearance to natural languages. At the same time, their functions are very high level and directly reflect the terminology and abstractions of the target domain.

Since DSL's are not flexible or powerful enough for writing entire applications, they are themselves implemented in the general purpose language, the *hosting* language, used to write the application. An application may support several DSL's at the same time. For example, a database front-end application may include a DSL for queries, a DSL for report generation and a DSL for screen design.

Some well known and commonly used DSL's include:

- Cascading style sheets (CSS) for web pages. The original motivation for these was to enable graphics designers, **not programmers**, to control the appearance of dynamically generated web pages through a declarative style without having to put their grubby little fingers into the beautiful page generation code.
- The syntax used by most make utilities is a DSL that expresses targets, dependencies, build rules etc.
- Even regular expressions syntax is a DSL that allows compact specification of how patterns are to be matched.
- Configuration files such as those used by Puppet or good old Windows .INI files can also be viewed as simple DSL's.

Internal and external DSL's

DSL's may be classified as *internal* or *external*. Internal DSL's are syntactically based on the host language and follow a similar structure. External DSL's on the other hand may follow a completely different syntax from the host language, often because the latter's structure is deemed too foreign from the perspective of the domain experts. Imagine for example if an accountant were asked to write auditing rules using C++ syntax[1]! The disadvantage of external DSL's is that you now have to write a parser for the DSL syntax which, although not hard for simple DSL's, does involve a non-trivial amount of work.

Happily, with Tcl we can get the benefits of a natural syntax without having to write an external DSL:

- Tcl's syntax as a list of words closely resembles natural language. For example, `deduct federal 10%` as a Tcl command is easily understood as English.
- There is no need for declarations, typing and other programming artifacts that an accountant may not be familiar with.
- The DSL can be executed using a separate Tcl interpreter with appropriate safety boundaries.
- Separate interpreters also allows unneeded elements of the language to be easily disabled **if so desired**. This is important because the DSL as seen by the domain expert should be as simple as possible.

For demonstration purposes, we will implement a little internal DSL that an accountant might use to generate paychecks. The following program is a sample that an accountant expects to be run against every employee record in a database to generate paychecks.

```
deduct insurance 100
deduct federal 10% when salary between 20000 and 30000
deduct federal 20% when salary above 30000
deduct state 5% when federal above 2500 ❶
generate paycheck
generate paystub
```

❶   Yes, silly I know but that's taxes for you

Our little language should be understandable to an accountant right away. Most hosting languages would need to implement the above as external DSL's. Not so in Tcl becauase **it is standard Tcl syntax**.

---

[1] It's hard enough for programmers!

We start with a couple of utility procedures to check whether a word is included in a permitted list and to ensure a word is an integer. Nothing to see here.

```
proc check_word {word allowed} {
    if {$word ni $allowed} {
        error "Invalid word '$word'. Must be one of [join $allowed ,]"
    }
}
proc check_number {word} {
    if {![string is integer -strict $word]} {
        error "'$word' is not an integer"
    }
}
```

We will assume employee data is passed to us in a dictionary. We will need a procedure to map the keywords in our DSL to keys in the employee dictionary. Once again, not much to see here either.

```
proc word_to_var {word} {
    dict get {
        salary    Salary
        federal   FederalTax
        state     StateTax
        insurance Insurance
    } $word
}
```

As we come to the meat of the implementation, we have a decision to make. The obvious design is to map the commands in the DSL to commands in the safe interpreter that are aliased in the master. At that point, we need to make choice — should the DSL be interpreted or compiled to a Tcl script. In the first case, our alias implementations would be called for every employee record. They would then take the appropriate actions to deduct taxes, print the check and so on. We will go with the second alternative and compile the DSL to a Tcl script. This Tcl script will then be directly evaluated for each employee record. The advantage is that the DSL program need not be repeatedly parsed for every employee record. This method is slightly more complex that the interpreted DSL implementation but since you have already seen metaprogramming in Section 10.8, it should not be hard to follow.

We create a class to store some context — the compiled Tcl script for the DSL code.

```
oo::class create Paymaster {
    variable Script
}
```

Our paymaster needs to be told the rules for generating a paycheck. These rules are provided by our accountant using our DSL. We will pass these in to our paymaster through its constructor.

```
oo::define Paymaster constructor {dslscript} {
    interp create dslengine -safe
    dslengine eval {namespace delete ::} ❶
    foreach alias {deduct generate} {
        dslengine alias $alias [self] $alias
    }
    dslengine eval $dslscript
    interp create calculator -safe
    calculator alias puts puts
}
```

❶ Deleting the global namespace will remove all commands from the slave interpreter

---

Our constructor needs some explanation. We create not one, but two, slave interpreters. The first one, dslengine, is fundamental to our design because it is the one that will parse the DSL script. We remove **all** commands from it and then add the aliases deduct and generate corresponding to the DSL language commands. These then will be the only commands accessible in that slave.

The second interpreter is not strictly necessary. Once dslengine has generated the Tcl script, we could run that in the master interpreter. Its presence is to add another layer of security. We do not trust our accountant. He may be annoyed about his annual raise and create a rule that "deducts" [exec reformat drive] instead of a number. Although our aliases will validate arguments, there is a chance that something will slip through. Running the generated script in another safe interpreter protects against oversights. Note we cannot use our dslengine for this because it does not have any commands other than the DSL-specific ones.

Next we define our destructor which has nothing to do but release the slaves.

```
oo::define Paymaster destructor {
    catch { interp delete dslengine }
    catch { interp delete calculator }
}
```

Time now to define our aliases that implement the DSL. Only two are needed for our simple language. We will start with the more complex one, deduct. Remember our design involves **transpiling** the DSL into Tcl so these methods need to generate fragments of Tcl that reflect the DSL statements. These fragments are concatenated to construct the transpiled Tcl script.

We have already seen techniques for constructing scripts in Section 10.7.1.2. We follow that method here by defining a script template that reflects the logic of the script fragment implementing the deduct DSL statement. We then construct the script fragments for each place holder in the template and then replace them at the end with string map. We will not go into further detail here as that is not the point of this exercise.

```
oo::define Paymaster method deduct {what deduction args} {
    check_word $what {federal state insurance}
    set category [word_to_var $what]
    if {[regexp {^(\d+)(%)?$} $deduction -> amount percent] == 0} {
        error "Invalid deduction amount $deduction"
    }
    set template {
        if {%CONDITION%} {
            if {%USE_PERCENT%} {
                set deduction [expr {int(($Salary * %AMOUNT%)/100)}] ❶
            } else {
                set deduction %AMOUNT% ❷
            }
            incr %CATEGORY% $deduction ❸
            incr NetAmount -$deduction ❹
        }
    }

    if {[llength $args] == 0} {
        set condition true ❺
    } else {
        if {[llength $args] < 4} {
            error "Incomplete line"
        }
        set args [lassign $args cond_keyword cond_var cond_cmp number]
        check_word $cond_keyword {when if}
        set cond_var [word_to_var $cond_var]
        check_number $number
        switch -exact -- $cond_cmp {
            over - above {
```

```
                    set condition "\[set $cond_var\] > $number"
            }
            under - below {
                set condition "\[set $cond_var\] < $number"
            }
            within - between {
                if {[llength $args] != 2 ||
                        [lindex $args 0] ne "and"} {
                    error "Invalid \"$cond_cmp\" arguments"
                }
                set upper [lindex $args 1]
                set condition "\[set $cond_var\] > $number && \[set $cond_var\] < $upper"
            }
        }
    }

    append Script [string map [list \
                                    %CATEGORY% $category \
                                    %CONDITION% $condition \
                                    %AMOUNT% $amount \
                                    %USE_PERCENT% [expr {$percent ne ""}]] \
                        $template]
}
```

❶ Deduction is a percentage
❷ Deduction is an absolute value
❸ Allocate deduction to a category...
❹ ...and remove net amount accordingly
❺ No condition for the deduction

The other DSL statement is generate. The generation of the script fragment here is much simpler.

```
oo::define Paymaster method generate {what} {
    switch -exact -- $what {
        paycheck {
            append Script {puts "Pay to $Name the amount of $NetAmount"} \n
        }
        paystub {
            append Script {
                puts "Paystub: $Name Salary=$Salary Net=$NetAmount"
                puts "         Fed=$FederalTax State=$StateTax Insurance=$Insurance"
            } \n
        }
        default { error "No means of generating \"$what\"" }
    }
}
```

Now we define the method that will be called by the application to pay an employee. The employee record is passed in the form of a dictionary with the employee's name and salary. It then executes the generated Tcl script within the scope of the dictionary in our calculator slave.

```
oo::define Paymaster {
    method pay {emp} {
        set emp [dict merge {
            FederalTax 0
            StateTax 0
            Insurance 0
        } $emp]
        dict set emp NetAmount [dict get $emp Salary]
```

```
        calculator eval [list set emp $emp]
        calculator eval [list dict with emp $Script]
    }
}
```

Our final method is just for the purposes of inspecting the Tcl script we generated. Just so we can see what havoc we have wrought.

```
oo::define Paymaster method script {} {
    return $Script
}
```

We are ready to try our DSL. Let us create our paymaster, passing in the rules for payment. Normally these would be read in from some resource edited by the accountant.

```
Paymaster create paymaster {
    deduct insurance 100
    deduct federal 10% when salary between 20000 and 30000
    deduct federal 20% when salary above 30000
    deduct state 5% when federal above 2500 ❶
    generate paycheck
    generate paystub
}
→ ::paymaster
```

Just for kicks let us see what our generated script looks like.

```
% paymaster script
→
            if {true} {
                if {0} {
    set deduction [expr {int(($Salary * 100)/100)}]; # Deduction is a percentage
                } else {
...Additional lines omitted...
```

Not pretty, but then again neither are we. At least it seems functional so we can try it out.

```
% paymaster pay {Name Guido Salary 25000}
→ Pay to Guido the amount of 22400
  Paystub: Guido Salary=25000 Net=22400
           Fed=2500 State=0 Insurance=100
% paymaster pay {Name Fredo Salary 35000}
→ Pay to Fredo the amount of 26150
  Paystub: Fredo Salary=35000 Net=26150
           Fed=7000 State=1750 Insurance=100
```

Extending this DSL is fairly easy. You just need to add another aliased method for the new statement, for example reimburse.

Let us reiterate what we have accomplished. In essence, we have implemented a domain specific language using syntax that would be natural to a non-programmer. Moreover we have done this without needing to write a parser and in just about a page of code.

And if you are not sufficiently impressed, think about how you would make our DSL **localized** so a French accountant could use the terms he is comfortable with. It would only take a few changes with the help of message catalogs (see Section 4.15).

That's Tcl; very few languages are as capable.

20.10. Chapter summary

This chapter described the use of multiple Tcl interpreters within a single application. Multiple interpreters bring two primary capabilities to the language:

- encapsulation of state into an independent computing unit
- provision for safe and controlled environments in which to run untrusted components

We also demonstrated the use of multiple interpreters for two common use cases — network servers and a DSL implementation — that benefit from these capabilities.

21

Coroutines

Programs are composed out of computational tasks which may themselves be recursively decomposed further into subtasks. There may be dependencies between these tasks where a task may depend on the result of one or more other tasks before it can proceed with its own computation. *Concurrency* refers to the ability of two or more such tasks to progress at the "same time". When we say "same time", we do not necessarily mean that both computations are running at any particular instant. The computations may in fact do that, running in parallel on separate processors, or they may be interleaved on a single processor so that each gets some share of the processor time. In either case, they all progress without any particular task having to be completed first.

Parallelism on the other hand refers to tasks being able to execute **simultaneously** on separate processors so that at any instant they are actually all changing state in some fashion. By definition then, parallelism implies availability of multiple processors each of which is running a separate task.

> For a short introduction to the two concepts and their semantic difference, see Rob Pike's conference talk.

We have already seen an example of concurrency in Tcl in our discussion of network server implementations. The service provided by a network server to each client can be thought of as a computation task and thus actively servicing multiple clients at the same time through the event loop and asynchronous I/O mechanisms can be thought of as one form of concurrent processing.

We will now look at another form of concurrency using *coroutines*. Like the event loop and asynchronous I/O mechanisms, this allows for concurrent tasks **but not parallel** ones. In the next chapter we will examine yet another alternative — *threads* — which allows tasks to progress not just concurrently, but in parallel as well.

At some level, even a procedure can be viewed as a computational task. It is called with some set of input values, performs a computation based on those values and produces a result which may include side effects. Coroutines are similar to procedures in terms of how they are invoked but with one important difference: they run on their own **separate** call stack that maintains the "state" of the computation. Unlike procedures, which lose their **implicit** internal state (local variables etc.) once they return, coroutines can return values (we call this *yielding*) **while maintaining their internal state**. When invoked again, they can then resume from where they left off.

We will start off by describing the mechanics of creating and using coroutines. That will allow us to then present their motivation, benefits and working in detail through examples in subsequent sections.

21.1. Creating coroutines: `coroutine`

A coroutine is created with the `coroutine` command.

```
coroutine CORO CMD ?ARG ...?
```

This creates a new coroutine context called *CORO* and an associated Tcl command of the same name. The coroutine context consists primarily of a call stack that is used for the execution of all code run in that coroutine's context. The coroutine begins with execution of the command *CMD* which is passed any additional arguments

provided. This command and any further commands it may call will all run off the newly created coroutine call stack. We will describe the call stacks further in a later section.

> Although the coroutine runs with a separate call stack, it still runs in the same interpreter context with the same procedure, global and namespace definitions.

The return value of the coroutine commmand is the value returned by *CMD* either when it completes normally, or a value it returns while suspending itself with the `yield` command. Since we have not discussed the latter yet, here is a trivial example of a coroutine that immediately completes normally.

```
proc adder args { return [tcl::mathop::+ {*}$args ] }
coroutine mycoro adder 1 2 3
→ 6
```

This command will create a new coroutine named `mycoro` with its own stack and execute the `adder` procedure within that coroutine context. Our simple procedure computes the sum and immediately returns. The coroutine, along with its associated context, are deleted on the procedure completion. The procedure return value is returned as the result of the `coroutine` command.

All in all, this is just an inefficient way of calling `adder` directly. There is not much point in a coroutine with an independent stack without a means to suspend its execution while preserving the stack context. That's where we will go next.

21.2. Suspending and resuming coroutines

Procedures run to completion returning a result when they are done. As we saw above, coroutines may also behave the same way but their distinguishing ability is being able to suspend their own execution while returning a result to the caller. While suspended, their call stack and context is preserved so that when next invoked they continue to execute from the point of suspension.

This magic is accomplished with the `yield` and `yieldto` commands.

21.2.1. Yielding to the caller: `yield`

Just like the `return` command in the case of procedures, the `yield` command invoked from a coroutine passes control back to the caller. The difference is that the coroutine call stack context is preserved.

```
yield ?RESULT?
```

The command may only be invoked from within a coroutine context, i.e. from the command passed to the `coroutine` command as its *CMD* argument or from any other command invoked from it to any depth. It preserves the state of the current coroutine context and reverts control back to the coroutine's caller returning *RESULT*, which defaults to the empty string, as the result of the coroutine invocation. While the caller continues execution, the coroutine is suspended inside the `yield` command. The coroutine may be called again at any time by its name and passed a single argument at which point it resumes from its suspended state inside of the `yield` command within the coroutine context. The argument passed to the coroutine command is returned as the result of the `yield` command.

From the caller's perspective, the whole sequence of invoking a coroutine looks similar to the following:

```
set result [coroutine CORO CMD ?ARG ...?]
set result [CORO ?ARG?]
set result [CORO ?ARG?]
```

The first line above creates and invokes the coroutine. When the coroutine yields, we can invoke it repeatedly using its name. On each such invocation, the corouting resumes its computation from the point of its suspension.

As always, an example will make it all, ahem, crystal clear.

```
coroutine mycoro eval {
    set j [yield 1]
    yield [incr j]
    return 0
}
→ 1
```

The above command creates a coroutine context called mycoro and a command of the same name. This coroutine begins executing the eval command, **running on a newly allocated stack**. When the first yield command is executed, the coroutine execution is suspended and the passed value 1 is returned to the caller of the coroutine command. This is what we see as the result above.

Since the coroutine **yielded** back to the caller and did not complete via an implicit or explicit return, it remains suspended with its call stack preserved and can be reinvoked with an optional argument.

```
mycoro 10 → 11
```

This passes control back to the mycoro coroutine which resumes execution by returning from the yield command. The result of yield, 10, is assigned to variable j. The coroutine execution continues with the increment of j which is then passed to the second call to yield resulting in mycoro returning 11 above.

We invoke the coroutine one more time.

```
mycoro → 0
```

Here we did not bother to pass the optional argument to mycoro and yield therefore would return an empty string as its result. When the coroutine resumes, the next command is a return, not a yield. The returned value 0 is the result of the mycoro call but having completed execution, the coroutine context and call stack are freed and the corresponding command removed. An attempt to call the coroutine will result in an error.

```
% mycoro
∅ invalid command name "mycoro"
```

There are a couple of other points about yielding from coroutines that should be noted. The first is that though our example yields directly from within a simple eval, a coroutine can yield from any level in nested procedure calls. Our example could also be mutated as follows:

```
proc demo1 {} {
    set j [yield 1]
    demo2 [incr j]
}
proc demo2 {val} {
    yield $val
}
proc demo {} {
    demo1
    return 0
}
```

We can then define the coroutine as below and run it in the same manner as before.

```
coroutine mycoro demo → 1
mycoro 10            → 11
```

You might expect yielding from a nested procedure call to work and wonder why we spell it out explicitly. This is only because some popular languages that support coroutines only allow yielding from the top level of the coroutine context.

The other point to note is that when a coroutine suspends with the `yield` command, it may only be called with at most one argument. Thus a call like

```
mycoro arg1 arg2
```

will fail. This is not a limitation because (a) multiple arguments can be wrapped and passed as a list, and (b) this capability can be easily added with a wrapper around the `yieldto` command that we will see in the next section.

21.2.2. Yielding after initialization

You will often see the main body of a coroutine contain what appears to be an extraneous `yield` before the main section. For example, we want the coroutine `naturals` to return successive integers starting with 1. We may write it in the following straighforward fashion.

```
proc natural_loop {} {
    while 1 {
        yield [incr i]
    }
}
coroutine naturals natural_loop
→ 1
```

However, the very first `yield` executed in a coroutine is the value of the `coroutine` command itself as seen above. So the first natural number is returned as the result of `coroutine` call and the first `naturals` invocation returns 2.

```
naturals → 2
```

This is a little awkard and has to be treated specially. Therefore, you will often find coroutines structured such that there is a special `yield` after the coroutine does any internal initialization required. The purpose of this `yield` is simply to return to the caller. The coroutine can then be invoked by its own name for its core functionality.

Our rewritten example would look like

```
proc natural_loop {} {
    set i 0 ❶
    yield ❷
    while 1 {
        yield [incr i] ❸
    }
}
coroutine naturals natural_loop
naturals
→ 1
```

❶❶ Initialization. Superfluous for our example.
❷   Returns to caller
❸   Core functionality

That gives us the desired behaviour. You will find this pattern more often than not in all but the simplest coroutines.

21.2.3. Yielding to arbitrary commands: `yieldto`

In Chapter 10, we described the `tailcall` command which returns from a procedure. However, unlike the `return` command, `tailcall` does not pass control back to the caller but instead invokes a target command replacing the call frame of the containing procedure invocation with a call to the target command. The result seen by the original caller of the procedure is then the result of the target of the tailcall.

The `yieldto` command is analogous to the `tailcall` command. Like `yield`, it suspends the coroutine in whose context it is invoked but instead of returning control to the caller, it invokes a target command using the same call frame that was used for the coroutine invocation. As with tailcall, the result of the original coroutine invocation as seen by the caller is the result of the target command invoked via `yieldto`.

```
yieldto CMD ?ARG ...?
```

Here *CMD* and optional *ARG* arguments comprise the target command to which control is to be passed. Note that *CMD* **may itself be another coroutine**, presenting a way of implementing a "co-operative multitasking" architecture.

Like `yield`, `yieldto` can only be invoked from within a coroutine context and results in that coroutine being suspended. It differs from `yield` in the following respects:

- Whereas `yield` always returns control to the caller of the coroutine, `yieldto` will transfer control to the specified target command.
- When a coroutine suspended using `yield` is resumed, it can be passed at most one argument. On the other hand, a coroutine suspended with `yieldto` can be passed multiple arguments on resumption. The return value from `yieldto` when the coroutine is resumed is the list of arguments that were passed to the coroutine invocation.

The first difference is illustrated by the following example.

```
% coroutine mycoro eval {
    yield
    yield abc ❶
    yieldto string length abc ❷
}
% mycoro
→ abc
% mycoro
→ 3
```

❶   Caller sees abc as the return value
❷   Caller sees result of `string length` as return value

The second difference with respect to `yield` allows a simple way to construct a `yieldm` that will behave like a `yield` in that it returns control to the caller but still allows the coroutine to be resumed with multiple arguments.

```
proc yieldm {{val {}}} {
    yieldto return -level 0 $val
}
```

(You may wish to revisit Section 11.3 for a refresher on the `return` command's `-level` option.) The implementation of `yieldm` uses `yieldto` to allow the coroutine to be resumed with multiple arguments while passing control to the `return` command to return the specified value to the caller.

As an example of use, consider implementing a coroutine that accumulates the sum of values passed to it and returns the accumulated total on each call.

```
proc accumulate {} {
    set sum 0
    while {1} {
        incr sum [+ {*}[yieldm $sum]]
    }
}
coroutine accumulator accumulate
→ 0
```

The use of `yieldm` in lieu of `yield` permits the multiple arguments to be passed to the coroutine in an invocation.

```
accumulator 5      → 5
accumulator 1 2 3 → 11
```

The above illustrated one feature of `yieldto`: the ability to accept multiple arguments on resumption of the coroutine. We will find applications of the other feature, handing off control to arbitrary commands, as we move along in this chapter.

21.3. Checking coroutine context: `info coroutine`

The `info coroutine` command returns the name of the current coroutine context or an empty string if called from outside a coroutine context.

```
% info coroutine ❶
% coroutine demo eval {
    yield
    yield [info coroutine]
    yield [info coroutine]
}
% demo
→ ::demo
% rename demo demo2
% demo2 ❷
→ ::demo2
```

❶ Empty string returned because not within a coroutine context
❷ Renaming a coroutine will be reflected in `info coroutine` as well

The command is generally used in library routines that may change their behaviour based on whether they are running within a coroutine context or not. For example, the `coroutine::auto` package in Tcllib [1] provides a set of I/O commands that use this to hide differences between coroutine and non-coroutine contexts. It implements a version of `read`, `gets` etc. that will behave like the standard `read` outside of a coroutine context. Within coroutines however, it will only block the calling coroutine while yielding to permit other coroutines and non-coroutine scripts to run.

21.4. Coroutine termination

A coroutine and its associated command will continue to exist until it runs to normal completion, raises an uncaught exception, or until the command is deleted with the `rename` command. We will look at exceptions in Section 21.5 so here we will consider just the other two methods.

The first of these is simple — when the command used to create the coroutine completes, the coroutine is also terminated.

---

[1] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

```
% coroutine demo eval {yield 0; return}
→ 0
% demo
% demo ❶
∅ invalid command name "demo"
```

❶　Error because the coroutine has completed.

There is a variation of this when a coroutine wants to terminate from within a nested procedure call. This is shown below.

```
% proc exit_coroutine {} {
    return -level [info level]
}
% proc demo_proc {} {
    puts "In demo"
    yield
    exit_coroutine
    puts "Exiting demo"
}
% coroutine demo demo_proc
→ In demo
% demo
% demo
∅ invalid command name "demo"
```

Notice that the Exiting demo message is never printed and moreover, the second call to demo fails as the coroutine no longer exists. See Section 11.3 if you need a refresher on the -level option of the return command.

An alternate method of terminating a coroutine is to rename the corresponding command to the empty string. The coroutine may even commit suicide by renaming itself within its own context. In such a case, the coroutine will continue to run until it yields at which point it will be deleted. The following example illustrates this.

```
% coroutine mycoro eval {
    yield
    rename mycoro "" ❶
    puts "Goodbye cruel world!" ❷
    yield ❸
    puts "Exiting coroutine"
}
% mycoro
→ Goodbye cruel world!
% mycoro ❹
∅ invalid command name "mycoro"
```

❶　Deletes the coroutine command.
❷　Coroutine continues to run until the next yield
❸　Coroutine destroyed after yielding and cannot be called again.
❹　We never see the Exiting coroutine message

There is one important difference in the two methods described above. Use of the return command can be caught within the coroutine context to prevent termination of the coroutine. This is not possible with the rename method which will terminate the coroutine with extreme prejudice after the next yield. Your choice will depend on what's appropriate for your application scenario.

21.4.1. Releasing resources on termination

The fact that termination via the rename command will not run any exception handlers has ramifications with regards to resource management. Let us look at the following simple example which returns a line from a file on each invocation.

```
coroutine getline {*}[lambda path {
    set chan [open $path]
    try {
        while {[gets $chan line] >= 0} {
            yield $line
        }
    } finally {
        close $chan
    }
}]
```

The code takes care to ensure that the opened channel is closed on both normal and exceptional completions. However, consider what happens if the coroutine has yielded and the caller does a

```
rename getline ""
```

The rename would literally make the coroutine just go Poof! — "disappear" without a trace. In particular the finally handler would never run leaving the file open resulting in channels being leaked.

A robust design warrants that we should guard against this possibility and variable traces provide a way. We can write the coroutine / procedure as follows:

```
coroutine getline {*}[lambda path {
    set chan [open $path]
    trace add variable DUMMY unset [list close $chan]
    while {[gets $chan line] >= 0} {
        yield $line
    }
}]
```

Now when the DUMMY variable goes out of scope, whether it is by the coroutine completing normally or with an exception or **even via the coroutine being deleted as described above**, our trace will trigger and close the file.

Note that we did not need to even have a value assigned to DUMMY (see Section 10.6.1.1).

21.5. Exception handling in coroutines

Any exceptions that are not caught within a coroutine are propagated to the top level of the coroutine context and passed back to the caller of the coroutine. Moreover, the **coroutine** context and associated command are deleted so that the coroutine cannot be invoked again.

We can verify this behaviour by passing our accumulator coroutine a value that is not a number.

```
% accumulator notanumber
Ø can't use non-numeric string as operand of "+"
% accumulator 3 ❶
Ø invalid command name "accumulator"
```

❶   Fails because the above error deletes the coroutine

In many cases, it is desirable to not terminate the coroutine but simply notify the caller of the error while preserving the coroutine context so it can be called again. We can accomplish this by combining yieldto

with one of the exception raising commands. We demonstrated this technique by modifying our `accumulator` implementation as shown below.

```
proc accumulate {} {
    set sum 0
    while {1} {
        if {[catch {
            set part_sum [+ {*}[yieldm $sum]]
        } result ropts]} {
            set part_sum [+ {*}[yieldto return -options $ropts $result]]
        }
        incr sum $part_sum
    }
}
coroutine accumulator accumulate
→ 0
```

Our modified implementation catches any exceptions raised in the sum operation so that they are not propagated to the top level of the coroutine which is therefore not terminated. At the same time, we need to pass the exception with its original error stack to the caller. We already saw how we can do this for the normal procedural case in Section 11.6.1. Here we cannot use that technique directly because that would again terminate the coroutine. Instead we use `yieldto` to pass the buck. That will suspend the coroutine and run the `return` target command in the context in which the coroutine was invoked. Thus we achieve the goal of returning the exception while preserving the coroutine.

Now when an error occurs, an exception is raised to the caller as before but the coroutine is not terminated and can be invoked again.

```
% accumulator 1 2 3
→ 6
% accumulator notanumber
∅ can't use non-numeric string as operand of "+"
% set errorInfo ❶
→ can't use non-numeric string as operand of "+"
      while executing
  "+ {*}[yieldm $sum]"
      invoked from within
  "accumulator notanumber"
% accumulator 4
→ 10
```

❶   Notice the error stack of the error is preserved as promised

21.6. Variable scopes in coroutines

As we have stated earlier, every new coroutine begins life with its own spanking new stack. The execution however begins in the global context. This is important to keep in mind as the following example shows.

Suppose we had written our `naturals` coroutine example from an earlier section as follows:

```
coroutine naturals eval {
    yield
    while {1} {
        yield [incr i]
    }
}
```

This seems to work as well as the previous version and does not need a separate procedure definition.

```
naturals → 1
naturals → 2
```

The problem is that the code actually modifies the variable i in the **global** scope.

```
set i    → 2
naturals → 3
set i    → 3
```

This is undesirable as global variables should be avoided as far as possible. For example, the above implementation would not permit two independent natural number generators to co-exist. Their use of the i global would cause interference between themselves.

For this reason, it is advisable to have the top level of a coroutine to be invoked as a named or anonymous procedure in all but the simplest cases. Any use of variables then is within the scope of that procedure. We follow this advice in all but the simplest of our examples.

21.6.1. Private variables

The fact that a coroutine runs on its own stack can be taken advantage of to implement "private" variables that are accessible only from procedures running in that coroutine's context. This is illustrated by the example below from Tcler's Wiki[2]. This defines a procedure corovars:

```
proc corovars {args} {
    foreach var $args {
        lappend vars $var $var
    }
    tailcall upvar #1 {*}$vars
}
```

The procedure is simply a wrapper to link local variables in the caller to the local variables of the same name at level 1 in the coroutine stack. It can be used as a "declaration" similar to how the global or variable commands are used to declare variables in the global and namespace scopes except that it declares variables in the "coroutine" scope.

Here is a demonstration of its use.

```
proc init {} {
    corovars x y
    set x 100
    set y 200
}
proc getx {} {
    corovars x
    yield $x
}
proc demo_helper {} {
    init
    yield
    getx
}
coroutine demo demo_helper
demo
→ 100
```

---

[2] http://wiki.tcl.tk

Notice how the "coroutine-private" variable x is referenced from two different procedures running in the coroutine context while remaining inaccessible outside.

21.7. Coroutines, uplevel and upvar

A point to be noted about coroutines vis-a-vis procedure calls is that because coroutines run off an independent stack, the uplevel and upvar commands do not work the same way in the two cases. A procedure may use these commands to run code or access variables in the caller's context. This is not possible with coroutines. We can see this simply by checking the info level command.

```
% coroutine mycoro while 1 {yield [info level]}
→ 0
% proc demo {} {puts "demo: [info level]"; puts "coro: [mycoro]"}
% demo
→ demo: 1
  coro: 0
```

Notice the coroutine is running in the global context and has no access to the stack of the caller demo. Of course, all nested procedure calls within a coroutine context can use upvar and uplevel as normal within that context.

There is however a way to emulate the functionality of upvar and uplevel with some explicit programming. The idea is to use yieldto to execute a script that does the needful in the caller's context and then calls back into the coroutine. The technique, accompanied with code and an explanation, is described in TIP #396: Symmetric Coroutines, Multiple Args, and yieldto.

21.8. Coroutines and multiple interpreters

Coroutines are limited to the interpreter in which they are defined. For example, the following generates an error.

```
% set ip [interp create]
→ interp0
% coroutine mycoro $ip eval {yield 1}
Ø yield can only be called in a coroutine
```

The error occurs because the coroutine context is defined within the master interpreter but the attempt to yield is made from the slave. If your situation calls for running coroutines within a slave, the coroutine must be defined in that slave. Then you can access it using interp eval.

```
$ip eval {
    coroutine mycoro try {yield 1; yield 2; yield 3}
}
$ip eval mycoro
→ 2
```

Optionally, you can create an alias for some syntactic sugar.

```
% interp alias {} mycoro $ip mycoro
→ mycoro
% mycoro
→ 3
```

21.9. Code injection

When a coroutine is resumed, it continues execution from the statement following the yield or yieldto command that had suspended the coroutine. The inject command offers a means of inserting code into the

coroutine at the point of resumption so that the next time the coroutine is executed, it resumes with the inserted code instead of with the statements following the yield.

The syntax of the command is

```
tcl::unsupported::inject CORO CMD ?ARG ...?
```

The first thing to note is that the command lies in the tcl::unsupported namespace. This means it is still experimental and may change syntax, behaviour or even be removed in future releases. It would not be wise to rely on it in production code despite Tcl's history of stability and compatibility. We describe the command here because it can be useful for debugging and troubleshooting.

The command arranges for CMD to be executed, within the context of the coroutine CORO, with any provided ARG arguments the next time CORO is invoked.

Let us define the following simple coroutine.

```
coroutine slogan apply {{} {
    while {1} {yield ; puts "Tastes great!"}
}}
```

Every time we invoke it, it returns from yield and prints our slogan and yields again.

```
% slogan
→ Tastes great!
```

Now we inject other code to run when the coroutine is invoked.

```
% tcl::unsupported::inject slogan puts "Less filling!"
```

Note that our code is not run yet. It will be executed when the coroutine is invoked next.

```
% slogan
→ Less filling!
  Tastes great!
```

As you see above, when the coroutine is resumed, it first runs our injected code, printing Less filling!. The injection is a one-time deal, not permanent so if you call the coroutine again, it will be back to the old behaviour.

```
% slogan
→ Tastes great!
```

Note that the coroutine can run valid commands in the coroutine context. For example, it could print and yield right away before the original code runs.

```
% tcl::unsupported::inject slogan try {puts "Less filling!" ; yield}
% slogan
→ Less filling!
```

Or even terminate the coroutine by executing a return.

```
% tcl::unsupported::inject slogan return
% slogan
```

The return causes the coroutine to complete on the following call we made above. So a succeeding attempt to run it again will meet with an error.

```
% slogan
→ invalid command name "slogan"
```

The primary purpose for inject should be for inspecting the state of a coroutine that has yielded. For example, a coroutine might have multiple yield points. To figure out where it has yielded and other state information, you can inject a command to dump the stack and yield. Then invoke the coroutine to collect the information.

As you can imagine, code injection can modify coroutine behaviour in powerful ways but its use should be limited to debugging and troubleshooting until it is moved out of the unsupported namespace.

21.10. Using coroutines

Having discussed the mechanics of working with coroutines, we now explore the motivation behind them and their use in some common software patterns. A key point to remember is that coroutines do **not** add any new capabilities to the language. What they do is enable simpler program structure that is easier to write and understand. We will attempt to justify that claim in the next few sections.

21.10.1. Explicit and implicit state

To understand how coroutines can simplify programming, let us first expound a little about the state encapsulated by a computation. This state can be thought of as being a combination of *explicit* and *implicit* states. The former is named such because the program has to explicitly maintain the state as a collection of global, namespace and object variables. The implicit state on the other hand is automatically kept (well, kind of) in the form of the call stack as execution proceeds through a sequence of procedure calls. At any point of time, the explicit and implicit states together tell us "where we are in the computation".

Let us illustrate the difference between the two with an example. Consider implementing a task which given a list will return the next element in the list on each subsequent call. When the end of the list has been reached, it should wrap back to return elements from the beginning of the list.

Our first implementation will use objects. We could use namespaces as well but that would involve a little more bookkeeping code to deal with the case where we want to use multiple such instances.

```
oo::class create Lgen {
    variable List
    variable Index
    constructor {l} {
        set List $l ❶
        set Index -1
    }
    method next {} {
        incr Index
        if {$Index >= [llength $List]} {
            set Index 0
        }
        return [lindex $List $Index]
    }
}
Lgen create onotes {do re mi}
→ ::onotes
```

❶ We will not bother with error conditions like empty lists

The state of computation is maintained through the member variable List and Index. Not exactly rocket science because the state information is very simple. We can verify that our implementation meets our requirements.

```
onotes next → do
onotes next → re
```

In contrast, here is a coroutine based implementation that does the same thing through a coroutine.

```
proc list_iterator {lst} {
    yield
    while {1} {
        foreach elem $lst {
            yield $elem
        }
    }
}
coroutine conotes list_iterator {do re me}
```

Just to prove that it works, we try it out.

```
conotes → do
conotes → re
```

The point of our little exercise was to contrast explicit and implicit state. The object based implementation stores explicit state, in the form of the object member variables List and Index, to keep track of the value to be returned on the next call. On the other hand, in the coroutine-based implementation, the state information is implicitly contained in the while and foreach loops. This makes the intent of the code more immediately obvious compared to the object-based implementation. This is true even in our trivial example but the difference can be even more pronounced in real world programming tasks with more complex program states.

21.10.2. Generators

One of the simplest uses for coroutines is for implementing *generators*. A generator is a command that returns successive elements from a collection on each invocation. Our naturals coroutine and the example in the previous section were essentially generators. In the former case, the "collection" did not actually exist in concrete form (it is infinite after all) but its elements were generated on demand. In the latter case, the generator returned elements of an existing simple (repeated) list.

Here we illustrate a couple of slightly more complex examples of generators, again with the purpose of demonstrating how coroutines simplify their implementation.

Consider a collection of integers stored as a nested list of **arbitrary** depth.

```
% set nested {0 {1 {2 3}} {4 5}}
→ 0 {1 {2 3}} {4 5}
```

Writing a recursive routine to invoke a command on each element of such a tree-like structure is very straightforward.

```
proc iterate {l cmd} {
    set n [llength $l]
    if {$n > 1} {
        foreach e $l {
            iterate $e $cmd
        }
    } elseif {[llength $l] == 1} {
        {*}$cmd [lindex $l 0]
    }
}
```

We can then apply some arbitrary operation to each element, for example output its square.

```
% iterate $nested [lambda i {puts [expr {$i*$i}]}]
→ 0
  1
  4
...Additional lines omitted...
```

However, this is not what we are after. What we actually want is an iterator command that would return the successive integers from this collection on each call.

Generators versus callbacks

The callback model of iteration where a script or command is invoked for every element in a collection is simple and adequate for many uses. However, generators are more flexible in some regards, for instance when the elements are not processed immediately. An example might be if each element was a token to be passed to a client on connecting. Clearly, this would not be suitable for a callback based implementation. Other cases where callbacks are not suitable is when processing of the element depends on significant local state which cannot be passed in the callback although uplevel and upvar can help some in that regard.

Converting recursive code to an iterator command is normally not possible because by its very nature the recursion encodes the state of the computation and will be lost when the iterator returns. On the other hand, writing a non-recursive implementation that explicitly keeps track of state involves quite a bit more bookkeeping.

Coroutines come to the rescue. Because they can return values while preserving execution state, we can easily wrap a coroutine around the above to construct an iterator command.

```
proc iterator_wrapper {collection} {
    yield
    iterate $collection yield
}
coroutine iterator iterator_wrapper $nested
```

We can now retrieve one element at a time.

```
iterator → 0
iterator → 1
iterator → 2
```

To better appreciate the simplicity of the above iterator implementation, the reader might want to write one without using coroutines.

Many algorithms, such as tree traversal above, are most clearly expressed in recursive form. In many cases, coroutines can encapsulate these recursive algorithms in a manner that allows them to be used in an iterative manner. Our example was a simple illustration of this.

As another more real world example assume we want an iterator that will return each file or directory within a directory tree. We saw the fileutil::find command in Chapter 9 that will return a list of files within a directory.

```
% print_list [fileutil::find c:/temp]
→ c:/temp/fromDir
  c:/temp/newDir
  ...Additional lines omitted...
```

However, we do not want this entire list of files at one shot for many reasons. It may consume too much memory, we may only be interested in the first few entries and so on. We would therefore like an iterator that will return a single entry at a time. Now it so happens that the `fileutil::find` command takes a second optional argument that is the name of a filter command. This command is invoked for each file or directory. If the command returns a boolean true value, the file is included in the returned list. We will misappropriate this feature for our own use.

```
proc yield_one {fname} {
    yield [file join [pwd] $fname]
    return 0
}
proc file_iterator {dir} {
    yield
    fileutil::find $dir yield_one
}
```

We have cunningly fooled `fileutil::find` to yield as part of its filter functionality. Let us try it out.

```
% coroutine nextfile file_iterator c:/temp
% nextfile
→ C:/temp/fromDir
% nextfile
→ C:/temp/newDir
% while {[set next [nextfile]] ne ""} { puts $next }
→ C:/temp/fromDir/fileA.txt
  C:/temp/fromDir/subDir
  C:/temp/newDir/fileA.txt
  ...Additional lines omitted...
```

Once again, the benefits of coroutines are best understood by implementing the above example without their use. **The ability to return values from a computation while preserving its implicit state greatly simplifies programming.**

21.10.2.1. The generator package

Generators are commonly used in programming and have essentially the same structure in most instances and need to support the same set of high-level operations such as map and fold, something we have not discussed in our earlier examples. Thus it makes sense to implement a framework that provides the common functionality required for generators. The generator package in Tcllib[3] does exactly that.

The package covers a lot of functionality with many commands and is well documented so we will only introduce basic usage here. You are encouraged to read its documentation to get a feel for all the ways it can be used.

```
package require generator
→ 0.1
```

All commands in the package are implemented via the generator ensemble command.

Our first example is to recreate the naturals command we implemented earlier, this time using the generator package. The generator define command is used to define the generator implementation.

```
generator define naturals {} {
    while 1 { generator yield [incr i] }
}
→ ::naturals
```

---

[3] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

This looks similar to our earlier implementation of naturals but there are a couple of notable differences. The obvious one is that the implementation uses generator yield, and not yield to yield values. Less obvious is that invoking naturals will not return the next natural number. Rather, it returns a *generator instance* which can then be used to iterate through the sequence.

```
% set nseq [naturals]
→ ::generator::generator1
```

The generator next command returns one or more elements from the generator instance.

```
generator next $nseq first                → (empty)
generator next $nseq second third fourth → (empty)
puts "$first, $second, $third, $fourth"  → 1, 2, 3, 4
```

Of course, we may create additional generator instances. These will all be independent.

```
set nseq2 [naturals]         → ::generator::generator2
generator next $nseq2 val → (empty)
set val                      → 1
```

The generator instances must be released with generator destroy when no longer required.

```
generator destroy $nseq $nseq2 → (empty)
```

So far, it does not appear there is a whole lot we gain from using the package as opposed to raw coroutines. The real benefits come from the additional commands in package that implement operations like iteration, folding, mapping and filters.

Iteration

The generator foreach command is like Tcl's foreach except that it iterates over the elements returned by a generator. We can iterate over the sequence of natural numbers using a very natural (pun unintended) syntax.

```
generator foreach n [naturals] {
    puts $n
    if {$n >= 2} {
        break
    }
}
→ 1
  2
```

On completion of the loop, the generator instance (returned by naturals above) used for the loop is automatically destroyed so we do not have clean up.

We have to break out of the above loop because naturals produces an infinite sequence. A finite sequence on the other hand will terminate the loop naturally. The next example defines such a finite generator, all integers between a specified range. We then iterate over it as above.

```
generator define range {low high} {
    for {set i $low} {$i <= $high} {incr i} { generator yield $i }
}
generator foreach n [range 4 6] { puts $n }
→ 4
  5
  6
```

Filtering

The generator filter command creates a new generator containing elements of an existing generator that satisfy a given predicate. Here we apply odd as the predicate condition on a range and iterate over the resulting generator.

```
proc odd {n} { tcl::mathop::& $n 1 }
generator foreach n [generator filter odd [range 2 8]] {
    puts $n
}
→ 3
  5
  7
```

Reducing

Another common operation is implemented by the generator reduce command. This combines an initial value supplied by the caller with each successive element by applying a specified operator.

```
generator reduce ::tcl::mathop::+ 0 [range 1 4] → 10
generator reduce ::tcl::mathop::* 1 [range 1 4] → 24
```

Map

The last operation we will mention is generator map which creates a new generator by applying a function to an existing generator.

```
set powers_of_2 [generator map {::tcl::mathop::** 2} [naturals]]
generator next $powers_of_2 i j k
puts "$i, $j, $k"
→ 2, 4, 8
```

These examples only provide a flavor of the generator package. The most important takeaways from these examples are:

- The elements of generated sequences are not actually instantiated until they are actually needed. This not only allows them to represent infinite sequences but also results in greatly reduced memory requirements for long finite ones.
- Despite the above, transforms and other operators can be applied to the generators as they would be for scalar values. The last example is an illustration of this.

21.10.3. Emulating objects

Coroutines can be used to implement simple object based frameworks. Here is the implementation of a simple calculator object.

```
proc plus {args} {
    corovars acc
    set acc [tcl::mathop::+ $acc {*}$args]
}
proc mul {args} {
    corovars acc
    set acc [tcl::mathop::* $acc {*}$args]
}
proc calculator {} {
    set acc 0
    set args [yieldm]
```

```
    while 1 {
        if {[llength $args] == 0} {
            set args [yieldm $acc]
        } else {
            switch -exact [lindex $args 0] {
                mul  -
                plus {
                    {*}$args
                    set args [yieldm]
                }
                default {
                    set args [yieldto error "Unknown method [lindex $args 0]"]
                }
            }
        }
    }
}
```

Our implementation could be simpler in the case of our example but we wanted to illustrate the use of private variables using the corovars procedure defined earlier to serve as member variables. It is even possible to extend this to support prototype-based inheritance with the help of the yieldto command.

Proof that our calculator works as designed:

```
% coroutine calc calculator
% calc plus 1 2
% calc mul 5
% calc div 2
0 Unknown method div
% calc
→ 15
```

Of course our little object system is missing pretty much all the features that Tcl's native OO system provides. But there are situations where a simple object type interface is useful in conjunction with uses of coroutines, for example, the multitasking system we describe in Section 21.10.6.

21.10.4. Producers, consumers, transformers and filters

The producer-consumer pattern is a common pattern in software where the former generates values out of thin air, figuratively speaking, and the latter consumes them for its own internal purposes. Related abstractions are transformers and filters. Both act as consumers, by retrieving values from a producer, as well as producers, by passing those values to a consumer. The difference between the two is that transformers pass on **all** values after potentially modifying them whereas filters may pass on only a subset of them. This has an effect on how they are implemented.

Below we discuss implementation of all these roles in the context of coroutines.

Producers

A coroutine behaves like a producer by returning values to the caller with the yield command. Because it is a producer, it does not take any input and in effect ignores the result of yield command itself.

We have already seen simple producers, such as the naturals example earlier, but it being a legal requirement to include a Fibonacci sequence generator in every discourse on programming, below is a producer that generates such a sequence.

```
proc fibonacci_generate {} {
    yield
    set prev 0
    set fib 1
    while 1 {
        yield $fib ❶
        lassign [list $fib [incr fib $prev]] prev fib
    }
}
coroutine fib_producer fibonacci_generate
```

❶   Note result of yield thrown away

Consumers

A consumer is a parasite, ingesting values without giving anything back in return. Thus a coroutine playing the role of a consumer does not pass any arguments to yield but processes its result. Our simple consumer simply prints what's passed to it.

```
coroutine print_consumer while 1 {puts [yield]} ❶
```

❶   Note no arguments passed to yield

Transformers

Transformers have both input and output and therefore pass arguments to yield as do producers, as well as use its return values as do consumers. We can write a filter that squares values before passing them on.

```
coroutine squarer {*}[lambda {} {
    set val [yield]
    while 1 {
        set val [yield [expr {$val*$val}]]
    }
}]
```

We can now hook up the producer and consumer with an intermediate transformer following the pattern

CONSUMER [TRANSFORMER [PRODUCER]]

In our example then,

```
print_consumer [squarer [fib_producer]] → 1
print_consumer [squarer [fib_producer]] → 1
print_consumer [squarer [fib_producer]] → 4
```

You can of course extend this chain by adding on additional transformers.

Filters

Filters are a generalized form of transformers in that they remove the restriction of a one-to-one correspondence between input and output values. Most commonly they pass on only a subset of input values (possibly modified) but could potentially insert additional values as well. This generalization means the form we showed above cannot be used for filters. For instance, imagine we had a filter that would only pass through even numbers. We cannot print a list of even fibonacci numbers as

```
print_consumer [evens [fib_producer]]
```

The problem is that if the generated number is not even, the filter cannot pass it to the consumer and nor can it call the producer to retrieve the next number as it is passed values, not the producer command itself. We have to therefore write filters in a slightly different form where we pass the producer command to the filter when it is created.

Our implementation of the a producer-filter-consumer pipeline that prints even fibonacci numbers would look like

```
proc filter_proc {producer} {
    yield
    while 1 {
        set number [$producer]
        if {[expr {$number & 1}] == 0} {
            yield $number
        }
    }
}
```

We can then invoke the pipeline as

```
coroutine fib_producer2 fibonacci_generate      → (empty) ❶
coroutine even_fibs filter_proc fib_producer2 → (empty)
print_consumer [even_fibs]                      → 2
print_consumer [even_fibs]                      → 8
```

❶ Fresh copy of our generator

An alternative to the above would be to have the consumer also be driven by the filter as shown below. As a variation, here we also **insert** additional output elements by preceding each the fibonacci number by its position in the **full** Fibonacci sequence.

```
coroutine fib_producer3 fibonacci_generate ❶
coroutine even_fibs2 {*}[lambda {producer consumer} {
    yield
    while 1 {
        incr seq
        set number [$producer]
        if {[expr {$number & 1}] == 0} {
            yieldto $consumer "Position: $seq"
            yieldto $consumer "Number:   $number"
        }
    }
}] fib_producer3 print_consumer
```

❶ Fresh copy of our generator

Trying it out,

```
even_fibs2 → Position: 3
even_fibs2 → Number:   2
even_fibs2 → Position: 6
even_fibs2 → Number:   8
```

In this last example, we use yieldto to call the consumer. As an exercise, consider how this is different from directly calling the consumer coroutine instead of via yieldto.

21.10.4.1. Coroutines and the event loop

Coroutines are invoked just like any other command so there is no great magic involved in calling them via the event loop. However, interaction with the event loop comes up in some common situations, like I/O and multi-tasking, that we describe in later sections so let us go through a simpler example first.

We would like to print the space taken up by each directory in a directory tree. Given that this might take some time on a large file system, we do not want to block the rest of the application in the meanwhile. One way to do this is through a coroutine that reschedules itself through the event loop after doing a block of work. We first define the worker procedure to recurse through the directory tree.

```
proc print_dir_size {dir} {
    set total 0
    foreach fn [glob -nocomplain [file join $dir *]] {
        incr total [file size $fn]
        if {[file isdirectory $fn]} {
            incr total [print_dir_size $fn]
        }
    }
    puts "$dir: $total"
    after idle after 0 [info coroutine]
    yield
    return $total
}
```

The script is very simple, in part because it ignores errors and special files such as links. After processing each directory, it reschedules itself using the idiom described in Chapter 15. It then gives up control through `yield`, allowing other parts of the application to run. Note the call to `info coroutine` which returns the name of the current coroutine. The event loop will then call back into the coroutine allowing it to resume.

Now if you run this as a coroutine in `wish` or `tclsh` with the event loop running, you will see the output being printed **without the rest of the application being blocked**.

```
% coroutine print_windir print_dir_size c:/windows/system32
→ c:/windows/system32/0409: 0
  c:/windows/system32/AdvancedInstallers: 3252576
  c:/windows/system32/AppLocker: 0
  c:/windows/system32/appraiser: 2532564
  c:/windows/system32/ar-SA/Licenses/OEM/Core: 0
  ...Additional lines omitted...
```

It is possible to rewrite the above without the use of coroutines but the resulting solutions are significantly more complex as in every round trip through the event loop you have to explicitly keep track of the current depth in the directory tree and the node within that depth.

We might have given the impression from our examples so far that coroutines are primarily useful in conjunction with recursive algorithms. That is not so. Coroutines are useful any time significant state information is retained in the call stack. Recursive algorithms are just a special category of these and their simple nature makes them particularly suitable for illustrative purposes.

21.10.5. Emulating blocking calls: `coroutine::util`

As you might have noticed from earlier chapters, particularly those dealing with I/O, code that uses synchronous blocking calls is both simpler to write and easier to read than its asynchronous counterpart. The drawback of course is that the entire application is blocked if a channel is not ready. Let us look at how coroutines might allow us the benefit of a synchronous style without its drawbacks.

Consider a simple application that acts as a network proxy connected to two remote end points. Its sole purpose in life is to copy data received on a connection to the other connection and vice versa. In real life, such a proxy might have a wide variety of uses such as firewalling, load balancing, protocol translation etc.

The core function of the proxy is very simple and ignoring peripheral matters such as connection management, error handling etc., it can be implemented by the following procedure.

```
proc proxy {from to} {
    while {[gets $from data] >= 0} {
        puts $from $data
    }
}
```

The procedure copies data from an input channel to an output channel. (There is an implicit assumption here that the two parties are using a text-based protocol.) Our application then is **conceptually** just a pair of calls, one for each direction of the connection.

```
proxy $chan_a $chan_b
proxy $chan_b $chan_a
```

This of course does not work. There are two problems that need to be solved:

- The I/O calls to read data should not block the application.
- The second call needs to run concurrently with the first whereas as written above it will not be invoked until the first completes.

Our coroutine-based version solves both of the above but needs the `coroutine` package from Tcllib [4]. This package includes several utility procedures that are useful when running in a coroutine context. Once again, this requires the Tcl event loop to be running.

```
package require coroutine → 1.1.3 ❶
```

❶   Required for coroutine::util::gets

We can then write our implementation as follows.

```
proc proxy {from to} {
    while {[coroutine::util gets $from data] >= 0} {
        puts $to $data
    }
}
coroutine atob proxy $chan_a $chan_b
coroutine btoa proxy $chan_b $chan_a
```

Before we get into the explanation of the code, note the similarity of structure with the (non-working) synchronous version we showed earlier. Within the `proxy` procedure we have replaced the `gets` call with a call to `coroutine::util gets`. This takes care of our first problem, blocking. Then instead of calling `proxy` directly, we call it through a coroutine. This takes care of the second problem, concurrency.

The key to the implementation is the `coroutine::util gets` command. This command has the same API as the standard Tcl `gets` command but is intended to be called from a coroutine context so that instead of blocking as `gets` does when no data is available, it invokes `yield` to allow other code to execute while keeping the caller, `proxy` in our case, suspended. It uses the event loop behind the scenes similar to what we described earlier to get

---

[4] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

called back when data is available. It is well commented and self-explanatory so we will not repeat it here. You can see it in the coroutine module of Tcllib or in the online repository[5].

In addition to gets, the coroutine package includes emulation of several other blocking calls, not all of which deal with I/O. For example, here is a short fragment that keeps writing a message to the console at regular intervals that uses an emulation of the blocking mode of the Tcl after command.

```
coroutine print_loop while 1 {
    coroutine::util after 1000; puts "Still going..."
}
```

The Tcl after command with a single argument will block until the specified interval has elapsed. The coroutine::util after command will block from the coroutine caller's perspective but **will yield under the covers allowing other parts of the application to run**.

21.10.6. Co-operative multitasking

We now move on to our final illustration of coroutine use — a framework for co-operative multitasking or *green threads*. Unlike preemptive multitasking which uses native threads at the operating system level, green threads are implemented purely at the user level and require each thread of computation to voluntarily give up the CPU to allow other threads to run. Hence the term *co-operative* multitasking. Tcl also supports native threads as discussed in detail in Chapter 22. We will compare and contrast the two in Section 22.15.

Actors versus Communicating Sequential Processes (CSP)

There are two commonly used models of concurrency — *actors* and *communicating sequential processes (CSP)*. The primary difference between the two is that the former uses asynchronous message passing as the communication mechanism. Messages are sent using a "fire and forget" policy where the sender does not wait for the message to be delivered and processed by the receiver. In the CSP model, messages are sent through synchronous *channels* (not to be confused with Tcl I/O channels) where the sender is blocked from proceeding until the receiver accepts the message. Both models can be implemented with Tcl coroutines as evinced by the fiber[6] and ycl coro relay[7] packages which implement a form of the actor model, and the csp[8] package which implements CSP.

An example actor implementation

For our illustration of coroutines for co-operative multitasking we will construct a package, task, that implements the actor model. Our concurrent units of computation are called tasks. They are implemented as coroutines and voluntarily give up control at appropriate times by yielding to a dispatcher which then gives other tasks a chance to run. Tasks may be completely independent of each other or may interact through *messages*.

Our package is rudimentary and is meant to highlight key considerations involved in implementing a co-operative multitasking framework in Tcl. Amongst other things, a production level package will need to be both more featureful and more robust with better error handling and recovery, resource limits on message queues, a more sophisticated scheduling algorithm to prevent starvations and so on.

We start off by importing the coroutine package from Tcllib[9] and defining our package namespace.

```
package require coroutine
namespace eval task {
    variable tasks {}
}
```

---

[5] http://core.tcl.tk/tcllib/dir?ci=tip&name=modules/coroutine
[6] https://sourceforge.net/projects/tclfiber/
[7] http://wiki.tcl.tk/44051
[8] https://github.com/securitykiss-com/csp/
[9] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

Creating a new task

The `task::task` command creates a new task in which the provided script will be evaluated.

```
proc task::task {script} {
    variable next_id
    variable tasks
    variable tasks_added

    set task_id [namespace current]::task#[incr next_id]
    dict set tasks $task_id State INIT
    dict set tasks $task_id Messages {}
    set tasks_added 1
    coroutine $task_id apply {script {
        yield
        try $script finally { task::cleanup [task::myself] }
    }} $script
    wakeup_dispatcher
    return $task_id
}
```

The `tasks` variable is a dictionary of active tasks keyed by task identifiers which are dynamically generated using the `next_id` counter. The task identifiers also serve as the name of the coroutine hosting the task. To create a new task, the command initializes the state of the task to INIT with an empty `Messages` that will hold any messages sent to it. It then creates a coroutine context in which to run the supplied script. As a general principle, we prefer to run the script within an anonymous procedure so as to avoid potentially polluting the global namespace.

The coroutine immediately yields on entry rather than executing the task script. The reason for this is that we want all execution to be controlled by the dispatcher. Moreover, we do not want the current task to be preempted by the new one unless it voluntarily gives up control explicitly.

Just before returning, we wake up the task dispatcher in case it was previously suspended.

Task identity

A script often needs to know the task under which it is running. The `myself` command returns the id of the current task. If called from outside a task, it raises an exception.

```
proc task::myself    {} {
    variable tasks
    set me [info coroutine]
    if {[dict exists $tasks $me]} {
        return $me
    }
    error "Not running in a task."
}
```

Task cleanup

Any structures allocated by our package need to be cleaned up when the task exits. The `cleanup` command is more or less self explanatory.

```
proc task::cleanup {task_id} {
    variable tasks
    if {[dict exists $tasks $task_id]} {
        catch {rename $task_id ""}
        dict unset tasks $task_id
    }
}
```

Note that this does not release any resources allocated by the task itself, for example open channels. That is the responsibility of the task script.

Sending messages

Now we start coming to the meat of the package, implementation of messages and dispatching of tasks. Our package itself does not care about the content of a message. Its interpretation is left entirely to the receiving task. Sending a message involves placing it on the task's message queue and waking up our task dispatcher in case it is not running. By design, we choose to ignore messages sent to non-existing tasks.

```
proc task::send {task_id message} {
    variable tasks
    if {[dict exists $tasks $task_id]} {
        dict with tasks $task_id {
            lappend Messages $message
        }
        wakeup_dispatcher
    }
    return
}

proc task::wakeup_dispatcher {} {
    variable dispatcher_alarm
    set dispatcher_alarm 1
}
```

We will explain wakeup_dispatcher later when we describe the dispatcher implementation.

Note that as stated earlier message passing is asynchronous in our framework. The sender does not wait for the receiver to accept or process the message.

The message queue

It is worth digressing a little bit at this point to elaborate on a design choice we made to use an explicit message queue as opposed to alternative implementations that you might find being used elsewhere.

The first of these dispenses with queueing of messages altogether. Sending a message to a task is implemented by calling the corresponding coroutine, either directly or via yieldto, passing the message as an argument. You might think of this as a *push* model where messages are pushed to the task whereas ours is a *pull* model where messages are explicitly read by the task with the receive command described later. There are two problems with the push method in a general purpose package like ours:

- The task (coroutine) might have yielded for reasons other than for the purpose of receiving messages. For example, the task may use calls such as coroutine::util gets or coroutine::util after that we saw in previous sections. In such cases, it expects to be resumed from the gets I/O handler or from the after timer implementation. Being resumed with a message argument at that point instead of the result of a gets would completely confuse the poor coroutine.

- Directly pushing messages to coroutines can lead to issues of reentrancy that can be difficult to diagnose and resolve.

The other implementation you might see is the use of Tcl's event queue for passing messages instead of the explicit queue in our design. This also is essentially a *push* model where it is the event dispatcher that is pushing messages to the tasks. This does not have the reentrancy problems but suffers from the first issue noted above.

Maintaining an explicit message queue where tasks pull messages avoids these issues. It also has the added benefit of allowing more sophisticated capabilities such as message priorities, pattern matching etc.

Receiving messages

The complementary command to send is recv which returns the next message from the tasks message queue.

```
proc task::recv {} {
    variable tasks
    set me [myself]
    while {1} {
        set msgs [dict get $tasks $me Messages]
        if {[llength $msgs] != 0} {
            dict set tasks $me Messages [lassign $msgs msg]
            return $msg
        } else {
            dict set tasks $me State RECEIVE
            suspend
        }
    }
}
```

The implementation is again more or less obvious. If the message queue is not empty, the first message is removed and passed back to the task. Otherwise, the task suspends itself thereby allowing other tasks to run.

Note that in our simple implementation, if a task continuously receives messages, it will not allow other tasks to run. In production quality software, a limit on the number of messages processed before suspension would be well advised. It could be up to the task to do this or our package could be enhanced to keep track of the number of messages received since the last suspension and suspend the thread even if the message queue is not empty.

Relinquishing the CPU

There may be tasks that run independently without sending or receiving messages. Being good citizens however, they may wish to relinquish the CPU to other tasks. The relinquish command lets them do that.

```
proc task::relinquish {} {
    variable tasks
    dict set tasks [myself] State RELINQUISHED
    suspend
}
```

We will see the purpose of the state RELINQUISHED later.

Suspending a task is nothing more than a simple yield.

```
proc task::suspend {} {
    yield
}
```

The task dispatcher

We finally come to the heart of our package, the dispatcher. The role of the dispatcher is to keep looping through the list of tasks looking for any that have messages pending in their receive queue and invoking them. There are several finer points that need discussion but first we show the code.

Our dispatcher is itself is a coroutine whose innards are implemented by the dispatch_loop procedure. This will loop continuously initializing some state variables and then running an inner loop that iterates over the **current** tasks invoking those that are ready to run. There is a subtle point here in that tasks may be added or removed by any of the current tasks. We will not schedule any new tasks to run until the current tasks have been invoked first. This is a matter of policy; you may choose otherwise.

A task is considered ready to run if

- it is a new task indicated by the INIT state.
- it has voluntarily given up the CPU and is awaiting its next turn as indicated by the RELINQUISHED state.
- it is waiting for messages as indicated by its RECEIVE state **and** there are messages pending in its queue.

```
proc task::dispatch_loop {} {
    variable tasks
    variable dispatcher_alarm
    variable tasks_added

    while {1} {
        set woken 0
        set tasks_added 0
        foreach task_id [dict keys $tasks] {
            if {! [dict exists $tasks $task_id]} continue
            dict with tasks $task_id {} ❶
            if {($State in {INIT RELINQUISHED}) ||
                ($State eq "RECEIVE" && [llength $Messages])} {
                incr woken
                dict set tasks $task_id State RUNNING
                if {[catch { $task_id } mesg]} {
                    puts stderr "$task_id ended with error $mesg"
                    cleanup $task_id
                }
            }
        }
        if {$woken == 0 && $tasks_added == 0} {
            coroutine::util vwait [namespace current]::dispatcher_alarm
        }
    }
}
```

❶   Copy dictionary keys to local variables

We need to make an important point about the task's state — why we mark a task as INIT, RELINQUISHED, RECEIVE or RUNNING. This gets back to our earlier discussion of the push versus pull model. A task in the RUNNING state may actually have yielded outside of our framework through a call such as coroutine::util gets. We must not invoke such tasks from our dispatcher for the same reasons discussed there. Maintaining state information allows us to eliminate these RUNNING tasks from consideration. The RECEIVE and RELINQUISHED states are distinguished by the former indicating the task should only be awoken **only** if it has messages waiting while the latter has no such requirement.

Invocation of a task is done by transferring control to its hosting coroutine. In case of any exceptions, we delete the task from the task database.

> Another option for transferring control to the task coroutine would have been to use yieldto instead of the direct call. You might find this used in some implementations. We prefer to call the task coroutine directly because our dispatcher would then regain control when the coroutine does a yield. This is important for some edge cases where the task yields but not using one of our package commands. For example, a task that sits in a loop using coroutine::util gets or the like would relinquish control but not to the dispatcher.

The other detail we need to take care of is with regards to the actions to take when none of the tasks are in a ready condition. Clearly we do not want the dispatcher to just spin endlessly without permitting other code to run. We would like it to suspend itself until there is a reason for it to run. We have a couple of options here. We can pass control to Tcl's event loop after scheduling ourselves to run after some short period of time. Instead of this polling mechanism, we choose the other option: we wait via the coroutine::util vwait call for a variable, dispatcher_alarm to be set. Correspondingly, the wakeup_dispatcher procedure, called on a send to any task, sets this variable. If the dispatcher is already active, it has no effect; otherwise, it releases the dispatcher from its suspended state.

Next we have to consider when the dispatcher should suspend itself via the vwait. Two conditions should be met for this:

- No tasks should be ready to run. We track this with the woken variable which is reset at the top of every dispatcher iteration and incremented every time a task is run. If this is 0 at the end of a complete loop through the task list, we know no tasks are ready.

- No new tasks must have been created since we initiated the iteration through the tasks. Again we use a variable, tasks_added, to keep track of this. The variable is reset at the top of the iteration and set when a new task is created within the task procedure.

The dispatcher suspends itself when both the above conditions are satisfied.

Starting the dispatcher

We have defined our dispatcher loop procedure but need to provide a means for the application to create it. So we do that now.

```
proc task::start_dispatcher {} {
    coroutine dispatcher dispatch_loop
}
```

Our package is now ready for use.

```
package provide task 1.0
```

We can try out our little package with some simple tasks. The first will simply print any message that is sent to it.

```
set logger [task::task {
    while 1 { puts "Received message: [task::recv]" }
}]
```

The second will send messages to the logger at regular intervals.

```
proc delay100 {} {coroutine::util after 100}
proc now {} {clock format [clock seconds] -format %T}
task::task "while 1 {task::send $logger \[now\]; delay100}"
```

Our final task is a variation of our previous example that printed directory sizes. This version sends the output to the logger task instead of printing it directly and is also considerate to relinquish the CPU.

```
proc dir_size_task {logger dir} {
    set total 0
    foreach fn [glob -nocomplain [file join $dir *]] {
        if {![catch {file size $fn} size]} {
            incr total $size
        }
        if {[file isdirectory $fn]} {
            if {![catch {dir_size_task $logger $fn} size]} {
                incr total $size
            }
        }
    }
    task::send $logger "$dir: $total"
    task::relinquish
    return $total
}
task::task "dir_size_task $logger c:/windows/system32"
```

We must not forget to start our task dispactcher.

```
task::start_dispatcher
```

Running the above commands in the wish shell or tclsh **with the event loop active** will result in the console printing the timestamps intermixed with the directory size data.

> In our example, we started the dispatcher after creating the task. This is of course not mandated. Tasks can be created after the dispatcher is running as well.

21.11. Chapter summary

In this chapter, we covered one of the mechanisms for doing concurrent computation in Tcl — coroutines, a lightweight and flexible mechanism adaptable to a wide variety of situations, some of which we also described. In the next chapter, we will look at an operating system facility for concurrency — threads, which provides access to parallel computation on multiple processors.

21.12. References

PIKE2013

Concurrency Is Not Parallelism, Rob Pike, https://vimeo.com/49718712

SOF2008

TIP #328: Coroutines, Sofer, Madden, http://tip.tcl.tk/328. The Tcl Implementation Proposal describing coroutines.

KBK2012

TIP #396: Symmetric Coroutines, Multiple Args, and yieldto, Kevin Kenny, http://tip.tcl.tk/396. The Tcl Implementation Proposal describing enhancements to coroutines.

22

Threads

In the previous chapter, we described the use of coroutines for concurrent computation. We now describe a more heavyweight, but also more powerful, alternative that uses operating system threads. We will contrast the two later in Section 22.15.

By default, Tcl applications run within a single operating system thread. Generally speaking, threads are used in software for one of several reasons:

- In languages with no or limited support for event-driven I/O, threads are used for performing I/O without blocking other computational tasks.
- Since each operating system thread runs on its own stack, threads maintain their own implicit context and state. This allows simpler coding of some programs where multiple tasks have to run concurrently and share results.
- Some application programming interfaces, database drivers for instance, may only provide a blocking mode of operation. In this case as well, threads may be used for these operations while allowing other computational tasks to proceed.
- Threads are also used for performance reasons in compute-intensive applications as each thread may run on its own processor in multiprocessor configurations.

When it comes to Tcl, the first two motivations above are not very compelling. The use of threads specifically for I/O is obviated by Tcl's first-class native support for asynchronous and event-driven I/O. Similarly, as discussed in Chapter 21, Tcl's coroutines also run on their own private stacks making the use of operating system threads purely for this purpose unnecessary.

Thus the use of threads in Tcl is driven primarily by the last two factors — a desire to utilize multiple processors within a program and for non-blocking operations with programming interfaces that only support a synchronous mode.

22.1. Enabling thread support: the Thread package

Use of threads within a Tcl interpreter requires the Tcl libraries to have been built with thread support. This has been the default on Windows platforms for many years. On other platforms, thread support requires the --enable-threads build option to have been specified to the configure script at the time of building the Tcl libraries. This is the default only in very recent Tcl releases for non-Windows platforms. Even then, some Tcl applications explicitly disable thread support for one of two reasons — non-threaded builds are slightly faster, and threaded builds do not work correctly in some cases involving binary extensions that invoke the fork system call.

You can check if the Tcl interpreter has thread support enabled by either inspecting the threaded element of the tcl_platform array or with the tcl::pkgconfig call.

```
puts $tcl_platform(threaded) → 1
tcl::pkgconfig get threaded  → 1
```

In addition to the core Tcl libraries being multithread-enabled, working with threads at the script level also requires the Thread package to be loaded.

```
package require Thread → 2.8.0
```

The commands implemented by the package are placed in four namespaces shown in Table 22.1 based on their functionality.

Table 22.1. Thread package namespaces

Namespace	Description
thread	Base commands for working with threads
tpool	Implements a pool of worker threads
tsv	Commands for sharing data between threads
ttrace	Enables sharing of interpreter state across threads

In addition to the above, other third-party packages are available that supplement or enhance the functionality of the Thread package. We however do not describe them in this book.

22.2. Threading model

Before going into the specifics of individual commands, a few words regarding the threading model used by Tcl are in order.

> Unless otherwise noted, the term "thread" will henceforth refer to any thread running a Tcl interpreter. A process may contain other threads as well, even those created in the background by the Tcl libraries for specific tasks like socket I/O on Windows. These are not germane to our discussion as they are not visible at the script level.

When a Tcl application starts up, there is only one thread executing Tcl code. We refer to this as the *main thread* and it has a special role as we will see in a bit. This thread executes Tcl code in a Tcl interpreter which may create threads each of which runs a Tcl interpreter. These may in turn create additional threads ad infinitum.

Threads and interpreters

We discussed the use of multiple Tcl interpreters at length in Chapter 20. The difference here is that the interpreter in a new thread is **not** a slave of the interpreter that created the thread. Each thread runs an independent "top-level" interpreter. Naturally, these interpreters may in turn create slave interpreters running in the same thread. As always, each interpreter, whether a top-level interpreter in a thread or a slave to one, is responsible for its own initialization such as loading any packages it needs, creating the proper namespaces and so on.

The Tcl threading model thus allows for multiple threads where each thread is (potentially) running multiple interpreters. **However each interpreter runs within the thread that created it.** There is no sharing of interpreter contexts between threads. Normal Tcl code therefore does not need to worry about having to synchronize access to its variables and data structures.

Thread run modes

A Tcl thread generally runs in one of two modes. It may execute a script and terminate, or it may sit in the event loop waiting for commands from other threads and executing them until it is asked to exit. Threads may be started and exit at any time with one important condition: **when the main thread terminates, the entire process exits.** It is therefore important for the main thread to monitor the status of the additional threads before it exits.

Thread reference counting

Some threads may act as worker threads wherein they run scripts sent to them by other threads. These threads have no defined exit point and have no way of knowing when their services are no longer required. Moreover, their "client" threads do not necessarily know who else might be requiring the services of that worker thread

either. To help with this situation, a reference count is maintained for each thread. The worker thread can choose to exit when this reference count reaches 0. Conversely, threads that have an interest in that thread being available can increment its reference count and only decrement it when the worker thread is no longer of interest.

Thread interaction

The design center for threads in Tcl is one where thread interaction is not expected to be fine grained. They are expected to run independently for the most part with occasional interaction and exchange of data. Thus unlike many programming languages, Tcl does not permit **normal** variables to be shared across threads (and in fact even across interpreters within a single thread). Threads interact by sending each other messages in the form of scripts which are executed by a target thread with the result being optionally returned to the source thread. This model simplifies multithread programming as the potential for race conditions, deadlocks and the like is greatly reduced.

One point to keep in mind is that the inter-thread communication "messages" are always received by the top-level interpreter of each thread. It can then be programmed to dispatch to any slave interpreters if so desired.

Shared properties

While threads run independently, there are some properties that shared. This is actually not specific to Tcl but is true for processes in general on most modern operating systems.

- The current working directory, as returned by the pwd command, is common to all threads. Changing it with the cd command in any thread changes it for the entire process.
- The environment variables, available in the env global array of the interpreter, is also shared amongst all threads. Changing a value in one thread is reflected across all threads in the process.

22.3. Creating threads: `thread::create`

Having summarized the threading model used by Tcl, we are now ready to describe the actual nitty-gritty of thread commands. Threads are created with the `thread::create` command.

```
thread::create ?-joinable? ?-preserved? ?SCRIPT?
```

The `-joinable` and `-preserved` options are related to managing thread lifetimes and are described in later sections. If the optional *SCRIPT* argument is not specified, the created thread will wait in a loop for messages to arrive. Otherwise, it will execute *SCRIPT* in its top-level interpreter and terminate when the script completes. Of course, as we will see *SCRIPT* itself may contain commands that activate the message communication loop. Note that the result of the script execution is not made available unless the script itself takes some action to return it as a message.

The command returns a thread id for the created thread that can be used for various purposes such as sending messages and synchronization.

Here is a short example that creates a thread to retrieve a URL in the background and save it to a file.

```
thread::create {
    package require http
    package require fileutil
    set tok [http::geturl http://www.example.com]
    fileutil::writeFile example.html [http::data $tok]
    http::cleanup $tok
}
→ tid000000000000014B4
```

The script loads any packages that it needs as it starts in a new interpreter in the created thread. It uses the blocking form of the `http::geturl` command but because it is running in a separate thread, our current thread can proceed with other computations. Once the script completes, the created thread will terminate.

Our simple example is missing many pieces. Since the threads run independently and asynchronously, the main thread has no way to know when the created thread has completed its task. There is also no mechanism for reporting errors and exceptions. We will come to these topics in a little bit.

22.4. Interthread communication

Threads communicate with each other through messages that are actually scripts executed in the receiving thread.

22.4.1. Waiting for messages: `thread::wait`

For a thread to receive a message, it must be running its event loop. A thread that is created without the *SCRIPT* argument specified to the `thread::create` command runs its event loop by default and is ready to receive messages. Alternatively, the script passed to the thread may use the `thread::wait` command to enter its event loop and process messages. The following commands are thus equivalent.

```
thread::create
thread::create {
    thread::wait
}
```

The latter form is generally used when there is some initialization required before messages are received. The pattern looks like

```
thread::create {
    ...load packages...
    ...other initialization...
    thread::wait
    ...cleanup...
}
```

The `thread::wait` command is similar to the `vwait` command in that it enters the thread's event loop dispatching events. The difference is that while `vwait` stays in the loop until a specified variable is set, `thread::wait` stays in the loop until the thread's reference count drops to 0 signalling it is time for the thread to exit. We discuss this in further detail in Section 22.2.

> Do **not** use `vwait` in a thread in lieu of `thread::wait` for the purposes of entering the event loop at the top level. The former is ignorant with respect to thread reference counts and therefore does not return when the count drops to 0. The thread will then not exit as expected.

Once the thread's event loop is running, events are handled just as we described in Chapter 15. Messages sent from other threads are just a special form of event. The associated event handler is the message itself which is a script to be executed in the context of the receiving thread's top-level interpreter.

22.4.1.1. Limiting message queue size: `thread::configure -eventmark`

The number of outstanding messages permitted on a thread's receive queue can be controlled through the configuration option `-eventmark` set with the `thread::configure` command.

```
thread::configure TID -eventmark ?QLIMIT?
```

If the *QLIMIT* argument is not specified, the command returns the current value of the option. If *QLIMIT* is 0, no limit is placed on the number of messages in the queue. Otherwise, it specifies the maximum number of messages allowed in the queue. When this limit is reached, even the asynchronous form of the `thread::send` command will block until the receiving thread processes enough messages for the number of queued messages to drop below the limit.

22.4.2. Sending messages: `thread::send`

On the sender's side, the `thread::send` command is used to send messages to a target thread.

```
thread::send ?-async? ?-head? TID SCRIPT ?VARNAME?
```

Here `TID` is the thread id of the target thread as returned by the `thread::create` command. The command appends, or prepends if the -head option is specified, the supplied `SCRIPT` argument to the target's event queue.

The command may operate either synchronously or asynchronously. If the -async option is not present, the command works in synchronous mode and waits for `SCRIPT` to be executed. If the `VARNAME` argument is not provided, the command returns the result of evaluation of `SCRIPT` in the target thread. If `VARNAME` is provided, the result of the command is the the **return code** from the script evaluation. For example, the command will return 0 on a successful evaluation of the script, 1 if an exception was raised and so on. The **result** of the script is stored in the variable `VARNAME`.

If the -async option is specified, the command returns immediately (with one exception described below) with an empty result. If the `VARNAME` argument is provided, the result will be placed in the variable of that name when the script completes. The sending thread can track completion of the script by either placing a trace on the variable or waiting on it with vwait. In case of errors or other exceptions this variable is still updated with the result, for example the error message. However, as the return code is not available, there is no way to distinguish this from a normal completion without somehow encoding this information in the result value.

Even with the -async option specified, there is one situation where the `thread::send` command may block. If there is an upper limit on the message queue size for the receiving thread as described in Section 22.4.1.1, the `thread::send` will block until the queue shrinks sufficiently for the new message to be placed on it.

Handling of erors during evaluation of the script is discussed in detail in Section 22.7.

> ⚠ You need to be careful to avoid deadlocks when using the synchronous form of `thread::send`. Thread A may send a script to Thread B. If as part of the evaluation of the script in Thread B, it executes a send back to Thread A, deadlock will result. Thread A cannot respond to this send because it is still blocked awaiting the result of the first send. That never completes because Thread B is still waiting for the result of the second send. This situation is not specific to Tcl or the `Thread` package. The same will apply to all synchronous communication mechanisms in any language or platform.

22.4.3. Broadcasting messages: `thread::broadcast`

Whereas the `thread::send` command sends a message to a specific thread, the `thread::broadcast` message sends a message to all existing threads that have been created with the `thread::create` command.

```
thread::broadcast SCRIPT
```

The command always works asynchronously and has no means of returning the response from each thread unless explicitly done via a `thread::send` from within the supplied script. Notice that the sending thread is not included amongst the recipients.

```
% thread::create
→ tid00000000000004F0
% thread::create
→ tid0000000000000660
% thread::broadcast [format {
    thread::send -async %s "puts {Ping from [thread::id]}"
} [thread::id]]
→ Ping from tid00000000000004F0
  Ping from tid0000000000000660
```

As an aside, the above example illustrates use of format to create the code fragment to be sent to the remote threads. This makes it a little less awkward to construct the script where part of the substitution happens in the current thread and part in the receiving thread. See Section 10.7.1.2.

22.5. Thread lifetime management

The issue of tracking and managing thread lifetimes comes up in a couple of scenarios:

- It is sometimes important to know when a thread has completed execution. For example, the main thread may fire off several threads to execute various tasks. It must make sure it does not exit before those threads as completion of the main thread causes the entire process to exit.

- Conversely, a thread may need to know when it itself may exit. For example, a server thread may service requests from multiple independent client threads. It needs some way of knowing when its services are no longer required so it can exit and not take up unnecessary resources. Moreover, the client threads may not be even aware of each other so any solution must not presume any coordination between them.

Although both these issues can be solved using explicit low-level synchronization primitives we discuss later, the scenarios are common enough that the package provides simpler built-in solutions for both.

22.5.1. Waiting for thread completion: thread::join

The thread::join command can be used to wait for the completion of a thread.

```
thread::join TID
```

Here *TID* must be the thread id of a *joinable* thread, one that is created using the -join option of the thread::create command. An attempt to join a thread that is not joinable will generate an error.

The command blocks and returns only when the specified thread has exited. The result of the command is the exit code for the thread. No events are processed in the thread calling thread::join until the command returns. At that point the thread being waited on has exited and the thread id *TID* no longer valid.

Any thread, not necessarily the parent thread, may use thread::join to wait for the completion of another thread. However, only one thread may issue a thread::join for a particular thread. Further attempts to wait on that same thread will raise an error.

Every joinable thread must be waited on by some thread::join command. Otherwise, when it completes it will stay around as a "zombie" taking up system resources.

To wait for the completion of multiple threads, you need to serially wait on each in turn. To expand a little on an earlier example, we can start two threads to download URL's and wait for their completion as below.

```
set fetch_script {
    package require http
    package require fileutil
    set tok [http::geturl http://www.example.com/%1$s]
    fileutil::writeFile %1$s.html [http::data $tok]
    http::cleanup $tok
}
set tid1 [thread::create -joinable [format $fetch_script page1]]
set tid2 [thread::create -joinable [format $fetch_script page2]]
thread::join $tid1
thread::join $tid2
puts "[file exists page1.html] [file exists page2.html]"
→ 1 1
```

An alternative to using `thread::join` to ensure a thread has exited is to use the `thread::names` command discussed in Section 22.9. This returns a list of threads created with `thread::create` that are still running. The command can be invoked periodically and the returned list checked for the thread(s) of interest. This method may be more suitable when you do not want the calling thread to block until the target thread exits.

22.5.2. Thread reference counting: `thread::preserve`, `thread::release`

We now move on to the next topic related to thread lifetimes — how a thread knows when to exit its event loop and terminate. More specifically, how does the `thread::wait` command which sits in the event loop processing messages know that it should stop looking for more events and return to the caller.

The answer lies in an internal reference count that is maintained for each thread. When the value of this reference drops to 0 or a negative number, the `thread::wait` loop is terminated regardless of any additional messages that may be pending in the queue. Note that the thread itself is not terminated. What happens next will depend on the commands following the `thread::wait` call. In the normal case, the thread should do clean up and exit.

The thread reference count can be manipulated with the `thread::preserve` and `thread::release` commands which increment and decrement the reference count respectively.

```
thread::preserve ?TID?
thread::release ?TID?
```

The commands operate on the reference count of the thread identified by `TID` which defaults to the current thread if unspecified. Both commands return the reference count value after it has been modified.

> ⚠ The reference documentation advises that the thread should not be referenced if the return value from `thread::release` is 0 or negative. This is misleading. You should not reference the released thread after a `thread::release` from the thread doing the release irrespective of the return value. This is because other threads holding a reference to the released thread might have released it in the meantime.

The simplest demonstration of thread management is creation of a single thread which will implicitly run its `thread::wait` loop.

```
set tid [thread::create] → tid0000000000001804
```

The new thread is created with a reference count of 0 by default. We can send this thread some compute-intensive tasks.

```
thread::send $tid {expr 1+1} → 2
thread::send $tid {expr 2*2} → 4
```

Once we are done with the thread, we call `thread::release` on it. This decrements its reference count causing the `thread::wait` loop to return allowing the thread to exit.

```
thread::release $tid → 0
```

In a slightly more complex scenario, multiple threads might be using this "server" thread we created in which case the application has to ensure that the thread does not disappear until all client threads are done with it. To ensure this, each client thread should call `thread::preserve` on the server thread and match that with a `thread::release` once they are done with it. This is true of the creating thread as well if it will also make use of the server thread. Alternatively, it can use the `-preserved` option on thread creation. As we noted earlier, new threads are created with a default reference count of 0. However, if the `-preserved` option is provided to `thread::create`, the new thread is created with a reference count of 1. The creating thread must then execute a corresponding `thread::release` at the appropriate time.

22.6. Canceling script execution: `thread::cancel`

The `thread::cancel` command cancels the evaluation of a script in a thread.

```
thread::cancel ?-unwind? TID ?RESULT?
```

The command cancels the evaluation of the script running in the target thread identified by *TID* by raising an exception. If the *RESULT* argument is provided, it is used as the result or error message of the exception. Otherwise a default error message is used. If the -unwind option is specified, the exception cannot be caught by the target thread and propagates all the way to the top level of the thread. Note that the thread is **not** terminated.

The following console sample session should clarify the operation. It creates a thread and defines a procedure demo within it that prints two lines separated by a delay. The delay is simply to permit us to type a cancel command in time.

```
% set tid [thread::create]
→ tid0000000000002EA0
% thread::send $tid {proc demo {} {puts foo; after 5000; puts bar}}
```

We then ask the thread to execute the procedure and immediately issue a cancel command.

```
% thread::send -async $tid demo
→ foo

% thread::cancel $tid
% Error from thread tid0000000000002EA0
→ eval canceled
      while executing
  "after 5000"
      (procedure "demo" line 1)
      invoked from within
  ...Additional lines omitted...
```

As seen, bar is never printed because the evaluation of the script was aborted in the meantime. However, the thread is still active and can process further messages. We can try it again but this time without cancellation.

```
% thread::send -async $tid demo
→ foo
  bar
```

22.7. Error handling in threads: `thread::errorproc`

The manner in which errors and exceptions are handled in threads depends on whether the thread is executing a script passed as an argument to `thread::create` or is running scripts via the `thread::wait` loop. Furthermore, the latter case is further distinguished by whether the script execution is synchronous or not.

The simpler case is that of a thread executing a script passed through the `thread::create` command. Any errors or exceptions will result in the thread being terminated.

> The above does not apply if the script invokes the `thread::wait` command and the exception is thrown in a script executed through its event loop. The behaviour for that case discussed later in this section.

By default, Tcl writes the error stack from the exception to the standard error channel. This behaviour can be changed by calling the thread::errorproc command.

```
thread::errorproc ?ERRORPROC?
```

If provided an argument, this command registers it as a procedure to be called to report the error. Tcl will append two additional arguments to the call: the id of the thread that generated the exception, and the error stack. An example procedure would be:

```
proc thread_error_handler {tid error_stack} {
    puts -nonewline stdout "[clock format [clock seconds]]: "
    puts "Thread $tid died. Error stack:\n$error_stack"
}
```

We can then register it and check out the effects.

```
% thread::errorproc thread_error_handler
% thread::create {error "Something horrible happened"}
→ tid0000000000000C60
  Sun Feb 19 22:56:34 IST 2017: Thread tid0000000000000C60 died. Error stack:
  Something horrible happened
      while executing
  "error "Something horrible happened""
```

The registered error handler is run in the interpreter that registered it irrespective of the thread creator. Moreover, that interpreter must have an active event loop running.

If thread::errorproc is called without any arguments, it returns the currently registered error handler.

```
% thread::errorproc
→ thread_error_handler
```

To reset the error handler to the default, call thread::errorproc with an empty string as the argument.

The other case to consider is that of exceptions thrown by scripts run by the thread::wait event loop, i.e. one sent by the thread::send command. Unlike for scripts passed as arguments to the thread::create command, exceptions in scripts run within thread::wait do not cause the thread to die by default. These exceptions are reported as described below but the thread continues to run the the event loop processing messages. The thread::configure command's -unwindonerror option may be used to change this behaviour.

```
thread::configure TID -unwindonerror ?UNWIND?
```

If UNWIND is 1, exceptions during execution of scripts from thread::wait will cause the thread to exit. The default value is 0. If the UNWIND argument is not given, the command returns the current value of the option.

Reporting of exceptions in scripts run with thread::send depends on whether the -async option was specified with the command. If present, exceptions are reported as described earlier for scripts passed to thread::create.

If the -async handler was not specified, the thread::send command executes synchronously. If the VARNAME argument is provided, the command behaves similar to the catch command. The result of the command is the return code and the script result is placed in the variable VARNAME.

```
% set tid [thread::create]
→ tid0000000000000FF4
% thread::send $tid {expr 1+1} result ❶
→ 0
% set result
```

```
→ 2
% thread::send $tid {error "An error!"} result ❷
→ 1
% set result
→ An error!
```

❶   Normal completion
❷   Error exception

If *VARNAME* was not provided, then a synchronous send results in exceptions being directly reflected as for any other command.

```
% thread::send $tid {error "An error!"}
Ø An error!
% set errorInfo
→ An error!
      while executing
   "error "An error!""
      invoked from within
   "thread::send $tid {error "An error!"}"
```

22.8. Threads and I/O channels: thread::transfer

As we described in Chapter 20, I/O channels are specific to an interpreter and not shared. The same then naturally applies to interpreters running in different threads. With the exception of standard I/O channels, a channel created in one thread is not available in other threads and cannot be shared. It can however be transferred from one thread to another with the thread::transfer command.

```
thread::transfer TID CHAN
```

The channel *CHAN* is made accessible to the **main** interpreter of the thread identified by *TID* and no longer available to the current interpreter. The channel name remains the same in the thread to which it is transferred. However, this name has to be explicitly provided to the target thread as otherwise it has no means to know how to reference the channel. The following trivial example illustrates the sequence.

We open a channel to a temporary file in our current interpreter and write to it.

```
set chan [file tempfile tempname]
puts $chan "File created by [thread::id]"
chan names
→ file413a920 stdin stdout rc28 stderr
```

Notice that chan names lists the channel amongst the open channels in this interpreter.

We then create a new thread and transfer the channel to it. We see that chan names no longer shows the channel as being available in the current interpreter. It does show up in the list of channels in the new thread.

```
set tid [thread::create]
thread::transfer $tid $chan
chan names
thread::send $tid {chan names}
→ file413a920 stderr stdout stdin
```

Although the channel is now available in the new thread, its name is not known to the interpreter in the thread. We have to inform it through some means. In our example, we simply set the global chan in the new thread to the channel name.

```
thread::send $tid [list set chan $chan]
→ file413a920
```

Finally we have the new thread write to the channel and close it.

```
thread::send $tid {
    puts $chan "Message from [thread::id]"
    close $chan
}
thread::release $tid
→ 0
```

As confirmation, we can read back the temporary file.

```
print_file $tempname
→ File created by tid0000000000002FE8
  Message from tid000000000000081C
```

There is one issue you have to be wary of when transferring channels. The `thread::transfer` command will block until the target thread has internally completed acceptance of the channel. This leads to the potential for deadlocks as illustrated by the following fragment.

```
thread::send $tid { set chan [file tempfile] }
thread::send $tid [list thread::transfer [thread::id] \$chan]
```

The thread executing the above code has the target thread open a temporary file. It then **synchronously** asks it to transfer the open channel back. However, because the `thread::send` command is used in synchronous mode, the first thread is blocked and cannot complete acceptance of the transfer. This results in the second thread being blocked as well since the `thread::transfer` command will not return until the channel transfer is complete. Deadlock results.

The general lesson in the above is to always be careful when using synchronous calls between threads. Again, this is nothing specific to Tcl or its Thread package.

An alternative to `thread::transfer` for moving channels between threads is the pair of commands `thread::detach` and `thread::attach`. The former removes the channel from the calling interpreter. The channel then stays in an "unowned" state until an interpreter (generally, not necessarily) in another thread calls `thread::attach` to claim ownership of the channel. This splitting of `thread::transfer` functionality is useful when the thread to which the channel should be transferred is not known a priori. This situation is common when using a "worker thread" model of the kind we will describe in Section 22.12.

Note that the same care for avoiding deadlocks must be taken with `thread::detach` / `thread::attach` as was described above for `thread::transfer`.

Transferring socket channels

There is one minor quirk to keep in mind regarding transfer of server sockets. As described in Chapter 18, a server socket is created through an accept callback on a listening socket. One common design in multithreaded servers is to hand off the socket to a separate thread for the actual data transfer[1]. For internal implementation reasons however, a server socket cannot be transferred to another thread from within the accept callback. In other words, the following accept callback will not work.

---

[1] This design is far less common in Tcl applications thanks to its strong support for async event-driven I/O

```
proc accept {chan remote_addr port} {
    set tid [thread::create]
    thread::transfer $tid $chan
}
```

The workaround is to reschedule the transfer of the channel through the event loop as shown below.

```
proc accept {chan remote_addr port} {
    after 0 [list transfer_socket $chan $remote_addr $port]
}
proc transfer_socket {chan remote_addr port} {
    set tid [thread::create]
    thread::transfer $tid $chan
}
```

This quirk originates from additional references held internally on the channel during the processing of the accept callback. These references prevent the channel from being transferred. Once the callback returns, these references are released allowing the transfer to take place in the rescheduled procedure.

22.9. Introspecting threads: thread::id, names, exists

All commands in the Thread package for introspecting threads only deal with threads created with the thread::create command. Threads that have been created by other means, such as at C level, are not considered by these commands.

The thread::id command returns the id of the current thread.

```
thread::id → tid0000000000002FE8
```

The thread::names command returns the list of currently executing threads.

```
% thread::names
→ tid0000000000000FF4 tid0000000000002FE8
```

You can check for the existence of a thread with the thread::exists command.

```
set tid [thread::create -joinable] → tid00000000000020E8
thread::exists $tid                → 1
thread::release $tid               → 0
thread::join $tid                  → 0
thread::exists $tid                → 0
```

22.10. Thread-shared variables

Variables in a Tcl interpreter are always contained within that interpreter. They are not directly accessible from other threads or even other interpreters within the same thread. Data is shared between threads by passing **values** through messages. There are times however where it is beneficial to have a shared **location** where data is stored and accessed from multiple threads:

- Situations where the same exact values for data items must be seen from multiple threads is greatly simplified if the data is stored in a single location and not passed around.
- When large amounts of data is being shared, passing it around by value entails memory copies with a significant cost in performance.

For these situations, the Thread package provides the *thread-shared variables*, or simply shared variables, and a set of commands for manipulating them under the tsv namespace.

Thread-shared variables have the following characteristics:

- They are not "real" Tcl variables in that they cannot be referenced using $ or the set command, they are not traceable and so on. They can only be manipulated through the tsv commands implemented by the Thread package.
- Shared variables exist outside of any interpreter or thread in that they continue to exist irrespective of the creating thread or any other thread terminating. When no longer required they have to be explicitly unset so as to free up memory resources.
- A shared variable is not a scalar but a collection of values indexed by a key similar to Tcl arrays and dictionaries.
- Access to these variables is always protected under a lock so threads do not need to explicitly synchronize when manipulating these variables. Note however that each variable has an independent lock so if a transaction involves multiple shared variables, additional locking may be required.

Commands for manipulating shared variables can be categorized as follows:

- Scalar operations such as setting, incrementing etc. on an individual element of a shared variable
- List operations similar to lappend, lsearch etc. on an individual element of a shared variable
- Array commands, similar to the standard Tcl array command, that operate on the entire shared variable, not just individual elements.
- Commands that treat the shared variable as a set of *keyed lists*, similar in utility to Tcl dictionaries
- Miscellaneous commands for introspection and utility functions

22.10.1. Scalar operations on shared variables

The commands tsv::set, tsv::unset, tsv::incr, tsv::append and tsv::exists work very similar to their standard Tcl counterparts set, unset, incr, append and info exists except that they operate on individual elements of a thread-shared variable rather than a Tcl variable.

```
% set tid [thread::create]
→ tid0000000000001A18
% tsv::exists mytsv myelem ❶
→ 0
% tsv::set mytsv myelem "A value" ❷
→ A value
% tsv::exists mytsv myelem
→ 1
% thread::send $tid {
    tsv::append mytsv myelem " in" " shared storage" ❸
}
→ A value in shared storage
% tsv::set mytsv myelem ❹
→ A value in shared storage
% tsv::set mytsv myinteger 0
→ 0
% tsv::incr mytsv myinteger 5 ❺
→ 5
```

❶ Check existence of myelem in the mytsv shared variable
❷ Set the value of the myelem element in the shared variable mytsv
❸ Append a string to the element from another thread
❹ Retrieve its value
❺ Increment the value of the myinteger element of the shared variable mytsv

There are additional commands that have no direct counterparts in the core Tcl command set. These are required because of the potential for race conditions between threads, something which is not an issue for normal variables. For example, consider the following code fragment:

```
if {[tsv::exists mytsv myelem]} {
    puts "Value is [tsv::set mytsv myelem]"
}
→ Value is A value in shared storage
```

The above seems to work but there is a hidden race condition. Although individual tsv commands operate under a lock protecting access to the shared variable, other threads may still change the variable between command invocations. For example, in the above code fragment, some other thread could unset the shared variable at some point between the tsv::exists check and the tsv::set invocation resulting in an exception.

The commands, tsv::get, tsv::move and tsv::pop, are provided to deal with common cases of this type where multiple base operations have to be performed atomically. The tsv::get command is meant for exactly the case described above.

```
tsv::get TSVAR ELEM ?VARNAME?
```

The command **atomically** checks for the existence of the element ELEM in the shared variable TSVAR and returns its value if it exists. If the VARNAME argument is provided, the command returns 1 if the element exists and 0 otherwise. The element value is stored in the variable VARNAME. Our previous (incorrect) code fragment can then be written as

```
if {[tsv::get mytsv myelem val]} {
    puts "Value is $val"
}
→ Value is A value in shared storage
```

If the VARNAME argument is not specified, the command returns the value of the shared variable element if it exists and raises an error otherwise.

The tsv::move command renames an element.

```
tsv::move mytsv myinteger yourinteger → (empty)
tsv::exists mytsv myinteger            → 0
tsv::set mytsv yourinteger             → 5
```

The tsv::pop command retrieves the value of an element while simultaneously removing it from the shared variable.

```
tsv::pop mytsv yourinteger    → 5
tsv::exists mytsv yourinteger → 0
```

22.10.2. List operations

The next set of commands also operate on individual elements of a shared variable but in this case the value stored in the element is treated as a list. Again, the majority of these commands parallel the standard Tcl commands of the same name for operating on lists. These are tsv::lappend, tsv::linsert, tsv::lreplace, tsv::llength, tsv::lindex, tsv::lrange, tsv::lset and tsv::lsearch. While some list modification commands operate on **variables** and others on list **values**, all tsv list modification commands operate on shared variable elements, not values (the latter would not make sense).

```
tsv::lappend mytsv mylist C D E                  → C D E ❶
thread::send $tid {
    tsv::linsert mytsv mylist 0 A B              ❷
    tsv::lreplace mytsv mylist end-1 end X [list Y Z] ❸
    tsv::lset mytsv mylist end 1 P               ❹
}                                                → A B C X {Y P}
tsv::lindex mytsv mylist 2                        → C ❺
tsv::lrange mytsv mylist 1 3                      → B C X ❻
tsv::lsearch mytsv mylist -exact Y               → -1 ❼
```

❶ Create a list
❷ Insert list elements from another thread
❸ Replace elements
❹ Set nested list element
❺ Retrieve a single list element
❻ Retrieve a range of list elements
❼ Search for exact match. Other options are `-glob` and `-regexp`

Just as in the case of scalar operations, shared variable lists also have commands for common multi-step sequences that need to be atomic. The `tsv::lpop` command is similar to `tsv::lindex` but in addition to returning the value at the specified index position, it also removes it from the shared variable element.

```
tsv::get mytsv mylist    → A B C X {Y P}
tsv::lpop mytsv mylist 2 → C
tsv::get mytsv mylist    → A B X {Y P}
tsv::lpop mytsv mylist   → A ❶
tsv::get mytsv mylist    → B X {Y P}
```

❶ If index is not specified, it defaults to 0.

The `tsv::lpush` command does the reverse operation.

```
tsv::lpush mytsv mylist J 3 → (empty)
tsv::get mytsv mylist       → B X {Y P} J
tsv::lpush mytsv mylist K    → (empty) ❶
tsv::get mytsv mylist       → K B X {Y P} J
```

❶ If index is not specified, it defaults to 0.

The `tsv::lpush` command always returns the empty string as its result.

22.10.3. Array operations

We have seen `tsv` commands that parallel the standard Tcl scalar and list operations. As you might expect, a similar set of commands exists that treat shared variables in a fashion similar to Tcl's array variables. Unlike the `tsv` commands previously discussed which operated on individual elements of a shared variable, these commands treat the shared variable itself as an array.

The commands related to `tsv` arrays are all implemented as the `tsv::array` ensemble. The ensemble subcommands `tsv::array set`, `tsv::array get`, `tsv::array names` and `tsv::array size` have the same function as their array command counterparts.

```
tsv::array set myarr {AB 1 BC 2 AC 3} → (empty) ❶
tsv::array get myarr                   → AC 3 BC 2 AB 1 ❷
tsv::array get myarr a*                → (empty) ❸
tsv::array size myarr                  → 3 ❹
```

❶   Create, set or modify elements
❷   Get all elements of an array
❸   Get all elements matching a pattern
❹   Get number of elements in the shared variable

The above commands operate on the shared variable as an array but it is still a shared variable and therefore can be accessed with the previously discussed commands. For example,

```
tsv::get myarr AB       → 1 ❶
tsv::set myarr XY 4     → 4 ❷
tsv::lappend myarr BC 5 → 2 5 ❸
tsv::array names myarr  → AC XY BC AB ❹
```

❶   Get the value of array key AB
❷   Creating a new element in the shared variable is equivalent to creating an element of the array
❸   List commands on an array element
❹   All element names. Notice it includes XY

Notice the symmetry between shared variables and Tcl arrays. A shared variable is analogous to a Tcl array where elements are indexed by a key. Just as elements of a Tcl array can be operated on by list and string commands, so can elements of a shared variable with the equivalent tsv commands.

There is one feature of thread shared variables that we do not discuss here. Shared variable arrays can be bound to persistent disk storage so that their contents are stored in a database. As of this writing the package supports the gdbm and ldbm databases. Depending on the platform, this feature may or may not be included. You can use the tsv::handlers command to list the available database backends. To use the feature, refer to tsv::bind in the Thread package documentation.

22.10.4. Keyed lists

One final basic Tcl structured data type whose analogue we have not described are dictionaries. Shared variables do not have any commands that work on dictionaries. However, they do have *keyed lists* which serve a similar purpose.

A keyed list is a list each element of which is itself a list with two elements, the first being the "key" and the second being the corresponding "value". For example, the keyed list

```
{Name {{First Sherlock} {Last Holmes}}} {Address {221B Baker Street}}
```

has two keys Name and Address. Moreover, the value associated with Name is itself a nested key list with keys First and Last.

The tsv commands related to keyed lists operate on the assumption that the element of the shared variable is a keyed list. The tsv::keylset command sets the values of elements in a keyed list.

```
tsv::keylset detectives holmes Name {{First Sherlock} {Last Holmes}}
tsv::keylset detectives holmes Address {221B Baker Street}
tsv::keylset detectives poirot Name {{First Hercule} {Last Poirot}} \
    Address {Whitehaven Mansions}
```

This creates elements `holmes` and `poirot` in the thread shared variable `detectives`.

We can retrieve elements as usual with `tsv::get`.

```
% tsv::get detectives holmes
→ {Name {{First Sherlock} {Last Holmes}}} {Address {221B Baker Street}}
```

To operate on individual fields, we use the `tsv` keyed list commands. The `tsv::keylset` command we used for creation can also be used to modify or add new keys. Field values are retrieved with `tsv::keylget`.

```
tsv::keylget TSVAR ELEM KEY ?RETVAR?
```

If the `RETVAR` argument is not specified, the command returns the value associated with `KEY` in the `ELEM` element of the shared variable `TSVAR`. Nested keys are accessed using special syntax where each key level is separated by a period.

```
% tsv::keylget detectives holmes Address
→ 221B Baker Street
% tsv::keylget detectives holmes Name
→ {First Sherlock} {Last Holmes}
% tsv::keylget detectives holmes Name.First ❶
→ Sherlock
```

❶ Nested field

The command will raise an error if the key does not exist.

When the `RETVAR` argument is provided, the command returns 1 if the key exists in the keyed list and 0 otherwise. In the former case the corresponding value is placed in the variable `RETVAR`.

```
tsv::keylget detectives mason Address addr   → 0
tsv::keylget detectives poirot Address addr  → 1
set addr                                     → Whitehaven Mansions
```

Deleting a key is accomplished with the `tsv::keyldel` command.

```
tsv::keyldel detectives poirot Address → (empty)
tsv::get detectives poirot             → {Name {{First Hercule} {Last Poirot}}}
```

The `tsv::keylkeys` command retrieves the list of keys.

```
tsv::keylkeys detectives holmes      → Name Address
tsv::keylkeys detectives holmes Name → First Last ❶
```

❶ Retrieve nested keys

22.10.5. Introspection and utility commands: `thread::names, object, lock`

The list of thread shared variables can be enumerated with the `tsv::names` command.

```
tsv::names       → myarr detectives mytsv ❶
tsv::names det* → detectives ❷
```

❶ All shared variables
❷ Shared variables matching a pattern

All shared variables should be deleted when no longer needed. The `tsv::names` command can be useful for cleaning up at an appropriate time.

```
foreach tsv [tsv::names my*] {
    tsv::unset $tsv
}
```

The `tsv::object` command is a convenience for working with a specific element in a shared variable. It creates an object command bound to a shared variable element with its methods redirected to the standard `tsv` commands. For example,

```
% set sherlock [tsv::object detectives holmes]
→ ::0000000004573C78
% $sherlock keylset Sidekick Watson
% $sherlock get
→ {Name {{First Sherlock} {Last Holmes}}} {Address {221B Baker Street}} {Sidekick Watson}
```

The created command is automatically deleted when the associated element is unset.

The `tsv::lock` command evaluates multiple commands while holding the internal lock associated with a shared variable.

```
tsv::lock TSVAR ARG ?ARG ...?
```

The command obtains the internal lock to the shared variable *TSVAR* creating the variable if necessary. It then evaluates the script formed through the concatenation of the *ARG* arguments and then releases the internal lock.

The `tsv::pop` command described earlier could be implemented as the following procedure.

```
proc pop {tsv elem} {
    tsv::lock $tsv {
        set val [tsv::get $tsv $elem]
        tsv::unset $tsv $elem
    }
    return $val
}
```

The lock ensures retrieval of value followed by unsetting of the element as an atomic operation.

22.11. Synchronization and locking

Threading models in many languages entail sharing of data between threads making the use of synchronization primitives commonplace. On the other hand, threads in Tcl are generally written as message-passing constructs with limited need for these primitives. Nevertheless, there are situations where synchronized access is required for commonly shared resources which may be internal, such as the shared variables we described in Section 22.10, or external, such as a set of files. The Thread package therefore provides mutexes and condition variables for such synchronization purposes.

> This section assumes you are familiar with locking and synchronization in multithreaded programs. We will not explain what mutexes and condition variables are and how they might be used. Our discussion is restricted to a description of the synchronization facilities commands available in Tcl.

22.11.1. Mutexes: `thread::mutex`, `thread::rwmutex`

The `Thread` package implements three types of mutexes that can be used to ensure mutual exclusion when accessing shared resources.

- An exclusive mutex can be locked at most once. A second attempt to lock it will block the thread attempting the lock, even if it is the same thread that is holding the lock. The second thread will be unblocked when the thread holding the lock releases it. Note that a second attempt to lock by the thread already holding the lock will result in a deadlock. Operations on exclusive mutexes are implemented by the `thread::mutex` command ensemble.

- A recursive mutex only allows at most one thread to lock it but that thread can recursively lock it any number of times. Attempts by another thread to lock the mutex will result in that thread being blocked until the first thread releases **all** the locks it is holding on the mutex. Operations on recursive mutexes are also implemented by the `thread::mutex` command ensemble.

- A reader-writer mutex differs in that it supports two modes of locking — reader and writer. Multiple threads may simultaneously hold reader locks on the mutex. However at most one thread may hold a writer lock. Morever, a reader lock cannot be obtained while a writer lock is held and vice versa. Threads holding reader locks are allowed by contract to read the shared resource being protected but operations that modify the resource require a writer lock to be obtained. Operations on read-write mutexes are implemented by the `thread::rwmutex` command ensemble.

The `create` subcommand of both ensembles creates a mutex of the appropriate type. A recursive mutex requires the `-recursive` option to be specified to the `thread::mutex` command.

```
set exclusive_mutex [thread::mutex create]             → mid0
set recursive_mutex [thread::mutex create -recursive] → rid1
set rw_mutex [thread::rwmutex create]                  → wid2
```

Exclusive and recursive mutexes are locked with the `thread::mutex lock` command.

```
thread::mutex lock $exclusive_mutex → (empty)
thread::mutex lock $recursive_mutex → (empty)
```

The locks are released with `thread::mutex unlock`.

```
thread::mutex unlock $exclusive_mutex → (empty)
thread::mutex unlock $recursive_mutex → (empty)
```

However for reader-writer mutexes, two separate commands, `thread::rwmutex rlock` and `thread::rwmutex wlock` are required corresponding to reader and writer locks.

```
thread::rwmutex rlock $rw_mutex   → (empty) ❶
thread::rwmutex unlock $rw_mutex → (empty)
thread::rwmutex wlock $rw_mutex   → (empty) ❷
thread::rwmutex unlock $rw_mutex → (empty)
```

❶   Obtain a reader lock
❷   Obtain a writer lock

All mutexes should be destroyed when no longer required with `thread::mutex destroy` or `thread::rwmutex destroy` depending on the type of mutex.

```
thread::mutex destroy $exclusive_mutex → (empty)
thread::mutex destroy $recursive_mutex → (empty)
thread::rwmutex destroy $rw_mutex       → (empty)
```

> **!** You must ensure there are no outstanding locks on a mutex before destroying it. Otherwise, undefined behaviour including process crashes may result depending on the platform.

Here is a trivial example of the use of mutexes. As we stated earlier, standard channels are available in all threads. If we want to ensure output from multiple threads is not intermixed, we can use a mutex for the purpose.

```
% set io_mutex [thread::mutex create]
→ mid3
```

Then threads have to **by convention** lock the mutex before doing I/O.

```
% thread::mutex lock $io_mutex
% puts "Line one"
→ Line one
% puts "Line two"
→ Line two
% thread::mutex unlock $io_mutex
```

There is one issue with the above pattern when more complex code is involved. If an exception is raised in the script, the mutex will stay locked. Safer practice is to enclose the block within a `try` or `catch` command which will release the lock even in error cases. This situation is common enough that the `Thread` package provides the `thread::eval` command as a convenience.

```
thread::eval ?-lock MUTEX? ARG ?ARG ...?
```

The command locks the specified mutex, or an internal global mutex if the `-lock` option is not provided. The command then executes the script formed from concatenating the ARG arguments and returns its result taking care to unlock the mutex even under exceptional returns.

Our example above could then be written with better error handling as

```
thread::eval -lock $io_mutex {
    puts "Line one"
    puts "Line two"
}
```

For an equally superficial illustration of reader-writer mutexes, assume we are keeping bank account information in shared variables `current` and `savings` whose elements are keyed by an account number. To ensure consistent data, we have to ensure we do not read in the middle of a transaction that modifies the data. However, there is no reason to disallow multiple readers so we can use a reader-writer lock. Our desire for a reader-writer lock precludes us from using `thread::eval`.

```
proc bank::init {} {
    variable acct_mutex [thread::rwmutex create]
    ...create and initialize account data...
}

proc transfer {from_type from_acct to_type to_acct amount} {
    variable acct_mutex
    thread::rwmutex wlock $acct_mutex
    try {
        tsv::incr $from_type $from_acct -$amount
        tsv::incr $to_type $to_acct $amount
    } finally {
        thread::rwmutex unlock $acct_mutex
```

```
    }
}

proc balance {acct_type acct} {
    variable acct_mutex
    thread::rwmutex rlock $acct_mutex
    try {
        set balance [tsv::get $acct_type $acct]
    } finally {
        thread::rwmutex unlock $acct_mutex
    }
    return $balance
}
```

22.11.2. Condition variables: thread::cond

Condition variables are used in conjunction with exclusive mutexes as a race-free mechanism for a thread to block until some predicate is true. The condition variable is used for notification while the mutex is used to ensure that the data underlying the predicate is not modified by another thread while being checked.

The general usage pattern is as follows. The initialization code

1. Creates an **exclusive** mutex
2. Creates a condition variable

Each waiting thread (there may be multiple of these) runs the following sequence of steps:

1. Lock the mutex. Since the mutex is held, the predicate can be safely checked in the next step without fear of interference from other threads.
2. Check the predicate. If the predicate evaluates to true, go to step 5. Otherwise go to the next step.
3. Block on the condition variable. Blocking on the condition variable also releases the held mutex lock allowing other threads to modify the protected predicate data.
4. When the blocking call returns, go back to Step 2. See the explanation below as to why. Note that the mutex is held when the blocking call returns.
5. Do the expected work. Note the mutex is held at this point.
6. Unlock the mutex.

Steps 3 and 4 warrant some additional explanation. The blocking call on the condition variable also internally releases the mutex. This allows another thread to modify the protected data possibly changing the predicate to evaluate to true. If so the thread will signal the condition variable (see below) thereby waking up the thread waiting on the condition variable. Before returning in Step 3, the woken thread also relocks the mutex thereby ensuring the predicate is again protected. **However between the time the thread was signalled and the time the mutex was relocked, the predicate might have changed again.** This can happen for any number of reasons. For example, in a worker thread model, multiple threads may wait on a condition variable in which all of them are woken up on the signal. The first one to succesfully grab the mutex may modify the predicate data, for example by removing a work item from a queue. When other worker threads lock the mutex, the predicate is then no longer true. For this reason, after a thread wakes up it needs to recheck the predicate as indicated by Step 4.

The sequence for the signalling threads is simpler.

1. Lock the mutex.
2. Modify the protected data.
3. If the predicate evaluates to true, signal the condition variable to wake up the waiting thread(s).
4. Unlock the mutex.

We will illustrate the use of condition variables by extending our previous examples to implement a worker thread model for fetching URLs. Our main thread will queue URL requests to a shared queue. A set of worker threads will pick up entries from the queue, fetch the URL and write the content to a file.

First we initialize the data — the URL queue, the condition variable and mutex — that will be shared between the threads. We have already described mutex creation but make a note that this must be an exclusive mutex. The condition variables is created with the `thread::cond create` command. The returned handles are stored in shared variables since they will need to be available to the worker threads as well.

```
tsv::set shared_data url {}          → (empty)
set mutex [thread::mutex create]     → mid4
tsv::set shared_data mutex $mutex    → mid4
set cond [thread::cond create]       → cid5
tsv::set shared_data cond $cond      → cid5
```

Next we define the script that each worker thread will run.

```
set worker_script {
    package require http
    package require fileutil
    proc url_to_file {url} {
        return [file join \
                    [fileutil::tempdir] \
                    [file rootname [file tail $url]]_content.html]
    }
    set mutex [tsv::get shared_data mutex]
    set cond  [tsv::get shared_data cond]
    while {1} {
        thread::mutex lock $mutex
        while {[tsv::llength shared_data urls] == 0} {
            thread::cond wait $cond $mutex
        }
        set url [tsv::lpop shared_data urls]
        thread::mutex unlock $mutex
        set tok [http::geturl $url]
        fileutil::writeFile [url_to_file $url] [http::data $tok]
        http::cleanup $tok
    }
}
```

The script initializes the thread by loading the omnipresent `http` package and retrieving the mutex and condition variable handles from shared memory. It then sits in an infinite loop waiting for work to be queued. Within each iteration of the loop, it follows the general pattern we described earlier. The predicate in this case is for the URL queue to have at least one entry in it. If the queue is empty, it waits on the condition variable with the `thread::cond wait` command.

```
thread::cond wait COND MUTEX ?TIMEOUT?
```

This command will unlock the specified mutex and then suspend execution until the condition variable *COND* is signalled or the optional timeout occurs. The timeout value *TIMEOUT* is specified in milliseconds. If unspecified or 0, the command will only return when the condition variable is signalled.

⚠ The timeout value has some caveats associated with it. It controls how long the command will internally wait for the condition to be signalled. However, even after being awoken, the `wait` command semantics call for the mutex to be re-acquired. There is no control over how long this might take if there are many threads competing for the mutex.

Before returning to the caller, the `wait` command re-locks the mutex. Our worker thread is then free to check the status of the URL queue without suffering race conditions from other threads. If the queue is empty (some other worker beat us to it), the code loops back into the conditional wait. If the queue is not empty, the thread removes one entry from it. It then unlocks the mutex to let other threads process additional entries if any, and proceeds to download the dequeued URL. The whole sequence is then repeated.

The work dispatcher is even simpler following our aforementioned pattern. We first start up a suitable number of worker threads, say 4, to run the worker thread script we showed earlier.

```
for {set i 0} {$i < 4} {incr i} {
    lappend workers [thread::create $worker_script]
}
```

We can then queue up URL's to be fetched in the background at any time with a sequence of calls similar to the following.

```
thread::mutex lock $mutex
tsv::lpush shared_data urls http://www.example.com/page.html end
thread::cond notify $cond
thread::mutex unlock $mutex
```

The only new command in this sequence is `thread::cond notify`. This command signals the specified condition variable causing all threads waiting on it to resume execution. The first thread to run and acquire the mutex will then dequeue and download the URL.

One final point about condition variables. They should be freed when no longer required by calling `thread::cond destroy`.

```
thread::cond destroy $COND
```

As with mutexes, care must be taken that the condition variable is not in use when the command is invoked.

22.12. Thread pools

In the previous section we used a minimal, incomplete and custom implementation of the commonly used worker thread model. The `Thread` package includes a generalized version of this functionality through its *thread pools*. A thread pool consists of a work queue where *jobs* can be posted and an associated set of worker threads that pull them off the queue and execute them. An application may create any number of such thread pools.

Each thread pool has a variable number of threads that can be configured to run in it with minimum and maximum limits. The pool starts out with the minimum number when it is created. If the number of active or queued jobs exceeds the current number of threads, new threads are created to handle the additional jobs until the maximum thread limit is reached. Beyond that point jobs remain queued until they are removed by some thread that has completed its previous job. If a thread finds there are no jobs remaining to be processed, it goes into an idle state. If it stays in that state for some period of time without getting a new job, it will exit provided the current number of threads is greater than the minimum limit.

All commands related to thread pools are contained in the `tpool` namespace.

22.12.1. Creating a thread pool: `tpool::create`

A thread pool is created with `tpool::create`.

```
tpool::create ?OPTIONS?
```

The command returns the id of the thread pool which can be used to post jobs to the work queue for the pool. The options shown in Table 22.2 may be specified to control various associated configuration parameters.

Table 22.2. Thread pool options

Option	Description
-exitcmd *SCRIPT*	Specifies a script to be run just before the thread exits. A thread will exit if it has been idle for some period of time.
-idletime *SECONDS*	Specifies the number of seconds that a thread has to remain idle before it exits as described above.
-initcmd *SCRIPT*	Specifies a script to be run in the worker thread when it first starts up. This can be used to load packages, initialize data and so on. If the script raises an error in a worker thread, the exception is reflected back into tpool::create or tpool::post depending on which call initiated the creation of the thread.
-maxworkers *COUNT*	The maximum number of worker threads that should be maintained in the thread pool. The default is 4,
-minworkers *COUNT*	The minimum number of worker threads that should be maintained in the thread pool. The default is 0 so that the pool starts with 0 threads until the first job is queued.

Any number of thread pools may be created. The command tpool::names returns the list of currently existing thread pools.

Let us create a thread pool version of the example in the last section with one difference. It returns the content of the URL back to the main thread instead of writing it to a file. This provides a more complete example as it shows communication in both directions and some rudimentary error handling.

First we define the initialization script each worker thread needs to run. This just loads the http package and defines a procedure to retrieve URL's.

```
set init_script {
    package require http
    proc fetch_url {url} {
        set tok [http::geturl $url]
        try {
            switch -exact -- [http::status $tok] {
                ok { return [http::data $tok] }
                eof { error "Server closed connection." }
                error { error [http::error $tok] }
                default { error "Unknown error retrieving $url" }
            }
        } finally {
            http::cleanup $tok
        }
    }
}
```

Creating the thread pool is straightforward. We will use the defaults for all options except -initcmd since we need to pass in the above script.

```
set tpool [tpool::create -initcmd $init_script] → tpool00000000041564D0
tpool::names                                     → tpool00000000041564D0 ❶
```

❶   Lists all thread pools

Just as for threads, we will mark the thread pool as "in-use". This is discussed further in Section 22.12.7.

```
tpool::preserve $tpool → 1
```

22.12.2. Posting jobs to a thread pool: `tpool::post, tpool::get`

Any thread can post a *job* to a thread pool with the `tpool::post` command. A job is nothing but a Tcl script to executed by a worker thread just as we referred to scripts sent to specific threads as messages. The difference in terminology is conceptual.

```
tpool::post ?-detached? ?-nowait? TPOOL SCRIPT
```

The result of the command is a *job identifier* unless the `-detached` option, described below, is specified. The command initiates evaluation of *SCRIPT* in a worker thread in the following manner:

- If an idle thread is available in the thread pool, the *SCRIPT* is passed to it for execution and the command returns.
- If the `-nowait` option is specified, the command places the job on the work queue and returns.
- If no threads are idle and the maximum thread count limit has not been reached for the thread pool, a new thread is created. The `tpool::post` command will then wait in the event loop of the current thread processing events until the new thread is initialized. It will then pass the script to the new thread and return.
- If no threads are idle and the thread limit is reached the command will wait for a thread to become idle. In this case as well, while waiting the current thread's event loop will continue processing events.

The `-detached` option provides a "fire and forget" capability. If specified, the command creates a detached job which cannot be canceled or waited on, and whose result cannot be obtained. In this case the command returns an empty string.

> ⚠ Make note of one difference between using the `thread::send` command to send messages to a specific thread and the `tpool::post` command. The script passed in the `tpool::post` command may be processed by **any** thread in the pool. Therefore do not rely on context to be maintained between two `tpool::post` calls. Thus the following sequence
>
> ```
> tpool::post $tpool {set x 1}
> tpool::post $tpool {puts $x}
> ```
>
> is likely to fail with an undefined variable error since the second script may not be executed by the same thread as the first.

Let us fetch two URL with our previously created thread pool. The second URL is invalid just so we can demonstrate error handling in fetching results.

```
set job [tpool::post $tpool {fetch_url http://www.example.com}]     → 1
set bad_job [tpool::post $tpool {fetch_url xyz://www.example.com}] → 2
```

Now that the jobs have been posted, we need to know when they complete so we can retrieve the results.

22.12.3. Waiting for job completion: `tpool::wait`

The `tpool::wait` command is used to wait for the completion of one or more posted jobs.

```
tpool::wait TPOOL JOBLIST ?VARNAME?
```

Here *JOBLIST* is a list of job identifiers as returned by the tpool::post command. The command enters the current thread's event loop processing events until at least one of the jobs in *JOBLIST* has completed. It then returns the list of completed job identifiers (there may be more than one). If the optional *VARNAME* argument is provided, the command stores the identifier for jobs still pending into the variable of that name.

We can wait for our previously posted jobs to complete with the following loop.

```
set jobs [list $job $bad_job]
while {[llength $jobs]} {
    tpool::wait $tpool $jobs jobs
}
```

22.12.4. Retrieving a job result: tpool::get

Once a job is completed, its result can be retrieved with the tpool::get command.

```
tpool::get TPOOL JOBID
```

The result of the job is the result of the script evaluation in the worker thread. We can get the content of our URL in our example.

```
% tpool::get $tpool $job
→ <!doctype html>
  <html>
  <head>
      <title>Example Domain</title>

...Additional lines omitted...
```

If the script threw an error, the tpool::get command will also throw an error with the same errorCode and errorInfo values set by the original error. The command will also throw an error on an attempt to retrieve the result of a job that has not completed yet.

We provided one URL above that was invalid. Accordingly, an attempt to retrieve the result will raise an error.

```
% tpool::get $tpool $bad_job
Ø Unsupported URL type "xyz"
% set errorInfo
→ Unsupported URL type "xyz"
      while executing
  "http::geturl $url"
      (procedure "fetch_url" line 2)
      invoked from within
...Additional lines omitted...
```

Note how the error message and stack are those raised by the worker thread.

22.12.5. Canceling a job: tpool::cancel

Jobs on the work queue that are still pending **and not yet assigned to worker threads** can be cancelled with the tpool::cancel command.

```
tpool::cancel TPOOL JOBLIST ?VARNAME?
```

Here *JOBLIST* is the list of identifiers for the jobs to be cancelled. The command returns the list of identifiers for the cancelled jobs. If the *VARNAME* argument is provided, the list of jobs that could not be cancelled is stored in a variable of that name.

22.12.6. Suspending thread pools: `tpool::suspend`, `tpool::resume`

A thread pool can be suspended at any time by calling the `tpool::suspend` command.

```
tpool::suspend TPOOL
```

Suspending a thread pool will suspend execution of all threads in the pool. However, a caller may still post jobs to the pool's work queue.

The suspension may be rescinded with the `tpool::resume` command.

```
tpool::resume TPOOL
```

The command will then cause the worker threads to resume execution. Idle workers will pick up any additional jobs placed on the work queue while the thread pool was suspended. However, no new threads are created to handle additional pending jobs **even if the maximum thread count limit has not been reached**.

22.12.7. Thread pool lifetimes: `tpool::preserve`, `tpool::release`

Just as for threads, thread pool lifetimes need to be managed. The reasons are much the same. There may be multiple users of a thread pool and they do not want the pool to be disappearing from under them while in use. The solution provided is also much the same. A reference count is associated with each thread pool. The reference count is incremented with `tpool::preserve` and decremented with `tpool::release`. The pool is freed when the reference count drops to 0. Beyond that point, attempts to post a job to the pool will result in a failure. However, this does not necessarily mean the threads in the pool have exited. They will do so on their own schedule.

Since we done with our URL fetching thread pool, we can inform the package accordingly.

```
tpool::release $tpool → 0
```

22.13. Distributing interpreter state: the `Ttrace` package

A common need when working with multiple threads is to ensure that they are all running in the same runtime "environment" in terms of namespaces, procedure definitions and the like. The `Ttrace` package partially satisfies this need. It allows for procedure and namespace definitions to be replicated across all threads.

Because this is a separate package and not part of `Thread`, we have to load it separately.

```
package require Ttrace → 2.8.0
```

Although the package contains many commands, we only describe one, `ttrace::eval`, which encapsulates the other commands into a simple-to-use interface. The other commands allow for finer grain control but we do not describe them here.

```
ttrace::eval ARG ?ARG ...?
```

The `ttrace::eval` command evaluates the script formed from concatenation of it arguments. Any changes in procedure or namespace definitions are then propagated to all threads.

The functionality is most easily illustrated with an example. Let us start up a thread. All threads making use of this feature must load the package as well.

```
set tid [thread::create {
    package require Ttrace
    thread::wait
}]
→ tid0000000000002730
```

Obviously this thread will not have the namespace ns or the ping procedure defined.

```
% set tid [thread::create]
→ tid0000000000002AAC
% thread::send $tid {namespace exists ns}
→ 0
% thread::send $tid ping
∅ invalid command name "ping"
```

Now we define both the namespace and the procedure within the **current** thread but do so within a
ttrace::eval.

```
ttrace::eval {
    namespace eval ns {}
    proc ping {} {return "Ping from [thread::id]"}
}
```

We can then verify that the namespace and procedure have been replicated in the current thread **and** in the other
thread.

```
namespace exists ns                   → 1
ping                                  → Ping from tid0000000000002FE8
thread::send $tid {namespace exists ns} → 1
thread::send $tid ping                → Ping from tid0000000000002AAC
```

If a new thread is created, that will also automatically have these definitions.

```
% set tid2 [thread::create {
    package require Ttrace
    thread::wait
}]
→ tid000000000000174C
% thread::send $tid2 {namespace exists ns}
→ 1
% thread::send $tid2 ping
→ Ping from tid000000000000174C
```

You can see how the Ttrace package simplifies keeping threads in sync in terms of the namespace and procedure
definitions. However, there are some important limitations to keep in mind. In particular, data and TclOO classes
and objects are not replicated.

22.14. Using extensions in threads

Generally speaking, packages that are purely script-based do not require any special consideration for use in
threads. However, use of binary extensions in a Tcl application that uses multiple threads takes some care. These
fall into three categories:

- Extensions that are written to be thread-safe and can be safely loaded and used in multiple threads.
- Extensions that are **not** thread-safe but can be safely loaded and used in a **single** thread within a multithreaded
  applications. If other threads want to make use of the extension functionality, they need to do so by sending
  messages to the thread in which the extension is loaded. The Tk extension is one such example.

- Extensions that not thread-safe and should not be used in a threaded Tcl application.

Refer to the documentation for each extension to determine its thread safety characteristics.

22.15. Comparing coroutines and threads

Having worked through both coroutines and threads, you can see that at some level they serve a similar purpose in enabling computational tasks to proceed in concurrent fashion. We now summarise their differences to help you choose the one appropriate for the problem you are trying to solve.

The primary differences between coroutines and threads arise from the fact that threads are operating system level constructs whereas coroutines are implemented purely in user mode.

- Multiple threads can be assigned to multiple processors leading to increased performance. Coroutines run within a single thread and therefore are limited to the processor on which the thread is running. They derive no benefit from the presence of multiple processors. Of course, there is no reason not to run coroutines within multiple threads but that is a different kettle of fish as coroutines can only communicate within the interpreter where they are defined and would have to use the thread mechanisms for anything else.

- Blocking operations in a thread do not block other threads. In contrast, a blocking operation in a coroutine will block other coroutines as well. This is often a primary consideration in the decision to move to a thread-based architecture. Certain operations, for example accessing some database implementations, are only implemented in blocking form by the database drivers. In such cases, moving those operations to a separate thread ensures other parts of an application continue to run while a long database operation is in progress.

- Coroutines are defined within a single interpreter and share its namespaces, commands, channels etc. There is limited isolation between coroutines. Threads on the other hand enclose independent interpreters. There is no sharing of any kind except through the threading commands discussed earlier. This means errors in one thread can be isolated to that thread.

- Threads are relatively expensive to create and consume significant system resources like memory while coroutines are cheap in that respect. Inter-thread communication is also slower as it forces operating system context switches.

- Partly because they are cheap and share code and data, coroutines can be used for implementing generators and the like. Threads are not suitable for such purposes.

Having contrasted the two, keep in mind that coroutines and threads are not mutually exclusive. It is common and perfectly reasonable to use both in an application architecture.

The article Modeling a Queuing System compares and contrasts multiple implementations of a queueing system model that include threads and coroutines in addition to other mechanisms.

22.16. Chapter summary

In this chapter, we covered the use of operating system threading capabilities from within Tcl. Multithreading permits the application to take advantage of multiple processors in the system. However, its use must be carefully considered as it adds significant complexity to an application. In some languages, multithreading is required for concurrent I/O; however, this is not the case in Tcl where the asynchronous I/O model through the event loop is not only adequate but often more efficient. Similarly, in cases where the purpose of multithreading is to simplify programming of concurrent independent tasks while writing in a "sequential" style, coroutines can fulfil the need at a cheaper cost.

Nevertheless there are instances where only threads meet the desired needs and Tcl provides for that possibility. In conjunction with Tcl's event loop and coroutines, practically any software architecture geared towards multitasking and concurrent computation can be implemented in Tcl.

22.17. References

MARK2015

Modeling a Queuing System, Arjen Markus, http://www.magicsplat.com/articles/queueing.html. Compares thread and coroutine-based implementations of a queueing system model.

23

Database Connectivity with TDBC

The *Tcl Database Connectivity* (TDBC) extension provides a Tcl API for accessing SQL databases. Because this API is independent of the underlying database system, most[1] of the code accessing databases in an application can run unchanged between different database implementations.

A bit of history

TDBC 1.0, authored by Kevin B. Kenny, was released with Tcl 8.6. Prior to its release, the lack of a standard Tcl database access API lead to a number of extensions with different interfaces including

- *tclodbc* for connecting to databases using ODBC
- *DIO* which is part of Apache Rivet
- *nstcl* and *nsdbi*, both derived from AOL Server web server
- Various database-specific extensions like *oratcl* for Oracle and *pgtcl* for PostgreSQL.

With the advent of TDBC, applications can now rely on a standard interface to databases from Tcl.

TDBC is broken up into two layers:

- The upper layer, which is what we cover here, is the interface used by applications to access the database.
- The lower layer consists of different drivers that implement access to specific databases. At the time of writing, the TDBC distribution includes drivers for MySQL, PostgreSQL, Sqlite3 and any database accessible via an ODBC interface. The TDBC documentation also defines an interface that allows new drivers to be written for other database implementations.

Due to space limitations, this book only covers the first of these — the application interface to databases.

23.1. Installing TDBC

The TDBC extension is included in the standard Tcl 8.6 source distributions as well as all binary distributions. However, individual database driver components may have to be installed separately. Most binary distributions of Tcl include drivers for SQLite and Windows has native support for ODBC. In other cases, the drivers, usually implemented as shared libraries, are available from the database vendor.

23.2. Loading TDBC

TDBC is loaded with the standard `package require` Tcl command. The specific package to be loaded depends on which database driver is desired. The packages included in the core distribution are shown in Table 23.1.

---

[1] Some code will be necessarily specific to databases because of differing capabilities and quirks in database implementations.

Table 23.1. Core TDBC driver packages

Package	Database
tdbc::sqlite3	Sqlite3
tdbc::postgres	PostgreSQL
tdbc::mysql	MySQL
tdbc::odbc	Any database that is provides an ODBC interface

In addition, open source TDBC drivers are also available such as ones for JDBC[2], CUBRID[3] and MonetDB[4].

Naturally, when dealing with multiple database implementations in an application, more than one of these packages may be loaded if desired.

For our code examples, we will make use of the Sqlite3 database and thus load the corresponding package

```
% package require tdbc::sqlite3
→ 1.0.4
```

23.3. Concepts

TDBC follows the same general pattern as other database access API's and involves the following steps:

1. First a *connection*[5] has to be established to the database (and database provider) of interest. In addition to identifying the database this may also include authorization credentials and other options. All subsequent interactions for the database are done through this connection object and its surrogates.

2. Next a SQL *statement* is *prepared* using the connection object and then executed with the results returned as a *result set*.

3. The *result set* is iterated over to operate on the returned data.

4. The statement and result sets are freed so as to not use up resources.

5. Steps 2-4 are repeated as needed.

6. When all done, the database connection is closed.

TDBC encapsulates all the above abstractions as the TclOO classes `tdbc::connection`, `tdbc::statement` and `tdbc::resultset`.

23.4. Connecting to databases

A database connection is represented by an object of the appropriate `tdbc::DRIVER::connection` class. To connect to a database, you create an object of this class specifying the database of interest. The manner in which the database is identified depends on the database driver in use.

23.4.1. Connecting to SQLite: `tdbc::sqlite3::connection`

The `tdbc::sqlite3` package must be loaded to access SQLite databases.

```
package require tdbc::sqlite3
```

A connection to a SQLite database is created by passing the path to the sqlite3 database file to the `tdbc::sqlite3::connection` command. For example, to open a SQLite database in the current directory,

---

[2] https://github.com/ray2501/TDBCJDBC
[3] https://github.com/ray2501/tclcubrid
[4] https://sites.google.com/site/tclmonetdb/
[5] Not to be confused with a *network* connection.

```
tdbc::sqlite3::connection create db my-database.sqlite3
```

This will create the object db representing the database connection to the my-database.sqlite3 database. As we see in a moment, we can operate on the database by invoking methods on this object.

For our sample code, we will create and make use of an in-memory SQLite database. The special token :memory: results in the database being created purely in memory with no disk store. It will be erased once closed which suffices for the sample code purposes.

```
% set dbconn [tdbc::sqlite3::connection new :memory:]
→ ::oo::Obj601
```

Note here we use the TclOO method new, as opposed to create, which generates the connection object name for us.

23.4.2. Connecting via ODBC: `tdbc::odbc::connection`

Open Database Connectivity (ODBC) is an industry standard API for accessing databases. Database implementations that support this interface can be accessed through the tdbc::odbc package.

```
package require tdbc::odbc
```

Windows comes with ODBC support built-in but to use ODBC on Unix or Linux, you may need to install an ODBC package such as unixODBC[6] or iODBC[7].

To connect to an ODBC database, pass its connection string to the tdbc::odbc::connection command. This takes the form of a series of attribute name and value pairs that specify the connection characteristics. For example, (assuming we are on Windows)

```
tdbc::odbc::connection create db "Driver={SQL
  Server};Server=localhost;Trusted_Connection=Yes;Database=YourDatabaseName;"
```

will return a connection object for the SQL Server database YourDatabaseName on the local system. Notice additional attributes can be specified in the connection string. For instance, the Trusted_Connection=Yes attribute and value specify that the credentials of the Windows account under which the application is running are to be used for authorization.

Depending on the system and the database, you can define a *data source name* (DSN) that stores the data used to construct a connection string. Then you can simply specify the DSN to connect to the database. So the above call would then become

```
tdbc::odbc::connection create db "DSN=YourDSN"
```

You can use the ODBC utilities in TDBC to define DSN's if the underlying ODBC implementation supports the ODBC Installer API. Alternatively, on Windows systems, you can use the ODBC applet in the Control Panel to define and configure DSN's. On Unix/Linux, unixODBC and iODBC both provide GUI and command line means of defining DSN's.

---

[6] http://www.unixodbc.org
[7] http://www.iodbc.org

Connection strings are ODBC driver specific and sometimes difficult to get right. The Connection Strings Reference web site is a useful resource to understand and construct these.

In addition to the common options (see Table 23.4) supported by all TDBC drivers, some ODBC environments support the `-parent` option which results in a prompt for user credentials if required. See the tdbc::odbc[8] reference for details on its use.

As described in Section 23.11, the package also implements some utility commands for interacting with the system ODBC manager.

23.4.3. Connecting to MySQL: `tdbc::mysql::connection`

Connecting to MySQL requires the `tdbc::mysql` package.

```
package require tdbc::mysql
```

The package differs from SQLite and ODBC in that the `tdbc::mysql::connection` command for establishing a database connection identifies the database to be connected through a set of options as opposed to a file name or connection string. These options are shown in Table 23.2.

Table 23.2. MySQL connection options

Option	Description
`-host` *HOSTNAME*	The name of the system on which the database server is running. Defaults to the local system.
`-port` *PORTNUMBER*	The TCP/IP port number on which the server is listening for connections.
`-socket` *PATH*	Connects to the Unix socket or named pipe specified by *PATH*.
`-user` *USERNAME*	The name of the user name to be used to access the database. Defaults to the current user id of the process.
`-password` *PASSWORD*	The password to be presented to the server. By default no password is presented.
`-database` *DATABASE*	The name of the database to default to if no database is specified in a query. Defaults to the default database for the user specified by the `-user` option.
`-db` *DATABASE*	Same as the `-database` option.
`-interactive` *BOOL*	If specified as `true`, sets the default timeout to be that for an interactive user; otherwise, the default timeout is set as for batch users.
`-ssl_ca` *PATH*	Specifies the **file** containing the list of trusted certificate authorities permitted for an SSL connection.
`-ssl_capath` *PATH*	Specifies the **directory** containing the files containing certificates for trusted authorities permitted for an SSL connection.
`-ssl_cert` *PATH*	Specifies the file containing the client certificate.
`-ssl_key` *PATH*	Specifies the file containing the client private key.
`-ssl_cipher` *CIPHERLIST*	Specifies the permissible ciphers to use for an SSL connection. *CIPHERLIST* is a list of cipher names separated by colons.

---

[8] http://www.tcl.tk/man/tcl8.6/TdbcodbcCmd/tdbc_odbc.htm

23.4.4. Connecting to PostgreSQL: `tdbc::postgres::connection`

Connecting to a PostgreSQL database is more or less identical to what was described previously for MySQL. The `tdbc::postgres` package is loaded

```
package require tdbc::postgres
```

and the connection is made via `tdbc::postgres::connection` using the options shown in Table 23.3.

Table 23.3. PostgreSQL connection options

Option	Description
-host _HOSTNAME_	The name of the system on which the database server is running.
-hostaddr _IPADDR_	The IP address of the system on which the database server is running. Takes precedence over the -host option.
-port _PORTNUMBER_	The TCP/IP port number on which the server is listening for connections.
-user _USERNAME_	The name of the user name to be used to access the database. Defaults to the current user id of the process.
-password _PASSWORD_	The password to be presented to the server. By default no password is presented.
-pw _PASSWORD_	Same as the -password option.
-database _DATABASE_	The name of the database to default to if no database is specified in a query. Defaults to the default database for the user specified by the -user option.
-db _DATABASE_	Same as the -database option.
-options _OPTS_	Specifies additional command line options to send to the server.
-sslmode disable|allow|prefer|require	A value of disable mandates a non-SSL connection to the server, require mandates an SSL connection, allow prioritizes a non-SSL over SSL and prefer (default) prioritizes SSL over non-SSL.
-service _SVCNAME_	Specifies that additional connection parameters are to be picked up from the entry corresponding to _SVCNAME_ in the pg_service.conf file.

23.4.5. Common connection options

All TDBC drivers understand the common set of options shown in Table 23.4.

Table 23.4. Connection options common to all TDBC drivers

Option	Description
-encoding _NAME_	Name of the character encoding to be used on the connection. _NAME_ should be one of the names returned by the Tcl encoding command. It is generally not necessary to specify this but be aware that drivers differ in their handling of this option.
-isolation _ISOLATION_	Specifies the transaction isolation level needed for transactions on the database. _ISOLATION_ must be one of readuncommitted, readcommitted, repeatableread, or serializable. See the tdbc::connection[9] reference for details.
-timeout MILLISECS	Specifies the interval after which an operation should time out with an error. The default value of 0 indicates no timeout. The operations to which the

[9] http://www.tcl.tk/man/tcl8.6/TdbcCmd/tdbc_connection.htm

Option	Description
	timeout is applicable differs between the various drivers and databases. Refer to the appropriate reference pages for each driver.
-readonly BOOLEAN	If specified as `true` or 1, the connection will not modify the database.

23.4.6. Configuring connections: *DBCONN* `configure`

The values of the options that can be specified at the time a `tdbc::connection` object is created can be retrieved via its `configure` method. The same method can also be used to modify the values of some options.

So to retrieve the configuration of our in-memory sample database.

```
% $dbconn configure
→ -encoding utf-8 -isolation serializable -readonly 0 -timeout 0
```

We can also pass one or more configuration options to be modified.

```
% $dbconn configure -timeout 1000
```

23.4.7. Releasing connection resources: *DBCONN* `close`

When no longer required, connections must be closed by invoking the `close` method on the `connection` object.

```
db close
```

This will also close and release resources related to `tdbc::statement` and `tdbc::resultset` objects created through the connection.

23.5. Executing SQL

Executing a SQL statement involves first *preparing* the statement and then running it one or more times with different parameter values.

23.5.1. Preparing a statement: *DBCONN* `prepare`

The first step in executing SQL is to create a `tdbc::statement` object via the `prepare` method of a `tdbc::connection`.

```
DBCONN prepare SQL
```

Here *SQL* is the SQL statement to be executed. The following creates a table in our sample database.

```
set stmt [$dbconn prepare {
    CREATE TABLE Accounts
    (Name text,
     AcctNo text NOT NULL PRIMARY KEY,
     Balance double)
}]
→ ::oo::Obj601::Stmt::1
```

We can then use the created `tdbc::statement` object to run the SQL script against the database.

23.5.2. Executing a prepared statement: *STMT* `execute`

Once a `tdbc::statement` is created, its `execute` method in invoked invoked to run the corresponding SQL.

```
% set res [$stmt execute]
→ ::oo::Obj602::ResultSet::1
```

The execute method returns a tdbc::resultset object which we will examine later. For now, we free up both objects by invoking their close method. Like tdbc::connection objects, tdbc::statement and tdbc::resultset objects should also be freed when no longer required.

```
% $stmt close
```

Note that closing the $stmt also closes any contained resultset objects so we do not need to explicitly close $res here. Similarly, tdbc::statement objects that are not closed explicitly will be closed when the owning tdbc::connection object is closed. However, for the sake of saving resources, it is generally a good idea to explicitly release them when no longer needed. Since we have more we want to do with the connection and are not closing it, we explicitly close $stmt.

Insertions and queries follow a similar pattern.

```
% set stmt [$dbconn prepare {INSERT INTO Accounts (Name, AcctNo, Balance) VALUES ('Tom', \
    'A001', 100.00)}]
→ ::oo::Obj601::Stmt::2
% $stmt execute
→ ::oo::Obj604::ResultSet::1
% $stmt close ❶
```

❶ Will also close the result set returned by execute

This multi-step sequence of prepare and execute can be a little tedious and TDBC provides some methods that act as wrappers and make it more convenient. We will discuss these and their pros and cons a little later.

23.5.3. Bound variables

The above example hard-coded the values that were to be inserted into the table. Naturally, that is not a viable option when values are not known apriori at the time a program is written.

TDBC allows for Tcl variable values to be passed into the SQL statement by binding names within the SQL that begin with : by their corresponding values. These values may either come from Tcl variables of the same name or from a dictionary passed in as an argument.

```
% set stmt [$dbconn prepare {
    INSERT INTO Accounts (Name, AcctNo, Balance) VALUES (:name, :acctno, :balance)
}]
→ ::oo::Obj601::Stmt::3
```

Here the bound variables are name, acctno and balance. In the script below, the values for these will be sourced from the Tcl variables of the same name.

```
foreach {name acctno balance} {
    Dick   A002 200.00
    Harry  A003 300.00
} {
    $stmt execute
}
```

Alternatively, we can pass in a dictionary to the execute command. The values will be picked up from the keys of the same name in the dictionary.

```
% $stmt execute {acctno A004 name Moe balance 100.00} ❶
→ ::oo::Obj606::ResultSet::3
```

❶ Order of elements does not matter

Note from the sequence above that a prepared statement can be executed multiple times with different values.

23.5.3.1. Bound variable configuration: *STMT* paramtype

Most databases drivers automatically figure out the type and direction (input, output, or both) of bound variables. A few require the application to provide this information. The paramtype method is provided for this purpose.

STMT paramtype *NAME* ?*DIRECTION*? *TYPE* ?*PRECISION*? ?*SCALE*?

Here *NAME* is the name of the bound variable. The *DIRECTION* argument specifies whether the bound variable is used to pass input (in), receive output (out) or both (inout). The *TYPE*, *PRECISION* and *SCALE* arguments correspond to the type, precision and scale column attributes shown in Table 23.5.

Although this is not required for SQLite, which figures out the information on its own, we could configure the balance variable in our statement as follows:

```
% $stmt paramtype balance in double
```

Conversely, the params method of the tdbc::statement object returns the configuration of the bound variables.

```
% print_dict [$stmt params]
→ acctno    = type Tcl_Obj precision 0 scale 0 nullable 1 direction in
  balance   = type Tcl_Obj precision 0 scale 0 nullable 1 direction in
  name      = type Tcl_Obj precision 0 scale 0 nullable 1 direction in
```

The result of the method is a nested dictionary keyed by the names of the bound variables. Each subdictionary contains details about the corresponding bound variable. The keys of this subdictionary are the same that were described for columns in Table 23.5 with an additional key, direction which may have the value in, out or inout.

23.5.4. Closing prepared statements: *STMT* close

Once a tdbc::statement has outlived its utility, resources associated with it should be freed via its close method.

```
% $stmt close
```

The result sets are also automatically freed when the associated tdbc::statement object or containing tdbc::connection object are closed.

23.5.5. Direct evaluation (MySQL only): *DBCONN* evaldirect

For cases where MySQL does not support data management language statements via the prepare call, the evaldirect method can be used to directly pass SQL code for execution in MySQL without going through a prepare call first.

DBCONN evaldirect *SQLSTMT*

This method is only supported by the MySQL TDBC driver and should only be used in those cases where MySQL does not support the statement via the prepare method.

23.6. Retrieving data from result sets

Any data returned from executing SQL is stored in a result set wrapped as a `tdbc::resultset` object.

```
% set stmt [$dbconn prepare {SELECT Name, Balance from Accounts}]
→ ::oo::Obj601::Stmt::4
% set res [$stmt execute]
→ ::oo::Obj610::ResultSet::1
```

23.6.1. Introspecting result sets: *RESULTSET* `columns`

The result set is a table whose column names can be retrieved with the `tdbc::resultset` object's `columns` method.

```
$res columns → Name Balance
```

The `rowcount` method returns the number of rows in the result set table.

```
$res rowcount → 1
```

23.6.2. Retrieving result set rows: *RESULTSET* `nextlist|nextdict|nextrow`

The data itself can be retrieved using one of several methods. The most basic of these are the `nextlist`, `nextdict` and `nextrow` methods.

```
RESULTSET nextlist VAR
RESULTSET nextdict VAR
RESULTSET nextrow ?-as lists|dicts? VAR
```

All three methods return 0 if there are no more rows in the result set. Otherwise they return 1 and store the next row from the result set into the variable named *VAR*. The difference between them is the format in which the row is stored in the variable:

- `nextlist` stores the row as a list in the same order as returned by the `columns` method.
- `nextdict` stores the row as a dictionary whose keys are the column names of the result set.
- `nextrow` stores the row either as a list or a dictionary depending on the value of the `-as` option (which defaults to `dicts`).

```
% $res nextlist val
→ 1
% puts $val
→ Tom 100.0
% while {[$res nextdict val]} {
    puts $val
}
→ Name Dick Balance 200.0
  Name Harry Balance 300.0
  Name Moe Balance 100.0
```

23.6.3. Multiple result sets: *RESULTSET* `nextresults`

Some databases support a single SQL statement returning multiple result sets, each of which may have a different column structure. The presence of additional result sets may be detected by calling the `nextresults` method. This

method must be called after the nextlist or nextdict command returns 0 indicating there are no more rows in the current result set.

```
$res nextresults → 0
```

The method returns 0, as in our example, if there are no more result sets. A return value of 1 indicates there are more result sets. The application can access them in the same manner as described above while noting that the columns may differ between result sets.

23.6.4. Releasing result sets: *RESULTSET* close

Once the data of interest within a result set is retrieved, associated resources should be released by calling the close method on the tdbc::resultset object.

```
% $res close
```

The result sets are also automatically freed when the associated tdbc::statement object or containing tdbc::connection object are closed.

23.6.5. Convenience wrappers for retrieval

As we have seen above, database operations involve calling the prepare and execute methods and then freeing the tdbc::statement and tdbc::resultset objects. To ensure proper cleanup, the sequence has to be wrapped in try or catch blocks. So in pseudocode the code looks roughly like this:

```
set stmt [$dbconn prepare SQL STATEMENT]
try {
    set res [$stmt execute]
    try {
        Loop using [$res nextdict] or [$res nextlist]
    } finally {
        $res close
    }
} finally {
    $stmt close
}
```

TDBC provides two convenience methods, allrows and foreach, that take care of all the boilerplate in the above and are supported by all the major TDBC classes, tdbc::connection, tdbc::statement and tdbc::resultset,

23.6.5.1. Retrieving a complete result set: allrows

The allrows method encapsulates the above pseudocode where the loop processing consists of simply collecting all results returned by nextdict or nextlist into a single list.

The method is implemented by tdbc::connection, tdbc::statement and tdbc::resultset.

```
RESULTSET allrows ?-as lists|dicts? ?-columnsvariable COLVAR?
STATEMENT allrows ?-as lists|dicts? ?-columnsvariable COLVAR? ?--? ?DICT?
DBCONN    allrows ?-as lists|dicts? ?-columnsvariable COLVAR? ?--? SQL ?DICT?
```

- In the case of tdbc::resultset, allrows simply iterates over the result set collecting the output of nextlist or nextdict methods.

- In the case of tdbc::statement, allrows executes the statement and then collects rows from the returned result set as described in the previous case. If the optional *DICT* argument is provided, it contains the bound variables else their values are taken from Tcl variables in the caller's context. See Section 23.5.3.

- In the case of tdbc::connection, allrows prepares the SQL passed as the *SQL* argument, then executes the returned statement as described in the previous case. The *DICT* argument has the same purpose as above.

The -as option controls whether each element of the returned list is itself a list containing the column values for a row or a dictionary keyed by column name (default). If the -columnsvariable option is specified, the column names for the result set are stored in the variable *COLVAR* in the caller's context.

In all cases, the allrows method takes care to free up objects and resources appropriately even in case of errors.

Below we illustrate a simple query using the different methods, first without using allrows.

```
% set query_values [dict create amount 200]
→ amount 200
% set stmt [$dbconn prepare {
    SELECT Name FROM Accounts WHERE Balance >= :amount
}]
→ ::oo::Obj601::Stmt::5
% set rows {}
% try {
    set res [$stmt execute $query_values]
    try {
        while {[$res nextdict row]} {
            lappend rows $row
        }
    } finally {
        $res close
    }
} finally {
    $stmt close
}
% print_list $rows
→ Name Dick
  Name Harry
```

Now the same code but using the allrows method of the tdbc::resultset object.

```
% set stmt [$dbconn prepare {
    SELECT Name FROM Accounts WHERE Balance >= :amount
}]
→ ::oo::Obj601::Stmt::6
% set rows {}
% try {
    set res [$stmt execute $query_values]
    try {
        set rows [$res allrows] ❶
    } finally {
        $res close
    }
} finally {
    $stmt close
}
→ {Name Dick} {Name Harry}
% print_list $rows
→ Name Dick
  Name Harry
```

❶ Replaces the inner loop in the previous example

Notice this returns the rows as dictionaries by default.

Above, we have only saved writing the innermost loop in the code. We can go another step further and use the `allrows` method of the `tdbc::statement` object. Note the difference from the `allrows` method of the `tdbc::resultset` in that here we need to pass in the values to be used for querying to the `allrows` method.

```
% set rows {}
% set stmt [$dbconn prepare {
    SELECT Name FROM Accounts WHERE Balance >= :amount
}]
→ ::oo::Obj601::Stmt::7
% try {
    set rows [$stmt allrows $query_values] ❶
} finally {
    $stmt close
}
→ {Name Dick} {Name Harry}
% print_list $rows
→ Name Dick
  Name Harry
```

❶ We do not have to explicitly deal with result sets

Finally, we go the whole hog and invoke `allrows` on the database connection itself. Obviously, in this case we have to tell it the SQL we want to run in addition to passing in the query values.

```
% set rows [$dbconn allrows {
    SELECT Name FROM Accounts WHERE Balance >= :amount
} $query_values] ❶
→ {Name Dick} {Name Harry}
% print_list $rows
→ Name Dick
  Name Harry
```

❶ We do not have to deal with statements

Given this last illustration is so much shorter than the previous examples, why would one pick any of the others? The Tcl Database Connectivity paper provides some hints:

- With very large databases and result sets, `allrows` may be unworkable because of the infeasibility of collecting all rows in memory before processing.
- Explicitly dealing with result sets allows for fine-grained control of the iteration, for example terminating the iteration based on some complex rules outside of SQL's capabilities.

The use of the `-as` and `-columnsvariable` is shown below.

```
% set rows [$dbconn allrows -as lists -columnsvariable cols {
    SELECT Name,Balance FROM Accounts WHERE Balance >= :amount
} $query_values]
→ {Dick 200.0} {Harry 300.0}
% print_list $rows
→ Dick 200.0
  Harry 300.0
% puts $cols
→ Name Balance
```

23.6.5.2. Iterating over result sets: `foreach`

The `foreach` method has a purpose very similar to `allrows` except that instead simply collecting results into a list, it executes a given script for every row in the result set. Like `allrows`, it is implemented by the `tdbc::connection`, `tdbc::statement` and `tdbc::resultset` classes.

```
RESULTSET allrows ?-as lists|dicts? ?-columnsvariable COLVAR? VAR SCRIPT
STATEMENT allrows ?-as lists|dicts? ?-columnsvariable COLVAR? ?--? VAR ?DICT? VAR SCRIPT
DBCONN    allrows ?-as lists|dicts? ?-columnsvariable COLVAR? ?--? SQL ?DICT? SCRIPT
```

The method will iterate over all rows in the result set evaluating *SCRIPT* after assigning the value of the row to the variable *VAR*. The options `-as` and `-columnsvariable`, as well as other arguments, have the same semantics as described for allrows in the previous section.

Because of its similarity to `allrows`, we do not discuss the method in detail but only illustrate it as invoked on a `tdbc::connection` object.

```
% $dbconn foreach row {
    SELECT Name FROM Accounts WHERE Balance >= :amount
} $query_values {
    puts $row
}
→ Name Dick
  Name Harry
```

Like `allrows`, `foreach` also takes care of all intermediate bookkeeping in terms allocating and release objects.

23.7. Database transactions

There are a couple of ways an application may make use of transactions. We describe both below.

23.7.1. Using the `transaction` method

The first is making use of the `transaction` method of `tdbc::connection`.

```
DBCONN transaction SCRIPT
```

This begins a transaction on the connection and evaluates the passed script. If the script completes with a return code of ok, `return`, `break` or `continue`, the transaction is committed. For other return codes, including errors, the transaction is rolled back and the error is rethrown.

Use of the method is illustrated by the simplistic example below to transfer funds from one account to another.

```
% set transfer { from "Tom" to "Dick" amount 50 }
→ from "Tom" to "Dick" amount 50
% $dbconn transaction {
    $dbconn allrows -as lists -- {
        UPDATE Accounts
        SET Balance = Balance - :amount
        WHERE Name=:from
    } $transfer ❶

    puts "Within transaction: [$dbconn allrows -as lists -- {
            SELECT Name, Balance FROM ACCOUNTS WHERE Name=:from
    } $transfer]" ❷

    error "Pretend something went wrong"
```

```
        $dbconn allrows -as lists -- {
            UPDATE Accounts
            SET Balance = Balance + :amount
            WHERE Name=:to
        } $transfer ❸
}
❷ Within transaction: {Tom 50.0}
  Pretend something went wrong
% $dbconn allrows -as lists -- {
        SELECT Name, Balance FROM Accounts WHERE Name=:from
} $transfer ❹
→ {Tom 100.0}
```

❶   Deduct from Tom's balance
❷   Verify balance updated within transaction
❸   Add to Dick's balance
❹   Verify Tom's balance restored to original

Notice that Tom's balance is restored as the transaction was aborted by an error exception.

23.7.2. Using `begintransaction`, `commit` and `rollback`

In cases where the sequence of operations in a transaction cannot be neatly wrapped in a script that can be passed to the `transaction` method, an application can explicitly manage the transaction itself by calling the `begintransaction` method of a `tdbc::connection` object.

DBCONN `begintransaction`

Then at some later point, it can call the `commit` or `rollback` methods to either complete or abort the transaction.

DBCONN `commit`
DBCONN `rollback`

We can rewrite the previous example as below.

```
% set transfer { from "Tom" to "Dick" amount 50 }
→  from "Tom" to "Dick" amount 50
%
% $dbconn begintransaction ❶
% $dbconn allrows -as lists -- {
     UPDATE Accounts
     SET Balance = Balance - :amount
     WHERE Name=:from
} $transfer
%
% puts "Within transaction: [$dbconn allrows -as lists -- {
            SELECT Name, Balance FROM ACCOUNTS WHERE Name=:from
} $transfer]"
→ Within transaction: {Tom 50.0}
%
% if {[catch {
     error "Pretend something went wrong"
     $dbconn allrows -as lists -- {
         UPDATE Accounts
         SET Balance = Balance + :amount
         WHERE Name=:to
     } $transfer
}]} {
```

```
    $dbconn rollback ❷
} else {
    $dbconn commit ❸
}
% $dbconn allrows -as lists -- {
    SELECT Name, Balance FROM Accounts WHERE Name=:from
} $transfer
→ {Tom 100.0}
```

❶   Begin the transaction
❷   On error, rollback the transaction
❸   On success, commit the transaction

23.8. Handling NULL values

Noting that the empty string "" is not the same as the SQL NULL value, there is no way to represent NULL in Tcl where everything is a string. In some applications, the distinction is not important and the empty string can be used interchangeably with NULL. In cases where the distinction is important, the dictionary-based interface to TDBC methods should be used as illustrated here.

Writing NULL values

To write NULL to a table column, pass a dictionary containing the bound variable values for the columns. A NULL will be stored in any column for which a corresponding key is not present in the dictionary.

```
% $dbconn allrows {
    INSERT INTO Accounts (Name, AcctNo, Balance) VALUES (:name, :acctno, :balance)
} {name Curly acctno C007}
```

Similarly, to retrieve data containing NULL, use one of the forms that returns rows as dictionaries. If a value is NULL, the returned dictionary for the row will not contain the corresponding key.

```
% $dbconn allrows {SELECT Name, Balance, AcctNo FROM Accounts WHERE Name='Curly'}
→ {Name Curly AcctNo C007}
```

Note that the key Balance is missing. You could have also used the tdbc::resultset::nextdict method for the same purpose.

Note however the result when list format is used.

```
% $dbconn allrows -as lists {SELECT Name, Balance, AcctNo FROM Accounts WHERE Name='Curly'}
→ {Curly {} C007}
```

In this case there is no way to distinguish whether the stored value in the database was actually "" or NULL.

23.9. Stored procedures: *DBCONN* `preparecall`

Stored procedures can be invoked with the `preparecall` method of a tdbc::connection object.

```
DBCONN preparecall CALL
```

The syntax of the stored procedure call is

```
?RESULTVAR =? STOREDPROCNAME (? arg, …?)
```

Like the prepare method, this also returns a tdbc::statement object which can then be used as described earlier.

23.10. Introspection

All TDBC classes allow for introspection and inspection of the meta-information associated with databases.

23.10.1. Enumerating objects: *DBCONN* statements

TDBC keeps track of the objects that are still open within a database connection. The statements and resultsets methods retrieve the names of any existing tdbc::statement and tdbc::resultset objects within the tdbc::connection.

```
% $dbconn statements
→ ::oo::Obj601::Stmt::4
% $dbconn resultsets
```

Clearly we forgot to release some objects. This is actually useful for a final cleanup, for example on a per request basis to a web server that leaves the database connection open.

23.10.2. Introspecting tables: *DBCONN* tables

We can introspect the tables within a database with the tables method of a tdbc::connection object.

```
DBCONN tables ?SQLPAT?
```

If the *SQLPAT* argument is not provided, the commands returns information about all tables in the database. Otherwise, only tables whose name matches *SQLPAT* are included in the result. Note that *SQLPAT* should be in SQL pattern syntax.

```
% $dbconn tables
→ accounts {type table name accounts tbl_name Accounts rootpage 2 sql {CREATE TABLE Accounts
      (Name text,
       AcctNo text NOT NULL PRIMARY KEY,
       Balance double)}}
% $dbconn tables A% ❶
→ accounts {type table name accounts tbl_name Accounts rootpage 2 sql {CREATE TABLE Accounts
      (Name text,
       AcctNo text NOT NULL PRIMARY KEY,
       Balance double)}}
```

❶   % is a wildcard in SQL pattern syntax

The command returns a nested dictionary with the first level keys being the table names. The second level keys and values are dependent on the specific database driver. Refer to the reference documentation for the driver for details.

23.10.3. Introspecting columns: *DBCONN* columns

Similarly, the columns method retrieves column information for one or more columns in a table.

```
DBCONN columns TABLE ?SQLPAT?
```

If the *SQLPAT* argument is not provided, the commands returns information about all columns in the table *TABLE*. Otherwise, only columns with names matching *SQLPAT* are included in the result.

```
% print_dict [$dbconn columns Accounts]
→ acctno   = cid 1 name acctno type text notnull 1 pk 1 precision 0 scale 0 nullable 0
  balance  = cid 2 name balance type double notnull 0 pk 0 precision 0 scale 0 nullable 1
  name     = cid 0 name name type text notnull 0 pk 0 precision 0 scale 0 nullable 1
% print_dict [$dbconn columns Accounts Bal%]
→ balance  = cid 2 name balance type double notnull 0 pk 0 precision 0 scale 0 nullable 1
```

The command result is a nested dictionary keyed by the column name. The second level dictionary for each column provides details about the column and contains the keys shown in Table 23.5.

Table 23.5. Column description keys

Key	Description
type	Data type of the column
precision	Column precision in bits, decimal digits or width in characters, depending on the column type
scale	Scale of the column, i.e. number of digits after the radix point
nullable	Has the value 1 if the column can contain SQL NULL values and 0 otherwise

Additional keys may be present in the second level dictionaries depending on the database driver in use. Refer to the TDBC documentation for the driver for details.

23.10.4. Introspecting keys: *DBCONN* `primarykeys|foreignkeys`

Information about the primary keys defined for a table can be obtained with the `primarykeys` method of a `tdbc::connection` object.

DBCONN `primarykeys` *TABLE*

The method returns a list of descriptors for the primary keys defined for the table. Each descriptor is a dictionary with at least the key `columnName` containing the name of a column that is a member of the primary key. The descriptor may contain other database-dependent keys.

```
% $dbconn primarykeys Accounts
→ {ordinalPosition 2 columnName AcctNo}
```

In a similar vein, the `foreignkeys` method retrieves information about foreign key relationships for the specified table.

DBCONN `foreignkeys` `?-primary` *TABLE*`? ?-foreign` *TABLE*`?`

If the `-foreign` option is specified, only the keys appearing in that table are included in the result. If the `-primary` option is specified, only the keys that refer to that table are included. It is recommended that one or both options be specified as otherwise the returned results depend on the database driver in use.

The method returns a list of descriptors of foreign key relationships. Each descriptor is a dictionary with the keys shown in Table 23.6. Depending on the database in use, the descriptor may contain additional keys apart from the ones shown in the table.

Table 23.6. Foreign key dictionary

Key	Description
foreignTable	The table containing the foreign key.
foreignColumn	The column containing the foreign key.
primaryTable	The table being referenced by the foreign key.
primaryColumn	The column being referenced by the foreign key.

23.11. ODBC utilities

The `tdbc::odbc` package provides some utility commands related to interacting with the ODBC manager. Some of these depend on the system ODBC manager's support of the ODBC Installer API.

The `tdbc::odbc::drivers` command enumerates the installed ODBC drivers on the system.

```
% package require tdbc::odbc
→ 1.0.4
% print_dict [tdbc::odbc::drivers]
→ SQL Server = APILevel=2 ConnectFunctions=YYY CPTimeout=60 DriverODBCVer=03.50 FileUsage=0
    ↳ SQLLevel=1 UsageCount=1
```

The `tdbc::odbc::datasources` command enumerates the configured ODBC data sources on the system. The command accepts the `-user` and `-system` options to limit the returned list to those configured for the current user and system respectively.

```
% print_dict [tdbc::odbc::datasources]
→ Excel Files              = Microsoft Excel Driver (*.xls, *.xlsx, *.xlsm, *.xlsb)
  MS Access Database       = Microsoft Access Driver (*.mdb, *.accdb)
  Visio Database Samples   = Microsoft Access Driver (*.mdb, *.accdb)
% print_dict [tdbc::odbc::datasources -user]
→ Excel Files              = Microsoft Excel Driver (*.xls, *.xlsx, *.xlsm, *.xlsb)
  MS Access Database       = Microsoft Access Driver (*.mdb, *.accdb)
  Visio Database Samples   = Microsoft Access Driver (*.mdb, *.accdb)
```

Finally, the ODBC driver provides an ensemble command, `tdbc::odbc::datasource` for management of ODBC data sources. It takes the one of the forms

```
tdbc::odbc::datasource SUBCMD DRIVER ?KEYWORD=VALUE?
```

Here *DRIVER* identifies the ODBC driver being targeted. The possible values for *SUBCMD* are shown in Table 23.7.

Table 23.7. tdbc::odbc::datasource commands

Command	Description
add	Adds a new user data source.
add_system	Adds a new system data source.
configure	Configures a user data source.
configure_system	Configures a system data source.
remove	Removes a user data source.
remove_system	Removes a system data source.

For all the above, the data source that is the target of the command is specified by a DSN entry in the list of keywords supplied to the command. See the reference documentation for examples.

23.12. Chapter summary

Databases comprise an important component of many software applications. Different database implementations provide different driver API's and there exist many Tcl extensions that provide programmatic access to specific databases.

TDBC is a means to access these different implementations through a uniform object-oriented API. In this chapter, we covered how you can generically use TDBC to

- establish connections
- prepare and execute statements
- execute transactions
- retrieve results

and also covered some specifics pertaining to the SQLite, MySQL, PostgreSQL and ODBC drivers.

23.13. References

TIPTDBC
> *Tcl Database Connectivity (TDBC)*, Kevin B. Kenny et al, Tcl Improvement Proposal #350, http://www.tcl.tk/cgi-bin/tct/tip/350

KBKPAPER
> *Tcl Database Connectivity*, Kevin B. Kenny. Tcl Conference Proceedings, 2008. The original paper describing TDBC. Available from http://www.tclcommunityassociation.org/wub/proceedings/Proceedings-2008/proceedings/tdbc/tcl2k8-kenny-withfonts.pdf

TDBCREF
> *TDBC reference pages*, http://www.tcl.tk/man/tcl8.6/TdbcCmd/contents.htm

WWWCONNSTR
> *Connection Strings Reference*, https://www.connectionstrings.com/

Testing and Performance

A very large part of every software development process is (or should be!) ensuring quality — making sure that the software works as expected and at a satisfactory performance level. This chapter describes some of the tools available within Tcl for meeting these goals in Tcl applications.

24.1. Testing

> Quality is not an act, it is a habit.
>
> — Aristotle

If developing software in Tcl, the interactive nature of Tcl streamlines the development of unit tests. As you code procedures and commands, you can, and **must**, interactively call them from a Tcl shell to verify they work correctly with various inputs. On failures, you can fix the code and reload the changes into the shell with the `source` command for further testing. This mode of development should, as the great philosopher stated, become a habit.

These interactive "off-the-cuff" tests can then be easily formalized, expanded and translated to a form suitable for inclusion in an automated suite. A comprehensive test suite is crucial to shipping robust software. The direct benefits are not just a better customer experience and reduced support costs but also freedom to refactor, optimize and enhance the software while maintaining a high level of confidence that working functionality is not broken by the changes.

Even when the software under test is written in a different language, by its very nature Tcl is very well suited as a language for building test automation suites and is widely used for that purpose. Several commercial test automation products as well as the open source DejaGNU[1] framework are based on Tcl.

Here we will only describe the `tcltest` package which comes with the Tcl source distribution and can be used for testing your own packages as well as applications.

24.1.1. The `tcltest` package

We will describe the `tcltest` package first as that is included in the Tcl core and used in the testing of not only Tcl itself but many other packages and extensions. The package is included in many but not all binary distributions. You can check if it is installed by attempting to load it.

```
package require tcltest → 2.4.0
```

If you get an error, your binary distribution probably does not include it. In that case, you will need to download a source distribution and extract the `tcltest` directory into a directory included in your Tcl installation's `auto_path` variable.

A `tcltest`-based test script consists primarily of a series of `tcltest::test` commands.

```
test NAME DESCRIPTION ?OPTION ...?
```

---

[1] http://www.gnu.org/software/dejagnu/

The *NAME* argument is a label given to the test for identification purposes. It is recommended to use names with some structure so as to be able select a subset of tests using glob wildcard matching. The *DESCRIPTION* argument is strictly for human consumption but clear descriptions greatly help in formulating a complete test suite. The rest of the arguments are specified as options and we will describe these as we go through our example.

We will demonstrate the use of the package with a simple example, a script containing some tests for Tcl's incr command. Test scripts using the package are simply Tcl scripts so you can use any Tcl commands, load other packages and so on. By convention, test scripts are given the .test extension so we name our test script incr.test.

The test script should start off by loading the tcltest package. You may also load any other packages required for testing, define supporting procedures and so on. For our simple example, we do not need anything else.

```
package require tcltest
```

A test case is defined with the tcltest::test command. For convenience we import this command to save us some typing.

```
namespace import tcltest::test
```

Let us write a test for the basic use of incr.

```
test incr-1.0 {
    Verify default increment of 1, the new value is returned and stored
} -setup {
    set i 1
} -body {
    list [incr i] $i
} -cleanup {
    unset i
} -result {2 2}
```

The first argument labels the test case and the second describes what we are testing. The -setup option specifies a script that does any required set up before the actual test can be run. In our case, it just initializes a variable to a known value. The -body option specifies the actual test itself. For our test case, it is meant to check the result of the command as well the new value of the variable. Many times the -setup script is combined with the -body script but we do not recommend this as it obfuscates exactly what is being tested. The -cleanup option specifies a script to run to remove any side-effects of the test case. Finally, the -result option specifies the value that is expected as the result of executing the script provided as the -body argument.

The options may be specified in any oder and not all need be specified though obviously it does not make sense to have a test case without a -body option. The -result option will default to the empty string. In test cases as simple as the one above you will generally see neither -setup nor -cleanup specified.

It is important for a test suite not only to verify correct behaviour on valid input, but to also verify that the command handles bad input correctly. So we will now define a test case to ensure invalid input is correctly rejected by the command.

```
test incr-err-1.0 {
    Verify non-integer variable generate an error.
} -setup {
    set i notaninteger
} -body {
    incr i
} -cleanup {
    unset i
} -returnCodes error -result "expected integer.*notaninteger" -match regexp
```

There are two additional options we see here. The `-returnCodes` option specifies the expected return code from evaluation of the -body script. By default, as in our previous test case, it is expected to be ok indicating a normal script completion. In this case, we expect an error exception and accordingly specify that as the `-returnCodes` value. The `-result` option now holds the expected error message. Since error messages may change slightly between builds, we do not want to specify an exact message value. We therefore use the `-match` option to indicate that the `-result` option value should be treated as a regular expression to be matched with the -body script result. You may also specify glob matching or even define your own matching criteria.

Having compiled a comprehensive set of tests, we end our test script with a call to clean up.

```
::tcltest::cleanupTests
```

The `tcltest::cleanupTests` command does two things. It cleans up any temporary files and other changes made by the test harness. It also prints out summary statistics from the test run.

Here is our entire test script.

```
# incr.test
package require tcltest

namespace import tcltest::test

test incr-1.0 {
    Verify default increment of 1, the new value is returned and stored
} -setup {
    set i 1
} -body {
    list [incr i] $i
} -cleanup {
    unset i
} -result {2 2}

test incr-err-1.0 {
    Verify non-integer variable generate an error.
} -setup {
    set i notaninteger
} -body {
    incr i
} -cleanup {
    unset i
} -returnCodes error -result "expected integer.*notaninteger" -match regexp

::tcltest::cleanupTests
```

Let us try running this test script from the Windows or Unix shell.

```
C:\temp>tclsh incr.test
incr.test:      Total    2      Passed  2      Skipped 0      Failed  0
```

In case of any failures, `tcltest` will print details about the failing test cases.

We can also choose to run a subset of the tests by matching test labels and ask for more verbose output.

```
C:\temp>tclsh incr.test -match *err* -verbose p
++++ incr-err-1.0 PASSED
incr.test:      Total    2      Passed  1      Skipped 1      Failed  0
```

Notice only the error test case was run and the other one skipped. There is also a `-skip` option that is complementary to `-match` and specifies tests that should be skipped.

There are a number of other features that we will not describe here. These include, among others

- ability to set up test constraints so that tests are only run if certain conditions are satisfied, for example limiting certain tests to specific platforms
- checking expected output on standard channels
- utility commands for creation and clean up of temporary files and directories.

> Hai Vu's blog[2] contains series of posts related to both basic and advanced use of the `tcltest` package. You may find it a more accessible tutorial for the `tcltest` package than the reference documentation.

24.1.2. Testing interactive applications

Testing of interactive applications poses some unique challenges in terms of simulating user actions and verifying the output which may be in the form of terminal output, a web page or a native user interface. The `tcltest` package is not suitable for this purpose so we list some packages below that can be used to supplement `tcltest` with the required functionality. Note that these written in Tcl, but with the exception of `TkTest`, are not limited to only testing Tcl applications.

The Expect extension

The Expect[3] extension, is widely used not just for testing but also for automation of systems administration tasks. It is primarily used on Unix systems; there is a Windows version also available but has some stability issues.

Expect is targeted toward applications that interact with the terminal. It derives its power through its ability through Tcl commands for managing pseudo-terminals, emulating user keystrokes, capturing program output and responding accordingly. It is not even limited to the local system as it can be used to drive telnet, ssh or similar terminal based network applications. This makes it suitable even for testing embedded devices, networking equipment etc. as long as they expose a telnet or serial line type of interface.

Expect itself does not provide any test framework and its use for testing interactive applications is through integration with `tcltest` or other frameworks like Caius discussed below.

The Caius test framework

The Caius[4] framework is an alternative to `tcltest` where Tests are written using Incr Tcl, an object-oriented extension of Tcl.

It has three distinguishing features:

- Integration with Expect for testing interactive applications
- Ability to interface to the Selenium Webdriver API for testing Web pages
- Support for continuous integration tools like Jenkins and report generation

> Although `tcltest` and Caius are both test frameworks, it is possible to run them in integrated fashion where one drives test cases written using the other.

The TkTest package

The TkTest[5] package is meant for testing GUI applications written in Tcl with the Tk extension. The package works by recording events and application state while you manually use the application. The recorded events can then be replayed and compared to the saved application state snapshots for regression testing purposes.

---

[2] https://wuhrr.wordpress.com/category/programming/tcl-programming/
[3] http://expect.sf.net
[4] http://caiusproject.com
[5] http://www.cwflynt.com/tktest/

24.1.3. Source code checkers

As a dynamic interpreted language, Tcl does not have a "front-end" compiler. A procedure body for example is compiled the first time it is invoked. Thus syntactic errors may not be detected until runtime. A couple of tools are available to help detect such errors through source code analysis. We will not describe them here but only mention them for your reference.

Nagelfar[6] performs syntactic analysis on Tcl source code to detect not only syntactic errors but some common programming errors such as uninitialized variables. It can also be extended with plugins for additional checks, call analysis etc.

Frink[7] is primarily a tool for formatting and pretty-printing Tcl but also includes some features for syntax checking and programming errors.

24.2. Improving performance

> If you optimize everything, you will always be unhappy.
>
> — Donald Knuth

Given that, improving performance involves multiple steps:

- profiling the application to determine which parts of the application are the "hotspots" that need attention
- measuring execution time for implementations of alternative algorithms for these hotspots
- **if needed**, examining the code for possible "microoptimizations" that can nevertheless significantly speed up the program under certain circumstances.

The next few sections go into these topics.

24.2.1. Profiling scripts

Writing a basic profiler for Tcl scripts is easy because of Tcl's malleability and introspective capabilities. For example, procedures can be redefined or wrapped to collect timing information on entry and exit. An alternative method is to use the command and execution tracing facility described in Section 10.6. You will find several such profilers referenced in Tcler's Wiki[8]. Here we will describe the profiler package included in Tcllib[9].

Note however that although writing the profiler is easy, interpreting the data needs to be done with care for many reasons. The statistics relating to call counts are generally accurate but timing information is less so, particularly for profilers measuring at the source line level. The profiler itself has significant impact on speed of execution. It is best to not rely on profiler output for fine grained measurements and use the output for **relative** comparison of where time is being spent.

When profiling an application, the profiler package must be loaded and the profiler::init command called **before** any of the procedures to be profiled are defined.

```
package require profiler → 0.3
profiler::init              → (empty)
```

Let us now write some sample code that we will use for demonstration purposes. The procedures do not do anything but pretend they are doing useful work by napping.

```
proc fiddle {} { after 10 ; twiddle }
proc twiddle {} { after 20 }
proc diddle {} { after 30 }
```

---

[6] http://nagelfar.sourceforge.net/
[7] http://catless.ncl.ac.uk/Programs/Frink/
[8] http://wiki.tcl.tk
[9] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

```
proc main {} {
    for {set i 0} {$i < 10} {incr i} {
        fiddle
        diddle
    }
}
```

Now we call main and then print the collected information.

```
% main
% profiler::print ::fiddle
→ Profiling information for ::fiddle
  ==============================================================
              Total calls:  10
      Caller distribution:
    ::main:  10
             Compile time:  42296
            Total runtime:  463968
          Average runtime:  46396
            Runtime StDev:  1442
           Runtime cov(%):  3.1
   Total descendant time:  312627
 Average descendant time:  31262
 Descendants:
    ::twiddle:  10
```

We have only printed the profiler data for fiddle. If we had not provided an argument to profiler::print it would have printed it for all procedures.

The profiler::print command require the **fully qualified** command name to be provided.

Most of the information is self-explanatory. The one item that needs some explanation is the Compile time line. The first time a procedure is invoked, Tcl compiles its body into bytecode. Thus the first invocation of a procedure is often significantly longer than subsequent invocation. This difference does not show up in our example because our artificial delay swamps any additional compilation time required.

An alternative to profiler::print is the profiler::dump command which returns the same information in a dictionary format. The package offers utility commands to suspend, resume and reset profiling specific procedures or all of them.

24.2.2. Timing scripts

Having pinpointed what parts of the application need to be worked on, we can try different algorithms to improve performance, say using a skip list versus a binary search tree. We then need a means for measuring execution time for each alternative. Tcl provides the time command for this purpose.

```
time SCRIPT ?COUNT?
```

The command evaluates SCRIPT a COUNT number (default once) of times and prints the average execution time for one evaluation. It is important to pick a sufficiently large COUNT value, not just to smooth variations, but because the first execution of a script will often result in the script being compiled to bytecode resulting in an additional cost. We have more to say about this later but first let us just go through an example.

We want to generate a list of the first N natural numbers. We write two procedures that use a for and a while respectively. (These are not two different algorithms, but never mind that.)

```
proc rangefor n {
    for {set i 1} {$i <= $n} {incr i} {
        lappend l $i
    }
    return $l
}

proc rangewhile n {
    set i 0
    while {$i < $n} {
        lappend l [incr i]
    }
    return $l
}
```

We then use time to measure their execution time.

```
time {rangefor 10000} 100   → 5522.35 microseconds per iteration
time {rangewhile 10000} 100 → 5103.37 microseconds per iteration
```

Let us get back to the question of what value to use for COUNT. A value that is too small will be skewed by the compilation time. Too large a value will take longer to measure. The Tcler's Wiki [10] page on the time [11] command suggests a minimum 2-second run with at least 500 iterations and provides a procedure for automating this.

```
proc measure args {
    set it [expr {int(ceil(10000./[lindex [time $args] 0]))}]
    set it [expr {int(ceil(2000000./[lindex [time $args $it] 0]))}]
    while {$it < 500} {set it 500}
    lindex [time $args $it] 0
}
```

We can then measure execution time as follows:

```
measure rangefor 10000 → 5611.71
```

As an aside, our example was for pedagogic purposes demonstrating the use of the time command. In general, it does not make sense to do microoptimization at this level as continued Tcl bytecode compiler enhancements may change the timings of specific commands.

You will sometimes find time being used as an iterative control structure for repeating a loop. For example, we could construct our integer range list with the following procedure.

```
proc rangetime n {
    time {lappend l [incr i]} $n
    return $l
}
measure rangetime 10000
→ 13020.888
```

Here the time is used in rangetime simply to repeat a script evaluation, not to actually measure time. As you can see, it is slower than the others but for interactive use in a shell it is less typing and convenient.

---

[10] http://wiki.tcl.tk
[11] http://wiki.tcl.tk/734

24.2.3. Performance hints

The top three factors that affect application performance are algorithms, algorithms and algorithms. Nevertheless, the speed of low level operations can come into play as well so we provide some hints here that you might keep in mind when writing Tcl.

Directly modify variables

Most Tcl commands accept values as arguments and return results as values. Commands like append, lappend, lset and incr on the other hand operate on variables. In cases where the result of a command is assigned back to the variable, these should be preferred. For example, prefer append to string interpolation

```
append var "foo"
set var "${var}foo"
```

or incr to expr

```
incr x $y
set x [expr {$x+$y}]
```

Similarly, wherever possible lappend and lset should be preferred to other list commands like linsert, lreplace. A simple timing test illustrates the difference:

```
set l [lrepeat 10000 X]; llength $l   → 10000
time {set l [lreplace $l 0 0 Y]} 100  → 262.42 microseconds per iteration
time {lset l 0 X} 100                 → 0.96 microseconds per iteration
```

As you can see, the difference can be very large although this is an extreme case.

The reason for the difference in performance is Tcl's reference counting mechanism described in Section 10.10.1.2. In the case of lreplace for example, a copy of the list storage has to be made for modification since it is has multiple references and therefore cannot be modified. In the case of lset, since we are modifying the variable itself, the associated value can be modified in place saving on memory and copying costs.

Explicitly drop references

Consider deleting the last element of a list stored in a variable. There is no command that operates directly on a variable so we are forced to use lreplace. Again this has the potential to be slow with large lists. We can time how long it takes us to drop the last element of a list.

```
set l [lrepeat 10000 X]; llength $l       → 10000
time {set l [lreplace $l end end]} 1000 → 84.186 microseconds per iteration
```

Here is an alternative implementation that is much faster. The explanation follows.

```
set l [lrepeat 10000 X]; llength $l                    → 10000
time {set l [lreplace $l[set l ""] end end]} 1000 → 0.925 microseconds per iteration
```

As you can see, that is again a very large difference. Although, it may not be obvious, the reason for the speed-up is the same as we described for modification of variables. When Tcl executes the statement

```
lreplace $l end end
```

the lreplace command has to return a new list contains the same elements as that stored in l except for the last element. It cannot modify the list directly because the variable l is still referencing it and the command semantics do not allow the variable to be modified. Thus, the cost of an additional memory allocation and copying of elements is incurred. Our second implementation fixes this.

```
lreplace $l[set l ""] end end
```

Now when Tcl parses the command arguments for `lreplace`, the first argument is still $l since the appended empty string does not change it (in fact the compiler will optimize it away). But now, it assigns an empty string to l which means when `lreplace` runs the variable l no longer points to the original list storage. Because the list storage is not referenced from anywhere else, the command is free to modify it in place by simply decrementing its element count. No memory allocation or copying of elements in needed.

We could thus write a complement to `lappend`, a `lpop` that removes elements from end of a list stored in a variable.

```
proc lpop {lvar} {
    upvar $lvar l
    lreplace $l[set l ""] end end
}
```

Though obviously not universally applicable, this trick is not limited to `lreplace` but can be used with other commands that modify large lists or strings. Its effectiveness depends on the specific operation. For example, if we were deleting the **first** element of the list instead of the **last** above, you would not see much benefit as only the memory allocation would be saved; the elements would still have to be copied to the previous slot.

It is also confusing to programmers unfamiliar with the idiom so selective usage with suitable comments is advised when put into use.

Avoid string generation

In Section 10.10.1.1 we described how Tcl stores data internally in different formats depending on its type. Its string representation is only generated when required. This happens either when some string operation is applied including I/O operations like printing to the console. This string representation incurs an unnecessary cost in memory and speed so should be avoided where possible. Examine the code below

```
% set l [list a b c] ; tcl::unsupported::representation $l
→ value is a list with a refcount of 3, object pointer at 0000000004BE7DC0, internal
    ↳ representation 0000000004B485C0:0000000000000000, no string representation
% if {$l eq ""} {
    puts "List is empty"
}
% tcl::unsupported::representation $l
→ value is a list with a refcount of 2, object pointer at 0000000004BE7DC0, internal
    ↳ representation 0000000004B485C0:0000000000000000, string representation "a b c"
```

The list starts out without a string representation but string operation

```
$l eq ""
```

causes a string representation to be generated. The internal type-specific representation is still a list as shown (which is a good thing) but nevertheless the generation of a string representation can slow down and consume memory unnecessarily in the case of large lists. The correct way to check for an empty list would be

```
if {[llength $l] == 0} ...
```

In short, for logical correctness as well as performance, stick to list operations for lists, dictionary operations for dictionaries etc.

Avoid shimmering

A similar (except worse!) situation is when in addition to a string representation being generated, the internal type-specific representation is also lost. For example,

```
% if {[string length $l] == 0} {
    puts "List is empty"
}
% tcl::unsupported::representation $l
→ value is a string with a refcount of 2, object pointer at 0000000004BE7DC0, internal
    ↳ representation 00000000044B77F0:0000000000000000, string representation "a b c"
```

Here even list internal representation is lost when the string length command is called. Any further list operations will then require the list representation to be recreated — a double whammy.

Again, most such cases are logical inconsistencies as much as performance issues.

Brace your expressions

Any expressions passed to commands like expr, if etc. should always be enclosed in braces. This important for reasons other than performance as we discussed in Section 7.2.2 but here we will only focus on the latter. Braced expressions are significantly faster to evaluate. This is true even for the simplest expressions.

```
set a 1                  → 1
time {expr $a} 10000     → 0.873 microseconds per iteration
time {expr {$a}} 10000   → 0.5083 microseconds per iteration
```

The reason for this is that braced expressions can be compiled by expr internally because the content inside the braces are not subject to substitution at the Tcl level. This is not true of unbraced expressions.

Use tcl::mathop commands for numerical lists

Do not forget the presence of the mathematical operator commands in tcl::mathop. Adding or multiplying a large list of numbers is much faster with these commands than with a loop using expr or incr.

```
set l {1 2 3}         → 1 2 3
tcl::mathop::+ {*}$l → 6
```

Use procedures instead of scripts

Procedure bodies are always compiled before execution unlike scripts. So for example when dynamically constructing code that will be called frequently, you are often better off constructing a procedure and calling that rather than constructing a script and running it with eval.

Use try instead of eval

Evaluating a script with try is faster than with eval (though still not as fast as a procedure body). This is true even for the single argument form of eval. Thus you should use try to evaluate scripts even if you have no need for the exception handler or finalization clauses. This is actually just a historical artifact and this difference may go away in a future release.

In general, dynamically created scripts are always slower as the compiler can do fewer optimizations.

Use local variables

Access to local variables is much faster than for any other type of variable. Therefore, cache namespace and global variables in locals in tight loops or other performance sensitive areas.

A special case of this relates to variables accessed in uplevel scripts. These run much faster if the caller precreates the variable. The example below from http://wiki.tcl.tk illustrates this.

```
set l [lrepeat 100000]
proc p l {
    uplevel 1 [list foreach v $l {}]
}
proc q l {
    p $l
}
proc r l {
    set v {}
    p $l
}
```

The procedures q and r are identical except that the latter initializes the variable v used in the uplevel call. This reserves a slot in the local variable table making access to it significantly faster as shown below.

```
% time {q $l} 10
→ 76058.7 microseconds per iteration
% time {r $l} 10
→ 66597.2 microseconds per iteration
```

Use string operations instead of patterns and regular expressions

Commands that operate on plain strings are usually faster than those that work with glob patterns or regular expressions and should be preferred where possible. For example, use string map in lieu of regsub for replacement of constant strings. Note however that some special cases are highly optimized in the regular expression engine so always "measure before cutting".

Minimize I/O calls

Tcl's I/O system is extremely flexible but that comes at some cost in performance. Consequently it makes sense to minimize the number of times the I/O boundary is crossed. One way to do this is to increase the size of the I/O buffer via the -buffersize option to chan configure and read data in larger chunks. At one extreme, unless the file is very large it is faster to read it all in one shot and process it as a string. For example, rather than use gets in a loop, use the idiom

```
set chan [open $path]
foreach line [split [read $chan [file size $path]] \n] {
    ...process line...
}
```

We reiterate that many of the optimizations mentioned above may reduce the clarity of the code and their use should be carefully considered and limited to the really performance sensitive areas based on actual measurements.

24.3. Chapter summary

Testing and performance tuning are often the focus only in the last few stages of software delivery. Tcl's interactive nature however encourages integration of these tasks right from the beginning of the development process. In this chapter we described the tools and packages that Tcl provides for the purpose.

Appendix A. Libraries and Extensions

The programming libraries available for a language are as important as the language itself. Here we list a tiny subset of those available for Tcl — the ones that are in the author's opinion commonly used or otherwise worthy of attention. There are several sites that have a more complete listing; the author's personal favorite is the Great Unified Tcl/Tk Extension Repository or GUTTER [1] [2] which lists and categorises libraries and extensions.

A.1. GUI toolkits

Package	Description
Tk	The most well known Tcl extension is of course the Tk graphical toolkit which is so closely associated with Tcl as to be referred collectively as Tcl/Tk. Tk's cross-platform portability, widget set and ease of use has made it the de facto graphical toolkit of choice not just for Tcl but also for other languages like Python, Ruby and Perl. Tk is generally included in all binary distribution of Tcl and available in source form from the same location [3] as Tcl. Most existing books on Tcl include Tk in their coverage and an excellent online book that includes the latest features is available from http://www.tkdocs.com.
gnocl	This is an alternative GUI toolkit based on GTK+/Gnome and comes with its own extensive set of widgets. This is however not widely used in the Tcl world and primarily meant for Unix systems. Available from http://www.gnocl.org.

A.2. Internet protocols

Package	Description
tls	Implements the SSL/TLS security protocols. Available from https://core.tcl.tk/tcltls/home.
http	Implements the client-side HTTP protocol. Available as part of the core Tcl distribution.
rl_http	An alternative implementation of the client-side HTTP protocol from RubyLane. Available from https://github.com/RubyLane/rl_http.
Tnm	Part of the Scotty network management suite, this package includes support for SNMP, ICMP and DNS protocols. Available from https://github.com/flightaware/scotty.
tclcurl	A wrapper around the well-known Curl [4] multiprotocol client library which supports a very wide range of Internet protocols including HTTP, FTP, Gopher, IMAP, LDAP, POP3, SMTP, Telnet and many others. Available from https://github.com/jdc8/tclcurl.
WS::Server, WS::Client	Server and client implementations of Web Services using Web Services over SOAP. Also supports JSON responses via REST. Available from http://core.tcl.tk/tclws/home.
Tcllib [5]	Contains modules for implementations of several network protocols like FTP, SMTP, NNTP and NTP. In some cases, the server side protocol is also supported.

---

[1] http://core.tcl.tk/jenglish/gutter/
[2] Yes, Tcl'ers often have a warped sense of humor.
[3] https://sourceforge.net/projects/tcl/files/Tcl/
[4] https://curl.haxx.se/
[5] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

A.3. Web servers and frameworks

Package	Description
tclhttpd	A pure Tcl web server with a Tcl based templating system. Available from https://sourceforge.net/projects/tclhttpd.
Rivet	An Apache module for creating web applications. Available from https://tcl.apache.org/rivet.
Naviserver	Web server with dynamic pages in Tcl, and high end features like database connection pooling and multithreading. Available from https://sourceforge.net/projects/naviserver.
Wub	The web server that hosts Tcler's Wiki[6]. Available from https://code.google.com/archive/p/wub/.
Woof!	A Web framework similar to Rails for Ruby. Web server agnostic and can be used with Apache, IIS, Lighttpd etc.
Tclssg	This is a static site generator using Markdown for content and tools for website management. Available from https://github.com/tclssg/tclssg.

A.4. Numeric computing

Package	Description
Tcllib[7]	Tcllib[8] includes modules for several types of numeric computation such as numerical integration, solving ODE's, combinatorics, fourier transforms, linear algebra, statistics, complex numbers, geometrical computations, and more.
mathemaTcl	The mathemaTcl project provides package wrappers for a variety of C and Fortran libraries for numerical computation. Home page at http://chiselapp.com/user/arjenmarkus/repository/mathemaTcl/index.
vectcl	The VecTcl extension is geared towards processing of numerical arrays with support for vectors, matrices and tensors. It also supports complex numbers. The extension is written in C for high performance and also offers a specialized syntax for numerical computation. Available from http://auriocus.github.io/VecTcl.
nap	The Tcl-nap (n-dimensional array processor) is another C based extension that implements efficient commands for processing n-dimensional arrays. It includes support for HDF and netCDF file storage formats. Available from http://tcl-nap.sourceforge.net.
mpexpr	This extension offers the ability to calculate with arbitrary precision. Unlike Tcl's native capability which has support for integers of unlimited size, mpexpr works with floating point numbers as well. Available from https://sourceforge.net/projects/mpexpr/files.

A.5. Database access

We described the generic core package for database access, TDBC, in Chapter 23. There are however also additional extensions that either target non-SQL databases, or are customized for specific database implementations. There are too many to enumerate here and there is no basis for which ones are "important" so we will just point you to their listing[9] in the GUTTER catalog.

---

[6] http://wiki.tcl.tk
[7] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html
[8] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html
[9] http://core.tcl.tk/jenglish/gutter/#cat-database

A.6. XML processing

Package	Description
tdom	The most widely used Tcl package for XML processing is tDOM. This extension is a very fast engine for parsing and generating XML with excellent XPath and XSLT support. You can also use the package to parse HTML. Available from http://core.tcl.tk/tdom.
tclxml	Another option for processing XML comes from the TclXML project. However, this is not under active development although it is stable enough for production use. It's biggest advantage over tDOM is that it has an optional implementation in pure Tcl that does not require any binary extensions. Available from https://tclxml.sourceforge.net.

A.7. Integration with other languages

Package	Description
ffidl	Allows direct calling of functions implemented in shared libraries from Tcl scripts as long as the functions use the C calling convention. Available from https://github.com/prs-de/ffidl.
tcc4tcl	This extension is a full C compiler that can compile and call C code embedded in a Tcl script at runtime. Note however that unlike most Tcl extensions, it is covered under the more restrictive LGPL license. Available from http://chiselapp.com/user/rkeene/repository/tcc4tcl/index
java	The Tcl Blend extension implements the java package which offers the ability to access and execute Java code from Tcl as well embed a Tcl interpreter into a Java application. Available from http://tcljava.sourceforge.net.
garuda	The Garuda extension integrates Tcl with the .Net platform. It allows access to libraries written in any .Net based language, such as C#, VB.Net etc. Available from http://eagle.to.
duktape	Implements bindings to the Duktape Javascript interpreter library. It can be used to run Javascript code from within Tcl. Available from https://github.com/dbohdan/tcl-duktape.
libtclpy	Supports calling Python code from Tcl and vice versa. Available from https://github.com/aidanhs/libtclpy.

A.8. Image processing

Package	Description
Img	The Img package adds support for a large number of image formats. Its primary use is in conjunction with the Tk graphical extension. Available from https://sourceforge.net/projects/tkimg/.
crimp	The CRIMP extension implements a set of commands for manipulation of raster images. Available from http://chiselapp.com/user/andreas_kupries/repository/crimp/home.
tclmagick	A wrapper for the GraphicsMagick and ImageMagick image processing libraries. Available from http://tclmagick.sourceforge.net/.
tclgd	A wrapper for the libGD graphics drawing library with support for JPEG, PNG, GIF and other popular formats. Available from https://flightaware.github.io/tcl.gd/.
pdf4tcl	A pure Tcl package for generating PDF files. Available from http://pdf4tcl.sourceforge.net.
tclhpdf	A wrapper around the Haru PDF library for generating PDF files. Available from http://reddog.s35.xrea.com/wiki/tclhpdf.html.

tclMuPdf	A wrapper for the MuPDF framework for rendering PDF or extracting data from a PDF file. See http://wiki.tcl.tk/48296.

A.9. Platform-specific extensions

A.9.1. Windows extensions

Package	Description
registry	Provides commands for reading and writing the Windows registry. The package is part of the core Tcl distribution.
dde	Implements the DDE protocol used for interprocess communication. The protocol itself is however obsolete and should not be used in new applications. This package is also part of the core Tcl distribution,
twapi	The Tcl Windows API package implements a large portion of the Windows API. It allows Windows services, as well as COM clients and servers, to be written as pure Tcl scripts. It provides access to the Windows event log, WMI, administrative API's, crypto services, systems services, desktop integration and more. Available from https://twapi.sf.net.
cawt	Includes modules for integrating with Microsoft Office applications and file formats. Available from http://www.posoft.de/html/extCawt.html.

A.9.2. Unix extensions

Package	Description
expect	The Expectextension, which we briefly mentioned in Chapter 24, is one of the most popular Tcl extensions. It is most commonly used in Unix environments for automation of tasks that require user interaction. Available from http://expect.sf.net.
tuapi	The Tcl Unix API package wraps several system calls. Currently this is only available on Linux. Available from https://chiselapp.com/user/rkeene/repository/tuapi/index.
tclx	The Extended Tcl (TclX) package also provides additional Posix interfaces to files, system services, signals etc. Available from https://sourceforge.net/projects/tclx.

A.9.3. Android extensions

In the case of the Android platform, the AndroWish [10] distribution includes a full set of built-in Android-specific commands, borg, sdltk, rfcomm, usbserial for interacting with the Android operating system.

---

[10] http://www.androwish.org

Appendix B. Utility scripts

We use some simple utility scripts in this book for purposes of pretty-printing etc. These are listed here. Some of these require the `fileutil` package from Tcllib [1] to be loaded.

```
package require fileutil
```

The `print_args` utility simply prints the arguments passed to it, separated by commas.

```
proc print_args {args} {
    puts "Args: [join $args {, }]"
}
```

The `print_list` utility prints each element of a list on a new line. The `print_sorted` utility is similar but prints them in sorted order.

```
proc print_list {l} {
    puts [join $l \n]
}

proc print_sorted {l} {
    print_list [lsort -dictionary $l]
}
```

The `print_dict` utility, transcribed from the Tcllib [2] debug module prints a formatted dictionary.

```
proc print_dict {dict args} {
    if {[llength $args] == 0} {
        set names [lsort -dict [dict keys $dict]]
    } else {
        set names {}
        foreach pattern $args {

            lappend names {*}[lsort -dict [dict keys $dict $pattern]]
        }
    }
    set maxl 0
    foreach name $names {
        if {[string length $name] > $maxl} {
            set maxl [string length $name]
        }
    }
    set maxl [expr {$maxl + 2}]
    set lines {}
    foreach name $names {
        set nameString [format %s $name]
        lappend lines [format "%-*s = %s"  $maxl $nameString  [dict get $dict $name]]
    }
    puts [join $lines \n]
}
```

The `print_array` prints the contents of an array or a subset thereof.

---

[1] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html
[2] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

```
proc print_array {args} {
    uplevel 1 parray $args
}
```

The print_file utility dumps the contents of a file while write_file writes specified contents to a file.

```
proc print_file {path} {
    fileutil::cat $path
}
```

```
proc write_file {path content} {
    fileutil::writeFile $path $content
}
```

The wait utility enters the event loop for the specified amount of time.

```
proc wait {ms} {
    after $ms [list set ::_wait_flag 1]
    vwait ::_wait_flag
}
```

The lambda command is syntactic sugar for defining an anonymous procedure.

```
proc lambda {params body args} {
    return [list ::apply [list $params $body] {*}$args]
}
```

The bin2hex command pretty prints binary data in hex.

```
proc bin2hex {args} {
    regexp -inline -all .. [binary encode hex [join $args ""]]
}
```