Tcl/Tk

Tcl/Tk
A Developer's Guide
Third Edition

Clif Flynt

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO
Morgan Kaufmann Publishers is an imprint of Elsevier

MORGAN KAUFMANN PUBLISHERS

Acquiring Editor: Todd Green
Development Editor: Robyn Day
Project Manager: A. B. McGee
Designer: Eric DeCicco

Morgan Kaufmann is an imprint of Elsevier
225 Wyman Street, Waltham, MA 02451, USA

Notices
Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods or professional practices, may become necessary. Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information or methods described herein. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent of the law, neither the Publisher nor the authors, contributors, or editors, assume any liability for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions, or ideas contained in the material herein.

Library of Congress Cataloging-in-Publication Data
Flynt, Clif.
Tcl/Tk : a developer's guide / Clif Flynt. – 3rd ed.
p. cm. – (The Morgan Kaufmann series in software engineering and programming)
ISBN 978-0-12-384717-1 (pbk.)
1. Tcl (Computer program language) 2. Tk toolkit. I. Title.
QA76.73.T44F55 2012
005.2'762–dc23

2011038927

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library.

For information on all Morgan Kaufmann publications,
visit our Web site at *www.mkp.com* or *www.elsevierdirect.com*

Printed in the United States of America
11 12 13 14 15 5 4 3 2 1

Foreword

It is a pleasure to introduce you to the first revision of Clif's opus on the popular Tool Command Language (Tcl). Tcl has become the Swiss Army knife for the busy programmer—you keep it within easy reach at all times simply because it has lots of uses in many different environments.

Clif's book is both a pleasure to read and a useful reference. Clif is a strong advocate for Tcl, explaining the many different scenarios where Tcl shines and the myriad ways the same facilities can be used over and over again.

In addition to the breadth that Clif brings to the subject, he also knows the details—and, to his credit, he doesn't overwhelm you with them. Instead, each chapter starts with the basics and builds to the fun stuff. There are even self-help questions at the end of each chapter to help you review the lessons Clif has taught.

Whether you're using Clif's book as an introduction to Tcl or as a comprehensive reference, I'm sure you'll find it a necessary addition to your professional toolkit.

Marshall T. Rose
Principal, Dover Beach Consulting, Inc.

Preface

TCL/TK: GUI PROGRAMMING IN A GOOEY WORLD

Alan Watts, the Episcopal priest who popularized Buddhist thought in 1960s California, once wrote that philosophical thinkers can be divided into two basic types, which he called "prickly" and "gooey." The prickly people, by his definition, "are tough-minded, rigorous, and precise, and like to stress differences and divisions between things. They prefer particles to waves, and discontinuity to continuity. The gooey people are tender-minded romanticists who love wide generalizations and grand syntheses. . . . Waves suit them much better than particles as the ultimate constituents of matter, and discontinuities jar their teeth like a compressed-air drill."[1]

Watts chose the terms *prickly* and *gooey* carefully, seeking to avoid making one term positive and the other pejorative, and he offered delightful caricatures of the strengths and weaknesses of each type of person. In his view, a constant tension between the two perspectives is healthy in promoting progress toward a more complete understanding of reality.

Western science has long made a similar distinction between two complementary approaches, theoretical and empirical. Prickly theoreticians seek to understand the world through the abstractions of thought, whereas gooey empiricists return ceaselessly to the real world for the ever-more-refined data that methodical experimentation can yield. The complementarity of the two approaches is widely recognized by scientists themselves. Although most scientists would place themselves squarely in either the theoretical or empirical camp, very few would go so far as to argue that the other approach is without merit. A constant dialectic between empiricism and theory is generally seen to promote the integrity and health of scientific inquiry.

More recently, the advent of computer programming has brought with it a new round in the prickly–gooey dialectic. By temperament, there seem to be two types of programmers, which I will call the planners and the doers. Although good programmers will of course plan their software before building it (and will actually build it after planning it), some programmers (the prickly planners) see planning and designing as the primary activity, after which system building is relatively automatic. Others (the gooey doers) perceive system design as a necessary and important preparatory stage before the real work of system building.

Both planners and doers can be excellent programmers, and the best of them excel at both design and execution. However, they may be categorized as planners or doers by their basic worldview, or primary orientation. They show different patterns of strengths and weaknesses, and they can with skilled management play complementary roles in large project teams. Still, certain tasks cry out for one type of programmer or the other. The design of software systems to control nuclear weapons, for example, is a task that demands exhaustive planning because the risk of unanticipated errors is too high. The design of a throwaway program that needs to be run just once to calculate the answer to a single question, on the other hand, should probably be built by the fastest methodology possible, robustness

[1] Watts, Alan, *The Book: On the Taboo Against Knowing Who You Are*, NY: Vintage Books, 1966, 1989.

be damned. Between these extremes, of course, lie nearly all real-world programming projects, which is why both approaches are useful. They are not, however, always equally well appreciated.

The modern managerial mentality thrives on careful organization, planning, and documentation. It should be no surprise that most managers prefer the planners to the doers; if they were programmers, they would probably be planners themselves. Since they are not programmers, they may easily fail to recognize the value of the skills doers bring to the table. Programmers, however, are less likely to make this mistake. In the meager (but growing) literature by and for serious programmers, the role of doers is more generally recognized. Indeed, planning-oriented programmers often still perceive themselves as the underdogs, and programmer humor shows a deeply rooted skepticism regarding overplanned projects.

Historically, the dialectic between planners and doers has been a driving force in the evolution of programming languages. As the planners moved from FORTRAN to Pascal, C, C++, and Java, the doers have moved from COBOL to Lisp, Basic, Perl, and Tcl. Each new generation of language reflected the innovations of both planners and doers in the previous generation, but each made most of its own innovations in a single realm, either planning or doing.

Historically, it has been surprisingly difficult to make money by inventing even the most wildly successful programming languages. Success is typically highly indirect. For example, being the birthplace of C is certainly a claim Bell Labs is proud to make, but it is difficult to argue that the language was spectacularly meaningful for AT&T (or Lucent's) bottom line. ParcPlace has perhaps come the closest to a genuine language-invention success story, but it is clearly no Microsoft. Imagine, then, the dilemma in which Sun Microsystems found itself, circa 1994, when its research labs were polishing up two new languages, one prickly and one gooey, and each a plausible contender for leadership in the next generation of programming languages. Could any corporation really afford two such high-status, low-profit "success" stories?

Sun's decision to promote Java more vigorously than Tcl is difficult to argue with. Because most decision makers are managers, and most managers are planners, Java has always been an easier product to sell. Sun's continued and parallel support for Tcl was correct, even noble, and probably wiser than one could expect from the average large corporation. Still, in the face of the Java juggernaut, the promotion of Tcl has been left, for the most part, to the gooey doers among us. Pragmatic programmers with dirt on their hands, many of us became Tcl advocates for the most practical of reasons: it helped us get our jobs done, and quickly.

One such gooey doer is the author, a veteran of countless programming languages who has fallen in love with Tcl for all the neat things it lets him do. Java advocates will often praise that language's abstractions, such as its virtual machine model. Tcl advocates such as the author prefer to grab on to the nuts and bolts and show you how quickly you can get real work done. That is what this book is intended to do, walking you all the way from the simplest basics to sophisticated and useful examples. If you are in a "doer" state of mind, you have picked up the right language and the right book. It can help you get a lot of jobs done.

Introduction

Thanks for purchasing this copy of *Tcl/Tk: A Developer's Guide*. (You did buy this, didn't you?) This book will help you learn Tcl/Tk and show you how to use these tools to increase your programming productivity.

By the time you've finished reading this book, you'll have a good idea of what the strengths and weaknesses of Tcl/Tk are, how you can use the Tcl/Tk libraries and interpreters, and what constitutes an effective strategy for using Tcl/Tk.

This book is aimed at both computer professionals (from novice to expert level) and students. If you are experienced with computer programming languages, you will be able to skim through the first few chapters getting a feel for how Tcl/Tk works, and then go on to the chapters on techniques and extensions. If you are less experienced, you should read the first chapters fairly carefully to be sure you understand the Tcl/Tk syntax. Tcl has some features that may not be obvious.

If your primary goal is to learn the Tcl/Tk basics as quickly as possible, visit one of the websites listed below for a tutorial. There are a large number of Tcl/Tk tutorials available in a large number of languages.

Every language has some simple tricks and techniques that are specific to that language. Tcl/Tk is no exception. I'll discuss how to best accomplish certain tasks with Tcl/Tk: how to use the list and associative array data structures effectively, how to build modular code using Tcl's software engineering features, and what kinds of programming constructs will help you get your project from prototype to product with the minimal amount of rewriting.

Finally, there are plenty of code snippets, examples, and bits of packages to help you see how Tcl/Tk can be used. These examples are chosen to provide you with some boilerplate procedures that you can add to your programs right now, to help you get from scribbles on paper to a working program.

This should give you a good idea of what's coming up. Feel free to skip around the book. It's written to be an overview and reference, as well as a tutorial.

WHERE TO GET MORE INFORMATION

Much as I would like to cover all the aspects of Tcl/Tk in this book, it isn't possible. Had I but world enough and time, I still wouldn't have enough of either. The Tcl/Tk community and the Tcl/Tk tools are growing too quickly for a book to do more than cover the central core of the language.

You can learn more about Tcl/Tk from these web references:

- `http://www.tcl.tk/`
  - Primary tcl.tk website
  - The definitive site for Tcl/Tk information.
- `http://wiki.tcl.tk/`
  - The Tcler's Wiki.
  - A collection of brief articles and discussions of Tcl features, tricks, tips, and examples.
- `http://www.tclcommunityassociation.org`
  - Tcl Community Association homepage

  - Promotes the use and development of Tcl/Tk. They run the annual Tcl/Tk Conference in the US, administer the Tcl contribution to Google Summer of Code, host the wiki and source websites and more.
- *www.purl.org/NET/Tcl-FAQ/*
  - Tcl FAQ Launch page.
  - URLs for several Tcl/Tk FAQs and *comp.lang.tcl*
- *http://core.tcl.tk/*
  - Tcl/Tk Fossil repository
  - The official repository for Tcl/Tk distributions.
- *http://sourceforge.net/projects/tcl/*
  - Tcl/Tk Archives
  - The mirror repository for Tcl/Tk distributions.
- *http://www.activestate.html*
  - Active State.
  - Tcl/Tk maintenance, ActiveTcl, development tools, and more.
- *http://www.tcl-tk.de/*
  - Tcl/Tk information in German
- *http://www.geocities.co.jp/SiliconValley/4137/*
  - Tcl Scripting Laboratory
  - Tcl/Tk information in Japanese
- *http://www.cyut.edu.tw/~ckhung/olbook/*
  - Chao-Kuei Hung's pages
  - Tcl/Tk information in Chinese
- *http://www.noucorp.com*
  - Noumena Corporation Home Page
  - Errata and extensions for this book, updated TclTutor

For additional links and resources please visit the companion site at: *http://www.elsevierdirect.com/9780123847171*.

Acknowledgments

You may like to think of the author sitting alone, creating a book in a few weeks. I'm afraid that this idea has as much to do with reality as the Easter Bunny and Santa Claus. The initial revision of this book took the better part of a year and a lot of help from my friends (some of whom I'd never heard of when I started the project). The second and now third editions have each taken over a year to write and introduced me to more new friends.

The first acknowledgment goes to my wife, Carol. For more than a year she's been: putting up with "I can't do any work around the house. I've got to finish this chapter," correcting the grammar on my rough drafts before I send them out for technical review, editing galleys, making sure that I ate occasionally, and keeping life (almost) sane around here. The book would not have happened without her help.

I also want to thank Ken Morton and his editorial assistants Samantha Libby and Jenn Vilaga. If they hadn't been willing to take a chance on an unproven author's ability to finish the first edition of this book, this edition would never exist.

Alex Safonov provided the technical review for the first edition. Larry Virden, and Leam Hall kept me honest in the second edition. Arnulf Wiedemann, Massimo Manghi, Richard Owlett, Brian Stretch, Virginia Gielincki, and Robert Clay provided much assistance with the third edition. Their comments greatly improved the book, and earned more thanks than I can give them.

My heartfelt and sincere thanks to the folks who read and commented on the early drafts. These folks pulled duty way above the call of friendship by reading the truly wretched first drafts. Their comments were all invaluable. My thanks to: Margaret Bumby, Clark Wierda, Terry Gliedt, Elizabeth Lehmann, Nick DiMasi, Hermann Boeken, Greg Martin, John Driestadt, Daniel Glasser, Theresa Arzadon, Lee Cave-Berry, William Roper, Phillip Johnson, Don Libes, Jeffrey Hobbs, John Stump, Laurent Demailly, Mark Diekhans, David Dougherty, Tod More, Hattie Schroeder, Michael Doyle, Sarah Daniels, Ray Johnson, Melissa Hirschl, Jan Nijtmans, Forest Rouse, Brion Sarachan, Lindsay F. Marshall, Greg Cronau and Steve Simmons.

And, finally, my thanks to John Ousterhout and his teams at UC Berkeley, Sun, Scriptics and Ajuba, and now to the Tcl Core Team and the host of maintainers. Programming in Tcl has been more productive and given me more pleasure than any tool I've used since I got my first chance to play Lunar Lander on an IBM 1130.

CHAPTER

Tcl/Tk Features

1

Your first question is likely to be "What features will Tcl/Tk offer me that other languages won't?" This chapter gives you an overview of the Tcl/Tk features. It covers the basics of what Tcl/Tk can offer you and what its strengths are compared with several alternatives.

Tcl is a multifaceted language package. You can use Tcl as a command scripting language, as a powerful multi-platform interpreted language, as a rapid prototyping platform, or as the backbone of an application that is primarily written in C or Java. You can use the Tcl and Tk libraries to embed powerful scripting capabilities into a compiled application like C, Java or FORTRAN. Tcl's simple syntax makes single-use scripts (to replace repetitive command typing or graphical user interface (GUI) clicking) quick and easy to write. Tcl's modularization, encapsulation and object oriented features help you develop large (100,000 + lines of code) projects. Tcl's extensibility makes it easy to use Tcl as the base language across a broad range of projects, from machine control to database applications to electronic design applications, network test devices, and more.

Tcl is both free software and a commercially supported package. The core Tcl language is primarily supported by a worldwide group of volunteers. Commercial support can be purchased from ActiveState, Noumena Corporation, Cygnus Solutions, and others.

The central site for Tcl/Tk information is `http://www.tcl.tk`. The source code repository and some binary snapshots are maintained at `http://sourceforge.net/projects/tcl/`. Tcl/Tk runtime packages are included with the Linux and FreeBSD packages, and with commercial UNIX distributions such as Solaris, HPUX and Mac OS X. The current binaries for selected systems (including MS-Windows, Linux, Mac OS X, and Solaris) are available from ActiveState (`http://www.activestate.com`).

One of the strengths of Tcl is the number of special-purpose extensions that have been added to the language. Extensions are commonly written in "C" and add high-performance special purpose features that not all users need to the language. The use of extensions keeps the Tcl base language small, while supporting a wide variety of uses.

The most popular Tcl extension is Tk, which stands for Tool Kit. This extension provides tools for graphics programming, including drawing canvases, buttons, menus, and so on. The Tk extension is considered part of the Tcl core and is included in the source code distributions at Sourceforge and most binary distributions.

Other popular extensions to Tcl include `[incr Tcl]` (which adds support for C++ style object-oriented programming to Tcl) and `expect`, an extension that simplifies controlling other applications and devices and allows many complex tasks to be easily automated.

Tcl/Tk: A Developer's Guide. DOI: 10.1016/B978-0-12-384717-1.00001-4

With Tcl 8.6 (released in 2010) object-oriented support is part of the core language. The Tcl-OO support provides object-oriented features that mimic smalltalk, Objective-C and C++. Tcl-OO is a powerful programming tool, and can be used to create pure Tcl extensions like `[incr Tcl]` and `XOTcl`. Creating these language extensions in pure Tcl, without any compiled code, assures that the extension will run on any platform that the Tcl interpreter can be built for.

Many extensions including `expect` (for controlling remote systems and routers), `TclX` (with commands for systems administration), Img and crimp (for image processing) are available both as source code and in binary format. Links to these resources are available from the companion website by visiting: *http://www.elsevierdirect.com/9780123847171*.

Dr. John Ousterhout received the 1997 Software System Award from the Association for Computing Machinery (ACM) for inventing Tcl/Tk. This award recognizes the development of a software system that has a lasting influence. The ACM press release says it well.

> *The Tcl scripting language and its companion Tk user interface toolkit have proved to be a powerful combination. Tcl is widely used as the glue that allows developers to create complex systems using preexisting systems as their components. As one of the first embeddable scripting languages, Tcl has revolutionized the creation of customizable and extensible applications. Tk provides a much simpler mechanism to construct graphical user interfaces than traditional programming languages. The overall Tcl/Tk system has had substantial impact through its wide use in the developer community and thousands of successful commercial applications.*

1.1 TCL OVERVIEW

Tcl (pronounced either as the three letters, or as "tickle") stands for Tool Command Language. This name reflects Tcl's strength as a scripting language for gluing other applications together into a new application.

Tcl was developed by Dr. John Ousterhout while he was at the University of California at Berkeley. He and his group developed simulation packages that needed macro languages to control them. After creating a few on-the-fly languages that were tailored to one application and would not work for another, they decided to create an interpreter library they could merge into the other projects. This provided a common parsing package that could be used with each project, and a common base language across the applications.

The original design goal for Tcl was to create a language that could be embedded in other programs and easily extended with new functionality. Tcl was also designed to execute other programs in the manner of a shell scripting language. By placing a small wrapper around the Tcl library, Dr. Ousterhout and his group created `tclsh`, a program that could be used as an interactive shell and as a script interpreter.

Dr. Ousterhout expected that this solid base for building specialized languages around a common set of core commands and syntax would be useful for building specialized tools. However, as programmers created Tcl extensions with support for graphics, database interaction, distributed processing, and so on, they also started writing applications in pure Tcl. In fact, the Tcl language turned out to be powerful enough that many programs can be written using `tclsh` as an interpreter with no extensions.

Today, Tcl is widely used for in-house packages, as an embedded scripting language in commercial products, as a rapid prototyping language, as a framework for regression testing, and for 24/7 mission-critical applications. The robustness of the Tcl interpreter is demonstrated by such mission-critical applications as controlling offshore oil platforms, product Q/A and Q/C operations, and running the NBC broadcasting studios. Many companies are open about their use of Tcl and many more consider Tcl their secret competitive edge.

1.1.1 The Standard Tcl Distribution

Tcl is usually distributed with two interpreters (`tclsh` and `wish`), documentation and support libraries. `Tclsh` (pronounced as *ticklish*) is a text-based interpreter and `wish` is that basic interpreter with Tk graphics commands added.

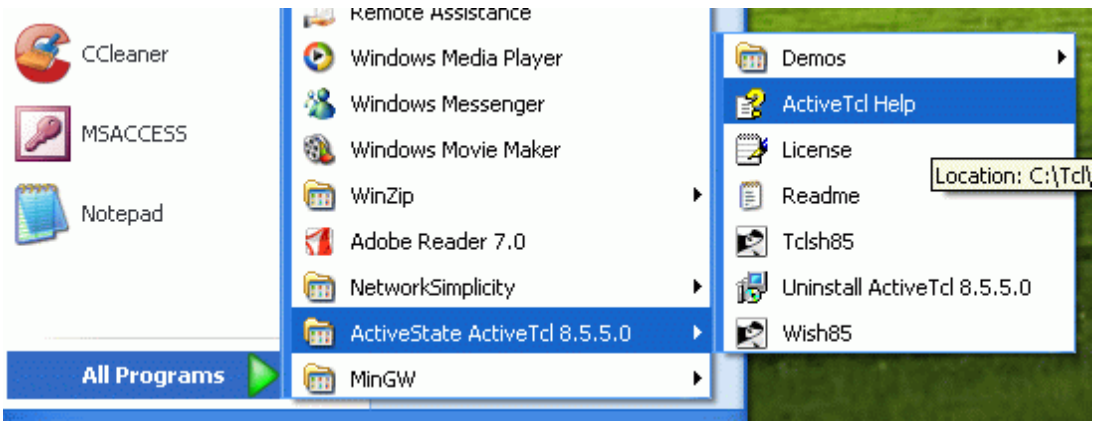
You can use the Tcl interpreter (`tclsh`) as you would use UNIX shell scripts or MS-DOS batch (`.bat`) scripts to control and link other programs or use `wish` to create GUI interfaces to these scripts. The `tclsh` interpreter provides a more powerful environment than the standard UNIX shell or `.bat` file interpreters.

The Tcl documentation facility integrates into the native look and feel of the platform Tcl is installed on.

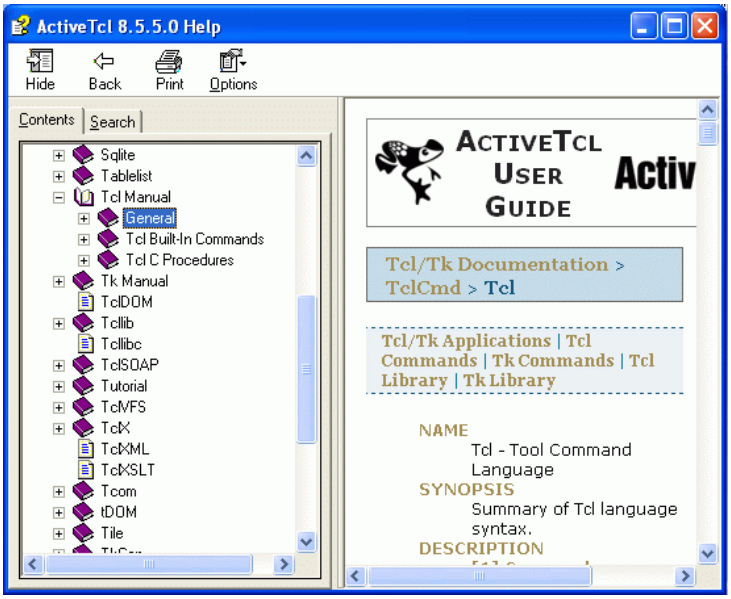
1.1.2 Documentation

On a UNIX/Linux platform, the man pages will be installed under *`installationDirectory/man/mann`*, and can be accessed using the `man` command. You may need to add the path to the installed manual pages to your `MANPATH` environment variable.

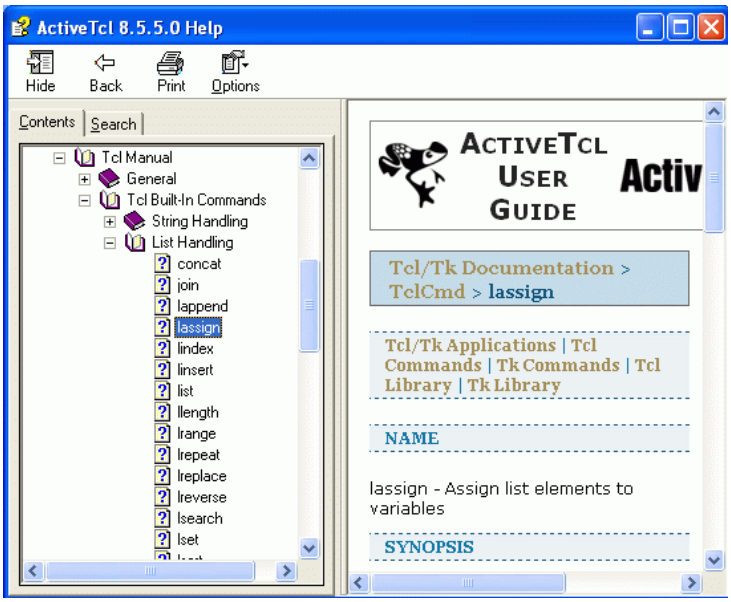
On Microsoft Windows platforms, you can access the Tcl help from the Start menu, shown in the following illustration.



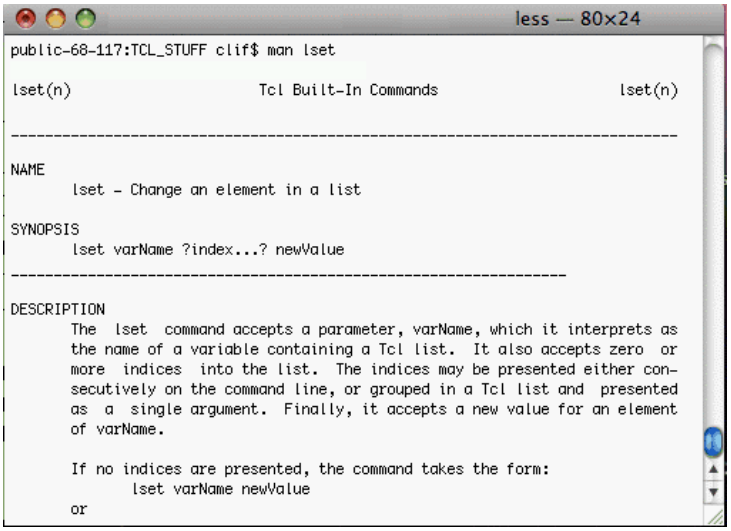
This will open a window for selecting which help you need. The window is shown in the following illustration.



Selecting **Tcl Manual/Tcl Built-In Commands/List Handling** from that menu will open a window like that shown in the following illustration. You can select from this window the page of help you need.

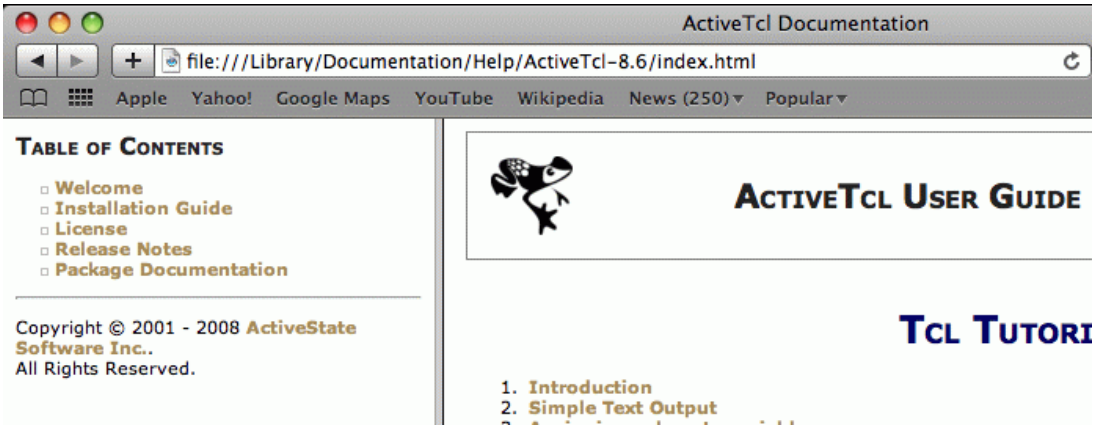


A Macintosh running Mac OS X comes with Tcl/Tk installed. You can access the man pages by opening a terminal window and typing `man` *`commandName`* as shown in the following illustration.



If you install a non-Apple version of the `tclsh` and `wish` interpreters (for instance whatever the latest ActiveState release is), the new man pages will be installed in a location defined by the installation.

The following image shows a Safari view of the ActiveState html man pages for Tcl 8.6.



1.2 TCL AS A GLUE LANGUAGE

A command glue language is used to merge several programs into a single application. UNIX programmers are familiar with the concept of using the output from one program as the input of another via pipes. With a glue language, more complex links between programs become possible. Instead of the linear data flow of a set of piped programs, you can have a tree-like data flow in which several sets of data flow into one application. For example, several programs that report network behavior can be merged with a glue language to create a network activity monitor.

There are good reasons to use simple glue language scripts in your day-to-day computing work.

- A script can glue existing programs into a new entity. It is frequently faster to glue together several small programs to perform a specific task than it is to write a program from scratch.
- A script can repeat a set of actions faster and more reliably than you can type. This is not to disparage your typing skills, but if you have to process 50 files in some consistent manner it will be faster to type a five-line script and have that loop through the file names than to type the same line 50 times.
- You can automate actions that would otherwise take several sets of window and mouse operations. If you have spent much time with GUI programs using Microsoft Windows, Mac OS, or the X Window System, you have probably noticed that there are certain operations you end up doing over and over again, without a hot button you can use to invoke a particular set of button and mouse events. With a scripting language, you can automate a set of actions you perform frequently.
- The script provides a record of commands and can act as procedure documentation.

If you have written scripts using the Bourne shell under UNIX, or `.bat` files under MS-DOS/MS-Windows, you know how painful it can be to make several programs work together. The constructs that make a good interactive user shell don't necessarily make a good scripting language.

Using Tcl, you can invoke other programs, just as you would with shell or `.bat` scripts, and read any output from those programs into the script for further processing. Tcl provides the string processing and math functionality of `awk`, `sed`, and `expr` without needing to execute other programs. It is easier to write Tcl scripts than Bourne shell scripts with `awk` and `sed`, since you have to remember only a single language syntax. Tcl scripts also run more efficiently, since the processing is done within a single executable instead of constantly executing new copies of `awk`, `sed`, and so on.

Note that you can only use a script to control programs that support a non-GUI interface. Many GUI-based programs have a command line interface suitable for scripting. Others may support script languages of their own or have a dialog-based mode. Under MS Windows, you can also interact applications using Tcl's DDE and COM extensions.

You can use a `wish` script as a wrapper around a set of programs originally designed for a text-based user interaction (query/response, or one-at-a-time menu choices) and convert the programs into a modern GUI-based package. This is a very nice way of adding a midlife kicker to an older package by hiding the old-style interactions from users more accustomed to graphical environments.

For example, we used a text-based problem-tracking system at one company. The user interface was a series of questions, such as "What product is the problem being reported against?" and "What revision is this product?" A new `wish` front end had buttons to select the products, text entry fields for the revisions and problem descriptions, and so on. This GUI invoked the old user interface when

the Submit button was pressed and relayed the selected values to that program as the prompts were received.

The GUI interface reduced the time and error count associated with filling out a report by reducing the amount of typing by providing a set of choices for models and revisions. This technique for updating the user interface is faster than rewriting the application and is less prone to introducing new errors, since all original (debugged) code is left untouched.

1.2.1 Tcl Scripts Compared with UNIX Shell Scripts

The following are advantages Tcl provides over UNIX shell scripts.

- *Easier handling of program output.* Parsing program output in a shell script is possible but not always easy. With Tcl, it is simple. Instead of saving the original command line variables, changing the field separator, reading a line of output and using the shell `set` command to parse the line into new command arguments, or invoking `sed` or `awk` to process some text, you can read a line from a program's output just as if it had been entered at a keyboard. Then you can use Tcl's powerful string and list operators to process the input, assign values to variables, perform calculations, and so on.
- *More consistent error handling in Tcl.* It can be difficult to distinguish between an error return and valid output in shell programming. Tcl provides a mechanism that separates the success/failure return of a program from the textual output.
- *Consistent language.* When you write UNIX shell scripts, you end up using copies of `awk` and `sed` to perform processing the shell cannot do. You spend part of your time remembering the arcane `awk` and `sed` command syntax, as well as the shell syntax. Using Tcl, you can use a single language to perform all of the tasks, such as program invocation, string manipulation, and computations. You can expend your effort solving the original problem instead of solving language issues.
- *Speed.* A single self-contained script running within the Tcl interpreter is faster and uses fewer machine resources than a UNIX shell script that frequently needs to spawn other programs.
  This is not to say that a program written in Tcl is faster than one written in the C language. A string-searching program written in Tcl is probably slower than a string-searching program written in C. But a script that needs to find strings may run faster with a string-searching subroutine in Tcl than one written for the UNIX shell that constantly forks copies of another executable. When it is appropriate to invoke a special-purpose program, it is easily done with the Tcl `exec` command.
- *GUI support.* You can use Tk to add a graphical interface to your scripts. Under UNIX, `wish` supports the `Motif` look and feel. Under Windows and Mac, Tk supports the native Windows or Macintosh look and feel.

According to reports in *comp.lang.tcl*, a GUI written with `wish` sometimes runs faster than the same GUI written in C with the `Motif` library. For lightweight GUIs, a Tk GUI is frequently faster than a Java GUI.

1.2.2 Tcl Scripts Compared with MS-DOS `.bat` Files

Tcl has so much more power than `.bat` files that there is actually very little comparison. The power of Tcl/Tk more closely resembles that of Visual Basic. Tcl/Tk is compared to Visual Basic in the next

section, in the context of comparison of general-purpose interpreters. The following are advantages a Tcl script provides over an MS-DOS `.bat` file.

- *Access to the output of programs invoked from script.* The `.bat` file interpreter simply allows you to run programs, not to access a program's output. With Tcl you can start a program, read its output into your script, process that data, send data back to that task, or invoke another task with the modified data.
- *Control structures.* Tcl has more control structures than `.bat` files, including `while` loops, `for` loops, `if/else`, and `switch`.
- *String manipulation.* The `.bat` files do not support any string manipulation commands. Tcl provides a very rich set of string searching and manipulation commands.
- *Math operations.* Tcl provides a full set of arithmetic functions, including arithmetic, trig, and exponential functions and multiple levels of parentheses.
- *Program control.* An MS-DOS `.bat` file has very limited control of other programs invoked by the `.bat` file. Tcl provides a great deal of control. In particular, when a GUI program is invoked from a `.bat` file under Windows 95 or Windows NT, the control may return to the `.bat` file before the program has ended. This is a problem if you were intending to make use of the results of the first program you invoked in the next program. Tcl can prevent the second program from starting until after the first program has completed its processing, or allow multiple programs to be run simultaneously.
- *GUI support.* You can use Tk to make your scripts into GUIs. `Wish` supports a look and feel that matches the standard MS Windows look and feel, and thus scripts you write with `wish` will look just like programs written in Visual C++, Visual Basic, or other Microsoft development environments.

1.3 TCL AS A GENERAL-PURPOSE INTERPRETER

The Tcl/Tk interpreter provides the following advantages over other interpreters.

- *Multi-platform.* The same script can be run under Microsoft Windows 3.1, Microsoft Windows 95/98, Microsoft Windows NT/2000/XP/Vista/System 7, Apple Mac OS and OS/X, and UNIX or Linux. Tcl has also been ported to Digital Equipment's VMS, PC-DOS, and real-time kernels, such as Wind River Systems' VxWorks, and even to platforms such as Palm OS and Win CE.
- *Speed.* Since version 8.0 (released in 1998), the Tcl interpreter performs a runtime compilation of a script into a byte code. Running the compiled code allows a Tcl script to run faster than Visual Basic or Perl.
- *Power.* Tcl supports most of the modern programming constructs.
  - Object-oriented programming with classes, supers, and mixins.
  - Modularization in the form of subroutines, libraries (including version control), and namespaces.
  - Standard program flow constructs: `if`, `while`, `for`, `foreach`, and `switch`.
  - Rich set of variable types: integers, floating point, strings, lists, dicts and associative arrays.
  - Exception handling. The Tcl `catch` and `error` commands provide an easy method of handling error conditions.
  - Support for traditional program flow and event-driven programming.

- *Rich I/O implementation.* Tcl can perform I/O operations with files, devices, keyboard/screen, other programs, or sockets.
- *Powerful string manipulation commands.* Tcl includes commands for searching and replacing strings or single characters, extracting and replacing portions of strings, and converting strings into lists, as well as commands that implement regular expressions for searching and replacing text.
- *Extensibility.* Tcl extensions add a few new commands to the Tcl/Tk language to extend the interpreter into a new application domain. This allows you to build on previous development and greatly reduces startup time when working on a new problem, in that you only need to learn a couple of new commands, instead of an entire new language.

1.3.1 Tcl/Tk Compared to Visual Basic

Of the currently available languages, Visual Basic is probably the closest in functionality to Tcl/Tk. The following are reasons Tcl/Tk is a better choice.

- Tcl/Tk scripts run on multiple platforms. Although the primary market for software may be the MS Windows market, you might also want to have Apple and UNIX markets available to your sales staff.
- Tcl/Tk scripts have support for Internet-style distributed processing, including a secure–safe interpreter that allows an application to run untrusted applications securely.
- Tk provides more and finer control of widgets. `Wish` allows you to bind actions to graphics objects as small as a single character in a text window or as large as an application window.
- Tcl has better support for library modules and version levels. A Tcl script can check for a particular version of its support libraries and not run if they are not available.

You might prefer Visual Basic over Tcl if you are writing a package that will only need to interact with the Microsoft applications. Many Microsoft applications use Visual Basic as their scripting language and there are off-the-shelf components written in Visual Basic you can merge into these applications. Native Tcl supports only the DDE interprocess communication protocol for interacting with other MS Windows applications.

Support for OLE and COM objects can be added to Tcl with the `TOCX` extension, available at *www.cs.cornell.edu/Info/Projects/zeno/tocx/*. Support for SOAP can be added with the Tcl SOAP extension, available at *http://tclsoap.sourceforge.net/*. The `eagle` implementation of Tcl , available at *http://eagle.to*, provides complete interaction with the .NET architecture within a CLR virtual machine.

1.3.2 Tcl/Tk Compared to Perl

Tcl/Tk often competes with Perl as an application scripting language. The following are advantages Tcl/Tk offers over Perl.

- *Simpler syntax.* Tcl code can be more easily maintained than Perl code. The rich set of constructs available to Perl programmers allows some very write-only programming styles. A Perl programmer can write code that strongly resembles UNIX shell language, C code, or `awk` scripts. Tcl supports fewer syntactic methods of accomplishing the same action, making Tcl code more readable.

- *Speed.* The Tcl 8.0 interpreter's byte-compiled code ran as fast or faster than Perl 5 interpreter code. The byte-code engine has continued to be enhanced with a major rework appearing in 8.6.
- *Better GUI support.* Native Perl has no GUI support. A couple of Tk-Perl mergers are available, but Tk is better integrated with Tcl.
- *Internationalization.* Tcl has had fully integrated Unicode support since the 8.1 release (in 1999). Tcl 8.6 includes a new time and date package that understands all time zones.
- *Thread safety.* While Tcl does not support multiple threads, the Tcl interpreter is thread safe, and can be used with multi-threaded applications. Multiple threads can be written in pure Tcl with the `threads` package described in Chapter 16.

1.3.3 Tcl/Tk Compared to Python

The base Python interpreter is probably the closest to Tcl in functionality and use. Both are interpreted scripting languages designed to either glue other applications into a new super-application or to construct new applications from scratch, and both support easy extension with new C library packages.

The Python and Tcl developers actively adopt each other's ideas, leading to a condition in which the best features of each language are in both languages (and if that is not true at this moment, it will be true again soon). Many programmers know both languages, and the use of one over the other is as much personal preference as anything.

The following are the advantages of Tcl.

- *Simpler syntax.* The Tcl language is a bit simpler and easier to learn than Python.
- *Integrated with Tk.* Both Python and Perl have adopted the Tk graphics library as their graphics package. The integration of Tk with Tcl is cleaner than the integration with either Perl or Python. (After all, Tk was designed to work with Tcl.)
- *Choice for object-oriented.* The Tcl interpreter supports dynamic object-oriented programming with the `TclOO` commands, or C++/Java style of object-oriented programming with the `[incr Tcl]` extension, but does not require that you use either.

Python is built around a concept of objects that does not quite match the C++/Java class model, and there is no Python extension to get a complete object-oriented behavior with public, protected, and private data. The `[incr Tcl]` extension supports the full C++/Java model of object oriented coding. TclOO supports many models including inheritance, delegation, and mixins, giving a developer the freedom to choose the architecture that best models the problem being solved.

1.3.4 Tcl/Tk Compared to Java

The primary language for multi-platform applications today is Java. Tcl and Java have some similarities but are really designed to solve two different problem sets. Java is a system programming language, similar to C and C++, whereas Tcl is an interpreted scripting language. In fact, you can write Tcl extensions in Java, and Tcl can load and execute Java code. The following are advantages Tcl offers over Java.

- *Better multi-platform support.* Tcl runs on more platforms than Java. Java requires thread support, which is unavailable on many older UNIX platforms. Tcl has also been ported to real-time kernels such as VxWorks and Q-nix.

- *Faster for prototyping.* The strength of an object-oriented language is that it makes you think about the problem up front and to perform a careful design before you code. Unfortunately, for many projects, you do not know the requirements up front and cannot design a class structure until you understand the solutions better. Tcl/Tk is great for whipping out a quick prototype, seeing what truly answers the user's needs and what was not necessary, and then creating a serious design from a solid set of requirements.
- *Better GUI support.* The Tk widgets are easier to work with and provide higher-level support than the Java graphics library.
- *Configurable security levels.* Tcl supports a different security model than the Java Applet model, making Tcl an ideal language for Internet program development. With Tcl, objects with various origins can be given varying levels of access to your system. By default, the Tcl browser plug-in module comes with three security levels: untrusted (no access to file system or sockets), slightly trusted (can read files, or sockets, but not both), and fully trusted (can access all I/O types.) With Tcl, you can configure your own security model, allowing access to a subset of the file system or specific devices, for instance.
- *Smaller downloadable objects.* Because the Tcl libraries live on the machine where an application runs, there is less to download with a Tcl/Tk application than a similar set of functionality written in Java.

1.4 TCL AS AN EXTENSIBLE INTERPRETER

Many programming groups these days have a set of legacy code, a set of missions, and a need to perform some sort of rapid development. Tcl can be used to glue these code pieces into new applications. Using Tcl, you can take the existing project libraries and turn them into commands within a Tcl interpreter. Once this is done, you have a language you can use to develop rapid prototypes or even shippable programs.

Merging the existing libraries into the interpreter gives you a chance to hide sets of calling conventions that may have grown over several years (and projects) and thus expose the application programmer to a consistent application programmer interface (API). This technique gives you a chance to graft a graphics interface over older, non GUI-based sets of code.

1.5 TCL AS AN EMBEDDABLE INTERPRETER

Programs frequently start as a simple proof of concept with hard-coded values. These values quickly evolve into variables that can be set with command line arguments. Shortly after that, the command lines get unwieldy, and someone adds a configuration file and a small interpreter to parse the configuration. The macro language then grows to control program flow as well as variables. Then it starts to get messy.

At the point where you need a configuration file, you can merge in a Tcl interpreter instead of writing new code to parse the configuration file. You can use Tcl calls to interpret the configuration file, and as the requirements expand you will already have a complete interpreter available, instead of hacking in features as people request them.

Embedding a Tcl interpreter is also a way to add functionality that isn't easily available in the original program. A C, Java or FORTRAN backbone can invoke Tcl scripts to create a GUI, add network support or user-level scripting.

1.6 TCL AS A RAPID DEVELOPMENT TOOL

Tcl has a simple and consistent syntax at its core, which makes it an easy language to learn. At the edges, Tcl has many powerful extensions that provide less commonly needed functionality. These extensions include the following:

- General-purpose programming extensions such as `TclX` (new commands to make Tcl more useful for sys-admins) and `[incr Tcl]` (a full-featured object-oriented extension).
- Application-specific extensions such as `OraTcl` and `SybTcl` (for controlling Oracle or Sybase database engines). Tcl 8.6 introduces the `tdbc` package with database-neutral support for many databases including MySQL and SQLite.
- Special-purpose hardware extensions such as the extensions for controlling Smartbits and Ixia broadband test equipment.
- Special-purpose software libraries such as the TSIPP extension, which lets you generate 3D images from a Tcl script using the SIPP library.

You can think of Tcl as a simple language with a rich set of libraries. You can learn enough of the core Tcl commands to start writing programs in under an hour, and then extend your knowledge as the need arises. When a task requires some new tools (SQL database interface, for instance), you have to learn only those new commands, not an entire new language. This common core with extensions makes your life as a programmer easier. You can take all the knowledge you gained doing one project and apply most of it to your next project.

This simplicity at the core with complexity in the extensions makes Tcl/Tk very suitable for rapid prototype development projects. During the 1995 Tcl/Tk Workshop, Brion Sarachan described the product that General Electric developed for NBC to control television network distribution (the paper describing this work is printed in the Conference Proceedings of the Tcl/Tk Workshop, July 1995).

The first meeting was held before the contract was awarded and included the engineers, management, and sales staff. After discussing the basic requirements, the sales and management groups continued to discuss schedules, pricing, and such, while the engineers went into another room and put together a prototype for what the system might look like. By the time the sales and management staff were done, the engineers had a prototype to show. This turnaround speed had a lot to do with GE being awarded that contract.

The GE group expanded their prototype systems with various levels of functionality for the people at NBC to evaluate. As the project matured, the specifications were changed on the basis of the experience with prototypes. The ease with which Tcl/Tk code can be extended and modified makes it an ideal platform for this type of a project.

The ability to extend the interpreter is a feature that separates Tcl from the other multi-platform and rapid development languages. Tcl interpreters can be extended in several ways, ranging from adding more Tcl subroutines (called `procs` in Tcl) to merging C, Java, or even assembly code into the interpreter.

Studies have found that 80% of a program's runtime is spent in 20% of the code. The extensible nature of Tcl allows you to win at that game by rewriting the compute-intensive portion in a faster language while leaving the bulk of the code in the more easily maintained and modified script language.

This form of extensibility makes Tcl/Tk useful as the basis for a suite of programs. In a rapid prototype phase, a Tcl/Tk project can evolve from a simple set of back-of-the-envelope designs to a quick prototype done entirely in Tcl. As the project matures, the various subroutines can be modified quickly and easily, as they are all written in Tcl. Once the design and the main program flow have stabilized, various Tcl/Tk subroutines can be rewritten in C or Java to obtain the required performance.

Once one project has evolved to this state, you have an interpreter with a set of specialized commands that can be used to develop other projects in this problem domain. This gives you a platform for better rapid prototyping and for code reuse.

1.7 GUI-BASED PROGRAMMING

The Tcl distribution includes the Tk graphics extensions of Tcl. The Tk extension package provides an extremely rich set of GUI tools, ranging from primitive widgets (such as buttons, menus, drawing surfaces, text windows, and scrollbars) to complex compound widgets such as file selectors.

Any visual object can have a script bound to it so that when a given event happens a particular set of commands is executed. For example, you can instruct a graphics object to change color when the cursor passes over it. You can bind actions to objects as trivial as a single punctuation mark in a text document or as large as an entire display.

Chapters 11 through 14 describe using Tk to build simple GUIs, active documents (such as maps and blueprints) that will respond to user actions, and complex, custom graphics objects.

1.8 SHIPPING PRODUCTS

When it comes time to ship a product, you can either ship the Tcl scripts and Tcl interpreter or merge your Tcl script into an interpreter to create a Tcl-based executable. The advantage of shipping the Tcl scripts is that competent users (or clever programs) can modify and customize the scripts easily. The disadvantages include the need for the user to have the proper revision level of Tcl available for the script to run and the possibility that someone will reverse engineer your program.

A Tcl script can be wrapped into a copy of the interpreter, to create a binary executable that will run only this script. (See the discussions of `starkit`, ActiveState's `tclapp`, and Dennis Labelle's `Freewrap` in Chapter 18.) With these techniques, you can develop your program in a rapid development environment and ship a single program that does not require installation of the Tcl interpreter and has no human-readable code.

1.9 BOTTOM LINE

Tcl/Tk is

- A shell scripting language
- A multi-platform language

- An extensible interpreter
- An embeddable interpreter
- A graphics programming language

Tcl/Tk is useful for

- Rapid prototyping
- Shippable product
- Use-once-and-dispose scripts
- Mission-critical, 24/7 applications
- GUI-based projects
- Multi-platform products
- Adding GUIs to existing text-oriented programs
- Adding new functionality to old code libraries

1.10 PROBLEMS

The following numbering convention is used in all Problem sections:

Number Range	Description of Problems
100–199	Short comprehension problems review material covered in the chapter. They can be answered in a few words or a 1–5 line script. These problems should each take under a minute to answer.
200–299	These quick exercises require some thought or information beyond that covered in the chapter. They may require reading a man page or making a web search. A short script of 1–50 lines should fulfill the exercises, which may take 10–20 minutes each to complete.
300–399	Long exercises may require reading other material or writing a few hundred lines of code. These exercises may take several hours to complete.

- *100* Can a Tcl script invoke other programs?
- *101* Can Tcl scripts perform math operations?
- *102* Is Tcl useful for rapid development?
- *103* Who developed the Tcl language? Where? Why?
- *104* Which of these statements is correct? Tcl is
  - A single language
  - A language core with many extensions
  - A graphics language
- *105* How would you deploy a Tcl/Tk program if you could not be certain the client will have the right Tcl/Tk version installed on their system?

- *106* Can a Tcl interpreter be extended with another library?
- *107* Can the Tcl interpreter be embedded into another application?
- *200* Why would you use Tcl instead of C?
- *201* Why would you use C or C++ instead of Tcl?
- *202* Why would you use Tcl instead of a `.bat` file?
- *203* What type of requirement would preclude Tcl as the sole development language?
- *300* Some computer languages were developed to solve problems in a particular domain (SQL for database manipulation, COBOL for business applications, FORTRAN for calculations). Others were developed to support particular programming styles (functional languages such as Scheme and Lisp, object-oriented languages such as C++ and Java). Other groups of languages were created by developers who wanted a new, better tool (C for system development, Perl for systems scripts). Describe the strengths and weaknesses of each of these design philosophies.
- *301* Tcl can be used to glue small applications into a single larger application, to merge the functionality of multiple libraries into a single application, or to write completely self-contained applications. Why would you use one technique over another?

CHAPTER

The Mechanics of Using the Tcl and Tk Interpreters

2

The first step in learning a new computer language is learning to run the interpreter/compiler and creating simple executable programs. This chapter explores the following.

- The mechanics of starting the interpreters
- Starting `tclsh` and `wish` scripts in interactive mode
- Exiting the interpreter
- Running `tclsh` and `wish` scripts from files

2.1 THE `tclsh` AND `wish` INTERPRETERS

The standard Tcl/Tk binary distribution includes the following two interpreters.

- `tclsh` A text-oriented interpreter that includes the core commands, looping constructs, data types, I/O, procedures, and so on. `Tclsh` is useful for applications that do not require a GUI.
- `wish` A GUI-oriented extension to the `tclsh` interpreter. The `wish` interpreter includes the core Tcl interpreter, with all commands `tclsh` supports plus the Tk graphics extensions.

The simplest way to get started playing with Tcl and Tk is to type your commands directly into the interpreter. If you do not already have Tcl installed on your system, see the instructions for installing Tcl/Tk on the companion website.

2.1.1 Starting the `tclsh` and `wish` Interpreters

You can invoke the `tclsh` and `wish` interpreters from a menu (in a GUI environment) or from a command line (with or without additional arguments). Invoking the interpreters from a menu or a command line without arguments starts the interpreter in interactive mode: the interpreter evaluates the commands as you type them. Invoking the interpreter from a command line with a file name argument (or by dragging a file with a Tcl script to the interpreter icon) will cause the interpreter to evaluate the script in that file.

The command line to invoke the `tclsh` or `wish` interpreter from a shell prompt under UNIX, Linux or Mac OS/X resembles the following. The question marks (?) indicate optional arguments.

```
/usr/local/bin/tclsh ?scriptName? ?options?
```

If you invoke `tclsh` from a `.bat` file, MS-DOS command window, or Run menu under Microsoft Windows, the command would resemble the following.

```
C:\tcl\bin\wish.exe ?scriptName? ?options?
```

Tcl/Tk: A Developer's Guide. DOI: 10.1016/B978-0-12-384717-1.00002-6

The exact path may vary from installation to installation, and the name of the `tclsh` or `wish` interpreter may vary slightly depending on how Tcl was installed. The default installation name for the interpreters includes the revision level (e.g., `wish8.6`), but many sites link the name `wish` to the latest version of the `wish` interpreter. The example paths and interpreter names used throughout this book will reflect some of the variants you can expect.

The command line options can tune the appearance of the `wish` graphics window. You can define the color map, the name of the window, the size of the initial window, and other parameters on the command line. Some of these options are system dependent, so you should check your on-line documentation for the options available on your system. See Section 1.1.2 for a discussion on how to access the on-line help on UNIX, Macintosh, and Windows platforms.

If there are flags the Tcl interpreter does not parse, those arguments will be ignored by the `tclsh` or `wish` interpreter and passed directly to the script for evaluation.

A command line such as the following would invoke the `wish` interpreter and cause it to evaluate the code in the file *myapp.tcl*.

```
wish myapp.tcl -name "My application" -config myconfig.cnf
```

The `wish` interpreter recognizes the `-name` flag, and thus the arguments `-name "My application"` would cause the window to be opened with the string `My application` in the window decoration. The interpreter does not support a `-config` flag, and thus the arguments `-config myconfig.cnf` would be passed to the script to evaluate.

2.1.2 Starting `tclsh` or `wish` under UNIX

Under UNIX, you start `tclsh` or `wish` as you would start any other program: type `tclsh` (or `wish`) at a shell prompt. If the directory containing the interpreter you want to invoke is in your `$PATH`, you can invoke the interpreter by name, as follows.

```
wish ?scriptName? ?wish options? ?script Arguments?
```

If your `$PATH` does not include the directory that contains the `tclsh` executable, you would need to use the full path, as follows:

```
/usr/local/bin/tclsh ?scriptName? ?scriptArguments?
```

When `tclsh` is run with no command line, it will display a `%` prompt to let you know the interpreter is ready to accept commands. When `wish` starts up, it will open a new window for graphics and will prompt the user with a % prompt in the window where you invoked `wish`. Starting `wish` and typing a short script at the prompt would resemble the following.

Errors Caused by Improper Installation

When the tclsh or wish interpreter starts, it loads several Tcl script files to define certain commands. The location for these files is set when the interpreter is installed. If an interpreter is moved without being reinstalled, or the support file directories become unreadable, you may get an error message resembling the following when you try to run tclsh.

```
application-specific initialization failed:
  Can't find a usable init.tcl in the following directories:
  /usr/local/lib/tcl8.6 /usr/local/lib/tcl8.6 /usr/lib/tcl8.6
  /usr/local/library /usr/library /usr/tcl8.6a4/library
  /tcl8.6a4/library /usr/local/lib/tcl8.6

This probably means that Tcl wasn't installed properly.
```

You might obtain the following when you try to run wish.

```
Application initialization failed:
  Can't find a usable tk.tcl in the following directories:
  /usr/local/lib/tk8.6 /usr/local/lib/tk8.6 /usr/lib/tk8.6
  /usr/local/library /usr/library /usr/tk8.6a4/library
  /tk8.6a4/library

This probably means that tk wasn't installed properly.
```

The tclsh and wish interpreters try several algorithms to find their configuration files. Sometimes this will result in the same directory being checked several times. This is normal, and does not indicate any problem.

If you receive an error message like these, you should reinstall Tcl/Tk, or restore the init.tcl, tk.tcl, and other files to one of the directories listed in the default search paths. Alternatively, you can find the directory that includes the appropriate init.tcl and tk.tcl and redefine where the interpreters will look for their configuration files by setting the environment variables TCL_LIBRARY and TK_LIBRARY to the new directories.

You can set the environment variable at your command line or in your .profile, .dtprofile, .login, .cshrc, or .bashrc configuration file. Using Bourne shell, Korn shell, or bash, this command would resemble:

```
TCL_LIBRARY=/usr/local/lib/tcl8.6 ; export TCL_LIBRARY
```

Using csh, tcsh, or zsh, the command would resemble:

```
setenv TCL_LIBRARY /usr/local/lib/tcl8.6
```

2.1.3 Starting tclsh or wish under Microsoft Windows

When you install Tcl under Microsoft Windows, it will create a Tcl menu entry under the Programs menu, and within that menu entries for tclsh and wish. When you select the tclsh menu item, Tcl will create a window for you with the tclsh prompt (%). If you select the wish menu item, wish will open two windows: one for commands and one to display graphics.

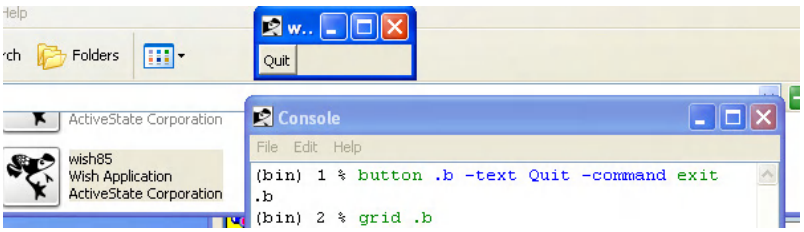
You can also invoke `tclsh` from an MS-DOS prompt by typing the complete path and command name. Note that you may need to use the 8.3 version of a directory name. Windows NT/2000/XP/Vista examples resemble this:

```
C:> \Program Files\tcl\bin\tclsh86.exe
C:> \Program Files\tcl\bin\wish86.exe
```

Windows 95/98/ME examples resemble this:

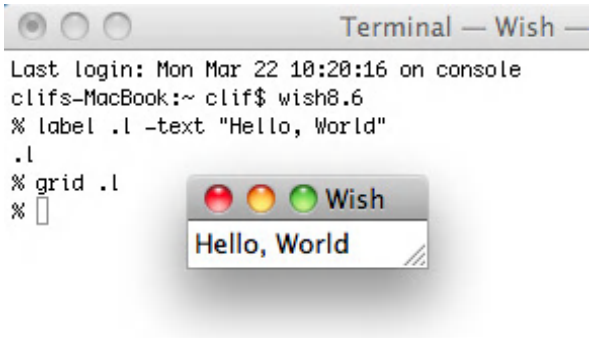
```
C:> \Progra~1\tcl\bin\tclsh.exe
C:> \Progra~1\tcl\bin\wish.exe
```

The following illustration shows `wish` being invoked by clicking an icon in the Windows File Explorer. Once you see the window with the % prompt (as shown in the following illustration), you can type commands directly to the `wish` shell. The commands you enter will be interpreted immediately. If you type a command which command creates a graphic widget, it will be displayed in the graphics window.



2.1.4 Starting `tclsh` or `wish` on the Mac

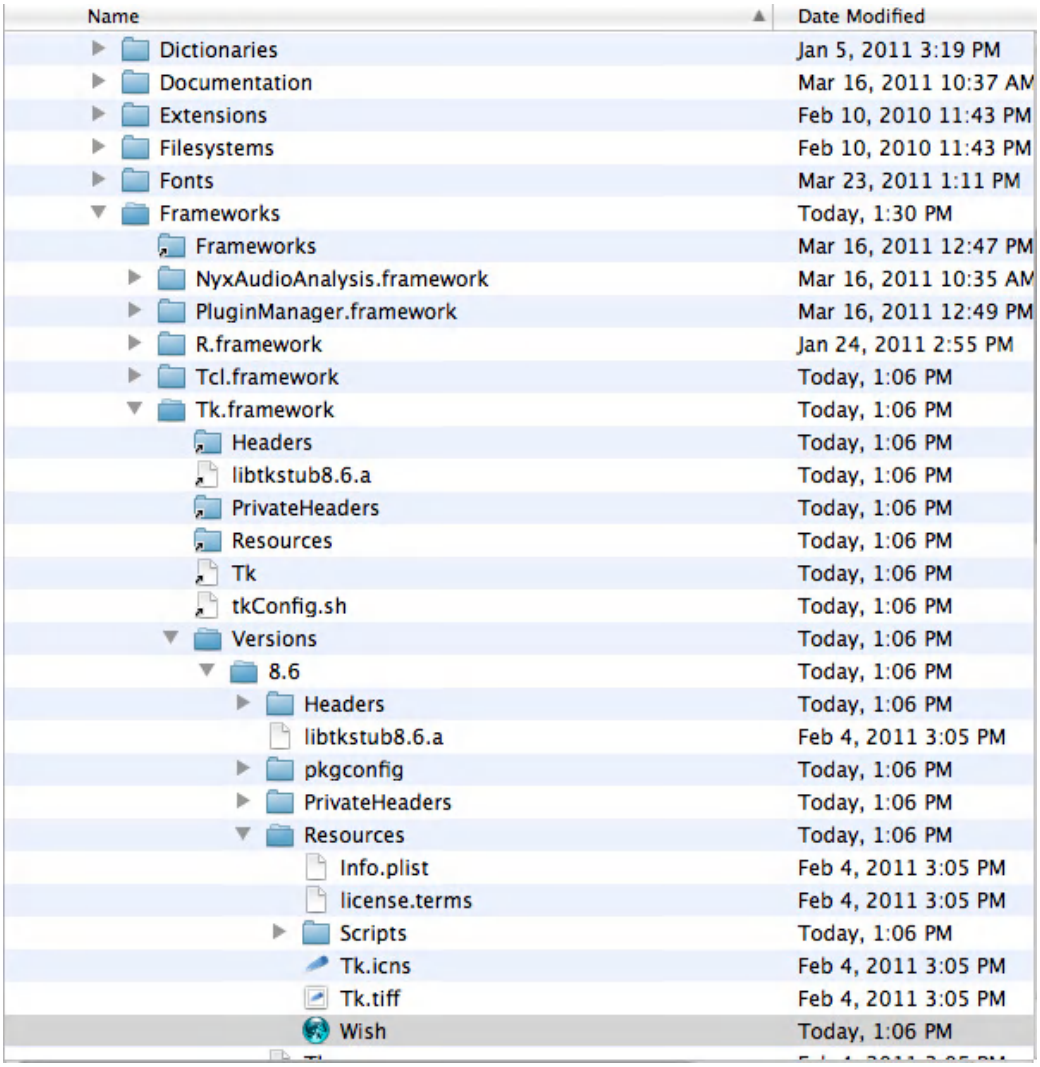
Mac OS X includes `tclsh` and `wish` interpreters. You can start the interpreter from a terminal session, just as you can with Linux/Unix.



Making a Desktop Icon

If you want to have wish available on your desktop, rather than starting it from a terminal:

1. Open a new finder window.
2. Select the Library, Frameworks, Tk.framework and Versions elements as shown in the following image.
3. Drag the Wish icon to your desktop.



2.1.5 Exiting `tclsh` or `wish`

You can exit a `tclsh` or `wish` interactive interpreter session by typing the command `exit` at the % prompt. Under UNIX, you can also exit a `wish` program by selecting a Destroy icon from the window manager or selecting Close or Destroy from a pull-down menu button.

On Microsoft Windows, you can exit a `wish` task by clicking on the X at the top right-hand corner of the graphics window. To exit `tclsh`, use the `exit` command.

On a Macintosh, you can exit a `wish` task by clicking on the red jellybutton on the top left-hand corner of the graphics window. To exit `tclsh`, use the `exit` command.

2.2 USING `tclsh/wish` INTERACTIVELY

The default `tclsh` prompt is a percent sign (%). When you see this prompt either in the window where you started `tclsh` or in a Tcl command window, you can type commands directly into the interpreter. Invoking `tclsh` in this mode is one way to experiment with commands and become familiar with their behavior.

2.2.1 Tclsh as a Command Shell

If you type a command like `ls` or `dir` that is not part of the Tcl language, the interpreter attempts to invoke that program as a subprocess and will display the child's output on your screen. This feature only exists when your session is in an interactive mode. This behavior allows you to use the Tcl interpreter as a command shell, as well as an interpreter.

Under UNIX and Mac OS/X, the command interpreters also evaluate shell programs. This is the equivalent of having the MS-DOS `.bat` file interpreter and `command.com` functionality available at the command line. Experienced users use this feature to write small programs from the command line to accomplish simple tasks.

For instance, suppose you give a friend a set of source code, and a few weeks later he returns it with a bunch of tweaks and fixes. You may want to go through all the source code to see what has been changed in each file before you commit the changes to your revision control system. Using a UNIX-style shell, you can compare all files in one directory to files with the same name in another directory and place the results in a file with a `DIF` suffix. The commands at the `Bourne`, `Korn`, or `Bash` shell prompt are as follows.

```
$ for i in *.c
    do
       diff $i ../otherDir/$i >$i.DIF
    done
```

You can use `tclsh` to gain this power in the Windows and Macintosh worlds. The code to accomplish the same task under Windows using a `tclsh` shell is:

```
% foreach i [glob *.c] {
     exec fc $i ../otherDir/$i >$i.DIF
  }
```

If you install the GNU UNIX tools distribution on your platform, you could use the diff program instead of fc, and have a "compare all files" script that runs on all platforms.

2.2.2 Tk Console (tkcon)—An Alternative Interactive tclsh/wish Shell

Jeffrey Hobbs wrote a useful console program, tkcon, which provides a nicer interface than the simple % prompt. A lightweight version program is the default console when you start tclsh or wish under MS Windows.

You can run tkcon on Mac OS/X and Linux/Unix systems as you would any other Tcl/Tk script. This program is included on the companion website.

The tkcon application provides the following features:

- A menu-driven history mechanism, to make it easy to find and repeat previous commands.
- An Emacs-like editor interface to the command line, to make it easy to fix typos or modify previous commands slightly.
- Brace and parenthesis matching, to help you keep your code ordered.
- Color-coded text, to differentiate commands, data, comments, and so on.
- The ability to save your commands in a file. This lets you experiment with commands and save your work.
- A menu entry to load and run Tcl command scripts from files.
- Support for opening several new tkcon displays.

Note that although tkcon has support for debugging Tcl/Tk scripts, is easier to work with than the interactive wish interpreter, can be used to invoke wish scripts, and is more powerful than the Windows command interpreter, tkcon does not deal well with programs that use stdin to read user input. It is designed to help develop Tcl scripts and to execute noninteractive or GUI-oriented executables, not as a replacement for the DOS Console or an xterm window.

2.2.3 Evaluating Scripts Interactively

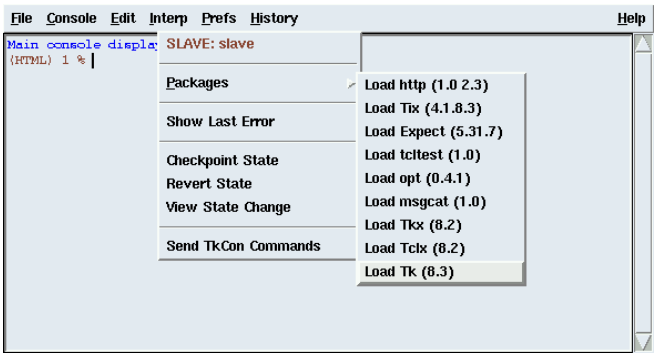
The tclsh and wish interpreters can be used interactively as Tcl program interpreters. This section will discuss typing a short program at the command prompt. The next section discusses how to run a program saved in a file.

The traditional first program, "Hello, world," is trivial in Tcl:

```
%  puts "Hello, world"
Hello, world
```

The puts command sends a string to an I/O channel. The default channel is the standard output device. Normally, the standard output is your command window, but the output can be assigned to a file, a TCP/IP socket, or a pipe to another program. The Tcl commands that create I/O channels are covered in following chapters.

Printing "Hello, world" is a boring little example, so let's make it a bit more exciting by using the Tk extensions to make it into a graphics program. In this case, you need to invoke the wish interpreter instead of the tclsh interpreter. If you are using the Tk Console program to run this example, you will need to load the Tk package by selecting the Interp menu, then Packages, and then selecting Load Tk from the Packages menu, as shown in the following illustration.



Typing the following code at the % prompt will display "Hello, world" in a small box in the graphics window.

Example 1

Script Code

```
label .l -text "Hello, world"
grid .l
```

The `label` command tells the `wish` interpreter to construct a graphics widget named `.l` and place the text "Hello, world" within that widget. The `grid` command causes the label to be displayed. These commands are covered in Chapter 11. When you run this script you should see a window resembling the following.

2.3 EVALUATING TCL SCRIPT FILES

Typing a short program at the command line is a good start, but it does not create a shippable product. The next step is to evaluate a Tcl script stored in a file.

2.3.1 The Tcl Script File

A Tcl script file is simply an ASCII text file of Tcl commands. The individual commands may be terminated with a newline marker (carriage return or line feed) or a semicolon.

Tcl has no restrictions on line length, but it can be difficult to read (and debug) scripts with commands that do not fit on a single display line. If a command is too long to fit on a single line, making the last character before the newline a backslash will continue the same command line on the next screen line. For example, the following is a simple `tclsh` script that prints several lines of information.

Example 2

Script Code

```
puts "This is a simple command"
puts "this line has two"; puts "commands on it";
puts "this line is continued \
on the next line, but prints one line"
```

Script Output

```
This is a simple command
this line has two
commands on it
this line is continued on the next line, but prints one line
```

Note that there are two spaces between the words `continued` and `on`. When the Tcl interpreter evaluates a line with a backslash in it, it replaces the `backslash newline` combination with a space.

You can write Tcl/Tk programs with any editor that will let you save a flat ASCII file.

In the UNIX world, vi, pico and Emacs are common editors. On the MS Windows platforms, Notepad, Brief, WordPerfect, and MS Word are suitable editors. The Macintosh editors MacWrite and Alpha (and other editors that generate simple ASCII files) are suitable. Note that you must specify an ASCII text file for the output and use hard newlines if you use one of the word processor editors. The Tcl interpreter does not read most native word processor formats.

There are several Tcl integrated development environments (IDEs) available, ranging from commercial packages such as ActiveState's Komodo and Neatware's MyrmocoX, to freeware such as Eclipse, to tools such as the editor Mike Doyle and Hattie Schroeder developed as an example in their book *Interactive Web Applications with Tcl/Tk* (*www.eolas.com/tcl*). IDEs are discussed in more detail in Chapter 18.

The free **Komodo Edit** application from ActiveState runs on Mac OS/X, Windows and several flavors of Unix. The **Komodo Edit** application is midway between a full IDE and a simple editor. It provides tips and highlighting for several languages.

Another free package is the Tcl Plugin for Netbeans, one of the Tcl Community's Google Summer of Code projects. This plugin also provides syntax highlighting. Details about this project are available at `http://wiki.tcl.tk/28292`.

You can also use the Tk Console to type in Tcl commands and then save them to a file via the **File > Save > History** menu choice. You will probably need to edit this file after you have saved it, to delete extra lines.

2.3.2 Evaluating Tcl Script Files

For the time being, let's assume you have created a file named *foo.tcl* containing the following text.

```
label .l -text "The arguments to this script are: $argv"
pack .l
```

There are several ways to evaluate this `wish` script file. The following are two that will work on any Mac OSX, Linux, Unix or Microsoft system. Other methods tend to be platform specific, and these are covered in the system-specific sections.

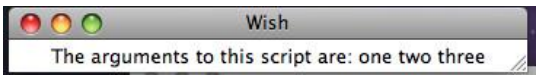
You can always evaluate a `tclsh` or `wish` script by typing the path to the interpreter, followed by the script, followed by arguments to the script. This works under Windows (from a Command window, or Run menu) or Unix, Linux or Mac OSX and would look as follows.

```
C:\tcl8.6\bin\wish86.exe foo.tcl one two three
```

or

```
/usr/local/bin/wish8.6 foo.tcl one two three
```

This will cause a window resembling the following to appear on your display.



You can also cause a script to be evaluated by typing *source foo.tcl* at the `%` prompt within an interactive `wish` interpreter session. The `source` command will read and evaluate a script but does not support setting command line arguments. Using the source command for the previous examples would result in a label that only displayed the text "The arguments to this script are:" with no arguments displayed.

```
$> wish
% source foo.tcl
```

2.3.3 Evaluating a Tcl Script File under UNIX

Under UNIX, you can use the technique of placing the name of the interpreter in the first line of the script file and using the `chmod` command to make the script executable.

```
chmod +x foo.tcl.
```

Unfortunately, the interactive Unix command interpreters (`bash`, `csh`, etc.) only search your current directory, */bin* and */usr/bin* for interpreters. If you keep `tclsh` in another directory (for instance, */usr/foo/bin*), you will need to start your script files with the complete path, as follows.

```
#!/usr/foo/bin/tclsh
```

This makes a script less portable, since it will not run on a system that has `tclsh` under */usr/bar/bin*. A clever workaround for this is to start your script file with the following lines.

```
#!/bin/sh
#\
exec wish "$0" "$@"
```

The trick here is that both Tcl and the shell interpreter use the # to start a comment, but the Bourne shell does not treat the \ as a line continuation character the way Tcl does.

The first line (`#!/bin/sh`) invokes the `sh` shell to execute the script. The shell reads the second line (`#\`), sees a comment, and does not process the `\`as a continuation character, so it evaluates the third line (`exec wish "$0" "$@"`). At this point, the `sh` shell searches the directories in your `$PATH` to find a `wish` interpreter. When it finds one, the `sh` shell overwrites itself with that `wish` interpreter to execute. Once the `wish` interpreter is running, it evaluates the script again.

When the `wish` interpreter reads the script, it interprets the first line as a comment and ignores it, and treats the second line (`#\`) as a comment with a continuation to the next line. This causes the Tcl interpreter to ignore `exec wish "$0" "$@"` as part of the comment started on the previous line, and the `wish` interpreter starts to evaluate the script.

2.3.4 Evaluating a Tcl Script File under Microsoft Windows

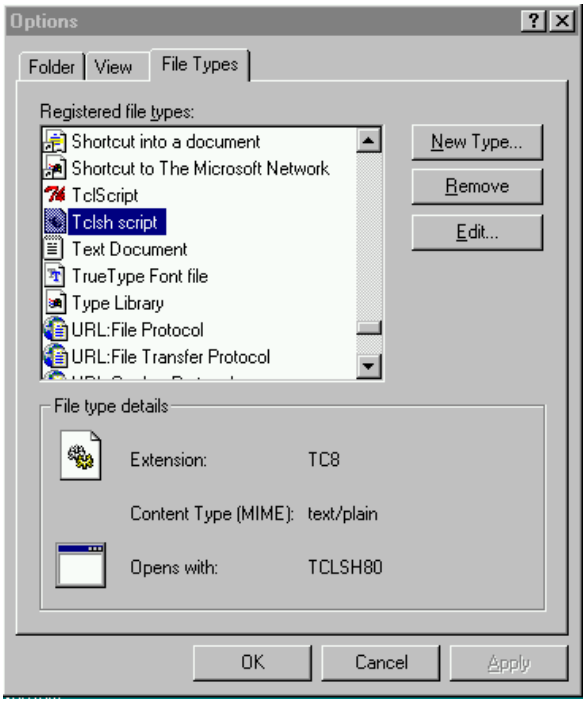
When Tcl is installed under Windows it establishes an association between the `.tcl` suffix and the `wish` interpreter. This allows you to run a `.tcl` script by typing the script name in the Run menu, or via mouse clicks using Windows Explorer, the `Find` application, and so on.

If you desire to use other suffixes for your files, you can define `tclsh` or `wish` to process files with a given suffix via the Microsoft Windows configuration menus. The following are examples of adding a file association between the suffix `.tc8` and the `tclsh` interpreter. This association will invoke the `tclsh` interpreter to run scripts that are text driven instead of graphics driven.

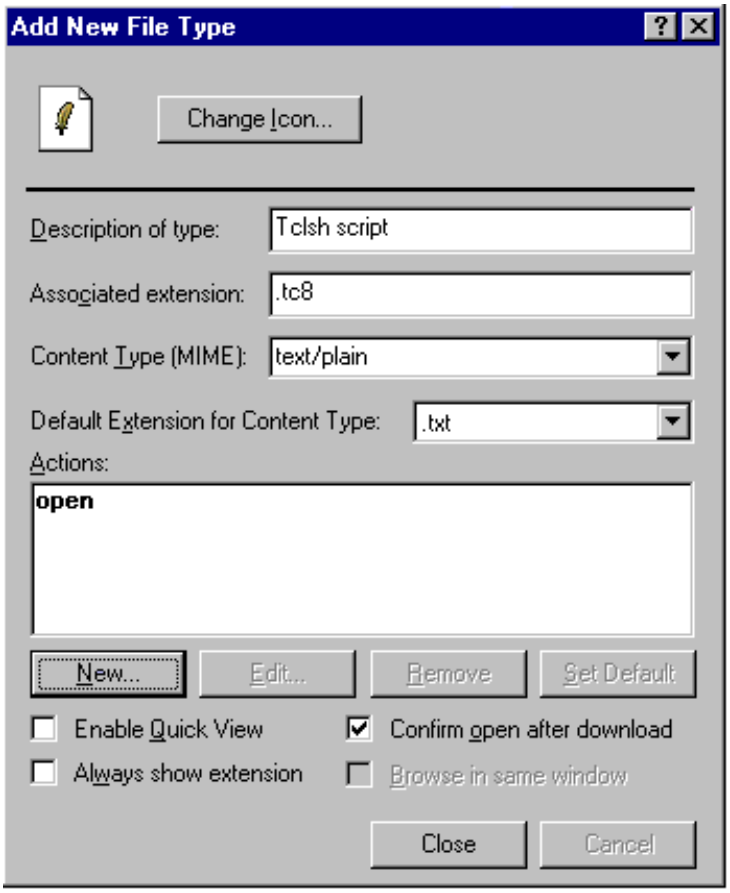
Changing File Association on Windows XP and Earlier

Select **My Computer > View > Options**, to get the **Options** display.

Select the **File Types** tab from this display and the `New Type` button from this card.



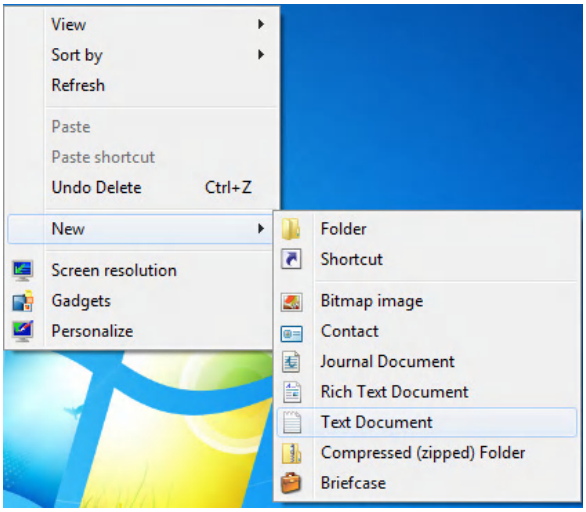
You must provide an `open` action that points to the interpreter you want to invoke to open files with this suffix. The application to perform the action needs a full path to the interpreter. Clicking the New button opens this window, where you can enter the `open` action, and the application to use to open the file.



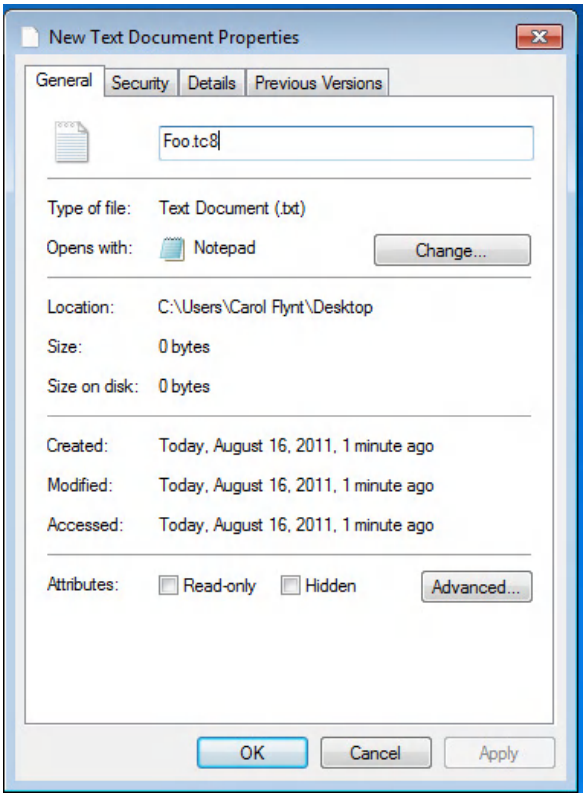
Changing File Association on Windows Vista and Windows 7

The simplest way to add a new file association in Windows Vista and Windows 7 is to create a new file with the extension you wish to add, right click the new file and select the `Open with` and add the new application for that extension as shown in the following illustration.

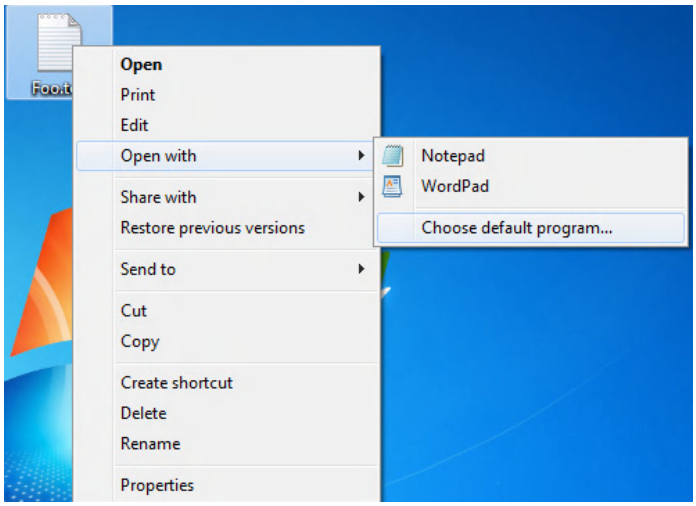
1. Create a new file:



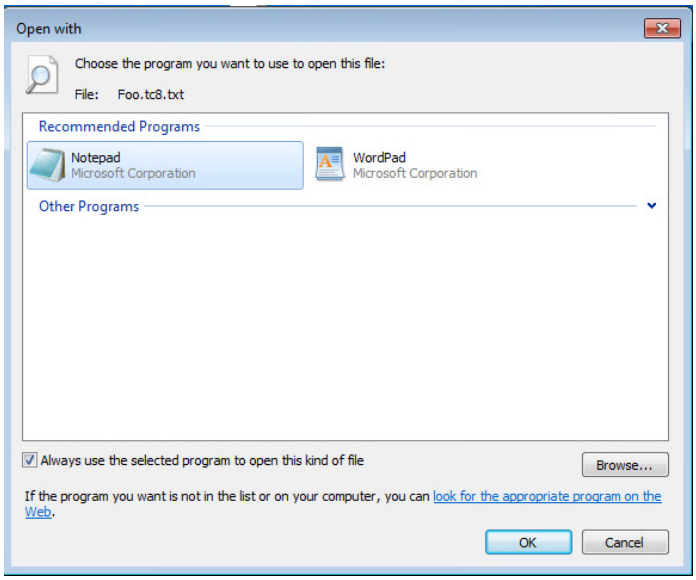
2. Give it a name with the new extension:



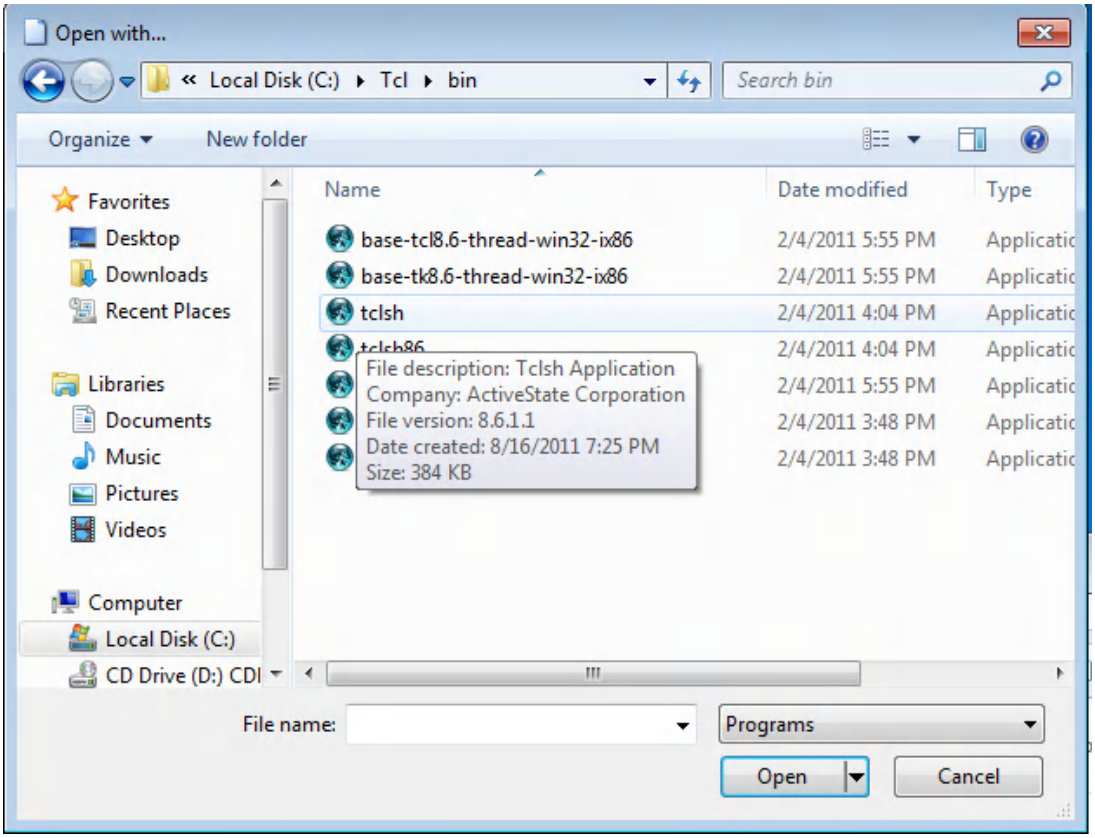
3. Right click the file icon, and select `Open with`, and select `Choose default program...`



4. Click the `Browse` button at the bottom.

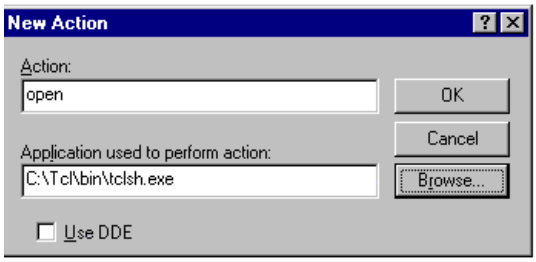


5. Edit the top line to point to the folder where the Tcl binaries are installed (the common default is C:Tcl). Select `bin` and finally select the application you wish to use to evaluate the `.tc8` files. This will probably be `tclsh` or `wish`.



Running scripts via the mouse works well for `wish`-based programs that do not require any command line arguments. If you need to invoke a script with command line options, you must invoke the script from a DOS command window, the Run selection, or via a shortcut.

You can create a shortcut by right-clicking on an empty space on the Windows screen and selecting the New and Shortcut menu items. In the Shortcut window, enter (or browse to) your Tcl script, followed by whatever options you wish, as shown in the following illustration.



This will create a shortcut on your desktop, and when you click that shortcut Windows will invoke the `wish` interpreter to evaluate the script with the command line options you entered. Using a shortcut, or the `Run` menu selection, Microsoft Windows will examine the registry to find out how to evaluate your script. If you want to invoke your script from a DOS-style command window, you will need to explicitly tell Windows what interpreter to use for the script. A technique that will always work is typing the complete path to the interpreter and the name of the script as follows.

```
C:\tcl\bin\wish myscript.tcl
```

If you do not wish to type ungainly long lines whenever you invoke a script, there are several options.

- You can add the path to the Tcl interpreter to your DOS path.
  ```
  PATH C:\Tcl\bin;C:\Windows;\C:\Windows\System32
  ```
- You can create a `.bat` file wrapper to invoke `tclsh`. This is similar to typing out the entire path in some respect, but allows you to put a single small `.bat` file in *C:\WINDOWS*, while leaving the Tcl interpreter and libraries in separate directories.
  ```
  C:\Tcl\bin\tclsh86  %1 %2 %3 %4 %5 %6 %7 %8 %9
  ```
- You can write your Tcl programs as `.bat` files and evaluate them as `filename.bat` with the following wrapper around the Tcl code.
  ```
  ::catch {};#\
  @echo off
  ::catch {};#\
  @"C:\Tcl\bin\tclsh.exe" %0 %1 %2
  ::catch {};#\
  @goto eof
  #your code here
  #\
  :eof
  ```

This is similar to the startup described for UNIX Tcl files. The lines with leading double colons are viewed as labels by the `.bat` file interpreter, whereas the Tcl interpreter evaluates them as commands in the global namespace.

These lines end with a comment followed by a backslash. The backslash is ignored by the `.bat` file interpreter, which then executes the next line. The Tcl interpreter treats the next line as a continuation of the previous comment and ignores it. The `catch` command is discussed in Chapter 5, and namespaces are discussed in Chapter 6.

If you prefer to run your scripts by single word (i.e., `filename` instead of `filename.bat`), you can change the line `@"C:\tcl\bin\tclsh.exe" %0 %1 %2` to `@"C:\tcl\bin\tclsh.exe" %0.bat %1 %2`. The problem with the `.bat` file techniques is that `.bat` files support a limited number of command line arguments. On the other hand, if you need more than nine command line arguments, you might consider using a configuration file.

2.3.5 Evaluating a Tcl Script on the Mac

Mac OS X is built over a UNIX kernel, and uses the techniques discussed for UNIX scripts. The concept of the executable script file does not really exist on the traditional Mac OS.

If you wish to evaluate a previously written script on the Mac, you can select the Source entry from the File menu, and then select the file to execute. Alternatively, you can type the command to `source` the script file at the % prompt. In this case, you must use the complete path, as follows.

```
source :demos:widget
```

2.4 BOTTOM LINE

- `tclsh` is an interpreter for text-based applications.
- `wish` is the `tclsh` interpreter with graphics extensions for GUI-based applications.
- These programs may be used as interactive command shells or as script interpreters.
- Either interpreter will accept a script to evaluate as a command line argument. A script is an ASCII file with commands terminated by newlines or semicolons. In a script, commands may be continued across multiple lines by making the final character of a line the backslash character.
- `tclsh` scripts may be made executable by wrapping them in `.bat` files under MS Windows.
- `tclsh` scripts may be made executable by setting the execute bit with `chmod` under UNIX.
- `wish` scripts that need no arguments may be executed from a File Explorer or Desktop under MS Windows, Mac OSX or Linux/Unix.
- `wish` scripts that need arguments may be executed from the command window or the Run menu under MS Windows.
- Arguments after the script file name that are not evaluated by the interpreter will be made available to the script being evaluated.
- The command to exit the `tclsh` or `wish` interpreters is `exit`.

2.5 PROBLEMS

The following numbering convention is used in all Problem sections:

Number Range	Description of Problems
100–199	Short comprehension problems review material covered in the chapter. They can be answered in a few words or a 1–5-line script. These problems should each take under a minute to answer.
200–299	These quick exercises require some thought or information beyond that covered in the chapter. They may require reading a man page or making a web search. A short script of 1–50 lines should fulfill the exercises, which may take 10–20 minutes each to complete.
300–399	Long exercises may require reading other material or writing a few hundred lines of code. These exercises may take several hours to complete.

- *100* Can the `wish` shell be used to execute other programs?
- *101* Can the text in the window decoration of a `wish` application be defined from the command line?

- *102* Can a `tclsh` or `wish` script access command line arguments?
- *103* If you have a private copy of `wish` in your personal `bin` account on a UNIX/Linux platform, and have that directory in your search path, can you start scripts with `#!wish` to have them evaluated with this copy of `wish`? Assume the script is located in the `bin` with the `wish` interpreter, and the current working directory is your $HOME.
- *104* If you install Tcl on MS Windows under `D:\Tcl8.6`, will scripts with a file name ending in `.tcl` be evaluated by `wish` when you select them from File Explorer?
- *105* What is the `wish` command to create a window with simple text in it?
- *106* What `wish` command will cause a window to be displayed?
- *107* What `tclsh` command will generate output to a file or device?
- *200* If you type `ls` during an interactive Tcl session, `tclsh` will attempt to execute the `ls` on the underlying operating system. Can you include an `ls` command in a `tclsh` script? Why or why not?
- *201* Write a `wish` script that has a label with your name in it.
- *202* Write a `tclsh` script that prints out several lines on the standard output.
- *203* What command line would start a `wish` interpreter with the title "My Wish Application" in the window decoration? (UNIX or MS Windows only.)
- *300* Each window created in `wish` needs to have a unique name starting with a lowercase letter. Write a `wish` script that creates labels: one label should contain your name, one your favorite color, and one a numeric value (which might be the air speed of a swallow).
- *301* One of the two standard interpreters (`wish` and `tclsh`) was ported to MS-DOS 5.0 (not Windows). Which was ported, and why not the other?

CHAPTER

Introduction to the Tcl Language

3

The next five chapters constitute a Tcl language tutorial. This chapter provides an overview of the Tcl syntax, data structures, and enough commands to develop applications. Chapter 4 discusses Tcl I/O support for files, pipes, and sockets. Chapters 5–8 introduce more commands and techniques and provide examples showing how Tcl data constructs can be used to create complex data constructs such as structures and trees. Chapters 9 and 10 introduce the TclOO object-oriented support package and explain some tricks in using dynamic and introspective object-oriented programming effectively.

This introduction to the Tcl language will give you an overview of how to use Tcl, rather than be a complete listing of all commands and all options. The on-line reference pages are the complete reference for the commands. See Chapter 1 for a discussion on how to access the on-line help on UNIX, Macintosh, and Windows platforms. The companion website contains a Tcl/Tk reference guide that contains brief listings of all commands and all options.

If you prefer a more extensive tutorial, see the *tutorials* list on the companion website. You will find a copy of `TclTutor`, a computer-assisted instruction program that covers all of the commands in Tcl, and most of the command options.

Chapters 11 through 14 constitute the Tk tutorial. If you are performing graphics programming, you may be tempted to skip ahead to those chapters and just read about the GUIs. Don't do it! Tcl is the glue that holds the graphic widgets together. Tk and the other Tcl extensions build on the Tcl foundation. If you glance ahead for the Tk tutorial, plan on coming back to fill in the gaps.

This book will print the command syntax using the font conventions used by the Tcl on-line manual and help pages. This convention is as follows.

`commandname`	The command name appears first in this type font.
`subcommandname`	If the command supports subcommands, they will also be in this type font.
`-option`	Options appear in italics. The first character is a dash (`-`).
`argument`	Arguments to a command appear in italics.
`?-option?`	Options that are not required are bounded by question marks.
`?argument?`	Arguments that are not required are bounded by question marks.

The following is an example.

Syntax: `puts` *`?-nonewline? ?channel? outputString`*

The command name is `puts`. The `puts` command will accept the options *`-nonewline`* and *`channel`* as arguments, and must include an `outputString` argument.

Tcl/Tk: A Developer's Guide. DOI: 10.1016/B978-0-12-384717-1.00003-8

3.1 OVERVIEW OF THE BASICS

The Tcl language has a simple and regular syntax. You can best approach Tcl by learning the overall syntax and then learning the individual commands. Because all Tcl extensions use the same base interpreter, they all use the same syntax. This consistency makes it easy to learn new sets of commands when the need arises.

3.1.1 Syntax

Tcl is a position-based language, not a keyword-based language. Unlike languages such as C, FORTRAN, and Java, there are no reserved strings. The first word of a Tcl command line must always be a Tcl command; either a built-in command, a procedure name, or (when `tclsh` is in interactive mode) an external command.

A complete Tcl command is a list of words. The first word must be a Tcl command name or a procedure name. The words that follow may be subcommands, options, or arguments to the command. The command is terminated by a newline or a semicolon.

For example, the word `puts` at the beginning of a command is a command name, but `puts` in the second position of a command could be a subcommand or a variable name. The Tcl interpreter keeps separate hash tables for the command names and the variable names, so you can have both a command `puts` and a variable named `puts` in the same procedure. The Tcl syntax rules are as follows.

- The first word of a command line is a command name.
- Each word in a command line is separated from the other words by one or more spaces.
- Words can be grouped with double quotes or curly braces.
- A list can be ungrouped with the three-character {`*`} operator.
- Commands are terminated with a newline or semicolon.
- A word starting with a dollar sign ($) must be a variable name. This string will be replaced by the value of the variable.
- A variable name followed by a value within parentheses (no spaces) is an associative array: `arrayName(index)`
- Words enclosed within square brackets must be a legal Tcl command. This string will be replaced by the results of evaluating the command.

The Tcl interpreter treats a few characters as having special meaning. These characters are as follows.

Substitution Symbols	
$	The word following the dollar sign must be a variable name. Tcl will substitute the value assigned to that variable for the `$varName` string.
[]	The words between the square brackets must be a Tcl command string. The Tcl interpreter will evaluate the string as a command. The value returned by the command will replace the brackets and string.

Grouping Symbols	
""	Groups multiple words into a single string. Substitutions will occur within this string.
{}	Groups multiple words into a single string. No special character interpretation will occur within this string. A newline within curly braces does not denote the end of a command, and no variable or command substitutions will occur.
Other	
\	Escapes the single character following the backslash. This character will not be treated as a special character. This can be used to escape a dollar sign to inhibit substitution, or to escape a newline character to continue a command across multiple lines.
;	Marks the end of a command.
<newline>	Marks the end of a command.
#	Marks the rest of the line as a comment. Note that the # must be in a position where a Tcl command name could be: either the first character on a line or following a semicolon (;).

Example 1

```
x=4
```

Not valid: The string `x=4` is interpreted as the first word on a line and will be evaluated as a procedure or command name. This is not an assignment statement.

```
Error Message:  invalid command name "x=4"
```

```
puts "This command has one argument";
```

Valid: This is a complete command.

```
puts one; puts two;
```

Valid: This line has two commands.

```
puts one puts two
```

Not valid: The first `puts` command is not terminated with a semicolon, so Tcl interprets the line as a `puts` command with three arguments.

```
Error Message:  bad argument "two": should be
"nonewline"
```

3.1.2 Grouping Words

The spaces between words are important. Since Tcl does not use keywords, it scans commands by checking for symbols separated by white space. Tcl uses spaces to determine which words are commands, subcommands, options, or data. If a data string has multiple words that must be treated as a single set of data, the string must be grouped with quotes (" ") or curly braces ({}).

Example 2

`if { $x > 2} {`
`  set greater true`
`}`
	Valid: If the value of `x` is greater than 2, the value of `greater` is set to "true."

`if{ $x > 2} {`
`  set greater true`
`}`
	Not valid: No space between `if` and test left brace.
	`Error Message: invalid command name "if{"`

`if {$x > 2}{`
`  set greater true`
`}`
	Not valid: No space between test and body left brace.
	`Error Message: invalid command name "}"`

`set x "a b"`	Valid: The variable `x` is assigned the value `a b`.
`set x {a b}`	Valid: The variable `x` is assigned the value `a b`.
`set x a b`	Not valid: Too many arguments to set. `Error Message: wrong # args: should be "set varName ?newValue?"`

The Tcl interpreter treats quotes and braces differently. These differences are discussed later in this chapter, and in more detail in Chapter 4.

3.1.3 Comments

A comment is denoted by putting a pound sign (#) in the position where a command name could be. The *Tcl Style Guide* recommends that this be the first character on a line, but the pound sign could be the first character after a semicolon.

Example 3

`# This is a comment`	Valid: This is a valid comment.
`puts "test" ; # Comment after a command.`	Valid: But not Recommended style.
`puts "test" # this is a syntax error.`	Not valid: The `puts` command was not terminated.

3.1.4 Data Representation

Tcl does not require that you declare variables before using them. The first time you assign a value to a variable name, the Tcl interpreter allocates space for the data and adds the variable name to the internal tables.

A variable name is an arbitrarily long sequence of letters, numbers, or punctuation characters. Although any characters (including spaces) can be used in variable names, the convention is to follow naming rules similar to those in C and Pascal; start a variable name with a letter, followed by a sequence of alphanumeric characters.

The usual convention is to start variable names with a lowercase letter.

The *Tcl Style Guide* recommends that you start variable and procedure names that are exported from a namespace with a lowercase letter and start variable and procedure names that are only for internal use with an uppercase letter. The rationale for this is that items that are intended for internal use should require more keystrokes than items intended for external use. This document is available at the Tcl/Tk resource (*www.tcl.tk/doc/styleGuide.pdf*) and on the companion website.

A variable is referenced by its name. Placing a dollar sign (`$`) in front of the variable name causes the Tcl interpreter to replace the $*varName* string with the value of that variable. The Tcl interpreter always represents the value of a Tcl variable as a printable string within your script. (Internally, it may be a floating-point value or an integer. Tcl interpreter internals are described in Chapter 15.)

Example 4

`set x four`	Set the value of a variable named `x` to four.
`set pi 3.14`	Set the value of a variable named `pi` to 3.14.
`puts "pi is $pi"`	Display the string: ”pi is 3.14”.
`set pi*2 6.28`	Set the value of a variable named `pi*2` to 6.28.
`set "bad varname" "Don't Do This"`	Set the value of a variable named `bad varname` to `Don't Do This`.

Note that the `*` symbol in the variable name `pi*2` does not mean to multiply. Since the `*` is embedded in a word, it is simply another character in a variable name. The last example shows how spaces can be embedded in a variable name. This is not recommended style.

3.1.5 Command Results

All Tcl commands return either a data value or an empty string. The data can be assigned to a variable, used in a conditional statement such as an `if`, or passed to another command.

The Tcl interpreter will evaluate a command enclosed within square brackets immediately, and replace that string with the value returned when the command is evaluated. This is the same as putting a command inside backquotes in UNIX shell programming.

For example, the `set` command always returns the current value of the variable being assigned a value. In the example that follows, when `x` is assigned the value of `"apple"`, the `set` command returns `"apple"`. When the command `set x "pear"` is evaluated within the square brackets, it returns `"pear"`, which is then assigned to the variable `y`.

Example 5

```
# The set x command returns the contents of the variable.
%  set x "apple"
apple
%  set y [set x "pear"]
pear
%  puts $y
pear
%  puts $x
pear
```

In the previous example, the quotes around the words `apple` and `pear` are not required by the Tcl interpreter. However, it is good practice to place strings within quotes.

3.1.6 Errors

Like other modern languages, Tcl has separate mechanisms for the status and data returns from commands and functions. If a Tcl command fails to execute for a syntactic reason (incorrect arguments, and so on), the interpreter will generate an error and invoke an error handler. The default error handler will display a message about the cause of the error and stop evaluating the current Tcl script.

A script can disable the default error handling by catching the error with the `catch` command, and can generate an error with the `error` command.

The `catch` command is used to catch an error condition without causing a script to abort processing.

Syntax: `catch` *`script ?varName?`*
`script` A script to evaluate
`?varName?` An optional variable name to receive the results of evaluating the script.

If the script generates an error, the `catch` command will return a true (1). If the script runs without error, the `catch` command will return a false (0). If an optional variable name is provided, the return value from running the command (either an error message or a return value) is set as that variable's value.

3.2 COMMAND EVALUATION AND SUBSTITUTIONS

Much of the power of the Tcl language is in the mechanism used to evaluate commands. The evaluation process is straightforward and elegant but, like a game of Go, it can catch you by surprise if you do not understand how it works.

Tcl processes commands in two steps. First, it performs command and variable substitutions, and then it evaluates the resulting string. Note that everything goes through this evaluation procedure. Both internal commands (such as `set`) and subroutines you write are processed by the same evaluation code. A `while` command, for example, is treated just like any other command. It takes two arguments: a test and a body of code to execute if the test is true.

3.2.1 Substitution

The first phase in Tcl command processing is substitution. The Tcl interpreter scans commands from left to right. During this scan, it replaces phrases that should be substituted with the appropriate values. Tcl performs two types of substitutions:

- A Tcl command within square brackets (`[...]`) is replaced by the results of that command. This is referred to as command substitution.
- A variable preceded by a dollar sign is replaced by the value of that variable. This is referred to as variable substitution.

After these substitutions are done, the resulting command string is evaluated.

3.2.2 Controlling Substitutions with Quotes, Curly Braces, and the Backslash

Most Tcl commands expect a defined number of arguments and will generate an error if the wrong number of arguments is presented to them. When you need to pass an argument that consists of multiple words, you must group the words into a single argument with curly braces or with quotes.

The difference between grouping with quotes and grouping with braces is that substitutions will be performed on strings grouped with quotes but not on strings grouped with braces. Examples 6 and 7 show the difference between using quotes and curly braces.

The backslash may be used to disable the special meaning of the character that follows the backslash. You can escape characters such as the dollar sign, quote, or brace to disable their special meaning for Tcl. Examples 8 and 9 show the effects of escaping characters. A Tcl script can generate an error message with embedded quotes with code, as in the following.

```
puts "ERROR: Did not get expected \"+OK\" prompt"
```

The following examples show how quotes, braces, and backslashes affect the substitutions. The first example places the argument to `puts` within curly braces. No substitutions will occur.

Example 6

Script Example

```
set x 2
set y 3
puts {The sum of $x and $y is returned by [expr $x+$y]}
```

Script Output

```
The sum of $x and $y is returned by [expr $x+$y]
```

In Example 7, `puts` has its argument enclosed in quotes, so everything is substituted.

Example 7

Script Example

```
set x 2
set y 3
puts "The sum of $x and $y is [expr $x+$y]"
```

Script Output

```
The sum of 2 and 3 is 5
```

In Example 8, the argument is enclosed in quotes, so substitution occurs, but the square brackets are escaped with backslashes to prevent Tcl from performing a command substitution.

Example 8

Script Example

```
set x 2
set y 3
puts "The sum of $x and $y is returned by \[expr $x+$y\]"
```

Script Output

```
The sum of 2 and 3 is returned by [expr 2+3]
```

Example 9 escapes the dollar sign on the variables to prevent them from being substituted and also escapes a set of quotes around the `expr` string. If not for the backslashes before the quotes, the quoted string would end with the second quote symbol, which would be a syntax error. Sets of square brackets and curly braces nest, but quotes do not.

Example 9

Script Example

```
set x 2
set y 3
puts "The sum of \$x + \$y is returned by \"\[expr \$x+\$y\]\""
```

Script Output

```
The sum of $x + $y is returned by "[expr $x+$y]"
```

Splitting Lists

The {*} operator will convert a list to its component parts before evaluating a command. This is commonly used when one procedure returns a set of values that need to be passed to another procedure as separate values, instead of as a single list.

The `set` command requires two arguments to assign a value to a variable - the name of the variable and the value. You cannot assign the variable name and value to a string and then pass that string to the `set` command.

```
# This is an error
set nameANDvalue "a 2"
set $nameANDvalue
```

The {*} operator can split the string `"a 2"` into two components: the letter `a` and the number `2`.

Example 10

Script Example

```
# This works
set nameANDvalue "a 2"
```

```
set {*}$nameANDvalue
puts $a
```

Script Output

```
2
```

3.2.3 Steps in Command Evaluation

When a Tcl interpreter evaluates a command, it makes only one pass over that command to perform substitutions. It does not loop until a variable is fully resolved. However, if a command includes another Tcl command within brackets, the command processor will be called recursively until there are no further bracketed commands. When there are no more phrases to be substituted, the command is evaluated, and the result is passed to the previous level of recursion to substitute for the bracketed string.

The next example shows how the interpreter evaluates a command. The indentation depth represents the recursion level. Let's examine the following command.

```
set x [expr [set a 3] + 4 + $a]
```

The `expr` command performs a math operation on the supplied arguments and returns the results. For example, `expr 2+2` would return the value `4`.

The interpreter scans the command from left to right, looking for a phrase to evaluate and substitute. The scanner encounters the left square bracket, and the command evaluator is reentered with that subset of the command.

```
expr [set a 3] + 4 + $a
```

The interpreter scans the new command, and again there is a bracket, so the command evaluator is called again with the following subset.

```
set a 3
```

There are no more levels of brackets and no substitutions to perform, so this command is evaluated, the variable `a` is set to the value `3`, and `3` is returned. The recursive call returned `3`, so the value 3 replaces the bracketed command, and the command now resembles the following.

```
expr 3 + 4 + $a
```

The variables are now substituted, and `$a` is replaced by `3`, making the following new command.

```
expr 3 + 4 + 3
```

The interpreter evaluates this string, and the result (`10`) is returned. The substitution is performed, and the command is now as follows.

```
set x 10
```

The interpreter evaluates this string, the variable `x` is set to `10` , and `tclsh` returns `10`. In particular, note that the variable `a` was not defined when this command started but was defined within the first bracketed portion of the command. If this command had been written in another order, as in the following,

```
set x [expr $a + [set a 3] + 4 ]
```

the Tcl interpreter would attempt to substitute the value of the variable `a` before assigning the value `3` to `a`.

- If `a` had not been previously defined, it would generate an error.
- If `a` had been previously defined, the command would return an unexpected result depending on the value. For instance, if `a` contained an alphabetic string, `expr` would be unable to perform the arithmetic operation and would generate an error.

A Tcl variable can contain a string that is a Tcl command string. Dealing with these commands is discussed in Chapter 5.

3.3 DATA TYPES

The primitive data type in Tcl is the string (which may be a numeric value). The composite data types are the list, dict and associative array. The Tcl interpreter manipulates some complex entities such as graphic objects, I/O channels, and sockets via handles. Handles are introduced briefly here, with more discussion in the following chapters.

Unlike C, C++, or Java, Tcl is a typeless language. However, certain commands can define what sort of string data they will accept. Thus, the `expr` command, which performs math operations, will generate an error if you try to add 5 to the string "You can't do that."

3.3.1 Assigning Values to Variables

The command to define the value of a variable is `set`. It allocates space for a variable and data and assigns the data to that variable.

Syntax: `set` *`varName ?value?`*

Define the value of a variable.

`varName`	The name of the variable to define.
`value`	The data (value) to assign to the variable.

`set` always returns the value of the variable being referenced. When `set` is invoked with two arguments, the first argument is the variable name and the second is a value to assign to that variable. When `set` is invoked with a single argument, the argument is a variable name and the value of that variable is returned.

Example 11

```
% set x 1
1
% set x
1
% set z [set x 2]
2
% set z
```

```
2
%  set y
can't read "y": no such variable
```

Because Tcl is often used as a string processing language, it's also useful to be able to add new characters to the end of the value in a variable. The `append` command will append a string to the end of a variable. If the variable was not previously defined, the `append` command will create the variable and will assign the initial value to the string.

Syntax: `append` *`varName ?value1? ?value2?`*

Append one or more new values to a variable.

`varName` The name of the variable to which to append the data.

`value` The data to append to the variable content.

Note that `append` appends only the data you request. It does not add any separators between data values.

Example 12

```
%  set x 1
1
%  append x 2
12
%  append x
12
%  append x 3 4
1234
%  append y new value
newvalue
```

3.3.2 Strings

The Tcl interpreter represents all data as a string within a script. (Within the interpreter, the data may be represented in the computer's native format.) A Tcl string can contain alphanumeric, pure numeric, Boolean, or even binary data.

Alphanumeric data can include any letter, number, or punctuation. Tcl uses 16-bit Unicode to represent strings, which allows non-Latin characters (including Japanese, Chinese, and Korean) to be used in strings. A Tcl script can represent numeric data as integers, floating-point values (with a decimal point), hexadecimal or octal values, or scientific notation.

You can represent a Boolean value as a 1 (for true) and 0 (for false), or as the string `"true"` or `"yes"` and `"false"` or `"no"`. Any capitalization is allowed in the Boolean string: `"TrUe"` is recognized as a Boolean value. The command that receives a string will interpret the data as a numeric or alphabetic value, depending on the command's data requirements.

Example 13

Legitimate Strings

```
set alpha "abcdefg"
```

Assign the string `"abcdefg"` to the variable `alpha`.

```
set validString "this is a valid string"
```

Assign the string `"this is a valid string"` to the variable `validString`.

```
set number 1.234
```

Assign the number `1.234` to the variable `number`.

```
set octalVal 0755
```

Assign the octal value `755` to the variable `octalVal`. Commands that interpret values numerically will convert this value to 493 (base 10). Support for the leading 0 to represent octal numbers is still supported in Tcl 8.6, but may be removed in later releases.

```
set hexVal 0x1ed
```

Assign the hex value `1ED` to the variable `hexVal`. Commands that interpret values numerically will convert this value to 493 (base 10).

```
set scientificNotation 2e2
```

Assign the string `2e2` to the variable `scientificNotation`. Commands that interpret values numerically will convert this value to 200.

```
set msg {Bad input: "Bogus". Try again.}
```

Assign the string `Bad input: "Bogus". Try again.` to the variable `msg`. Note the internal quotes. Quotes within a braced string are treated as ordinary characters.

```
set msg "Bad input: \"Bogus\". Try again."
```

Assign the string `Bad input: "Bogus". Try again.` to the variable `msg`. Note that the internal quotes are escaped.

Bad Strings

```
set msg "Bad input: "Bogus". Try again."
```

The quotes around `Bogus` are not escaped and are treated as quotes. The quote before `Bogus` closes the string, and the rest of the line causes a syntax error.

```
Error Message:  extra characters after close-quote
```

```
set badstring "abcdefg
```

Has only one quote. The error message for this will vary, depending on how the missing quote is finally resolved.

```
set mismatch {this is not a valid string"
```

Quote and brace mismatch. The error message for this will vary, depending on how the missing quote is finally resolved.

```
set noquote this is not a valid string
```

This set of words must be grouped to be assigned to a variable.

```
Error Message:  wrong # args: should be "set varName ?newValue?"
```

3.3.3 String Processing Commands

The `string`, `format`, and `scan` commands provide most of the tools a script writer needs for manipulating strings. The regular expression commands are discussed in Section 5.6. The `string` subcommands include commands for searching for substrings, identifying string matches, trimming unwanted characters, determining the contents of a string, replacing substrings and converting case. The `format` command generates formatted output from a format descriptor and a set of data (like the C library `sprintf` function). The `scan` command will extract data from a string and assign values to variables (like the C library `scanf` function).

All Tcl variables are represented as strings. You can use the string manipulation commands with integers and floating-point numbers as easily as with alphabetic strings. When a command refers to a position in a string, the character positions are numbered from 0, and the last position can be referred to as `end`.

There is more detail on all of the string subcommands in the Tcl reference and the companion website tutorials. The following subcommands are used in the examples in the next chapters.

The `string match` command searches a target string for a match to a pattern. The pattern is matched using the glob match rules. The rules for glob matching are as follows.

`*` Matches 0 or more characters.

`?` Matches a single character.

`[]` Matches a character in the set defined within the brackets.

- `[abc]` Defines `abc` as the set.
- `[m-y]` Defines all characters alphabetically between `m` and `y` (inclusive) as the set.

`\?` Matches the single character ?.

Note that the glob rules use `[ ]` in a different manner than the Tcl evaluation code. You must protect the brackets from `tclsh` evaluation, or `tclsh` will try to evaluate the phrase within the brackets as a command and will probably fail. Enclosing a glob expression in curly braces will accomplish this.

Syntax: `string match` *`pattern string`*

Returns 1 if *`pattern`* matches *`string`*, else returns 0.

`pattern` The pattern to compare to *`string`*.

`string` The string to match against the *`pattern`*.

Example 14

```
%  set str "This is a test, it is only a test"
This is a test, it is only a test
%  string match "*test*" $str
```

```
1
%  string match "not present" $str
0
```

The `string tolower` command converts a string to lowercase letters. Note that this is not done in place. A new string of lowercase letters is returned. The `string toupper` command converts strings to uppercase using the same syntax.

Syntax: `string tolower` *`string`*

Syntax: `string toupper` *`string`*

`string` The string to convert.

Example 15

```
%  set upper [string toupper $str]
THIS IS A TEST, IT IS ONLY A TEST
%  set lower [string tolower $upper]
this is a test, it is only a test
```

The `string first` command returns the location of the first instance of a substring in a test string, or −1 if the pattern does not exist in the test string. The `string last` returns the character location of the last instance of the substring in the test string.

Syntax: `string first` *`substr string`*

Syntax: `string last` *`substr string`*

Return the location of the first (or last) occurrence of *`substr`* in *`string`*.

`substr` The substring to search for.

`string` The string to search in.

Example 16

```
%  set str "This is a test, it is only a test"
This is a test, it is only a test
%  set st_first [string first st $str]
12
%  set st_last [string last st $str]
31
```

The `string length` command returns the number of characters in a string. With Tcl 8.0 and newer, strings are represented internally as 2-byte Unicode characters. The value returned by `string length` is the number of characters, not bytes.

Syntax: `string length` *`string`*

Return the number of characters in *`string`*.

`string` The string.

Example 17

```
set len [string length $str]
33
```

The `string range` command returns the characters between two points in the string.

Syntax: `string range` *`string first last`*

Returns the characters in *`string`* between *`first`* and *`last`*.

`string` The string.

`first` The position of the first character to return

`last` The position of the last character to return

Example 18

```
%  set subset [string range $str $st_first $st_last]
st, it is only a tes
```

The `string map` command replaces one or more substrings within a string with new values.

Syntax: `string map` *`map string`*

Return a modified *`string`* based on values in the *`map`*.

`map` a set of string pairs that describe the changes to be made to the string. Each string pair is an *`old`* string and a *`new`* string which will replace *`old`* string.

`string` A string to be modified.

Example 19

```
%  string map {"a test" "an exam"} $str
This is an exam, it is only an exam
# The map may contain multiple pairs as
#     {old1 new1 old2 new2 ...}
%  string map \
     {This These is are a {} test tests it they} $str
These are tests, they are only tests
```

The `string is` command will report what type of data is in a string. It can test to see if a string is an integer, double, printable string, boolean and more. See the man page for all the types of tests the `string is` command can perform.

Syntax: `string is` *`type string`*

Test to see if the string matches the type.

`type` The type of data that might be contained in string. Options include:

`digit`	Any unicode digit
`integer`	An integer value. May include leading or trailing whitespace.
`double`	A number
`alnum`	A letter or number
`upper`	Uppercase letters
`lower`	Lowercase letters
`space`	Any unicode space character

Example 20

```
% string is integer "123"
1
% string is integer "123a"
0
% string is integer "123.0"
0
% string is double "123.0"
1
% string is alnum "123a"
1
% string is alnum "123.0a"
0
```

The `format` command generates formatted strings and can perform some data conversions. It is equivalent to the C language `sprintf` command.

Syntax: `format` *`formatString ?data1? ?data2? ...`*

Return a new formatted string.

`formatString`	A string that defines the format of the string being returned.
`data#`	Data to substitute into the formatted string.

The first argument must be a format description. The format description can contain text strings and % fields. The text string will be returned exactly as it appears in the format description. The % fields will be substituted with formatted strings derived from the data that follows the format descriptor. A literal percent symbol can be generated with a %% field.

The format for the % fields is the same as that used in the C library. The field definition is a string consisting of a leading percent sign, two optional fields, and a ‘`formatDefinition`, as follows.

`% ?justification? ?field width? formatDefinition`

- The first character in a % field is the `%` symbol.
- The *`field justification`* may be a plus or minus sign. A minus sign causes the content of the % field to be left justified. A plus sign causes the content to be right justified. By default the data is right justified.
- The *`field width`* is a numeric field. If it is a single integer, it defines the width of the field in characters. If this value is two integers separated by a decimal point, the first integer represents the total width of the field in characters, and the second represents the number of digits to the right of the decimal point to display for floating-point formats.
- The *`formatDefinition`* is the last character. It must be one of the following.

s — The argument should be a string.
Replace the field with the argument.

```
% format %s 0xf
0xf
```

c — The argument should be a decimal integer.
Replace the field with the ASCII character value of this integer.

```
% format %c 65
A
```

d or i — The argument should be a decimal integer.
Replace the field with the decimal representation of this integer.

```
% format %d 0xff
255
```

u — The argument should be an integer.
Replace the field with the decimal representation of this integer treated as an unsigned value.

```
% format %u -1
4294967295
```

o — The argument should be an decimal integer value.
Replace the field with the octal representation of the argument.

```
% format %o 0xf
17
```

X or x — The argument should be a decimal integer.
Replace the field with the hexadecimal representation of this integer.

```
% format %x -1
ffffffff
```

f — The argument should be a numeric value.

Replace the field with the decimal fraction representation.

```
% format %3.2f 1.234
1.23
```

E or e — The argument should be a numeric value.

Replace the field with the scientific notation representation of this integer.

```
% format %e 0xff
2.550000e+02
```

G or g — The argument should be a numeric value.

Replace the field with the scientific notation or floating-point representation.

```
% format %g 1.234e2
123.4
```

Example 21

```
%  format {%5.3f} [expr 2.0/3]
0.667
%  format {%c%c%c%c%c} 65 83 67 73 73
ASCII
%  set def "%-12s %s"
%  puts [format $def "Author" "Title"]
%  puts [format $def "Clif Flynt" \
     "Tcl/Tk: A Developers Guide"]

Author       Title
Clif Flynt   Tcl/Tk: A Developers Guide
```

The `scan` command is the flip side to `format`. Instead of formatting output, the `scan` command will parse a string according to a format specifier. The `scan` command emulates the behavior of the C `sscanf` function. The first argument must be a string to scan. The next argument is a format description, and the following arguments are a set of variables to receive the data values.

Syntax: `scan` *`textString formatString ?varName1? ... ?varNameN?`*

Parse a text string into one or more variables.

`textString` — The text data to scan for values.

`formatString` — Describes the expected format for the data. The format descriptors are the same as for the `format` command.

`varName*` — The names of variables to receive the data.

The `scan` command returns the number of percent fields that were matched. If this is not the number of percent fields in the `formatString`, it indicates that the `scan` command failed to parse the data.

The format string of the scan command uses the same % descriptors as the format command, and adds a few more.

[...] The value between the open and close square brackets will be a list of characters that can be accepted as matches.

Characters can be listed as a range of characters ([a-z]). A leading or trailing dash is considered a character, not a range marker.

All characters that match these values will be accepted until a nonmatching character is encountered.

```
% scan "a scan test" {%[a-z]} firstword
1
% set firstword
a
```

[^...] The characters after the caret (^) will be characters that cannot be accepted as matches. All characters that do not match these values will be accepted until a matching character is encountered.

```
% scan "a scan test" {%[^t-z]} val
1
% set val
a scan
```

In the following example, the format string {%s %s %s %s} will match four sets of non-whitespace characters (words) separated by whitespace.

Example 22

```
%  set string {Speak, Friend and Enter}
Speak, Friend and Enter
% scan $string {%s %s %s %s} a b c d
4
% puts "The Password is: $b"
The Password is: Friend
```

A format string can also include literal characters that will be included in a format return, or must be matched by the scan command. For instance, the scan command in the previous example could also be written as follows.

```
% scan $string {Speak %s} password
```

This would extract the password from Speak Friend and Enter, but would not extract any words from "The password is sesame", since the format string requires the word *Speak* to be the first word in the string.

String and Format Command Examples

This example shows how you might use some string, scan, and format commands to extract the size, from, and timestamp data from an e-mail log file entry and generate a formatted report line.

Example 23

Script Example

```
# Define the string to parse.
set logEntry {Mar 25 14:52:50 clif sendmail[23755]:
from=<tcl-core-admin@lists.sourceforge.net>,
size=35362, class=-60, nrcpts=1

# Extract "From" using string first and string range
set openAnglePos [string first "<" $logEntry]
set closeAnglePos [string first ">" $logEntry]
set fromField [string range $logEntry $openAnglePos $closeAnglePos]

# Extract the date using scan
scan $logEntry {%s %d %d:%d} mon day hour minute
# Extract the size using scan and string cmds.
set sizeStart [string first "size=" $logEntry]
set substring [string range $logEntry $sizeStart end]

# The formatString looks for a word composed of the
# letters 'eisz' (size will match) followed by an
# equals sign, followed by an integer. The word
# 'size' gets placed in the variable discard,
# and the numeric value is placed in the variable
# sizeField.
scan $substring {%[eisz]=%d} discard sizeField

puts [format {%-12s %-40s %-s} "Timestamp" "From" "Size"]
puts [format {%s %d %d:%d %-40s %d} \
    $mon $day $hour $minute $fromField $sizeField]
```

Script Output

```
Timestamp    From                                     Size
Mar 25 14:52 <tcl-core-admin@lists.sourceforge.net> 35362
```

3.3.4 Lists

A Tcl list can be represented as a string that follows some syntactic conventions. (Internally, a string is represented as a list of pointers to Tcl objects, which are discussed later.)

- A list can be represented as a list of list elements enclosed within curly braces.
- Each word is a list element.
- A set of words may be grouped with curly braces.
- A set of words grouped with curly braces is a list element within the larger list, and also a list in its own right.
- A list element can be empty (it will be displayed as {}).

For example, the string {apple pear banana} can be treated as a list. The first element of this list is apple, the second element is pear, and so on. The order of the elements can be changed with Tcl commands for inserting and deleting list elements, but the Tcl interpreter will not modify the order of list elements as a side effect of another operation.

A list may be arbitrarily long, and list elements may be arbitrarily long. Any string that adheres to these conventions can be treated as a list, but it is not guaranteed that any arbitrary string is a valid list. For example, "this is invalid because of an unmatched brace {" is not a valid list.

With Tcl 8.0, lists and strings are treated differently within the interpreter. If you are dealing with data as a list, it is more efficient to use the list commands. If you are dealing with data as a string, it is better to use the string commands.

The following are valid lists.

```
{This is a six element list}
{This list has {a sublist} in it}
{Lists may {be nested {arbitrarily deep}}}
"A string like this may be treated as a list"
```

The following are invalid lists.

```
{This list has mismatched braces
{This list {also has mismatched braces
```

3.3.5 List Processing Commands

A list can be created in the following ways.

- By using the set command to assign a list to a variable
- By grouping several arguments into a single list element with the list command
- By appending data to an unused variable with the lappend command
- By splitting a single argument into list elements with the split command
- By using a command that returns a list. (The array names command returns a list of associative array indices. It will be discussed in Section 3.3.7.)

The list command takes several units of data and combines them into a single list. It adds whatever braces may be necessary to keep the list members separate.

Syntax: list *element1 ?element2? ... ?elementN?*

Creates a list in which each argument is a list element.

*element** A unit of data to become part of the list

Example 24

```
% set mylist [list first second [list three element sublist] fourth]
first second {three element sublist} fourth
```

The lappend command appends new data to a list, creating and returning a new, longer list. Note that this command will modify the existing list, unlike the string commands, which return new data without changing the original.

Syntax: `lappend` *`listName ?element1? ... ?elementN?`*

Appends the arguments onto a list.

`listName`	The name of the list to append data to.
`element*`	A unit of data to add to the list.

Example 25

```
% lappend mylist fifth
first second {three element sublist} fourth fifth
```

The `split` command returns the input string as a list. It splits the string wherever certain characters appear. By default, the split location is a `whitespace` character: a space, tab, or newline.

Syntax: `split` *`data ?splitChar?`*

Split data into a list.

`data`	The string data to split into a list.
`?splitChar?`	An optional character (or list of characters) at which to split the data.

Example 26

```
% set commaString "1,2.2,test"
1,2.2,test
% # Split on commas
% set lst2 [split $commaString ,]
1 2.2 test
% # Split on comma or period
% set lst2 [split $commaString {,.}]
1 2 2 test
% # Split on empty space between letters
% # (each character becomes a list element)
% set lst2 [split $commaString {}]
1 , 2 . 2 , t e s t
```

Tcl also includes several commands for manipulating lists. These include commands to convert a list into a string, return the number of elements in a list, search a list for elements that match a pattern, retrieve particular elements from a list, and insert and replace elements in a list.

The `join` command converts a list into a string.

Syntax: `join` *`list ?separator?`*

Joins the elements of a list into a string.

`list`	The list to convert to a string.
`separator`	An optional string that will be used to separate the list elements. By default, this is a space.

The join command can be used to convert a Tcl list into a comma-delimited list for import into a spreadsheet.

Example 27

```
% set numbers [list 1 2 3 4]
1 2 3 4
% join $numbers :
1:2:3:4
% join $numbers ", "
1, 2, 3, 4
```

The llength command returns the number of list elements in a list. Note that this is not the number of characters in a list, but the number of list elements. List elements may be lists themselves. These lists within a list are each counted as a single list element.

Syntax: llength *list*

Returns the length of a list.

list The list.

Example 28

```
% set mylist [list first second [list three element sublist] fourth]
first second {three element sublist} fourth
% llength $mylist
4
```

The lsearch command will search a list for elements that match a pattern. The lsearch command uses the glob-matching rules by default. These are described with the previous string match discussion. The regular expression rules are discussed in Chapter 5.

Syntax: lsearch *?-option? list pattern*

Returns the index of the first list element that matches *pattern* or -1 if no element matches the pattern. The first element of a list has an index of 0.

?-option? Controls how the lsearch command will behave. Multiple options may be used together to control the behavior of the lsearch command. Some of the modifiers include:

- -exact List element must exactly match the pattern.
- -glob List element must match pattern using the glob rules. This is the default matching algorithm.
- -regexp List element must match pattern using the regular expression rules.

`-all`	Return all the values that match the pattern, instead of only the first element.
`-inline`	Return the element instead of the index of the element.
`-start` *`position`*	Return the first value after *`position`* that matches the pattern.
`-not`	Inverts the test and returns values that do not match the pattern.

`list` The list to search.

`pattern` The pattern to search for.

Example 29

```
% set mylist [list first second [list three element sublist] fourth]
first second {three element sublist} fourth
% lsearch $mylist second
1
% # three is not a list element - it's a part of a list element
% lsearch $mylist three
-1
% lsearch $mylist "three*"
2
% lsearch $mylist "*ou*"
3
% lsearch -all $mylist "*s*"
0 1 2
% lsearch -all -inline $mylist "*s*"
first second {three element sublist}
% lsearch -start 1 -all -inline $mylist "*s*"
second {three element sublist}
% lsearch -not $mylist "*s*"
3
```

You can extract elements from a list with the `lindex` command.

Syntax: `lindex` *`list index`*

Returns a list element. The first element is element 0. If this value is larger than the list, an empty string is returned.

`list` The list.

`index` The position of a list entry to return.

Example 30

```
% set mylist [list first second [list three element sublist] fourth]
first second {three element sublist} fourth
```

```
% lindex $mylist 0
first
% lindex $mylist 2
three element sublist
% lindex $mylist [lsearch $mylist *ou*]
fourth
```

The `linsert` command returns a new list with the new elements inserted. It does not modify the existing list.

Syntax: `linsert` *`list position element1 ... ?elementN?`*

Inserts an element into a list at a given position.

`list`	The list to receive new elements.
`position`	The position in the list at which to insert the new list elements. If this value is `end` or greater than the number of elements in the list, the new values are added at the end of the list.
`element*`	One or more elements to be inserted into the list.

Example 31

```
% set mylist [list first second [list three element sublist] fourth]
first second {three element sublist} fourth
% set longerlist [linsert $mylist 0 zero]
zero first second {three element sublist} fourth
% puts $mylist
first second {three element sublist} fourth
```

Like `linsert`, the `lreplace` command returns a new list. It does not modify the existing list. The difference between the *`first`* and *`last`* elements need not match the number of elements to be inserted. This allows the `lreplace` command to be used to increase or decrease the length of a list.

Syntax: `lreplace` *`list first last element1 ... ?elementN?`*

Replaces one or more list elements with new elements.

`list`	The list to have data replaced.
`first`	The first position in the list at which to replace elements. If this value is `end`, the last element will be replaced. If the value is greater than the number of elements in the list, an error is generated.
`last`	The last element to be replaced.
`element*`	Zero or more elements to replace the elements between the *`first`* and *`last`* elements.

Example 32

```
% set mylist [list first second [list three element sublist] fourth]
first second {three element sublist} fourth
% set newlist [lreplace $mylist 0 0 one]
one second {three element sublist} fourth
% set shortlist [lreplace $mylist 0 1]
{three element sublist} fourth
```

The next example demonstrates using the list commands to split a set of comma and newline delimited data (a common export format for spreadsheet programs) into a Tcl list and then reformat the data for display.

Example 33

List Commands Example

```
# Define the raw data

set rawData {Package,Major,Minor,Patch
Tcl,8,6,0
math::geometry,1,0,3
math::complexnumbers,1,0,2}

# Split the raw data into a list using the newlines
#     as list element separators.
# This creates a list in which each line becomes a
#    list element
set dataList [split $rawData "\n"]

# Create a list of the column headers.
set columns [split [lindex $dataList 0] ","]

foreach line [lrange $dataList 1 end] {
  # Convert the line of data into a list
  set rowValues [split $line ","]\

  # Create a new list from $rowValues that includes
  # all the elements after the first (package name).
  set revList [lrange $rowValues 1 end]

  # and rejoin them into a "." separated string
  set revision [join $revList "."]

  # Display a reformatted version of the data line
```

```
    puts [format "%s: %s Revision: %s" [lindex $columns 0]\
        [lindex $rowValues 0] $revision]
}
```

Script Output

```
Package: Tcl Revision: 8.6.0
Package: math::geometry Revision: 1.0.3
Package: math::complexnumbers Revision: 1.0.2
```

3.3.6 Dictionaries

The `dict` command was added to Tcl in version 8.5. Conceptually, a dictionary is an ordered list of key-value pairs. A dict looks like this:

```
puts [dict create key1 val1 key2 val2]

key1 val1 key2 val2
```

The concept of key-value pairs is simple enough that you may be tempted to just write procedures to handle such lists. Writing procedures to handle lists of key-value pairs will be demonstrated in Chapter 6.

Unless you are constrained to use versions of Tcl that don't have dictionary support, you should use the `dict` command. The improvements of the dictionary over home-grown keyed-list procedures are:

- the dict supports nesting dictionaries within a dictionary (just as lists can be nested).
- the `dict` command is implemented in fast C code with hash tables.
- there is a rich set of `dict` subcommands to search and manipulate dictionaries.
- the `dict` command contains both field and value information, making it useful as a data construct to use with a database extension.

As with Tcl lists, a dict variable can be initialized in several ways including `dict create` , `dict append` and `dict lappend`.

The `dict create` command creates a complete dictionary without assigning the value to a variable, just as you might use the `format` command to initialize a string.

Syntax: `dict create` *`key1 val1 key2 val2 ...`*

Create a dictionary of keys and values.

`key*` A key in the dictionary.

`val*` The value to associate with the previous key.

The next example creates a dictionary of movie titles and notable quotes. If you print the contents of the `$quotes` variable it will look just like a list of movie titles and quotes.

Example 34

Creating a Dict

```
set quotes [dict create \
  "Casablanca" "Play it, Sam." \
  "Star Wars"  "I get a bad feeling about this." \
  "Indiana Jones" "Snakes, Why did it have to be snakes." \
  "Looney Tunes" "What's Up Doc?"]

set movie Casablanca
puts "A notable quote from $movie is: [dict get $quotes $movie]"
```

Script Output

```
A notable quote from Casablanca is: Play it, Sam.
```

In the previous example, the values associated with each key are a grouped set of words, but there is no higher level of grouping. If we want each quote to be a single unit (so we can have multiple quotes for some movies), the `dict create` command will need to add the grouping as shown in the next example.

Example 35

Creating a Dict of Quotes

```
set quotes [dict create \
  "Casablanca" [list "Play it, Sam." "Round up the usual suspects"] \
  "Star Wars"  [list "I get a bad feeling about this."] \
  "Indiana Jones" [list "Snakes, Why did it have to be snakes."]]

set movie Casablanca
puts "A notable quote from $movie is: \
    [lindex [dict get $quotes $movie] 0]"
```

Script Output

```
A notable quote from Casablanca is: Play it, Sam.
```

The `dict append` and `dict lappend` commands will modify an existing dict variable, or create a new variable if the variable did not previously exist. This is the same way that the `append` and `lappend` commands work with string and list variables.

Syntax: `dict append` *`dictName key value`*

Appends the given value to the given key.

`dictName`	Name of the variable to be modified.
`key`	Key within this dictionary to be modified.
`value`	Value to append onto the current value of this key.

Syntax: `dict lappend` *`dictName key value`*

Appends the given value to the given key.

`dictName`	Name of the variable to be modified.
`key`	Key within this dictionary to be modified.
`value`	Value to lappend onto the current value of this key.

Example 36

Modifying a Dict

```
# Add a new quote to the Star Wars quotes
dict lappend quotes "Star Wars" "Feel the Force, Luke."

# Create a new entry for ET quotes.
dict lappend quotes "ET" "E.T. Phone Home."

set movie "Star Wars"
puts "Notable quotes from $movie include:"
foreach quote [dict get $quotes $movie] {
  puts "  $quote"
}
```

Script Output

```
Notable quotes from Star Wars include:
I get a bad feeling about this.
Feel the Force, Luke.
```

Because the values in this dictionary are lists, modifying an individual list element must be done by extracting the value from the dictionary, modifying it and replacing it. For example, the complete *Casablanca* quote is "Play it, Sam. Play *As Time Goes By.*"

The `dict replace` command will let us replace a value with a new value. Like the `lreplace` command, this command does not replace the value in the dictionary, it returns a new dictionary with the value modified.

Syntax: `dict replace` *`$dict key new_value`*

Return a new dictionary with the value associated with a key replaced with a new value.

`$dict`	The dictionary to be modified.
`key`	The key to be modified.
`new_value`	The new value for this key.

Example 37

Modifying a Dict Value

```
set vals [dict get $quotes Casablanca]
set val [lindex $vals 0]
append val "  Play 'As Time Goes By'."
set vals [lreplace $vals 0 0 $val]
set quotes [dict replace $quotes Casablanca $vals]

set movie Casablanca
puts "The full quote from $movie is: \
    [lindex [dict get $quotes $movie] 0]"
```

Script Output

```
The full quote from Casablanca is:
Play it, Sam. Play 'As Time Goes By'.
```

A dictionary value can be modified in place with the `dict set` command.

Syntax: `dict set` *`dictName key new_value`*

Sets the contents of a dictionary key to a new value.

`$dict`	The name of the dictionary to be modified.
`key`	The key to be modified.
`new_value`	The new value for this key.

The previous example—adding a string to one movie quote—becomes a bit simpler by using the `dict set` command instead of `dict replace`.

Example 38

Modify a Dict Value

```
set vals [dict get $quotes Casablanca]
set val [lindex $vals 0]
append val "  Play 'As Time Goes By'."
set vals [lreplace $vals 0 0 $val]
dict set quotes Casablanca $vals

set movie Casablanca
puts "The full quote from $movie is: \
    [lindex [dict get $quotes $movie] 0]"
```

Script Output

```
The full quote from Casablanca is:
Play it, Sam. Play 'As Time Goes By'.
```

3.3.7 Associative Arrays

The associative array is an array that uses a string to index the array elements, instead of a numeric index the way C, FORTRAN, and BASIC implement arrays. A variable is denoted as an associative array by placing an index within parentheses after the variable name.

For example, `price(apple)` and `price(pear)` would be associative array variables that could contain the price of an apple or pear. The associative array is a powerful construct in its own right and can be used to implement composite data types resembling the C struct or even a C++ class object. Using associative arrays is further explored in Chapter 6.

Example 39

`set price(apple) .10`	`price` is an associative array. The element referenced by the index `apple` is set to .10.
`set price(pear) .15`	`price` is an associative array. The element referenced by the index `pear` is set to .15.
`set quantity(apple) 20`	`quantity` is an associative array. The element referenced by the index `apple` is set to 20.
`set discount(12) 0.95`	`discount` is an associative array. The element referenced by the index `12` is set to 0.95.

3.3.8 Associative Array Commands

An array element can be treated as a simple Tcl variable. It can contain a number, string, list, or even the name of another array element. As with lists, there is a set of commands for manipulating associative arrays. You can get a list of the array indices, get a list of array indices and values, or assign many array indices and values in a single command. Like the `string` commands, the `array` commands are arranged as a set of subcommands of the `array` command.

The list of indices returned by the `array names` command can be used to iterate through the content of an array.

Syntax: `array names` *`arrayName ?pattern?`*

Returns a list of the indices used in this array.

`arrayName`	The name of the array.
`pattern`	If this option is present, `array names` will return only indices that match the pattern. Otherwise, `array names` returns all the array indices.

Example 40

```
set fruit(apples) 10
set fruit(pears) 5
foreach item [array names fruit *] {
  puts "There are $fruit($item) $item."
}
```

```
There are 5 pears.
There are 10 apples.
```

The `array get` command returns the array indices and associated values as a list. The list is a set of pairs in which the first item is an array index and the second is the associated value. The third list element will be another array index (first item in the next pair), the fourth will be the value associated with this index, and so on.

Syntax: `array get` *arrayName*

Returns a list of the indices and values used in this associative array.

arrayName The name of the array.

Example 41

```
% array get fruit
pears 5 apples 10
```

The `array set` command accepts a list of values in the format that `array get` generates. The `array get` and `array set` pair of commands can be used to copy one array to another, save and restore arrays from files, and so on.

Syntax: `array set` *arrayName* {*index1 value1 ... indexN valueN*}

Assigns each value to the appropriate array index.

arrayName The name of the array.

*index** An index in the array to assign a value to.

*value** The value to assign to an array index.

Example 42

```
% array set fruit [list bananas 20 peaches 40]
% array get fruit
bananas 20 pears 5 peaches 40 apples 10
```

The next example shows some simple uses of an array. Note that while the Tcl array does not explicitly support multiple dimensions the index is a string and you can define multidimensional arrays by using a naming convention, such as separating fields with a comma, period, dash, and so on, that does not otherwise appear in the index values.

Example 43

Array Example

```
# Initialize some values with set
set fruit(apple.cost) .10
set fruit(apple.count) 5
```

```
# Initialize some more with array set
array set fruit {pear.cost .15 pear.count 3}
# At this point the array contains:

# Index        Value
# apple.cost   .10
# pear.cost    .15
# apple.count  5
# pear.count   3

# Count the number of different types of fruit in the
# array by getting a list of unique indices, and then
# using the llength command to count the number of
# elements in the list.
set typeCount [llength [array names fruit *cost]]
puts "There are $typeCount types of fruit in the fruit array"
# You can use another variable to hold all,
# or a part of an array index.
set type apple
puts "There are $fruit($type.count) apples"
set type pear
puts "There are $fruit($type.count) pears"
array set new [array get fruit]
set type pear
puts "There are $new($type.count) pears in the new array"
```

Script Output

```
There are 2 types of fruit in the fruit array
There are 5 apples
There are 3 pears
There are 3 pears in the new array
```

3.3.9 Binary Data

Versions of Tcl prior to 8.0 (pre 1998) used null-terminated ASCII strings for the internal data representation. This made it impossible to use Tcl with binary data that might have NULLs embedded in the data stream.

With version 8.0, Tcl moved to a new internal data representation that uses a native-mode data representation. An integer value is saved as a long integer, a real value is saved as an IEEE floating point value, and so on. The new method of data representation supports binary data, and a command was added to convert binary data to integers, floats, or strings. Tcl is still oriented around printable ASCII strings but the binary command makes it possible to handle binary data easily.

The `binary` command supports two subcommands to convert data to and from a binary representation. The `format` subcommand will transform an ASCII string to a binary value, and the `scan` subcommand will convert a string of binary data to one or more printable Tcl variables. These subcommands require a descriptor to define the format of the binary data. Examples of these descriptors follow the command syntax.

Syntax: `binary format` *`format arg1 ?arg2? ... ?argN?`*

Returns a binary string created by converting one or more printable ASCII strings to binary format.

`format` A string that describes the format of the ASCII data.

`arg*` The printable ASCII to convert.

Syntax: `binary scan` *`binaryData format arg1 ?varName1? ... ?varNameN?`*

Converts a string of binary data to one or more printable ASCII strings.

`binaryData` The binary data.

`format` A string that describes the format of the ASCII data.

`varName*` Names of variables to accept the printable representation of the binary data.

The components of *`format`* are similar to the format strings used by `scan` and `format` in that they consist of a descriptor (a letter) and an optional count. If the count is defined, it describes the number of items of the previous type to convert. The count defaults to 1. Common descriptors include the following.

Example 44

Script Example

```
binary scan "Tk" H4 x
puts "X: $x"
# Assign three integer values to variables
set a 1415801888
set b 1769152615
set c 1919246708
# Convert the integers to ASCII equivalent
puts [binary format {I I2} $a [list $b $c]]
```

Script Output

```
X: 546b
Tcl is great
```

The string used to define the format of the binary data is very powerful, and allows a script to extract fields from complex C structures.

h Converts between binary and hexadecimal digits in little endian order.

`binary format h2 34` - returns "C" (0x43).

`binary scan "4" h2 x` - stores 0x43 in the variable x

H Converts between binary and hexadecimal digits in big endian order.

`binary format H2 34` - returns "4" (0x34).

`binary scan "4" H2 x` - stores 0x34 in the variable x

c Converts an 8-bit value to/from ASCII.
`binary format c 0x34` - returns “4” (0x34).
`binary scan "4" c x` - stores 0x34 in the variable x

s Converts a 16-bit value to/from ASCII in little endian order.
`binary format s 0x3435` - returns “54” (0x35 0x34).
`binary scan "45" s x` - stores 13620 (0x3534) in the variable x

S Converts a 16-bit value to/from ASCII in big endian order.
`binary format S 0x3435` - returns ”45” (0x34 0x35).
`binary scan "45" S x` - stores 13365 (0x3435) in the variable x

i Converts a 32-bit value to/from ASCII in little endian order.
`binary format i 0x34353637` - returns ”7654” (0x37 0x36 0x35 0x34).
`binary scan "7654" i x` - stores 875902519 (0x34353637) in the variable x

I Converts a 32-bit value to/from ASCII in big endian order.
`binary format I 0x34353637` - returns ”4567” (0x34 0x35 0x36 0x37).
`binary scan "7654" I x` - stores 926299444 (0x37363534) in the variable x

f Converts 32-bit floating point values to/from ASCII.
`binary format f 1.0` - returns the binary string ”0x00803f”.
`binary scan "0x00803f" f x` - stores 1.0 in the variable x

Example 45
Script Examples

C Code to Generate a Structure

```
#include <stdio.h>
#include <fcntl.h>
main () {
  struct a {
    int i;
    float f[2];
    char s[20];
    } aa;
    FILE *of;

    aa.i = 100;
    aa.f[0] = 2.5;
    aa.f[1] = 3.8;

    strcpy(aa.s, "This is a test");
```

Tcl Code to Read the Structure

```
# Open the input file, and read data
set if [open tstStruct r]
set d [read $if]
close $if

# scan the binary data into variables.

binary scan $d "i f2 a*" i f s
# The string data includes any binary
# garbage after the NULL byte.
# Strip off that junk.

set 0pos [string first\
   [binary format c 0x00] $s]
incr 0pos -1
set s [string range $s 0 $0pos]
```

```
    of = fopen("tstStruct", "w");
    fwrite(sizeof(aa), 1, of);
    fclose(of);
}
```

```
# Display the results
puts $i
puts $f
puts $s
```

Script Output

```
100
2.5 3.79999995232
This is a test
```

3.3.10 Handles

Tcl uses handles to refer to certain special-purpose objects. These handles are returned by the Tcl command that creates the object and can be used to access and manipulate the object. When you open a file, a handle is returned for accessing that file. The graphic objects created by a `wish` script are also accessed via handles, which will be discussed in the `wish` tutorial. The following are types of handles.

`channel` A handle that references an I/O device such as a file, serial port, or TCP socket. A `channel` is returned by an `open` or `socket` call and can be an argument to a `puts`, `read`, `close`, `flush`, or `gets` call.

`graphic` A handle that refers to a graphic object created by a `wish` command. This handle is used to modify or query an object.

`http` A handle that references data returned by an `http::geturl` operation. An `http` handle can be used to access the data that was returned from the `http::geturl` command or otherwise manipulate the data.

There will be detailed discussion of the commands to manipulate handles in sections that discuss that type of handle.

3.4 ARITHMETIC AND BOOLEAN OPERATIONS

The commands discussed so far directly manipulate particular types of data. Tcl also has a rich set of commands for performing arithmetic and Boolean operations and for using the results of those operations to control program flow.

3.4.1 Math Operations

Math operations are performed using the `expr` and `incr` commands. The `expr` command provides an interface to a general-purpose calculation engine, and the `incr` command provides a fast method of changing the value of an integer.

The `expr` command will perform arbitrarily complex math operations. Unlike most Tcl commands, `expr` does not expect a fixed number of arguments. It can be invoked with the arguments grouped as a string or as individual values and operators. The `expr` command is optimized to handle arithmetic

strings. You will see a performance improvement if you pass the `expr` command a string enclosed in curly braces, rather than using quotes which pushes some processing into the Tcl interpreter, instead of the `expr` command code.

The arithmetic arguments to `expr` may be grouped with parentheses to control the order of math operations. The `expr` command can also evaluate Boolean expressions and is used to test conditions by the Tcl branching and looping commands.

Syntax: `expr` *`mathExpression`*

Tcl supports the following math operations (grouped in decreasing order of precedence).

$-+\sim!$	Unary minus, unary plus, bitwise NOT, logical NOT.
$*/\%$	Multiply, divide, modulo (return the remainder).
$+-$	Add, subtract.
$\ll\gg$	Left shift, right shift.
$<><=>=$	Less than, greater than, less than or equal, greater than or equal.
$==!=$	Equality, inequality.
&	Bitwise AND.
^	Bitwise exclusive OR.
|	Bitwise OR.
&&	Logical AND. Produces a 1 result if both operands are nonzero; 0 otherwise.
||	Logical OR. Produces a 0 result if both operands are zero; 1 otherwise.
x?y:z	If-then-else, as in C. If *x* evaluates to nonzero, the result is the value of *y*. Otherwise, the result is the value of *z*. The *x* operand must have a numeric value. The *y* and *z* operands may be variables or Tcl commands.

Note that the bitwise operations are valid only if the arguments are integers (not floating-point or scientific notation). The `expr` command also supports the following math functions and conversions.

Trigonometric Functions

sin	`sin(`*`radians`*`)`
	`set sin [expr sin($degrees/57.32)]`
cosine	`cos (`*`radians`*`)`
	`set cosine [expr cos(3.14/2)]`
tangent	`tan (`*`radians`*`)`
	`set tan [expr tan($degrees/57.32)]`
arcsin	`asin (`*`float`*`)`
	`set angle [expr asin(.7071)]`

arccosine	`acos(`*float*`)`
	`set angle [expr acos(.7071)]`
arctangent	`atan(`*float*`)`
	`set angle [expr atan(.7071)]`
hyperbolic sin	`sinh(`*radians*`)`
	`set hyp_sin [expr cosh(3.14/2)]`
hyperbolic cosine	`cosh(`*radians*`)`
	`set hyp_cos [expr cosh(3.14/2)]`
hyperbolic tangent	`tanh(`*radians*`)`
	`set hyp_tan [expr tanh(3.14/2)]`
hypotenuse	`hypot(`*float, float*`)`
	`set len [expr hypot($side1, $side2)]`
arctangent of ratio	`atan2(`*float, float*`)`
	`set radians [expr atan2($numerator, $denom)]`

Exponential Functions

natural log	`log(`*float*`)`
	`set two [expr log(7.389)]`
log base 10	`log10(`*float*`)`
	`set two [expr log10(100)]`
square root	`sqrt(`*float*`)`
	`set two [expr sqrt(4)]`
exponential	`exp(`*float*`)`
	`set seven [expr exp(1.946)]`
power	`pow(`*float, float*`)`
	`set eight [expr pow(2, 3)]`

Conversion Functions

Return closest int	`round(`*float*`)`
	`set duration [expr round($distance / $speed)]`
Largest integer less than a float	`floor(`*float*`)`
	`set overpay [expr ceil($cost / $count)]`
Smallest integer greater than a float	`ceil(`*float*`)`
	`set each [expr ceil($cost/$count)]`
Floating Point remainder	`fmod(`*float, float*`)`
	`set missing [expr fmod($cost, $each)]`

Convert int to float	`double(`*`int`*`)`
	`set average [expr $total / double($count)]`
Convert float to int	`int(`*`float`*`)`,
	`set leastTen [expr (int($total) / 10) * 10]`
Absolute Value	`abs(`*`num`*`)`
	`set xDistance [expr abs($x1 - $x2)]`

Random Numbers

Seed random number	`srand(`*`int`*`)`
	`expr srand([clock seconds])`
Generate random number	`rand()`
	`set randomFloat [expr rand() ]`

Example 46

```
% set card [expr rand()]
0.557692307692
% set cardNum [expr int($card * 52)]
29
% set cardSuit [expr int($cardNum / 13)]
2
% set cardValue [expr int($cardNum % 13)]
3
% expr floor(sin(3.14/2) * 10)
9.0
% set x [expr int(rand() * 10)]
4
% expr atan(((3 + $x) * $x)/100.)
0.273008703087
```

The `incr` command provides a shortcut to modify the content of a variable that contains an integer value. The `incr` command adds a value to the current content of a given variable. The value may be positive or negative, thus allowing the `incr` command to perform a decrement operation. The `incr` command is used primarily to adjust loop variables.

Syntax: `incr` *`varName ?incrValue?`*

`incr`	Add a value (default 1) to a variable.
`varName`	The name of the variable to increment. NOTE: This is a variable name, not a value. Do not start the name with a `$`. This variable must contain an integer value, not a floating-point value.
`?incrValue?`	The value to increment the variable by. May be a positive or negative number. The value must be an integer between −65,536 and 65,535, not a floating-point value.

Example 47

```
% set x 4
4
% incr x
5
% incr x -3
2
% set y [incr x]
3
% puts "x: $x y: $y"
x: 3 y: 3
```

3.4.2 Conditionals

The `if` Command

Tcl supports both a single-choice conditional (`if`) and a multiple-choice conditional (`switch`). The `if` command tests a condition, and if that condition is true, the script associated with this test is evaluated. If the condition is not true, an alternate choice is considered, or alternate script is evaluated.

Syntax: `if {`*`testExpression1`*`} {`
`body1`
`} ?elseif {`*`testExpression2`*`} {`
`body2`
`}? ?else {`
`bodyN`
`}?`

`if`	Determine whether a code body should be evaluated based on the results of a test. If the test returns true, the first body is evaluated. If the test is false and a body of code exists after the `else`, that code will be evaluated.
`testExpression1`	If this expression evaluates to true, the first body of code is evaluated. The expression must be in a form acceptable to the `expr` command. These forms include the following. An arithmetic comparison `{$a < 2}`. A string comparison `{ $string` ***`ne`*** `"OK"}`. The results of a command `{ [eof $inputFile]}`.

	A variable with a numeric value. Zero (0) is considered false, and nonzero values are true.
body1	The body of code to evaluate if the first test evaluates as true.
`elseif`	If *testExpression1* is false, evaluate *testExpression2*.
testExpression2	A second test to evaluate if the first test evaluates to false.
body2	The body of code to evaluate if the second test evaluates as true.
`?else` *bodyN?*	If all tests evaluate false, this body of code will be evaluated.

In the following example, note the placement of the curly braces ({}). The *Tcl Style Guide* describes the preferred format for `if`, `for`, `proc`, and `while` commands. It recommends that you place the left curly brace of the body of these commands on the line with the command and place the body on the next lines, indented two spaces. The final, right curly brace should go on a line by itself, indented even with the opening command. This makes the code less dense (and more easily read).

Putting the test and action on a single line is syntactically correct Tcl code but can cause maintenance problems later. You will need to make some multiline choice statements, and mixing multiline and single-line commands can make the action statements difficult to find. Also, what looks simple when you start writing some code may need to be expanded as you discover more about the problem you are solving. It is recommended practice to lay out your code to support adding new lines of code.

A Tcl command is normally terminated by a newline character. Thus, a left curly brace must be at the end of a line of code, not on a line by itself. Alternatively, you can write code with the new line escaped, and the opening curly brace on a new line, but this style makes code difficult to maintain.

Example 48

A Simple Test

```
set x 2
set y 3
if {$x < $y} {
  puts "x is less than y"
}
```

Script Output

```
x is less than y
```

The `switch` Command

The `switch` command allows a Tcl script to choose one of several patterns. The `switch` command is given a variable to test and several patterns. The first pattern that matches the test phrase will be evaluated, and all other sets of code will not be evaluated.

Syntax: `switch` *`?opt? str pat1 bod1 ?pat2 bod2 ...? ?default defaultBody?`*

Evaluate 1 of N possible code bodies, depending on the value of a string.

`?opt?` One of the following possible options.

- `-exact` Match a pattern string exactly to the test string, including a possible "`-`" character.
- `-glob` Match a pattern string to the test string using the `glob` string match rules. These are the default matching rules.
- `-regexp` Match a pattern string to the test string using the regular expression string match rules.
- `--` Absolutely the last option. The next string will be the string argument. This allows strings that start with a dash (`-`) to be used as arguments without being interpreted as options.

`str` The string to match against patterns.

`patN` A pattern to compare with the `string`.

`bodN` A code body to evaluate if `patN` matches `string`.

`default` A pattern that will match if no other patterns have matched.

`defaultBody` The script to evaluate if no other patterns were matched.

The options `-exact`, `-glob`, and `-regexp` control how the `string` and `pattern` will be compared. By default, the `switch` command matches the patterns using the glob rules described previously with `string match`.

You can use regular expression match rules by including the `-regexp` flag. The regular expression rules are similar in that they allow you to define a pattern of characters in a string but are more complex and more powerful. The `regexp` command, which is used to evaluate regular expressions, is discussed in Chapter 5. The `switch` command can also be written with curly braces around the patterns and body.

Syntax:
```
switch ?option? string {
    pattern1 body1
    ?pattern2 body2?
    ?default defaultBody?
}
```

When the `switch` command is used without braces (as shown in the first `switch` statement below that follows), the `pattern` strings may be variables, allowing a script to modify the behavior of a `switch` command at runtime. When the braces are used (the second example following), the `pattern` strings must be hard-coded patterns.

Example 49

Script Example

```
set x 7
set y 7
# Using no braces substitution occurs before the switch
# command looks for matches.
# Thus a variable can be used as a match pattern:

switch $x \
  $y {puts "X=Y"} \
  {[0-9]} {puts "< 10"} \
  default {puts "> 10"}

# With braces, the $y is not substituted to 7, and switch looks
# for a match to the literal string $y

switch -glob $x {
  "1" {puts "one"}
  "2" {puts "two"}
  "3" {puts "three"}
  "$y" {puts "X=Y"}
  {[4-9]} {puts "greater than 3"}
  default {puts "Not a value between 1 and 9"}
}
```

Script Output

```
X=Y
greater than 3
```

If you wish to evaluate the same script when more than one pattern is matched, you can use a dash (`-`) in place of the body to cause the `switch` command to evaluate the next body, instead of the body associated with the current pattern. Part of a folk music quiz might resemble the following.

Example 50

Script Example

```
puts "Who recorded 'Mr Tambourine Man' "
gets stdin artist  ;# User types Bob Dylan
switch $artist {
  {Bob Dylan} -
  {Judy Collins} -
  {Glen Campbell} -
  {William Shatner} -
  {The Chipmunks} -
  {The Byrds} {
```

```
    puts "$artist recorded 'Mr Tambourine Man' "
  }
  default {
    puts "$artist probably recorded 'Mr Tambourine Man' "
  }
}
```

Script Output

```
Who recorded 'Mr Tambourine Man'
Bob Dylan
Bob Dylan recorded 'Mr Tambourine Man'
```

3.4.3 Looping

Tcl provides commands that allow a script to loop on a counter, loop on a condition, or loop through the items in a list. These three commands are as follows.

`for`	A numeric loop command
`while`	A conditional loop command
`foreach`	A list-oriented loop command

The `for` *Command*

The `for` command is the numeric loop command.

Syntax: `for` *`start test next body`*

Set initial conditions and loop until the *`test`* fails.

`start`	Tcl statements that define the start conditions for the loop.
`test`	A statement that tests an end condition. This statement must be in a format acceptable to `expr`.
`next`	A Tcl statement that will be evaluated after each pass through the loop. Normally this increments a counter.
`body`	The body of code to evaluate on each pass through the loop.

The `for` command is similar to the looping `for` in C, FORTRAN, BASIC, and others. The `for` command requires four arguments; the first (*`start`*) sets the initial conditions, the next (*`test`*) tests the condition, and the third (*`next`*) modifies the state of the test. The last argument (*`body`*) is the body of code to evaluate while the *`test`* returns true.

Example 51

Script Example

```
for {set i 0} {$i < 2} {incr i} {
  puts "I is: $i"
}
```

Script Output

```
I is: 0
I is: 1
```

The `while` *Command*

The `while` command is used to loop until a test condition becomes false.

Syntax: `while` *`test body`*

Loop until a condition becomes false.

`test` A statement that tests an end condition. This statement must be in a format acceptable to `expr`.

`body` The body of code to evaluate on each pass through the loop.

Example 52

While Loop Example

```
set x 0;
while {$x < 5} {
  set x [expr $x+$x+1]
  puts "X: $x"
}
```

Script Output

```
X: 1
X: 3
X: 7
```

The `foreach` *Command*

The `foreach` command is used to iterate through a list of items.

Syntax: `foreach` *`listVar list body`*

Evaluate *`body`* for each of the items in *`list`*.

`listVar` This variable will be assigned the value of the list element currently being processed.

`list` A list of data to step through.

`body` The body of code to evaluate on each pass through the loop.

Example 53

Script Example

```
set total 0
foreach num {1 2 3 4 5} {
  set total [expr $total + $num]
}
puts "The total is: $total"
```

Script Output

```
The total is: 15
```

With Tcl release 7.5 (1996) and later, the `foreach` command was extended to handle multiple sets of list variables and list data.

Syntax: `foreach` *`valueList1 dataList1 ?valueList2 dataList2?...`* {
`body`
}

If the *`valueList`* contains more than one variable name, the Tcl interpreter will take enough values from the *`dataList`* to assign a value to each variable on each pass. If the *`dataList`* does not contain an even multiple of the number of *`valueList`* elements, the variables will be assigned an empty string.

Example 54

Script Example

```
foreach {pres date} { {George Washington} {1789-1797}
                      {John Adams} {1797-1801}
                      {Thomas Jefferson} {1801-1809}
                      {James Madison} {1809-1817}
                      {James Monroe} {1817-1825}
        } state {Virginia Massachusetts Virginia Virginia Virginia} {
  puts "$pres was from $state and served from $date"
}
```

Script Output

```
George Washington was from Virginia and served from 1789–1797
John Adams was from Massachusetts and served from 1797–1801
Thomas Jefferson was from Virginia and served from 1801–1809
James Madison was from Virginia and served from 1809–1817
James Monroe was from Virginia and served from 1817–1825
```

3.4.4 Exception Handling in Tcl

When the Tcl interpreter hits an exception condition the default action is to halt the execution of the script and display the data in the `errorInfo` global variable. The information in `errorInfo` will describe the command that failed and will include a stack dump for all the procedures that were in process when this failure occurred. The simplest method of modifying this behavior is to use the `catch` command to intercept the exception condition before the default error handler is invoked.

Syntax: `catch` *`script ?varName?`*

Catch an error condition and return the results rather than aborting the script.

`script` The Tcl script to evaluate.

`varName` Variable to receive the results of the script.

The `catch` command catches an error in a script and returns a success or failure code rather than aborting the program and displaying the error conditions. If the script runs without errors, `catch` returns 0. If there is an error, `catch` returns 1, and the `errorCode` and `errorInfo` variables are set to describe the error.

Sometimes a program should generate an exception. For instance, while checking the validity of user-provided data, you may want to abort processing if the data is obviously invalid. The Tcl command for generating an exception is `error`.

Syntax: `error` *`informationalString ?Info? ?Code?`*

`error`	Generate an error condition. If not caught, display the *`informationalString`* and stack trace and abort the script evaluation.
`informationalString`	Information about the error condition.
`Info`	A string to initialize the `errorInfo` string. Note that the Tcl interpreter may append more information about the error to this string.
`Code`	A machine-readable description of the error that occurred. This will be saved in the global `errorCode` variable.

The next example shows some ways of using the `catch` and `error` commands.

Example 55

Script Example

```
proc errorProc {first second} {
  global errorInfo
  # $fail will be non–zero if $first is non–numeric.
  set fail [catch {expr 5 * $first} result]
  # if $fail is set, generate an error
  if {$fail} {
    error "Bad first argument"
  }
  # This will fail if $second is non–numeric or 0
  set fail [catch {expr $first/$second} dummy]
  if {$fail} {
    error "Bad second argument" \
    "second argument fails math test\cback n\$errorInfo"
  }
  error "errorProc always fails" "evaluating error" \
    [list USER {123} {Non–Standard User–Defined Error}]
}
# Example Script
puts "call errorProc with a bad first argument"
set fail [catch {errorProc X 0} returnString]
```

```
if {$fail} {
  puts "Failed in errorProc"
  puts "Return string: $returnString"
  puts "Error Info: $errorInfo\n"
}
puts "call errorProc with a 0 second argument"
if {[catch {errorProc 1 0} returnString]} {
  puts "Failed in errorProc"
  puts "Return string: $returnString"
  puts "Error Info: $errorInfo\n"
}
puts "call errorProc with valid arguments"
set fail [catch {errorProc 1 1} returnString]
if {$fail} {
  if {[string first USER $errorCode] == 0} {
    puts "errorProc failed as expected"
    puts "returnString is: $returnString"
    puts "errorInfo: $errorInfo"
  } else {
    puts "errorProc failed for an unknown reason"
  }
}
```

Script Output

```
call errorProc with a bad first argument
Failed in errorProc
Return string: Bad first argument
Error Info: Bad first argument
  while executing
"error "Bad first argument""
  (procedure "errorProc" line 10)
  invoked from within
"errorProc X 0"
call errorProc with a 0 second argument
Failed in errorProc
Return string: Bad second argument
Error Info: second argument fails math test
divide by zero
  while executing
"expr \$first/\$second"
  (procedure "errorProc" line 15)
  invoked from within
"errorProc 1 0"
call errorProc with valid arguments
errorProc failed as expected
returnString is: errorProc always fails
errorInfo: evaluating error
  (procedure "errorProc" line 1)
```

```
  invoked from within}
"errorProc 1 1"
```

Note the differences in the stack trace returned in `errorInfo` in the error returns. The first, generated with `error` *`message`*, includes the `error` command in the trace, whereas the second, generated with `error` *`message Info`*, does not.

If there is an *`Info`* argument to the `error` command, this string is used to initialize the `errorInfo` variable. If this variable is not present, Tcl uses the default initialization, which is a description of the command that generated the exception. In this case, that is the `error` command. If your application needs to include information that is already in the `errorInfo` variable, you can append that information by including `$errorInfo` in your message, as done with the second test.

The `errorInfo` variable contains what should be human-readable text to help a developer debug a program. The `errorCode` variable contains a machine-readable description to enable a script to handle exceptions intelligently. The `errorCode` data is a list in which the first field identifies the class of error (ARITH, CHILDKILLED, POSIX, and so on), and the other fields contain data related to this error. The gory details are in the on-line manual/help pages under `tclvars`.

If you are used to Java, you are already familiar with the concept of separating data returns from status returns. If your background is C/FORTRAN/BASIC type programming, you are probably more familiar with the C/FORTRAN paradigm of returning status as a function return, or using special values to distinguish valid data from error returns. For example, the C library routines return a valid pointer when successful, and a NULL pointer for failure.

If you want to use function return values to return status in Tcl, you can. Using the `error` command (particularly in low-level procedures that application programs will invoke) provides a better mechanism. The following are reasons for using `error` instead of status returns.

- An application programmer must check a procedure status return. It is easy to forget to check a status return and miss an exception. It takes extra code (the `catch` command) to ignore bad status generated by `error`.
- This makes the fast and dirty techniques for writing code (not checking for status, or not catching errors) the more robust technique. If a low-level procedure has a failure, the intermediate code must propagate the failure to the top level. Doing this with status returns requires special code to propagate the error, which means all functions must adhere to the error-handling policy.
- The `error` command automatically propagates the error. Procedures that use a function that may fail need not include exception propagation code. This moves the policy decisions for how to handle an exception to the application level, where it is more appropriate.

3.5 MODULARIZATION

Tcl has support for all modern software modularization techniques.

- Subroutines (with the `proc` command)
- Multiple source files (with the `source` command)
- Libraries (with the `package` command)

The `package` commands are discussed in detail in Chapter 8.

3.5.1 Procedures

The procedure is the most common technique for code modularization. Tcl procedures:

- Can be invoked recursively.
- Can be defined to accept specific arguments.
- Can be defined to accept arguments that have default values.
- Can be defined to accept a variable number of arguments.

The `proc` command defines a Tcl procedure.

Syntax: `proc` *`procName argList body`*

Defines a new procedure.

`procName`	The name of the procedure to define.
`argList`	The list of arguments for this procedure.
`body`	The body to evaluate when this procedure is invoked.

Note how the argument list and body are enclosed in curly braces in the following example. This is the normal way for defining a procedure, since you normally do not want any substitutions performed until the procedure body is evaluated. Procedures are discussed in depth in Chapter 7.

Example 56

Proc Example

```
# Define the classic recursive procedure to find the
# n'th position in a Fibonacci series.
proc fib {num} {
  if {$num <= 2} {return 1}
  return [expr [fib [expr $num -1]] + [fib [expr $num -2]] ]
}
for {set i 1} {$i < 6} {incr i} {
  puts "Fibonacci series element $i is: [fib $i]"
}
```

Script Output

```
fibonacci series element 1 is: 1
fibonacci series element 2 is: 1
fibonacci series element 3 is: 2
fibonacci series element 4 is: 3
fibonacci series element 5 is: 5
```

3.5.2 Loading Code from a Script File

Splitting functionality into separate files, so that each file contains closely related procedures, makes code easier to maintain. The `source` command loads a file into an existing Tcl script. It is similar to the `#include` in C, the `source` in C-shell programming, and the `require` in Perl. This command lets you build source code modules you can load into your scripts when you need particular functionality.

This allows you to modularize your programs. This is the simplest of the Tcl commands that implement libraries and modularization. The `package` command is discussed in Chapter 8.

Syntax: `source` *`fileName`*

Load a file into the current Tcl application and evaluate it.

`fileName` The file to load.

Macintosh users have two options to the `source` command that are not available on other platforms.

Syntax: `source -rsrc resourceName ?fileName?`

Syntax: `source -rsrcid resourceId ?fileName?`

These options allow one script to source another script from a TEXT resource. The resource may be specified by `resourceName` or `resourceID`.

3.5.3 Examining the State of the Tcl Interpreter

Any Tcl script can query the Tcl interpreter about its current state. The interpreter can report whether a procedure or variable is defined, what a procedure body or argument list is, the current level in the procedure stack, and so on. The next examples will only use a few of the `info` subcommands. See the on-line documentation for details of the other subcommands.

Syntax: `info` *`subCommand arguments`*

Provide information about the interpreter state.

`subCommand` Defines the interaction. Interactions include:

`exists varName`	Returns `True` if a variable has been defined.
`proc globPattern`	Returns a list of procedure names that match the `glob` pattern.
`body procName`	Returns the body of a procedure.
`args` *`procName`*	Returns the names of the arguments for a procedure.
`nameofexecutable`	Returns the full path name of the binary file from which the application was invoked.

This example shows a procedure that counts the number of times values appear in a list and returns a list of values and the number of times they occur. It uses the `info exists` command to determine whether or not a value has been found (and counted) yet.

Example 57

Using `info` ***Commands***

```
proc countListElements {lst} {
  # Step through each element in the list
  foreach l $lst {
```

```
    # If the index exists, increment the count
    if {[info exists counts($l)]} {
      incr counts($l)
    } else {
      # If the index did not exist, initialize it
      set counts($l) 1
    }
  }
  return [array get counts]
}

if {[info proc countListElements] eq "countListElements"} {
  set testList {a b a a c b a}
  puts "The countListElements procedure is defined."
  puts "The countListElements procedure returns"
  puts [countList $testList]
  puts "for the list {$testList}"
}
```

Script Output

```
The countListElements procedure is defined.
The countListElements procedure returns
a 4 b 2 c 1
for the list {a b a a c b a}
```

3.6 BOTTOM LINE

This covers the basics of the Tcl language. The next chapter introduces the Tcl I/O calls, techniques for using these commands, and a few more commands.

- Tcl is a position-based language rather than a keyword-based language.
- A Tcl command consists of
  - A command name
  - Optional subcommand, flags, or arguments
  - A command terminator [either a newline or semicolon (;)]
- Words and symbols must be separated by at least one *whitespace* (space, tab, or escaped newline) character.
- Multiple words or variables can be grouped into a single argument with braces ({}) or quotes (" ").
- Substitution will be performed on strings grouped with quotes.
- Substitutions will not be performed on strings grouped with curly braces ({}).
- A Tcl command is evaluated in a single pass.
- The Tcl evaluation routine is called recursively to evaluate commands enclosed within square brackets.

- Some Tcl commands can accept flags to modify their behavior. A flag will always start with a hyphen. It may proceed or follow the arguments (depending on the command) and may require an argument itself.
- Values are assigned to a variable with the `set` command.
  Syntax: `set` *`varName value`*
- Math operations are performed with the `expr` and `incr` commands.
  Syntax: `expr` *`mathExpression`*
  Syntax: `incr` *`varName ?incrValue?`*
- The branch commands are `if` and `switch`.
  Syntax: `if` {*`test`*} {*`bodyTrue`*} *`?elseif`* {*`test2`*} {*`body2`*}*`?`* *`?else`* {*`bodyFalse`*}*`?`*
  Syntax: `switch` *`?option? string pattern1 body1\`*
  `?pattern2 body2? ?default defaultBody?`
- The looping commands are `for`, `while`, and `foreach`.
  Syntax: `for` *`start test next body`*
  Syntax: `while` *`test body`*
  Syntax: `foreach` *`listVar list body`*
- The list operations include `list`, `split`, `llength`, `lindex`, and `lappend`.
  Syntax: `list` *`element1 ?element2? ... ?elementN?`*
  Syntax: `linsert` *`list position element1 ... ?elementN?`*
  Syntax: `lappend` *`listName ?element1? ... ?elementN?`*
  Syntax: `split` *`data ?splitChar?`*
  Syntax: `join` *`list ?joinString?`*
  Syntax: `llength` *`list`*
  Syntax: `lindex` *`list index`*
  Syntax: `lsearch` *`list pattern`*
  Syntax: `lreplace` *`list position1 position2 element1 ?... elementN?`*
- The string processing subcommands include `first`, `last`, `length`, `match`, `toupper`, `tolower`, and `range`.
  Syntax: `string first` *`substr string`*
  Syntax: `string last` *`substr string`*
  Syntax: `string length` *`string`*
  Syntax: `string match` *`pattern string`*
  Syntax: `string toupper` *`string`*
  Syntax: `string tolower` *`string`*
  Syntax: `string range` *`string first last`*
- Formatted strings can be generated with the `format` command.
  Syntax: `format` *`format ?data? ?data2? ...`*
- The `scan` command will perform simple string parsing.
  Syntax: `scan` *`textstring format ?varName1? ?varName2? ...`*
- The array processing subcommands include `array names`, `array set`, and `array get`.
  Syntax: `array names` *`arrayName ?pattern?`*
  Syntax: `array set` *`arrayName`* {*`index1 value1 ...`*}
  Syntax: `array get` *`arrayName`*

- Values can be converted between various ASCII and binary representations with `binary scan` and `binary format`.
- The `source` command loads and evaluates a script.
  Syntax: `source` *`fileName`*
- The `info` command returns information about the current state of the interpreter.
  Syntax: `info` *`proc`*
  Syntax: `info` *`args`*
  Syntax: `info` *`body`*
  Syntax: `info` *`exists`*
  Syntax: `info` *`nameofexecutable`*

3.7 PROBLEMS

The following numbering convention is used in all Problem sections:

Number Range	Description of Problems
100–199	Short comprehension problems review material covered in the chapter. They can be answered in a few words or a 1–5-line script. These problems should each take under a minute to answer.
200–299	These quick exercises require some thought or information beyond that covered in the chapter. They may require reading a man page, or making a web search. A short script of 1–50 lines should fulfill the exercises, which may take 10–20 minutes each to complete.
300–399	Long exercises may require reading other material, or writing a few hundred lines of code. These exercises may take several hours to complete.

- *100* What will the following code fragments display?
  - ```
    set a 1
    puts "$a"
    ```
  - ```
    set a 1
    puts {$a}
    ```
  - ```
    set a 1
    puts [expr $a + 1]
    ```
  - ```
    set a b
    set $a 2
    puts "$b"
    ```
  - ```
    set a 1
    puts "\$$a"
    ```
- *101* What are the Tcl's three looping commands?
- *102* What conditional commands does Tcl support?
- *103* What command will define a Tcl procedure?
    ```

- *104* Can a Tcl procedure be invoked recursively?
- *105* What is the first word in a Tcl command line?
- *106* How are Tcl commands terminated?
- *107* Can you use binary data in Tcl?
- *108* How does a Tcl procedure return a failure status?
- *109* What commands can modify the content of a variable?
- *110* How could you change an integer to a floating-point value with the `append` command?
- *111* Write a pattern for `string match` to match:
  - Strings starting with the letter A.
  - Strings starting with the letter A followed by a number.
  - Strings starting with a lowercase letter followed by a number.
  - Strings in which the second character is a number.
  - Three character strings of uppercase letters.
  - A question.
- *112* What characters are returned by:
  - `string range "testing" 0 0`
  - `string range "testing" 0 1`
  - `string range "testing" 0 99`
  - `string range "testing" 0 end`
  - `string range "testing" 99 end`
  - `string range "testing" end end`
- *113* Write a format definition that will:
  - Use 20 spaces to display a string, and left justify the string.
  - Use 20 spaces to display a string, and right justify the string.
  - Display a floating point number less than 100 with 2 digits to the right of the decimal point.
  - Display a floating point number in scientific notation.
  - Convert an integer to an ASCII character (i.e., convert 48 to ”0”, 49 to ”1”, and so on).
- *114* What is the second list element in the following lists?
  - {one two three four}
  - {one {two three} four}
  - { {} one two three}
  - { {one two} {three four} }
- *115* Which array command will return a list of the indices in an associative array?
- *116* Which Tcl command could be used to assign a value to a single element in an associative array?
- *117* Which Tcl command could be used to assign values to multiple elements in an associative array?

- *118* What `binary scan` format definition would read data that was written as this C structure?: struct { int i[4]; char c[25]; float f; }
- *119* If `x` and `y` are two Tcl variables containing the length of the opposite and adjacent sides of a triangle, write the `expr` command that would calculate the hypotenuse of this angle.
- *120* Write a procedure that uses info commands to report all the procedures and arguments that exist in an interpreter.
- *200* The classic recursive function is a Fibonacci series, in which each element is the sum of the two preceding elements, as in the following.
  ```
  1 1 2 3 5 8 13 21 ..
  ```
  Write a Tcl proc that will accept a single integer, and will generate that many elements of a Fibonacci series.
- *201* Write a procedure that will accept a string of text, and will generate a histogram of how many times each unique word is used in that text.
- *202* Write a procedure that will accept a set of comma-delimited lines, and will generate a formatted table from that data.
- *203* Write a procedure that will check to see whether a string is a palindrome (if it reads the same backward and forward). Examples of palindromes include the words *noon* and *radar*, and the classic sentence *Able was I ere I saw Elba*.
- *300* The bubble sort works by stepping through a list and comparing two adjacent members and swapping them if they are not in ascending order. The list is scanned repeatedly until there are no more elements in the wrong position.
  a. Write a recursive procedure to perform a bubble sort on a list of data.
  b. Write a loop-based procedure to perform a bubble sort on a list of data.
- *301* Tcl has an `lsort` command that will sort a list. Use the `lsort` command to check the results of the bubble sort routines constructed in the previous exercise.
- *302* A trivial encryption technique is to group characters in sets of four, convert that to an integer, and print out the integers. Write a pair of Tcl procedures that will use the `binary` command to convert a plaintext message to a list of integers, and convert a list of integers into a readable string.

CHAPTER

Navigating the File System, Basic I/O and Sockets

4

This chapter describes the Tcl tools for:

- Navigating a file system
- Finding files and their attributes
- Reading and writing data to files or other applications
- Using client- and server-side sockets

4.1 NAVIGATING THE FILE SYSTEM

Most modern operating systems represent file systems as some form of tree-structured system, with a single root node, and drives and directories descending from that single node. They represent the trees as a string of words defining the drives and subdirectories separated by some character. Most operating systems provide a library of system calls to interact with the file system. Unfortunately, many modern operating systems use different symbols for the directory separator, have different conventions for naming drives and subdirectories, and provide different library calls for interacting with the file system.

The Tcl solution to multiple platforms is to provide a set of Tcl commands to generalize the interface between Tcl scripts and the underlying libraries. Interactions with the file system are handled by the `cd`, `pwd`, `glob`, and `file` commands.

Your script can learn or change its current working directory with the `pwd` (Print Working Directory) and `cd` (Change Directory) commands.

Syntax: `pwd`

Returns the current working directory.

Syntax: `cd` *`newDirectory`*

Changes the default working directory.

`newDirectory` A directory to make the current default directory for `open`, `glob`, and so on.

Tcl/Tk: A Developer's Guide. DOI: 10.1016/B978-0-12-384717-1.00004-X

Example 1

Script Example

```
puts "Working Directory: [pwd]"
cd /tmp
puts "Working Directory: [pwd]"
```

Script Output

```
Working Directory: /home/clif
Working Directory: /tmp
```

The Tcl `glob` command allows a script to search a path for items with names or types that match a pattern. You can use `glob` to write scripts that will perform some action on each file in one or more directories without knowing in advance what files will be present. The `glob` command matches file names using the `glob`-style patterns described with the `string match` command in Section 3.3.3

Syntax: `glob ?-nocomplain? ?-types` *`typeList`*`? ?--? ` *`pattern`*

`-nocomplain` Do not throw an error if no files match the pattern.

`-types` *`typeList`* Return only the items that match the *`typeList`*. The *`typeList`* is a string of letters that describes the types of file system entities available. If the *`typeList`* includes these elements, they are combined with a logical OR operation.

`b` A block-mode device

`c` A character device

`d` A directory

`f` A normal file

`l` A symbolic link

`p` A named pipe

`s` A socket

If the *`typeList`* includes these elements, they are combined with a logical AND operation.

`r` A file with read access.

`w` A file with write access.

`????` On MacOS 9 and earlier: A four-letter type is the type or creator of the file. For instance, TEXT for a TEXT-type file, or APPL for an Application-type file. OS X applications do not assign a value to type or creator.

`--` Identifies the end of options. All following arguments are patterns even if they start with a dash.

`pattern` A `glob`-style pattern to match.

Example 2

Script Example

```
foreach fileName [glob *.c *.h] {
   puts "C Source: $fileName"
}
```

Script Output

```
...
C Source: tclUtil.c
C Source: tclVar.c
C Source: regcustom.h
C Source: regerrs.h
...
```

The `-types` option can be used to select only those types of file system entities you want to deal with.

Example 3

Script Example

```
puts "The subdirectories under /usr are: [glob -types d /usr/*]"
```

Script Output

```
The subdirectories under /usr are: /usr/bin /usr/lib/usr/libexec
/usr/sbin /usr/share /usr/X11R6 /usr/dict/usr/etc /usr/games
/usr/include /usr/local /usr/src /usr/kerberos/usr/i386-glibc21-linux
/usr/man
```

4.1.1 Constructing File Paths

The `pwd`, `cd`, and `glob` commands provide an interface to the navigating of a file system that all operating systems support. The `file` commands provide an interface to services that may be platform specific. Like the `string` commands, the `file` commands are implemented as subcommands from the main `file` command.

The `file` command includes many subcommands, including creating new directories, copying files, reading and modifying file information, creating links and creating new file names and paths.

Tcl maintains file paths as Unix style paths inside a script, using a forward slash (/) to separate the subdirectories in a file path. The internal format path is automatically translated to native format when you pass a filename to an application with the `exec` command.

For example, code like this works fine on a Windows system:

```
exec notepad.exe "C:/My Documents/somefile.txt"
```

Do not use the Windows-style backward slash within a Tcl script. The backward slash is used by Tcl to escape the character following it. You will end up confusing yourself with multiple layers of

backward slashes escaping more backward slashes. For example this command:

```
set channel [open "C:/data/datafile" r]
```

will work on Windows platforms, but this command:

```
set channel [open "C:\data\datafile" r]
```

will return an error that `C:datadatafile` cannot be opened.

If you need the actual native file path (for instance, to write the path to a file), use the `file nativename` command.

Syntax: `file nativename` *`path`*

Return a native format path.

`path`	A path in Tcl format.

Example 4

Script Example

```
puts [file nativename "C:/My Documents/letter.doc"]
```

Script Output

```
Windows: C:\My Documents\letter.doc
Linux, Unix, Mac OS X: C:/My Documents/letter.doc
```

You can use either relative or absolute paths within a Tcl script. If you need the absolute path, use the `file normalize` command to remove symbolic links, relative paths, and "/../" sequences from a file path.

Syntax: file normalize *`path`*

Returns an absolute path with no symlinks or relative portions

`path`	A calculated path that may be relative, use symbolic links, etc.

Example 5

```
%  cd $env(HOME)
puts [file normalize "My Documents/myDoc.doc"]
```

Script Output

```
Unix: /home/clif/My Documents/myDoc.doc
Windows: C:/Documents and Settings/Clif/My Documents/myDoc.doc
Mac OS X: /Users/clif/My Documents/myDoc.doc
```

You can construct file paths using the `string`, `split` and `list` commands described in Chapter 2. It is usually easier to use the `file split` and `file join` commands to split a full path into subdirectories or join a set of path elements into a full path.

`file split path`

Returns the `path` as a list split on the OS-specific or canonical directory markers.

`file join list`

Merges the members of list into a canonical file path with each list member separated by a forward slash.

The `file split` command will convert either a native format file path or a Unix style file path to a Tcl list on any platform. It will only split a file path with backward slashes (a Windows style filepath) to a list on a Windows platform.

The `file join` command will join a Tcl list into a file path using forward slashes.

The `file` command includes more subcommands to simplify manipulating file paths.

`file dirname path`

Returns the portion of `path` before the last directory separator.

`file tail path`

Returns the portion of `path` after the last directory separator.

`file rootname path`

Returns the portion of `path` before the last extension marker.

Example 6

Script Example

```
set path [file join "program files" Tcl bin wish.exe]"
puts "join: $path"
puts "dirname: [file dirname $path]"
puts "root: [file rootname $path]"
puts "tail: [file tail $path]"
puts "normal: [file normalize $path]"
```

Script Output

```
join: program files/Tcl/bin/wish.exe
dirname: program files/Tcl/bin
root: program files/Tcl/bin/wish
tail: wish.exe
normal: /Users/clif/program files/Tcl/bin/wish.exe
```

4.2 PROPERTIES OF FILE SYSTEM ITEMS

Before a script attempts to open a file, it may need to know information about the file: its size, creation date, permissions, whether it has been backed up, and so on. The system library calls that report these data differ from platform to platform. The information can be accessed within a Tcl script using some of the `file` subcommands.

The `file` command supports the following subcommands that return information about files.

Reporting a File's Existence

`file exist` *`path`* Returns true if a file exists, and false if it does not exist.

Reporting a File's Type

`file type` *`path`* Returns the type of file referenced by path.

The return value will be one of the following.

`file`	*`$path`* is the name of a normal file.
`directory`	*`$path`* is the name of a directory.
`characterSpecial`	*`$path`* is the name of a UNIX character I/O device (such as a `tty`).
`blockSpecial`	*`$path`* is the name of a UNIX block I/O device (such as a disk).
`fifo`	*`$path`* is the name of a UNIX fifo file.
`link`	*`$path`* is the name of a hard link.
`socket`	*`$path`* is the name of a named socket.

For common tests, the `file` command includes the following subcommands.

`file isdirectory` *`path`* Returns 1 if *`path`* is a directory; 0 if not.

`file isfile` *`path`* Returns 1 if *`path`* is a regular file; 0 if not.

Reporting Statistics about a File

`file stat` *`path varName`*

`file lstat` *`path varName`* Treats `varName` as an associative array and creates an index in that array for each value returned by the `stat` or `lstat` library call. If the underlying OS does not support a status field, the value will be −1. The new indices (and values) will be:

`atime`	Time of last access
`ctime`	Time of last change to directory information
`mtime`	Time of last modification of file content
`dev`	Device type
`gid`	Group ID of owner

`ino`	Inode number for file
`mode`	Protection mode bits
`nlink`	Number of hard links
`size`	Size in bytes
`type`	Type of file
`uid`	User ID number of owner

`file attributes` *`path`*

Return a list of platform-specific attributes of the item referenced by *`path`*.

These values are returned as a name/value list. The values returned on a Unix system are `group`, `owner`, and `permissions`. The values returned on a Windows system are `archive`, `hidden`, `longname`, `readonly`, `shortname`, and `system`. The values returned on a Mac system are `creator`, `readonly`, `hidden`, and `type`.

`file attributes` *`path attributeName`*

Return the value of a named platform-specific attribute of the item Referenced by *`path`*.

`file attributes` *`path attributeName newValue`*

Sets the value of the named attribute to `newValue`.

`file nativename` *`path`*

Returns `pathName` in the proper format for the current platform.

Example 7

Script Example

```
proc reportDirectory {dirName} {
# file join merges the existing directory path with
# the * symbol to match all items in that directory.
foreach item [glob -nocomplain [file join $dirName *]] {
  if { [string match [file type $item] "directory"]} {
      reportDirectory $item
   } else {
      puts "$item is a [file type $item]"
      puts "Attributes are: [file attributes $item]"
      file stat $item tmpArray
      set name [file tail $item]
      puts "$name is: $tmpArray(size) bytes\n"
    }
  }
}
# Start reporting from the current directory
reportDirectory { }
```

Script Output

```
# Unix output resembles:
html/index.html is a file
Attributes are: -group root -owner root -permissions 00644
index.html is: 1945 bytes
html/poweredby.png is a file
Attributes are: -group root -owner root -permissions 00644
poweredby.png is: 1154 bytes
# Windows output resembles:
C:/WINDOWS/Favorites/Personal Files.LNK is a file
Attributes are: -archive 1 -hidden 0
  -longname {C:/WINDOWS/Favorites/Personal Files.LNK}
  -readonly 0 -shortname C:/WINDOWS/FAVORI~1/PERSON~1.LNK
  -system 0
Personal Files.LNK is: 247 bytes
C:/WINDOWS/Favorites/Graphics.LNK is a file
Attributes are: -archive 1 -hidden 0
  -longname C:/WINDOWS/Favorites/Graphics.LNK
  -readonly 0 -shortname C:/WINDOWS/FAVORI~1/GRAPHICS.LNK
  -system 0
Graphics.LNK is: 379 bytes
# Mac Classic (OS 7, 8, 9) output resembles:
:Build:Drag Drop Tclets is a file
Attributes are: -creator WIsH -hidden 0 -readonly 0 -type APPL
Drag Drop Tclets is: 257683 bytes
:Build:Tclapplescript.shlb is a file
Attributes are: -creator TclL -hidden 0 -readonly 0 -type shlb
Tclapplescript.shlb is: 21056 bytes
# Mac OS/X output resembles
reportDirectory.tcl is a file
Attributes are: -group _lpoperator -owner clif -permissions 00644 -readonly 0
  -creator {} -type {} -hidden 0 -rsrclength 0
reportDirectory.tcl is: 410 bytes
```

Note that names of the attributes in the `file attributes` command include the dash (`-`) character, and are returned as key/value pairs. Many Tcl extensions use this format to return collections of data. Data returned as key/value pairs can be easily accessed with the `dict` commands, or assigned to an array using the `array set` command.

```
array set dataArray [file attributes $filePath]
set owner [dict get [file attributes $filePath] -owner]
```

4.3 REMOVING FILES

The last thing a program needs to do with a file is remove it. The `file` command also provides access to the operating system function that will remove a file.

file delete pathName
Deletes the file referenced by *pathName*.

4.4 INPUT/OUTPUT IN TCL

It is difficult to perform any serious computer programming without accepting data and delivering results. Tcl abstracts the input and output commands into two input commands and one output command. Using these three commands, you can read or write to any file, pipe, device, or socket supported by `tclsh` or `wish`. A GUI-based `wish` script can perform user I/O through various graphics widgets but will use these three commands to access files, pipes to other programs, or sockets.

All I/O in Tcl is done through channels. A channel is an I/O device abstraction similar to an I/O stream in C. The Tcl channel device abstraction is extended beyond the stream abstraction to include sockets and pipes. The Tcl channel provides a uniform abstract model of the UNIX, Mac OS, and MS Windows socket calls, so they can be treated as streams.

The Tcl commands that open a channel return a handle that can be used to identify that channel. A channel handle can be passed to an I/O command to specify which channel should accept or provide data. These channels are predefined in Tcl, as follows.

`stdin`	Standard input: keyboard or a redirected file.
`stdout`	Standard output: usually the screen.
`stderr`	Error output: usually the screen. Note that Mac OS before OS/X and MS Windows do not distinguish between `stdout` and `stderr`.

4.4.1 Output

The `puts` command sends output to a channel. It requires a string to output as an argument. By default, this command will append a newline character to the string.

Syntax: `puts` *?-nonewline? ?channel? outputString*

Send a string to a channel.

?-nonewline?	Do not append a newline character to the output.
?channel?	Send output to this channel. If this argument is not used, send to standard output.
outputString	The data to send.

Example 8

```
   % puts "Hello, "; puts "World "
Hello,
World
   % puts -nonewline "Hello, "; puts "World"
Hello, World
```

4.4.2 Input

The Tcl commands that input data are the `gets` and `read` commands. The `gets` command will read a single line of input from a channel and strip off any newline character. The `gets` command may return the data that was read or the number of characters that were read. The `read` command will read a requested number of characters, or until the end of data. The `read` command always returns the data that was read.

The `gets` command is best for interactive I/O, since it will read a single line of data from a user. The `read` command is best used when there is a large body of text to read in a single read command, such as a file or socket, or when there is a need for single-character I/O.

Syntax: `gets` *`channelID ?varName?`*

Read a line of data from a channel up to the first newline character. The newline character is discarded.

`channelID` Read from this channel.

`?varName?` If this variable name is present, `gets` will store the data in this variable, and will return the number of characters read. If this argument is not present, `gets` will return the line of data.

Example 9

```
   % gets stdin  # A user types "Hello, World" at keyboard.
Hello, World
% gets stdin inputString # A user types "Hello, World" at keyboard.
12
% puts $inputString
Hello, World
```

A Tcl script can invoke the `read` command to read a specified number of characters, or to read data until an End-Of-File is encountered. If a script invokes the `read` command to read a specified number of characters and `read` encounters an End-Of-File before reading the requested number of characters, the `read` command will return the available characters and not generate an error. The `read` command returns the characters read. The `read` command may strip off the final newline character but by default leaves newlines intact.

Syntax: `read` *`channelID numBytes`*

Read a specified number of characters from a channel.

`channelID` The channel from which to read data.

`numBytes` The number of bytes to read.

Syntax: `read` *`?-nonewline? channelID`*

Read data from a channel until an End-Of-File (EOF) condition.

?-nonewline?	Discard the last character if it is a newline.
channelID	The channel from which to read data.

Example 10

```
# Read from stdin until an End-Of-File is encountered
#
% read stdin  # A user types Hello, World EOF
Hello, World
# Read 5 characters from stdin
% read stdin 5 # A user types Hello, World
Hello
```

When the output channel is the standard output device, Tcl buffers text until it has a complete line (until a newline character appears). To print a prompt and allow a user to enter text on the same line, you must either reset the buffering with `fconfigure` or use the `flush` (see Section 4.5.2) command to force Tcl to generate output.

Example 11

Script Example

```
puts -nonewline "What is your name? "
flush stdout
gets stdin name ;# User types name
puts "Hello, $name. Shall we play a game?"
```

Script Output

```
What is your name? Dr. Falken
Hello, Dr. Falken.  Shall we play a game?
```

4.4.3 Creating a Channel

A Tcl script can create a channel with either the `open` or `socket` command. The `open` command can be used to open a channel to a file, a device, or a pipe to another program. The `socket` command can be used to open a client socket to a server or to open a server socket that will listen for clients.

Syntax: `open` *fileName ?access? ?permissions?*

Open a file, device, or pipe as a channel and return a handle to be used to access this channel.

fileName — The name of the file or device to open.

If the first character of the name is a pipe (|), the filename argument is a request to open a pipe connection to a command. The first word in the file name will be the command name, and subsequent words will be arguments to the command.

?access? — How this file will be accessed by the channel. The *access* option may be one of:

- r — Open file for read-only access.
- r+ — Open file for read and write access. File must already exist.
- w — Open file for write-only access. Truncate file if it exists.
- w+ — Open file for read and write access. Truncate the file if it exists, or create it if it does not exist.
- a — Open file for write-only access. New data will be appended to the file. For versions of Tcl before 8.3, the file must already exist. With Tcl8.3, the behavior was changed to create a new file if it does not already exist.
- a+ — Open file for read and write access. Create the file if it does not exist. Append data to an existing file.

?permissions? — If a new file is created, this parameter will be used to set the file permissions. The *permissions* argument will be an integer, with the same bit definitions as the argument to the creat system call on the operating system being used.

Example 12
Script Example

```
# Open a file for writing - Note square brackets cause the Tcl command
# to be evaluated, and the channel handle returned by open
```

```
# is assigned to the variable outputFile.
set outputFile [open "testfile" "w"]

# send a line to the output file.
puts $outputFile "This is line 1"
puts $outputFile "This is line 2"
puts $outputFile "This is line 3"

# Close the file.
close $outputFile
# Reopen the file for reading
set inputFile [open "testfile" "r"]
# Read a line of text
set numBytes [gets $inputFile string]
# Display the line read
puts "Gets returned $numBytes characters in the string: $string"

# Read the rest of the file
set string2 [read $inputFile]
puts "Read: $string2"

# Announce intent
puts "\nOpening a Pipe\n"

# and open a pipe to the ls command
set pipe [open "|ls /" "r"]

# Equivalent command under MS-Windows is:
# set pipe [open "!command.com /c dir" "r"]

# read the output of the ls command:
while {![eof $pipe]} {
   set length [gets $pipe lsLine]
   puts "$lsLine is $length characters long"
}
```

Script Output

```
Gets returned 14 characters in the string: This is line 1
Read: This is line 2
This is line 3
Opening a Pipe
bin is 3 characters long
boot is 4 characters long
bsd is 3 characters long
```

```
dev is 3 characters long
...
```

4.4.4 Closing Channels

The Tcl interpreter supports having multiple channels open simultaneously. The exact number is defined at compile time and by the underlying operating system. In order to process many files, your script will need to close channels after it has completed processing them.

Syntax: `close` *`channel`*

Close a channel.

`channel` The handle for the channel to be closed.

When a file that was opened for write access is closed, any data in the output buffer is written to the channel before it is closed.

4.5 SOCKETS

Most client/server applications such as Telnet, ftp, e-mail, web browsers, large database systems, chat programs, and so on use TCP/IP sockets to transfer data. The Tcl `socket` command creates a channel connected to a TCP/IP socket. The `socket` command returns a channel handle that can be used with `gets`, `read`, and `puts` just as a file channel is used. Unlike connections to files or pipes, there are two types of sockets.

- Client socket (very much like a connection to a file or pipe)
- Server socket (waits for a client to connect to it)

Using the `socket` command to establish a client socket is as straightforward as opening a file.

Syntax: `socket` *`?options? host port`*

Open a socket connection.

`?options?` Options to specify the behavior of the socket.

- *`-myaddr addr`* Defines the address (as a name or number) of the client side of the socket. This is not necessary if the client machine has only one network interface.
- *`-myport port`* Defines the port number for the server side to open. If this is not supplied, the operating system will assign a port from the pool of available ports.

	-async	Causes the `socket` command to return immediately, whether the connection has been completed or not.
host		The host to open a connection to. May be a name or a numeric IP address.
port		The name or number of the port to open a connection to on the host machine.

4.5.1 Using a Client Socket

The outline of a Tcl TCP/IP client looks as follows.

```
set server SERVERADDRESS
set port PORTNUMBER

set connection [socket $server $port]

puts $connection "COMMAND"
flush $connection
gets $connection result

analysisProcedure $result
```

The next example demonstrates how to open a socket to a remote Post Office Protocol (POP) server to check for mail and shows how Tcl I/O and string commands can be used to develop an Internet client. The POP 3 message protocol is an ASCII conversation between the POP 3 client (your machine) and the POP 3 server (the remote machine). If you used POP from a Telnet client to learn if you have mail waiting, the conversation would resemble the following.

```
$> telnet pop.example.com pop3
Trying 123.456.789.012
Connected to 123.456.789.012
Escape character is '^]'.
+OK QPOP (version 2.2-krb-IV) at pop.example.com starting.
          <28004.887684070@pop.example.com>
user ImaPopper
+OK Password required for ImaPopper.
pass myPassword
+OK ImaPopper has 1 message (1668 octets).
```

The following example will contact a remote machine and report if any mail is waiting. The machine name, user name, and password are all hard-coded in this example. The test for `+OK` is done differently in each test to demonstrate some different methods of checking for one string in another.

Example 13

Script Example

```
# Open a socket to a POP server. Report if mail is available

# Assign a host, login and password for this session

set popHost example.com

set popLoginID myID
set popPasswd SecretPassword

# Open the socket to port 110 (POP3 server)

set popClient [socket $popHost 110]

# Get the first line:
# +OK QPOP (version ..) at example.com starting...

set line [gets $popClient]

# We can check for the 'OK' reply by confirming that 'OK'
# is the first item in the string
if {[string first "+OK" $line] != 0} {
  puts "ERROR: Did not get expected '+OK' prompt"
  puts "Received: $line"
  exit;
}

# send the user name
# Note that the socket can be used for both input and output

puts $popClient "user $popLoginID"

# The socket is buffered by default. Thus we need to
# either fconfigure the socket to be non-buffered, or
# force the buffer to be sent with a flush command.

flush $popClient

# Receive the password prompt:

# +OK Password required for myID.

set response [gets $popClient]
# We can also check for the 'OK' using string match
if {[string match "+OK*" $response] == 0} {
  puts "ERROR: Did not get expected '+OK' prompt"
  puts "Received: $response"
  exit;
```

```
}

# Send Password

puts $popClient "pass $popPasswd"
flush $popClient

# Receive the message count:
# +OK myID has 0 messages (0 octets).

set message [gets $popClient]

if {![string match "+OK*" $message]} {
  puts "ERROR: Did not get expected '+OK' prompt"
  puts "Received: $message"
  exit;
}

puts [string range $message 3 end]
```

Script Output

```
myID has 2 messages (2069 octets).
```

You can put together a client/server application with just a few lines of Tcl. Note that the error messages use an apostrophe to quote the `+OK` string. In Tcl, unlike C or shell scripts, the apostrophe has no special meaning.

4.5.2 Controlling Data Flow

When a script tries to read past the end of a file, Tcl will return −1 as the number of characters read. A script can test for an End-Of-File condition with the `eof` command.

`eof` *`channelID`*

Returns true if the channel has encountered an End-Of-File condition.

By default, Tcl I/O is buffered, like the stream-based I/O in the C standard library calls. This is usually the desired behavior. If this is not what your project needs, you can modify the behavior of the channel or flush the buffer on demand.

The `fconfigure` command lets you define the behavior of a channel. You can modify several options, including whether to buffer data for this channel and the size of the buffer to use. The `fconfigure` command allows you to configure more options than are discussed in this book. Read the on-line documentation for more details.

Syntax: `fconfigure` *`channelID ?name? ?value?`*

Configure the behavior of a channel.

channelID The channel to modify.

name The name of a configuration field, which includes the following.

-blocking boolean

If set true (the default mode), a Tcl program will block on a `gets` or `read` until data is available.

If set false, `gets`, `read`, `puts`, `flush`, and `close` commands will not block.

-buffering newValue

If `newValue` is set to `full`, the channel will use buffered I/O.

If set to `line`, the buffer will be flushed whenever a full line is received.

If set to `none`, the channel will flush whenever characters are received.

By default, files and sockets are opened with `full` buffering, whereas `stdin` and `stdout` are opened with `line` buffering.

The `fconfigure` command was added to Tcl in version 7.5. If you are writing code for an earlier revision of Tcl, or if you want to control when the buffers are flushed, you can use the `flush` command. The `flush` command writes the buffered data immediately.

Syntax: `flush` *channelID*

Flush the output buffer of a buffered channel.

channelID The channel to flush.

The previous POP client example could be written without the `flush $popClient` commands if the following lines had been added after the `socket` command.

```
# Turn off buffering
fconfigure $popClient -buffering none
```

You can write simple applications using the traditional top-to-bottom flow of initiating a read and blocking until the data is available. If necessary, you could implement a round-robin loop by using the `fconfigure` command to set a channel to be non-blocking. More complex applications (perhaps with multiple open channels) require some sort of event-driven programming style. The Tcl `fileevent` command allows you to implement event-driven I/O in a script. The `fileevent` command registers

a script to be evaluated when a channel either has data to read or is ready to accept data in its write buffer. Registering a callback script is a common paradigm for event-driven programming in Tcl.

Syntax: `fileevent` *`channel direction script`*

`channel`	The channel to watch for file events on.
`direction`	The direction of data flow to watch. May be `readable` or `writable`.
`script`	A script to invoke when the condition becomes true. (A channel has data or can be written to.)

We can write a simple `syslog` daemon, as shown in the next example. Pay attention to the use of quotes and backslashes in the `fileevent` script. The quotes are used because the `$logFile` and `$socket` substitution must be done before the script is registered. The square brackets are escaped because that substitution must be done when the script is evaluated. The script that is registered with the `fileevent` will look something like the following.

```
puts file2 [gets sock123].
```

If the script were enclosed in curly braces, instead of quotes, no substitution would occur, and the script to be evaluated would be `"puts $logFile \[gets $socket\]"`. When data becomes available on the channel, this script would be evaluated, and would throw an error, since the `logFile` variable is a local variable that was destroyed when the `createLogger` procedure stopped being active.

Example 14

```
proc createLogger {socket} {
  # Open a file for data to be logged to
  set logFile [open C:\event.log w]

  # When the socket is written to, read the data
  # and write it to the log file.
  fileevent $socket readable "puts $logFile \[gets $socket\]"
}
```

4.5.3 Server Sockets

Creating a TCP/IP server is a little different from a client-side socket. Instead of requesting a connection, and waiting for it to be established, the server-side `socket` command registers a script to be evaluated when a client socket requests a connection to the server.

Syntax: `socket` *`-server script port`*

`script`	A script to evaluate when a connection is established. This script will have three arguments appended to the script before it is evaluated.

`channel`	The I/O channel that is assigned to this session.
`IP Address`	The address of the client requesting a connection. (You can use this for simple Access List security tests.)
`port`	The port being used by the client-side socket.

`port` The port to accept connections on.

The outline for a Tcl TCP/IP server follows. Note the test for the End-Of-File in the `readLine` procedure and the `vwait` command at the end of the script. When the other end of a socket is closed it can generate spurious `readable` events that must be caught in the file event-handling code. If not, the server will receive continuous `readable` events, the `gets` command will return an empty string (and `-1` characters), and the system will be very busy doing nothing. This test (or something similar) will catch that condition and close the channel, which will unregister the `fileevent`.

The normal `tclsh` script is read and evaluated from top to bottom, and then the program exits. This behavior is fine for a filter-type program, but not good for a server. A server program must sit in an event loop and wait for connections, process data, and so on. The `vwait` command causes the interpreter to wait until a variable is assigned a new value. While it is waiting, it processes events.

Syntax: `vwait` *`varName`*

`varName` The variable name to watch. The script following the `vwait` command will be evaluated after the variable's value is modified.

```
proc serverOpen {channel addr port} {

   # Set up fileevent to be called when input is available

   fileevent $channel readable "readLine $channel"
}

proc readLine {channel} {

   set len [gets $channel line]

   # If we read 0 or -1 characters, check for EOF.
   # If EOF, close the channel and delete that client entry.

     if {($len <= 0)&&[eof $channel]} {
        close $channel
        } else {
        # Process Line of Data
    }
}
```

```
set serverPort PORTNUMBER
set server [socket -server serverOpen $serverPort]

# Initialize a variable and wait for it to change
# to keep tclsh from exiting.
set running 1
vwait running
```

The next example shows a simple server that will watch a file, and send a notice whenever the file is changed. This type of script might be used on a firewall to report if the registry, or files in `/bin` or `C:/winnt`, are modified. In this case, the script is watching the UNIX password file. If this file changes size, it means a new user has been added to the system.

The `after` command will register a script to be evaluated after a time interval. This is discussed in more detail in Chapter 9.

Example 15

```
proc serverOpen {channel addr port} {
  # Set up fileevent to be called when input is available
  initializeData /etc/passwd
  after 2000 "examineFiles $channel /etc/passwd"
}
proc initializeData {filename} {
  global fileData}
  file stat $filename fileData
}
proc examineFiles {channel filename} {
  global fileData
  file stat $filename newData
 foreach index [array names newData] {
  if {$newData($index) != $fileData($index)}{
    puts $channel "$filename: $index has changed"
    puts $channel "  was: $fileData($index) now: $newData($index)"
    flush $channel
  }
 }
 file stat $filename fileData
 after 2000 "examineFiles $channel $filename"
}
set serverPort 12345
set server [socket -server serverOpen $serverPort]
set done 0
vwait done
```

To test this program, you might telnet to the server in one window, and modify the test file in another to see if you get a report. The IP address `127.0.0.1` is always your current host computer. This is called the loopback address. This example watches the `/etc/passwd` file, and will report whenever the file is modified. When someone attempts to log into the system, or a program checks that a user name is valid, the access time (`-atime`) attribute will change.

```
C:> telnet 127.0.0.1 12345
Trying 127.0.0.1...
Connected to localhost (127.0.0.1).
Escape character is '^]'.
/etc/passwd: atime has changed
was: 1023112769 now: 1023112872
/etc/passwd: atime has changed
was: 1023112872 now: 1023112888
```

4.6 BOTTOM LINE

- The `pwd` command will return the current working directory.
- The `cd newDir` command will change the current working directory.
- The `glob` command will return file system items that match a pattern or type.
- Multiplatform file system interactions are supported with the following `file` commands.

`file exist path`

Returns true if a file exists, and false if it does not exist.

`file type path`

Returns the type of file referenced by `path`. The return value will be `file`, `directory`, `characterSpecial`, `blockSpecial`, `fifo`, `link`, or `socket`.

`file attributes path`

Returns a list of platform-specific attributes of the item referenced by `path`.

`file stat path varName`

Treats `varName` as an associative array and creates an index in that array for each value returned by the `stat` library call.

`file split path`

Returns the `path` as a list, split on the OS-specific directory markers.

`file join list`

Merges the members of list into an OS-specific file path, with each list member separated by the OS-specific directory markers.

file dirname path

Returns the portion of `path` before the last directory separator.

file tail path

Returns the portion of `path` after the last directory separator.

file rootname path

Returns the portion of `path` before the last extension marker.

file attributes path attributeName

Returns the value of a named platform-specific attribute of the item referenced by *path*.

file attributes path attributeName newValue

Sets the value of the named attribute to *newValue*.

- The `open` command will open a channel to a file or pipe.
- The `socket` command will open a client or server-side socket.
- The `puts` command will send a string of data to a channel.
- The `gets` command will input a line of data from a channel.
- The `read` command will input one or more bytes of data from a channel.
- The `fconfigure` command will modify the behavior of a channel.
- The `fileevent` command will register a script to be evaluated when a channel has data to read, or can be written to.
- The `eof` command returns true if the End-Of-File has been reached or a socket has been closed.
- The `close` command will close a channel.

4.7 PROBLEMS

The following numbering convention is used in all Problem sections:

Number Range	Description of Problems
100–199	Short comprehension problems review material covered in the chapter. They can be answered in a few words or a 1–5-line script. These problems should each take under a minute to answer.
200–299	These quick exercises require some thought or information beyond that covered in the chapter. They may require reading a man page or making a web search. A short script of 1–50 lines should fulfill the exercises, which may take 10–20 minutes each to complete.
300–399	Long exercises may require reading other material or writing a few hundred lines of code. These exercises may take several hours to complete.

- *100* Can a Tcl script change the current working directory?
- *101* Can a Tcl `open` command be given a relative path to the file to open, or must it be an absolute path?
- *102* What Tcl command will return a list of all the files in a directory?
- *103* What features of a file are reported by the `file attributes` command?
- *104* Why should you use `file join` instead of assembling a file path using the `string` commands?
- *105* Can Tcl be used for event-driven programming?
- *106* Can a TCP/IP server written in Tcl have more than one client connected at a time?
- *107* What Tcl command or commands would convert `/usr/src/tcl8.4.1/generic/tcl.h` to the following?
  - `tcl.h`
  - `/usr/src/tcl8.4.1/generic/tcl`
  - `tcl`
  - `/usr/src/tcl8.4.1/generic`
  - `generic`
- *108* Can a single Tcl `puts` command send data to more than one channel?
- *109* What Tcl command can be used to input a single byte of data from a channel?
- *110* Given an executable named `reverse` that will read a line of input from `stdin` and output that line of text with the letters reversed, what commands would be used to open a pipe to the `reverse` executable, send a line of text to the `reverse` process, and read back the reply?
- *111* Which two commands will cause an output buffer to be flushed?
- *112* If you wish your application to flush the output buffer whenever a newline is received, what Tcl command would you include in the script?
- *200* Example 4.2.3-1 watches a single file. Modify this example so that the `initializeData` and `examineFiles` procedures can accept a list of files to watch.
- *201* Write a Tcl script that will read lines of text from a file and report the number of words in the file.
- *202* Write a script that will list the object files in a directory that are older than the appropriate source file.
- *203* Write a script that will list files in a directory in size order.
- *204* Write a script that will report the oldest file in a set of directories by recursively searching subdirectories under a parent directory.
- *205* Write a script that will recursively search directories for files that contain a particular pattern.

- *206* Write a TCP/IP server that will accept a line of text, and will return that line with the words in reverse order.
- *300* Write a TCP/IP client that will:
  - Prompt a user for a port
  - Open a client socket to IP Address 127.0.0.1 and the port provided by the user
  - In a loop:
    i. Prompt the user for input
    ii. Send the text to a server
    iii. Accept a line of data from the server
    iv. Display the response to the user

  Do not forget to append a newline character to the user input and `flush` the output socket.
- *301* Write a TCP/IP server that will send three questions (one at a time) to a client, accept the input, and finally generate a summary report. The conversation might resemble the following.

```
Server: What is your name?
Client: Lancelot
Server: What is your quest?
Client: To find the holy grail
Server: What is your favorite color?
Client: Blue
Server: Your name is Lancelot.
     You like Blue clothes.
     You wish To find the holy grail.
```

  Test this server by connecting to it with a Telnet client, or using the test client described in the previous example.
- *302* Write a TCP/IP server that will accept a line of numbers and will return the sum of these values. Write a client that will exercise and test this server. The client-server conversation might resemble the following.

```
Client: 1 2 3 9 8 7
Server: 30
```

- *303* Modify the previous server so that it will return the wrong answer if the correct sum is 30. Confirm that your client will catch this error.

CHAPTER

Using Strings and Lists

5

This chapter describes how to use Tcl strings and lists for common data-searching applications. A common programming task is extracting one piece of information from a large mass of data. This section discusses several methods for doing that and introduces a few more Tcl concepts and commands along the way.

For the application, we will find the Noumena Corporation home page in a list of uniform resource locators (URLs) for some Tcl information and archive sites. The following is the starting data, extracted and modified from a browser's bookmark file.

```
% set urls {
wiki.tcl.tk "Tcler's Wiki"
core.tcl.tk "Tcl/Tk Fossil Core"
sourceforge.net/projects/tcl/ "Sourceforge Tcl"
sourceforge.net/projects/tktoolkit/ "Sourceforge Tk"
www.noucorp.com "Noumena Corporation"
www.activestate.com "ActiveState"
www.tcl.tk/ "Tcl Developer's Exchange"
expect.nist.gov/ "Expect Home Page"
www.tclcommunityassociation.org "Tcl/Tk Community Association"
}
```

5.1 CONVERTING A STRING INTO A LIST

Since the data has multiple lines, one solution is to convert the lines into a list and then iterate through the list to check each line.

Example 1

Splitting Data into a List

```
# Split data into a list at the newline markers
set urlList [split $urls "\n"]
# display the list
puts $urlList
```

Tcl/Tk: A Developer's Guide. DOI: 10.1016/B978-0-12-384717-1.00005-1

Script Output

```
{} {wiki.tcl.tk "Tcler's Wiki"}
{core.tcl.tk "Tcl/Tk Fossil Core"}
{sourceforge.net/projects/tcl/ "Tcl at Sourceforge"}
{sourceforge.net/projects/tktoolkit/ "Tk at Sourceforge"}
{www.noucorp.com "Noumena Corporation"}
{www.activestate.com "ActiveState"}
{www.tcl.tk/ "Tcl Developer's Exchange"}
{expect.nist.gov/ "Expect Home Page"}
{www.tclcommunityassociation.org "Tcl/Tk Community Association"}
{}
```

Note that the empty lines after the left bracket and before the right bracket were converted to empty list entries. This is an artifact of the way the starting data was defined. If it had been defined as:

```
% set urls {wiki.tcl.tk "Tcler's Wiki"
core.tcl.tk "Tcl/Tk Fossil Core"
sourceforge.net/projects/tcl/ "Sourceforge Tcl"
sourceforge.net/projects/tktoolkit/ "Sourceforge Tk"
www.noucorp.com "Noumena Corporation"
www.activestate.com "ActiveState"
www.tcl.tk/ "Tcl Developer's Exchange"
expect.nist.gov/ "Expect Home Page"
www.tclcommunityassociation.org "Tcl/Tk Community Association"}
```

the empty list elements would not be created, but the example would be a bit less readable. This example code will not be bothered by the empty lists, but you may need to check for that condition in other code you write. The `split` command will convert text to a list at every occurrence of the split characters, whether you expect it to convert at that location or not.

5.2 EXAMINING THE LIST WITH A `for` LOOP

Now that the data is a list, we can iterate through it using the numeric `for` loop introduced in Chapter 3. The section following the example explains in detail what happens when this script is evaluated.

Syntax: `for` *`start test next body`*

Example 2

Search Using a `for` Loop

```
for {set pos 0} {$pos < [llength $urlList]} {incr pos} {
  set testLine [lindex $urlList $pos]
  if {[string first "Noumena" $testLine] >= 0} {
```

```
    puts "NOUMENA PAGE:\n $testLine"
  }
}
```

Script Output

```
NOUMENA PAGE:
www.noucorp.com "Noumena Corporation"
```

```
for {set pos 0} {$pos < [llength $urlList]} {incr pos} {
```

This line calls the `for` command. The `for` command takes four arguments: `start`, `test`, `next`, and a body to evaluate. Notice that this line shows only three arguments (`start`, `test`, and `next`) but no body. There is only a left curly brace for the body of this command.

The curly braces cause all characters between the braces to be treated as normal text with no special processing by the Tcl interpreter. Variable names preceded by dollar signs are not replaced with their value, the code within square brackets will not get evaluated, and the newline character is not interpreted as the end of a command. So, placing the curly braces at the end of the `for` line tells the Tcl interpreter to continue reading until it reaches the matching close bracket, and treat that entire mass of code as the body for this `for` command. The following code would generate an error.

```
% for {set i 1} {$i < 10} {incr i}
  {
  set x [expr $x + $i]
  }
```

A Tcl interpreter reading the line `for {set i 1} {$i < 10} {incr i}` would find a `for` command (and `start`, `test`, and `next` arguments) and would then see the End-Of-Line and nothing to tell the interpreter that the next line might be part of this command. The interpreter would return the following error to inform the programmer that the command could not be parsed correctly.

```
% wrong # args: should be "for start test next command"
```

The *Tcl Style Guide* created by the Tcl development group at Sun Microsystems recommends that you place the body of an `if`, `for`, `proc`, or `while` command on a separate line from the rest of the command, with just the left bracket on the line with the command to inform the compiler that the body of the command is continued on the following lines. The following code is correct.

```
for {set i 0} {$i < 100} {incr i} {
  puts "I will write my code to conform to the standard"
}
```

Note that the arguments to the `for` command in both sets of example code are grouped with curly braces, not quotes. If the arguments were grouped with quotes, a substitution pass would be performed on the arguments before passing them to the `for` command. The variable `pos` has not been defined, so attempting a substitution would result in an error. If `pos` had already been defined as 10 (perhaps in a previous `for` loop), variable substitutions would be performed and the first line would be passed to the `for` command, as in the following.

```
for {set pos 0} "10 < 10" {incr pos} {
```

With the variables in the `test` already substituted, the `test` will always either fail or succeed (depending on the value of the variable), and the loop will not do what you expect. For the same reason, the body of a `for` command should be grouped with braces instead of quotes. You do not want any variables to be substituted until the loop body is evaluated. Given that no substitution happens to the variables enclosed in curly braces, you may be wondering how the code in one scope (within the `for` command) can access the variables in another scope (the code that invoked the `for` command).

The Tcl interpreter allows commands to declare which scope they should be evaluated in. This means that commands such as `for`, `if`, and `while` can be implemented as procedures and the body of the command can be evaluated in the scope of the code that called the procedure. This gives these commands access to the variables that were enclosed in curly braces and allows the substitution to be done as the command is being evaluated, instead of before the evaluation.

For internal commands such as `for`, `if`, and `while`, the change of scope is done within the Tcl interpreter. Tcl script procedures can also use this facility with the `uplevel` and `upvar` commands, which are described in Chapter 7. There are examples using the `uplevel` and `upvar` commands in Chapters 7–12, 16, and 19.

The `start` and `next` arguments to the `for` command are also evaluated in the scope of the calling command. Thus, `pos` will have the last value that was assigned by the `next` phrase when the `for` loop is complete and the line of code after the body is evaluated.

```
set testLine [lindex $urlList $pos]
```

This extracts the list element at position `$pos` and assigns it to the variable `testLine`. The value of the variable `pos` is incremented on each pass through the loop, thus the value of `testLine` steps through the list on each pass through the loop.

This line could have been omitted, and the `lindex` command could be used to extract the list element each time it was needed. If a value is going to be created and used multiple times, it's usually better code style to assign it to a variable, rather than duplicating the code that creates the value.

```
if {[string first "Noumena" $testLine] >= 0} {
```

Like the `for` command, the `if` command accepts an expression and a body of code. If the expression is true, the body of code will be evaluated. As with the `for` command, the test expression goes on the same line as the `if` command, but only the left brace of the body is placed on that line. The test for the `if` is grouped with brackets, just as is done for the `for` command.

Within the `test` expression, there is a nested Tcl command. This will be evaluated before the expression is evaluated. The command `[string first "Noumena" $testLine]` will be replaced by either the position of the string `Noumena` in the target string or −1 if `Noumena` is not in the list element being examined.

```
puts "NOUMENA PAGE:\n $testLine"
```

If the `test` evaluates to true, the `puts` command will be evaluated; otherwise, the loop will continue until `$pos` is no longer less than the number of entries in the list. The argument to `puts` is in quotes rather than curly braces to allow Tcl to perform substitutions on the string. This allows Tcl to replace the `\n` with a newline and `$testLine` with the list element at `$pos`.

5.3 USING THE foreach COMMAND

Using a for command to iterate through a list is familiar to people who have coded in C or FORTRAN, which are number-oriented languages. Tcl is a string- and list-oriented language. Tcl has better ways to iterate through a list. For instance, we could use the foreach command instead of the for command to loop through the list.

Syntax: foreach *varname list body*

Example 3

Search Using a* foreach *Loop

```
foreach item $urlList  {
  if {[string first "Noumena" $item] >= 0} {
    puts "NOUMENA PAGE:\n $item"
  }
}
```

Script Output

```
NOUMENA PAGE:
www.noucorp.com "Noumena Corporation"
```

Using the foreach command the code is somewhat simpler, because the foreach command returns each list element instead of requiring the lindex commands to extract the list elements.

5.4 USING string match INSTEAD OF string first

There are other options that can be used for the test in the if statement.

Example 4

Search Using a* string match *Test

```
% foreach item $urlList {
  if {[string match {*[Nn]oumena*} $item]} {
    puts "NOUMENA PAGE:\n $item"
  }
}
```

Script Output

```
NOUMENA PAGE:
www.noucorp.com "Noumena Corporation"
```

The `string match` command will compare a *glob* style pattern to a string and return `true` if the pattern matches the string. The *glob* pattern rules are described in Section 3.3.

Note that the pattern argument to the `string match` command is enclosed in braces to prevent Tcl from performing command substitutions on it. The pattern includes the asterisks because `string match` will try to match a complete string, not a substring. The asterisk will match to any set of characters. The pattern {`*[Nn]oumena*`} causes `string match` to accept as a match a string that has the string `noumena` or `Noumena` with any sets of characters before or after.

5.5 USING `lsearch`

We could also extract the Noumena site from the list of URLs using the `lsearch` command. The `lsearch` command will search a list for an element that matches a pattern.

Syntax: `lsearch` *?mode? list pattern*

Return the index of the first list element that matches `pattern` or `−1` if no element matches the pattern.

`?mode?` The type of match to use in this search.

`?mode?` may be one of:

`-exact`	The list element must exactly match the pattern.
`-glob`	The list element must match pattern using the `glob` rules.
`-regexp`	The list element must match pattern using the regular expression rules.

`list` The list to search.

`pattern` The pattern to search for.

Example 5

Search Using an `lsearch`

```
set index [lsearch −glob $urlList "*Noumena*"]
if {$index >= 0} {
    puts "NOUMENA PAGE:\n[lindex $urlList $index]"
}
```

Script Output

```
NOUMENA PAGE:
www.noucorp.com "Noumena Corporation"
```

Note that this solution will only find the first list element that matches `*Noumena*`.

For versions of Tcl before 8.5, to find multiple matches, you will need a loop as follows.

Example 6

Script Example

```
set l {phaser tricorder {photon torpedo} \
  transporter communicator}
# Report all list elements with an 'a' in them.
while {[set p [lsearch $l "*a*"]] >= 0} {
  puts "There's an 'a' in: [lindex $l $p]"
  incr p
  set l [lrange $l $p end]
}
```

Script Output

```
There's an 'a' in: phaser
There's an 'a' in: transporter
There's an 'a' in: communicator
```

Tcl version 8.5 added new options to the `lsearch` command to make finding multiple matches easier:

`-start num`	Return the first list element that matches the pattern after this location.
`-all`	Return a list of all the list elements that match the pattern.
`-inline`	Return the list element instead of position.

Using the `-start` option removes the need to reduce the size of the list on each pass through the `while` loop.

Example 7

Script Example

```
set l {phaser tricorder {photon torpedo} \
  transporter communicator}
# Report all list elements with an 'a' in them.
set first 0
while {[set p [lsearch –start $first $l "*a*"]] >= 0} {
  puts "There's an 'a' in: [lindex $l $p]"
  set first [incr p]
}
```

Script Output

```
There's an 'a' in: phaser
There's an 'a' in: transporter
There's an 'a' in: communicator
```

The `-start` option will find the next list element that matches a pattern at or after the defined location. That's why the value of `first` is incremented to be one location past the location of the previous match.

The solution gets even simpler if you use the `-all` option to get a list of all the indices that match the pattern using just one call to the `lsearch` command.

```
set l {phaser tricorder {photon torpedo} \
  transporter communicator}
# Report all list elements with an 'a' in them.
foreach pos [lsearch -all $l "*a*"]} {
  puts "There's an 'a' in: [lindex $l $pos]"
}
```

The final simplification is to use the `-inline` option to get the actual list element, instead of using the `lindex` command to extract the list element.

```
set l {phaser tricorder {photon torpedo} \
    transporter communicator}
# Report all list elements with an 'a' in them.
foreach val  [lsearch -inline -all $l "*a*"]} {
  puts "There's an 'a' in: $val"
}
```

5.6 THE `regexp` COMMAND

The regular expression commands in Tcl provide finer control of string matching than the `glob` method, more options, and much more power.

5.6.1 Regular Expression Matching Rules

A regular expression is a collection of short rules that can be used to describe a string. Each short rule is called a *piece*, and consists of an element that can match a single character (called an *atom*) and an optional count modifier that defines how many times this atom may be matched in the string. If the count modifier is omitted, the atom will be matched once.

Basic Regular Expression Rules

This table shows the ways an atom can be defined.

Definition	Example	Description
A single character	`x`	will match the character `x`.
A range of characters enclosed in brackets	`[a-q]`	will match any lowercase letter between `a` and `q` (inclusive).
A period	`.`	will match any character.
A caret	`^`	matches the beginning of a string.

(*Continued*)

Definition	Example	Description
A dollar sign	`$`	matches the end of a string.
A backslash sequence	`\^`	inhibits treating `*`, `,` `$`, `+`, `^`, and so on as special characters, and matches the exact character.
A regular expression enclosed in parentheses	`([Tt]cl)`	will match that regular expression.

A regular expression could be as simple as a string. For example, `this is a regular expression` is a regular expression consisting of 28 atoms, with no count modifiers. It will match a string that is exactly the same as itself. The range atom (`[a-z]`) needs a little more explanation, since there are actually several rules to define a range of characters. A *range* consists of square brackets enclosing the following.

Definition	Example	Description
A set of characters	`[tcl]`	Any of the letters within the brackets may match the target.
Two characters separated by a dash	`[a-z]`	The letters define a range of characters that may match the target.
A character preceded by a caret, if the caret is the first letter in the range	`[^,]`	Any character *except* the character after the caret may match the target.
Two characters separated by a dash and preceded by a caret, if the caret is the first letter in the range	`[^a-z]`	Any character *except* characters between the two characters after the caret may match the target.

The regular expression `[Tt][Cc][Ll]` would match a string consisting of the letters T, C, and L, in that order, with any capitalization. The regular expression `[TtCcLl]*` would match those letters in any order.

A count modifier may follow an atom. If a count modifier is present, it defines how many times the preceding atom can occur in the regular expression. The count modifier may be the following.

Definition	Example	Description
An asterisk	`a*`	Match 0 or more occurrences of the preceding atom (`a`).
A plus	`a+`	Match 1 or more occurrences of the preceding atom.
A question mark	`[a-z]?`	Match 0 or 1 occurrence of the preceding atom.

(*Continued*)

Definition	Example	Description
A *bound*	{3}	An integer that defines exactly the number of matches to accept.
	{3,}	The comma signifies to match at least three occurrences of the previous atom, and perhaps more.
	{3,5}	A pair of numbers representing a minimum and maximum number of matches to accept.

Support for count modifier bounds was added in Tcl 8.1. Versions of Tcl earlier than that only support the asterisk, plus, and question mark count modifiers.

5.6.2 Examples of Regular Expressions

As a very simple example, the regular expression `A*` would match a string of a single letter A, a string of several letter As, or a string with no A at all.

The regular expression `[A-Z]+[0-9A-Z]*` is more useful. This regular expression describes a string that starts with at least one uppercase alphabetic character, followed by 0 or more alphanumeric characters. This regular expression describes a legal variable name in many programming languages.

Regular expression pieces can be placed one after the other to define a set of items that must be present, or they can be separated by a vertical bar to indicate that one piece or the other must be present. Pieces can be grouped with parentheses into a larger atom. To match a literal parentheses, you must escape the character with a backslash.

The regular expression `(Tcl)|(Tk)` will match either the string `Tcl` or the string `Tk`. The same strings would be matched by the regular expression `T((cl)|k)`. The regular expression `((Tcl)|(Tk)|/)+` will match a set of `Tcl`, `Tk`, or slash. It would match the string `Tcl`, `Tk`, `Tcl/Tk`, `Tk/Tcl`, `TclTk/`, and so on, but not `Tlc-kT` or other variants without a `Tcl`, `Tk`, or slash.

The regular expression `\([^ \)]*\)` would match a parenthetical comment (like this). The backslashed left parenthesis means to match a literal left parenthesis, then 0 or more characters that are not right parenthesis, and finally a literal right parenthesis.

5.6.3 Advanced and Extended Regular Expression Rules

This defines most of the basic regular expression rules. These features (with the exception of the *bounds* count modifier) are supported by all versions of Tcl. The 8.1 release of Tcl included a large number of modifications to the string handling. The largest change was to change the way strings were stored and manipulated. Prior to 8.1, Tcl used 8-bit ASCII for strings. With version 8.1, Tcl moved to 16-bit Unicode characters, giving Tcl support for international alphabets.

Part of revamping how strings are handled required reworking the regular expression parser to handle the new-style character strings. Henry Spencer, who wrote the original regular expression library for Tcl also did the rewrite. While he was adding support for 16-bit Unicode he also added support for the Advanced and Extended regular expression rules.

The rest of this discussion concerns the rules added in Tcl version 8.1.

Minimum and Maximum Match

When the regular expression parser is looking for matches to rules, it may find multiple sets of characters that will match a rule. The decision for which set of characters to apply to the rule is:

1. The set of characters that starts furthest to the left is chosen
2. The longest set of characters that match from that position is chosen

This behavior is referred to as *greedy* behavior. You can change this behavior to match the minimum number of characters by placing a question mark after the count modifier. For example, the following regular expression

```
<.*>
```

would match an HTML tag. It would work correctly with a string such as `<IMG SRC="comic.gif">`, but would match too many characters when compared to a string such as `<IMG SRC="comic.gif"><P>`. The greedy match is all characters between the first "<" and the last ">", i.e., the entire string.

Adding the question mark after the star will cause the minimal matching algorithm to be used, and the expression will match to a single HTML tag, as follows.

```
<.*?>
```

Internationalization

Prior to Version 8.1, all Tcl strings were pure ASCII strings, and the content of a regular expression would also be ASCII characters. Several new features were added to support the possible new characters. Unicode support is discussed in more detail in Chapter 12.

Non-ASCII Values

You can search for a character by its hexadecimal value by preceding the hex value with `\x`. For example, the æ character is encoded as 0xe6 in hexadecimal, and could be matched using a script such as the following.

```
% set s [format "The word salvi%c is Latin" 0xe6]
% regexp {([^ ]*\xe6[^ ]*)} $s a b
1
% puts $a
saliæ
```

Character Classes, Collating Elements, and Equivalence Classes

Another part of reworking the regular expression engine was to add support for named character classes, collating elements, and equivalence classes. These features make it possible to write a single regular expression that will work with multiple alphabets.

A named character class defines a set of characters by using a name, instead of a range. For example, the range `[A-Za-z]` is equivalent to the named character class `[[:alpha:]]`. A named character class is defined by putting a square bracket and colon pair around the name of the character class.

The advantage of named characters classes is that they automatically include any non-ASCII characters that exist in the local language. Using a range such as `[A-Za-z]` might not include characters

with an umlaut or accent. The named character classes supported by Tcl include those outlined in the following table.

`alpha`	The alphabetic characters
`lower`	Lowercase letters
`upper`	Uppercase letters
`alnum`	Alphabetic and numeric characters
`digit`	Decimal digits
`xdigit`	Hexadecimal digits
`graph`	All printable characters except blank
`cntrl`	Control characters (ASCII values < 32)
`space`	Any whitespace character
`<`	The beginning of a word
`>`	The end of a word

A word is a set of alphanumeric characters or underscores preceded and followed by a non-alphanumeric or underscore character (such as a whitespace).

A collating element is a multi-character entity that is matched to a single character. This is a way to describe two-letter characters such as æ. A collating element is defined with a set of double square brackets and periods: `[[. .]]`. Thus, the Latin word *salviæ* would be matched with the regular expression `salvi[[.ae.]]`.

An equivalence class is a way to define all variants of a letter that may occur in a language. It is denoted with a square bracket and equal sign. For example, `[[=u]]` would match to the character `u`, `ú`, or `ü`. Note that the internationalization features are only enabled if the underlying operating system supports the language that includes the compound letter. If your system does not support a collating element or equivalence class, Tcl will generate the following error.

```
couldn't compile regular expression pattern: invalid collating element
```

Tcl Commands Implementing Regular Expressions

Regular expression rules can be used for pattern matching with the `switch` and `lsearch` commands. Tcl also provides two commands for parsing and manipulating strings with regular expressions.

`regexp`	Parses a string using a regular expression, may optionally extract portions of the string to other variables.
`regsub`	Substitutes sections of a string that match a regular expression.

Syntax: `regexp` *`?opt? expr string ?fullmatch? ?submatch?`*

Returns 1 if `expr` has a match in `string`. If `matchVar` or `subMatchVar` arguments are present, they will be assigned the matched substrings.

`opt`	Options to fine-tune the behavior of `regexp`. Options include:

-nocase	Ignores the case of letters when searching for a match.
-indices	Stores the location of a match, instead of the matched characters, in the submatch variable.
-line	Performs the match on a single line of input data. This is roughly equivalent to putting a newline atom at the beginning and end of the pattern. This option was added with Tcl release 8.1.
--	Marks the end of options. Arguments that follow this will be treated as a regular expression even if they start with a dash.
expr	The regular expression to match with string.
string	The string to search for the regular expression.
?fullmatch?	If there is a match, and this variable is supplied to regexp, the entire match will be placed in this variable.
?submatch?	If there is a match, and this variable is supplied to regexp, the Nth parenthesized regular expression match will be placed in this variable. The parenthesized regular expressions are counted from left to right and outer to inner.

Example 8

Example Script

```
Match a string of uppercase letters,
# followed by a string lowercase letters
# followed by a string uppercase letters
regexp {([A-Z]*)(([a-z]*)[A-Z]*)} "ABCdefgHIJ" a b c d e
puts "The full match is: $a"
puts "The first parenthesized expression matches: $b"
puts "The second parenthesized expression matches: $c"
puts "The third parenthesized expression matches: $d"
puts "There is no fourth parenthesized expression: $e"
```

Script Output

```
The full match is: ABCdefgHIJ
The first parenthesized expression matches: ABC
The second parenthesized expression matches: defgHIJ
The third parenthesized expression matches: defg
There is no fourth parenthesized expression:
```

Syntax: `regsub` *?options? expression string subSpec varName*

Copies `string` to the variable `varName`. If `expression` matches a portion of `string`, that portion is replaced by `subSpec`.

`options`	Options to fine-tune the behavior of `regsub`. May be one of:
`-all`	Replaces all occurrences of the regular expression with the replacement string. By default, only the first occurrence is replaced.
`-nocase`	Ignores the case of letters when searching for match.
`--`	Marks the end of options. Arguments that follow this will be treated as regular expressions, even if they start with a dash.
`expression`	A regular expression that will be compared with the target string.
`string`	A target string with which the regular expression will be compared.
`subSpec`	A string that will replace the regular expression in the target string.
`varName`	A variable in which the modified target string will be placed.

Example 9

`regsub` ***Example***

```
set bad "This word is spelled wrung"
regsub "wrung" $bad "correctly" good
puts $good
```

Script Output

```
This word is spelled correctly
```

The portions of a string that match parenthesized atoms of a regular expression can also be captured and substituted with the `regsub` command. The substrings are named with a backslash and a single digit to mark the position of the parenthesized atom. As with the `regexp` parenthesized expressions, they are numbered from left to right, and outside to inside. The subexpressions are numbered starting with `\1`, with `\0` being the entire matching string.

Example 10

`regsub` ***Example***

```
set wrong {Don't put the horse before the cart}
```

```
regsub {(D[^r]*)(h[^ ]*)( +[^c]*)(c.*)} $wrong \
    {\1\4\3\2\} right
puts $right
```

Script Output

```
Don't put the cart before the horse
```

The regular expression `(D[^r]*)(h[^ ]*)( +[^c]*)(c.*)` breaks down to the following.

Piece	Description	Matches
`(D[^r]*)`	A set of characters starting with `D` and including any character that is not an `r`	`Don't put the`
`(h[^ ]*)`	A set of characters starting with `h` and including any character that is not a `space`	`horse`
`( +[^c]*)`	One or more spaces, followed by a set of characters that are not `c`	`before the`
`(c.*)`	A set of characters starting with `c`, followed by one or more of any character until the end of the string	`cart`

The substitution phrase {\1\4\3\2} reorders the piece such that the first piece is followed by the fourth (`cart`), and the second piece (`horse`) becomes the last.

5.6.4 Back to the Searching URLs

We could use the `regexp` command in place of the `string` or `lsearch` commands to search for `Noumena` . This code would resemble the following.

Example 11

Search Using a `regexp` Test

```
foreach item $urlList  {
  if {[regexp {[Nn]oumena} $item]} {
    puts "NOUMENA PAGE:\n$item"
  }
}
```

Script Output

```
NOUMENA PAGE:
www.noucorp.com "Noumena Corporation"
```

Alternatively, we could just use the `regexp` command to search the original text for a match rather than the text converted to a list. This saves us a processing step.

Example 12

Search All Data Using `regexp`

```
set found [regexp "(\[^\n]*\[Nn\]oumena\[^\n]*)" $urls\
    fullmatch submatch]
if {$found} {
 puts "NOUMENA PAGE:\n $submatch"
}
```

Script Output

```
NOUMENA PAGE:
www.noucorp.com "Noumena Corporation"
```

Let's take a careful look at the regular expression in Example 12.

First, the expression is grouped with quotes instead of braces. This is done so that the Tcl interpreter can substitute the `\n` with a newline character. If the expression were grouped with braces, the characters`\n` would be passed to the regular expression code, which would interpret the backslash as a `regexp` escape character and would look for an ASCII `n` instead of the newline character.

However, since we have enclosed the regular expression in quotes, we need to escape the braces from the Tcl interpreter with backslashes. Otherwise, the Tcl interpreter would try to evaluate `[^\n]` as a Tcl command in the substitution phase. Breaking the regular expression into pieces, we have:

`[^\n]*`	Match zero or more characters that are not newline characters
`[Nn]oumena`	Followed by the word `Noumena` or `noumena`
`[^\n]*`	Followed by zero or more characters that are not newline characters

This will match a string that includes the word `Noumena` and is bounded by either newline characters or the start or end of the string. If `regexp` succeeds in finding a match, it will place the entire matching string in the `fullmatch` variable. The portion of the string that matches the portion of the expression between the parentheses is placed in `submatch`. If the regular expression has multiple sets of expressions within parentheses, these portions of the match will be placed in submatch variables in the order in which they appear.

The `regexp` command in recent versions of Tcl supports a `-line` option. The `-line` option will limit matches to a single line. Using this option, we do not need the `\n` to mark start and end of the line, so we can simplify the regular expression by using curly braces instead of quotes, and simple atoms instead of ranges for the character matches, as follows.

```
set found [regexp -line {(.*[Nn]oumena.*)} \
        $urls fullmatch submatch ]
```

Adding the `-nocase` option will further simplify the regular expression, as follows.

```
set found [regexp -line -nocase {(.*noumena.*)}
        $urls fullmatch submatch ]
```

Alternatively, we could use the submatching support in `regexp` to separate the URL from the description, as follows

Example 13

Script Example

```
set found [regexp —line —nocase {(.*) +(.*noumena.*)} \
        $urls full url desc]
if {$found} {
    puts "full: $full"
    puts "url: $url"
    puts "desc: $desc"
}
```

Script Output

```
full: www.noucorp.com "Noumena Corporation"
url: www.noucorp.com
desc: "Noumena Corporation"
```

5.7 CREATING A PROCEDURE

Identifying one datum in a set of data is an operation that should be generalized and placed in a procedure. This procedure is a good place to reduce the line of data to the information we actually want.

5.7.1 The `proc` Command

A Tcl subroutine is defined with the `proc` command and thus is commonly called a proc.

The `proc` command takes three arguments that define a procedure: the procedure name, the argument list, and the body of code to evaluate when the procedure is invoked. When the `proc` command is evaluated, it adds the procedure name, arguments, and body to the list of known procedures. The command looks very much like declaring a subroutine in C or Fortran, but `proc` is a command that modifies the procedure table when it is evaluated, not a declaration that will be evaluated before the rest of the code. Procedures must be created with the `proc` command before they can be used.

Syntax: `proc` *`name args body`*

Create a new procedure and add it to the procedure list

`name`	The name of the new procedure
`args`	A list of arguments for the new procedure (commonly enclosed in curly brackets)
`body`	The body of the new procedure (commonly enclosed in curly brackets)

Example 14

Script Example

```
#  Define a proc
proc demoProc {arg1 arg2} {
  puts "demoProc called with $arg1 and $arg2"
}
#  Now, call the proc
demoProc 1 2
demoProc alpha beta
```

Script Output

```
demoProc called with 1 and 2
demoProc called with alpha and beta
```

5.7.2 A `findUrl` Procedure

This `proc` will find a line that has a given substring, extract the URL from that string, and return just the URL. It can accept a single line of data or multiple lines separated by newline characters.

Example 15

Script Example

```
#  findUrl —
#    Finds a particular line of data in a set of lines
#  Arguments:
#   match   A string to match in the data
#   text    Textual data to search for the pattern.
#             Multiple lines separated by newline
#             characters.
#  Results:
#   Returns the line which matched the target string
#
proc findUrl {match text} {
  set url ""
  set found [regexp —line —nocase \
      "(.*) +(.*$match.*)" $text full url desc]
  return $url
}
# Invoke the procedure to
# search for a couple of well known sites
#
puts "Noumena site: [findUrl Noumena $urls]"
```

```
puts "Tcl Community site: [findUrl Community $urls]"
if {[string match [findUrl noSuchSite $urls] ""]} {
  puts "noSuchSite not found"
}
```

Script Output

```
Noumena site: www.noucorp.com
Tcl Community site: www.tclcommunityassociation.org
noSuchSite not found
```

5.7.3 Variable Scope

Most computer languages support the concept that variables can be accessed only within certain scopes. For instance, in C, a subroutine can access only variables that are either declared in that function (local scope) or have been declared outside all functions (the extern, or global, scope).

Tcl supports the concept of local and global scopes. A variable declared and used within a `proc` can be accessed only within that `proc`, unless it is made global with the `global` command. The Tcl scoping rules are covered in detail in Chapters 7 and 8.

Tcl also supports namespaces, similar in some ways to the FORTRAN named common (a set of variables that are grouped) or a C++ static class member. The `namespace` command is discussed in Chapter 8.

Global variables must be declared in each procedure that accesses a global variable. This is the opposite of the C convention in which a variable is declared static and all functions get default access to that variable. The `global` command will cause Tcl to map a variable name to a global variable, instead of a local variable.

Syntax: `global` *`varName1 varName2...`*

Map a variable name to a global variable.

`varName*` The name of the variable to map

Example 16

Script Example

```
set a 1
set b 2
proc tst {} {
  global a
  set a "A"
  set b "B"
  puts "$a $b"
}
```

Script Output

```
% tst
A B
% puts "$a $b"
A 2
```

Within the proc `tst` the variable `a` is mapped to the `a` in the global scope, whereas the variable `b` is local. The set commands change the global copy of `a` and a local copy of `b`. When the procedure is complete, the local variable `b` is reclaimed, while the variable `a` still exists. The global variable `b` was never accessed by the procedure `tst`.

5.7.4 Global Information Variables

Tcl has several global variables that describe the version of the interpreter, the current state of the interpreter, the environment in which the interpreter is running, and so on. Some of these variables are simple strings or lists, and some are associative arrays. Associative arrays will be discussed in more detail in the next chapter. The information variables include the following.

`argv` — A list of command line arguments.

`argc` — The number of list elements in `argv`.

`env` — An associative array of environment variables.

`tcl_version` — The version number of a Tcl interpreter.

`tk_version` — The version number of the Tk extension.

`tcl_pkgPath` — A list of directories to search for packages to load.

`errorInfo` — After an error occurs, this variable contains information about where the error occurred within the script being evaluated.

`errorCode` — After an error occurs, this variable contains the error code of the error.

`tcl_platform` — An associative array describing the hardware and operating system the script is running under. The `tcl_platform` associative array has several indices that describe the environment the script is running on. These indices include:

- `user` — The username of the user running the interpreter.
- `byteOrder` — The order of bytes on this hardware. Will be `LittleEndian` or `BigEndian`.
- `osVersion` — The version of the OS on this system.
- `machine` — The CPU architecture (i386, sparc, and so on).
- `platform` — The type of operating system. Will be `macintosh`, `unix`, or `windows`.

`os`	The name of the operating system. On a UNIX system this will be the value returned by `uname -s`. For MS Windows systems it will be `Windows NT`, or `Windows 95`. Microsoft Windows platforms are identified as the base version of the OS. Windows 2000 is identified as Windows NT, and Windows 98 and ME are identified as Windows 95.

New array indices have been added in Tcl8.5 and Tcl8.6. These include:

`threaded`	Will be 1 if the interpreter is compiled with threading enabled, else 0
`pointerSize`	The size of a pointer on this machine.
`wordSize`	The size of a word on this machine.

5.8 MAKING A SCRIPT

As a final step in this set of examples, we will create a script that can read the bookmark file of a well-known browser, extract a URL that matches a command line argument string, and report that URL. This script will need to read the bookmark file, process the input, find the appropriate entry or entries, and report the result. Because the bookmark files are stored differently under UNIX, Mac OS, and MS Windows, the script will have to figure out where to look for the file.

We will use the `tcl_platform` global variable to determine the system on which the script is running. Once we know the system on which the script is running, we can set the default name for the bookmark file, open the file, and read the content.

5.8.1 The Executable Script

Aside from those additions, this script uses the code we have already developed.

```
##+##########################################################################
# geturl.tcl
# Clif Flynt -- clif@cflynt.com
#
# Extracts a URL from a mozilla/firefox bookmark file
#
##+##########################################################################
# findUrl -
# Finds a line in the text string that matches the
# pattern string. Extracts the URL from that line.
#
# Arguments:
# match The pattern to try and match
# text The string to search for a match in.
```

```
#
# Results:
# Returns the matched URL, or ""
proc findUrl {match text} {
  set url ""
  set expression [format {"http://(.*?%s.*?)"} $match]
  regexp -line -nocase $expression $text full url
  return $url
}
##############################################################################
#
# Check for a command line argument
#
if {$argc != 1} {
  puts "geturl.tcl string"
  exit -1;
}

#
# Find a Firefox bookmark file.
# The path to this file depends on the platform and operating system.
#
switch $tcl_platform(platform) {
  unix {
    # unix could be Linux, Darwin, Solaris or something else.
    switch $tcl_platform(os) {
      Darwin {
        set paths [glob [file join $env(HOME) Library \
            "Application Support" Firefox Profiles *default bookmarks.html]]
        set bookmarkName  [lindex $paths end]
      }
      Linux {
        set paths [glob [file join $env(HOME) .mozilla firefox \
            *default bookmarks.html]]
        set bookmarkName  [lindex $paths 0]
      }
      default {
        set paths [glob [file join $env(HOME) .mozilla firefox \
            *default bookmarks.html]]
        set bookmarkName  [lindex $paths 0]
      }
    }
  }
  windows {
    set paths [glob [file join C:/ "Documents and Settings" $env(USERNAME) \
        "Application Data" Mozilla Firefox Profiles *.default bookmarks.html]]
```

```
    set path [lindex [glob $paths] 0]
    # Strip potential { and } from the name returned by glob.
    set bookmarkName [string trim $path "{}"]
  }
  mac -
  macintosh {
    # Mac OS 8, 9 - No Firefox, look for an old Netscape bookmark file.
    # Find the exact path, with possible unprintable characters.
    set path [file join $env(PREF FOLDER) Netsc* * Bookmarks.html]
    # If there are multiple personalities, return just one file
    set path [lindex [glob $path] 0]
    # Strip potential { and } from the name returned by glob.
    set bookmarkName [string trim $path "{}"]
  }
  default {
   puts "I don recognize platform: $tcl platform(platform)"
   exit -1;
  }
}
#
# Open the bookmark file, and read in the data
#
 set bookmarkFile [open $bookmarkName ]
 set bookmarks [read $bookmarkFile]
 close $bookmarkFile

 puts [findUrl $argv $bookmarks]
```

5.9 SPEED

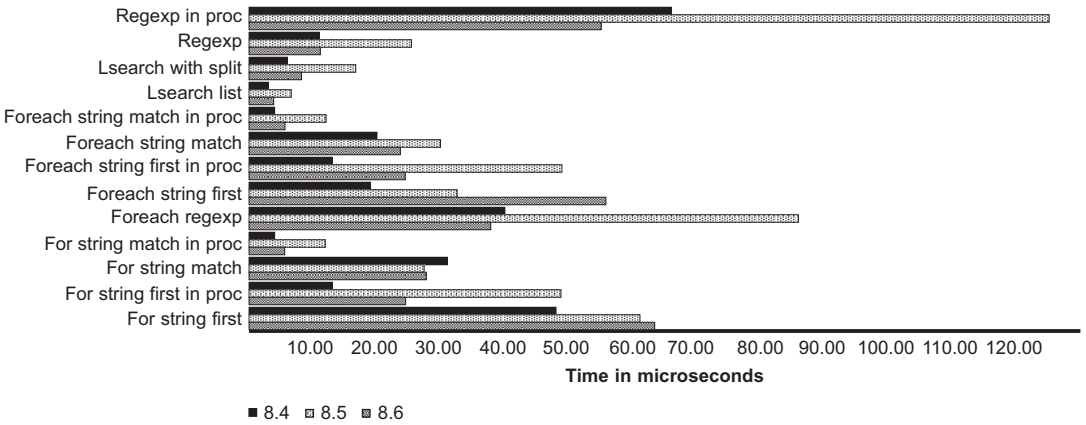
One question that always arises is "How fast does it run?" This is usually followed by "Can you make it run faster?" The `time` command times the speed of other commands.

Syntax: `time` *`cmd ?iterations?`*

Returns the number of microseconds per iteration of the command.

`cmd`	The command to be timed. Put within curly braces if you do not want substitutions performed twice.
`?iterations?`	The number of times to evaluate the command. When this argument is defined, the `time` command returns the average time for these iterations of the command.

The `time` command evaluates a Tcl command passed as the first argument. The `cmd` argument will go through the normal substitutions when `time` evaluates it, so you probably want to put the `cmd` variable within curly braces. We have tried several methods of extracting a single datum from a mass of data. Now, let's look at the relative speeds, as depicted in the following illustration.



5.9.1 Comparison of Execution Speeds (Linux Celeron @ 2.6 GHz)

The graph shows execution time in microseconds of various ways to find the `Noumena` url in the list of urls. Notice that a `foreach` loop is faster than a `for` loop for processing a list of data. This is because using the `foreach` loop doesn't require an extra `lindex` step to extract the list element being tested. Placing code inside a procedure will also speed it up. The procedures are byte-code compiled the first time they are used and the faster byte-code compiled version is retained for future use.

Regular expressions are very powerful, but are generally slower than simple string or list search commands. The first step in using a regular expression is to compile the expression into an internal format, and that can be expensive.

If you can search your entire set of data with a single `lsearch` or `regexp` command that will be faster than iterating through a loop.

Finally, these timing tests were accurate when I ran them. The Tcl interpreter is always being improved, either with new features or new optimizations. Some new features will make the interpreter slower but more powerful until the next round of optimizations. If speed and performance are truly an issue for you, you should run your own tests and consider rewriting the more compute intensive sections of your application in C as described in Chapters 15 and 18.

5.10 BOTTOM LINE

There are some tricks in Tcl that may not be apparent until you understand how Tcl commands are evaluated.

- Tcl variables exist in either a local or global scope, or can be placed in private namespaces.
- A Tcl procedure may execute in its default scope or in the scope of a procedure in its call tree.
- Searching a list with `lsearch` is faster than iterating through a list, checking each item.
- The `time` command can be used to tune your code.
  Syntax: `time` *`cmd ?iterations?`*
- Regular expression string searches are performed with the `regexp` command.
  Syntax: `regexp` *`?opt? expr str ?fullmatch? ?submatch?`*

- Regular expression string substitutions are performed with the `regsub` command.
  Syntax: `regsub` *?opt? expr str subSpec varName*
- You can search a set of data for items matching a pattern with:
  Syntax: `string match` *pattern string*
  Syntax: `string first` *pattern string*
  Syntax: `string last` *pattern string*
  Syntax: `lsearch` *list pattern*
  Syntax: `regexp` *?opt? expr str ?fullmatch? ?submatch?*

5.11 PROBLEMS

The following numbering convention is used in all Problem sections:

Number Range	Description of Problems
100–199	Short comprehension problems review material covered in the chapter. They can be answered in a few words or a 1–5 line script. These problems should each take under a minute to answer.
200–299	These quick exercises require some thought or information beyond that covered in the chapter. They may require reading a man page or making a web search. A short script of 1–50 lines should fulfill the exercises, which may take 10–20 minutes each to complete.
300–399	Long exercises may require reading other material or writing a few hundred lines of code. These exercises may take several hours to complete.

- *100* Some operations are well suited to a `for` loop, whereas others are better suited to a `foreach` or `while` loop. Which loop construct is best suited to each of the following operations?
  - Calculating the Y coordinate for a set of experimentally derived X values
  - Calculating the Y coordinate for X values between 1 and 100
  - Examining all files in a directory
  - Scanning a specific port on a subnetwork to find systems running software with security holes
  - Waiting for a condition to change
  - Inverting a numerically indexed array
  - Iterating through a tree
  - Reversing the order of letters in a string
- *101* Can you use a single `lsearch` command to find two adjacent elements in a list?
- *102* Can you use an `lsearch` command to find the third occurrence of a pattern in a list?
- *103* What regular expression atom will match the following?
  - One occurrence of any character
  - One occurrence of any character between A and L
  - One occurrence of any character except Q

  - The word *Tcl*
  - A single digit

- *104* Given the following Tcl command:
  `regexp $exp $s full first`
  with the variable `s` assigned the string "An image is worth 5 × 10^3 pixels," what string would be assigned to the variable `first` for the following values of `exp`?
  - {({0-9]+)}
  - {({0-9]{2})}
  - {A[^]* +(i[^]*)}
  - {(w.*?[[:>:]])}

- *105* What global variable contains a list of command line arguments?

- *106* What global variable can be used within a script to discover if the script is being evaluated on a Windows or UNIX platform?

- *200* The `lsearch` command will return the index of the first match to a pattern. Write a one-line command (using square brackets) to return the first list element that matches a pattern, instead of the index.

- *201* Write an `lsearch` command that would find the element that:
  - Starts with the letter A and has other characters
  - Starts with the letter A followed by 0 or one integer
  - Starts with the letter A followed by 1 or more integers
  - Starts with the letter A followed by 1 or more integers, with the final integer either 0 or 5
  - Is a number between 0 and 199
  - Is a string with 5 characters
  - Is a string with less than 3 characters

- *202* Write a `regexp` command that will extract the following substrings from the string "`Regular expressions are useful and powerful`". Note that these substrings can be matched by a regular expression that is not a set of atoms identical to the substring.
  - Regular expressions are us
  - expressions
  - pow
  - useful and powerful

- *203* Write a `regexp` command that will extract:
  - The first word from a string
  - The second word from a string
  - A word with the letters `ss` in it
  - A word of 2 to 4 letters long

- *300* Write a Tcl procedure that will split a set of data into a list on multi-character markers, instead of the single-character marker used by `lsearch`. For example, this procedure should split `aaaSPLITbbbSPLITcccSPLITddd` into the list {`aaa bbb ccc ddd`}.

- *301* Write a procedure that will accept a list and return the longest element in that list.
- *302* The `lsearch` command will return the index of the first match to a pattern. Write a procedure that will return the index of the Nth match. The procedure should accept a list, pattern, and integer to define which match to return.
- *303* Modify the procedure from the previous exercise to return the Nth list element that matches a pattern, instead of the Nth index.
- *304* Write a procedure that will accept a list and two patterns and return a list of the indices for the element that matches the first pattern followed by the element that matches the second pattern.
- *305* Modify the procedure from the previous exercise to return the indices of the elements that match the two patterns when the elements are adjacent. This may not be the first occurrence of either element.

CHAPTER

Complex Data Structures with Lists, Arrays and Dicts

6

This chapter demonstrates how lists, arrays and dicts can be used to group data into constructs similar to C structs, linked lists, and trees. Tcl has been accused of being unsuited for serious programming tasks because of the simplicity of its data types. Whereas Java has integers, floats, pointers, structs, and classes, Tcl had only strings, lists and associative arrays. The 8.5 release added the dict data structure to Tcl, and 8.6 adds classes and more. The simple constructs are frequently sufficient. This chapter will show some techniques for using Tcl data constructs in place of the more traditional structs, linked lists, and so on.

The first examples show how lists can be used to group data in ordered and unordered formats, then using keyed lists and simple dicts to collect sparse or duplicated data. The next section explores using associative arrays instead of structs. The final set of examples shows how dicts can be used with nested data sets.

The last section discusses Tcl's support for interacting with SQL databases. The Tcl dict provides a good data structure for this interaction since it can reflect whether a value is NULL or is present but empty.

If you are familiar with compiled languages such as C or Pascal, you may want to consider why you use particular constructs in your programs instead of others. Sometimes, you may do so because of the machine and language architecture, rather than because the problem and the data structure match. In many cases the Tcl data types solve the problem better than the more familiar constructs.

For example, when programming in compiled languages such as C or Pascal, there are several reasons for using linked lists.

- Linked lists provide an open-ended data structure; you do not need to declare the number of entries at compile time.
- Data can be added to or deleted from linked lists quickly.
- Entries in a linked list can be easily rearranged.
- Linked lists can be used as container classes for other data.

The Tcl list supports all of these features. In fact, the internal implementation of the Tcl list allows data items to be swapped with fewer pointer changes than exchanging entries in a linked list. This allows commands such as `lsort` to run very efficiently.

The important reason for using linked lists is that you can represent the data as a list of items. The Tcl list is ideal for this purpose, allowing you to spend your time developing the algorithm for processing your data, instead of developing a linked list subroutine library.

A binary tree is frequently used in C or Pascal to search for data efficiently. The Tcl interpreter stores the indices of an associative array in a very efficient hash table. Rather than implementing

Tcl/Tk: A Developer's Guide. DOI: 10.1016/B978-0-12-384717-1.00006-3

a binary tree for data access purposes, you can use an associative array and get the speed of a good hash algorithm for free. As a tree grows deeper, the hash search becomes faster than a binary search, without the overhead of balancing the tree.

Most applications that use Tcl are not speed critical, and the speed of the list or array is generally adequate. If your application grows and becomes speed bound by Tcl data constructs, you can extend the interpreter with other faster data representations. Extending the interpreter is covered in Chapter 15.

6.1 USING THE TCL LIST

A Tcl list can be used whenever the data is conceptualized as a sequence of data items. These items could be numeric (such as a set of graph coordinates) or textual, such as a list of fields from a database or even another list.

6.1.1 Manipulating Ordered Data with Lists

Lists can be used to manipulate data in an ordered format. Spreadsheet and database programs often export data as strings of fields delimited by a field separator. The Tcl list is an excellent construct for organizing such data within a program. Each spreadsheet row can be a separate list element consisting of the row data. As the rows are processed, the elements can be extracted into a set of variables with the `lassign` command or extracted as needed with the `lindex` command.

Syntax: `lassign` *`list varName1 varName2`*

Extracts N elements from a list and assigns their values to N variables.

`list`	The list to extract values from.
`varName*`	A set of variables to extract values into.

Using the `lassign` command to extract the values of a list in a single command improves code maintenance, since all of the field definitions are in one line of code.

If you need to split the field access across several areas of code, the code maintenance becomes simpler if you use a set of variables to define the locations of the fields in a list, rather than hard-coding the positions in the `lindex` commands. The mnemonic content of the variable names makes the code more readable. This technique also allows you to add new fields without having to go through all your code to change hard-coded index numbers.

Chapters 3 and 5 showed how a comma-delimited line could be split into a Tcl list, with each field becoming a separate list element. The next example manipulates three records with fields separated by colons, similar to data exported by a spreadsheet or saved in a system configuration file (such as `/etc/passwd`).

In this example, each record has four fields in a fixed order: unique key, last name, first name, and e-mail address. The example converts the records to lists with the `split` command and then merges the lists into a single list with the `lappend` command. After the data has been converted to a list, the `lsearch` command is used to find individual records in this list, the `lreplace` command is used to modify a record, and then the list is converted back to the original format. The `join` command is the flip side of the `split` command. It will join the elements of a list into a single string.

Syntax: `join` *`list ?joinString ?`*

Join the elements of a list into a single string.

`list`	The list to join into a string.
`?joinString?`	Use this string to separate list elements. Defaults to a space.

The `lreplace` command will replace one or more list elements with new elements or can be used to delete list elements.

Syntax: `lreplace` *`list first last ?element1 element2 ...?`*

Return a new list, with one or more elements replaced by zero or more new elements.

`list`	The original list.
`first`	The position of the first element in the list to be replaced.
`last`	The position of the last element in the list to be replaced.
`element*`	A list of elements to replace the original elements. If this list is shorter than the number of fields defined by `first` and `last`, elements will be deleted from the original list.

Example 1

Position-oriented Data Example

```
# Define textual data
set text {
KEY1:Flynt:Clif:clif@cflynt.com
KEY2:Doe:John:jxd@example.com
KEY3:Doe:Jane:janed@example.com
}
#  Set up a list
foreach line [split $text \n] {
  # Skip any blank lines
  if {$line eq ""} {
    continue
  }
  lappend data [split $line :]
}

#  data is a list of lists.
#
#  { {KEY1 Flynt Clif clif@cflynt.com}
#    {KEY2 Doe John jxd@example.com} ...}

#  Assign the record positions to mnemonically named variables
set keyIndex 0;
set lastNameIndex 1;
```

```
set firstNameIndex 2;
set eMailIndex 3;

#  Find the record with KEY2
set position [lsearch $data  "KEY2 *"]

#  Extract a copy of that record
set record [lindex $data $position]

#  Display fields from that record
puts  "The Email address for Record [lindex $record $keyIndex] \
    ([lindex $record $firstNameIndex]) was \
    [lindex $record $eMailIndex] "

#  Modify the eMail Address
set newRecord [lreplace $record $eMailIndex $eMailIndex \
    "joed@example.com"]

#  Confirm change
puts  "New Email address for Record [lindex $newRecord $keyIndex] \
    ([lindex $newRecord $firstNameIndex]) is \
    [lindex $newRecord $eMailIndex] "

#  Update the main list
set data [lreplace $data $position $position $newRecord]

#  Convert the list to colon-delimited form, and display it.
foreach record $data {
  puts  "[join $record :]"
}
```

Script Output

```
The Email address for Record KEY2 (John) was jxd@example.com
The Email address for Record KEY2 (John) is joed@example.com
KEY1:Flynt:Clif:clif@cflynt.com
KEY2:Doe:John:joed@example.com
KEY3:Doe:Jane:janed@example.com
```

6.1.2 Manipulating Data with Keyed Lists

In some applications information may become available in an indeterminate order, some fields may have multiple sets of data, and some fields may be missing. It may not be feasible to build a fixed-position list for data such as this. For example, the e-mail standard does not define the order in which header fields must occur, some fields (such as `Subject`) need not be present and there may be multiple `Received` fields.

One solution to representing data such as this is to use a string to identify each piece of data and create pairs of identifier and data. As the data becomes available, the identifier/data pair is appended to the list.

Since a Tcl list can contain sublists, you can use the list to implement a collection of key/value pairs. The records in the next example consist of two-element lists. The first element is a field identifier, and the second element is the field value. The order of these key/value pairs within a record is irrelevant. There is no position-related information, because each field contains an identifier as well as data.

The keyed list data structure is supported in the `tclX` extension. It can also be implemented with a few simple procedures. The `dict` data structure is a more extensive implementation of the keyed list concepts. It will be described in the next section.

The following example shows a set of procedures to store and retrieve data in a keyed list. The sample script places information from an e-mail header into a keyed list, and then retrieves portions of the data.

Example 2

Keyed Pair List Procedures

```
##########################################################################
# proc keyedListAppend {list key value}
#    Return a list with a new key/value element at the end
# Arguments
#    list: Original list
#    key: Key for new element
#    value: Value for new element
proc keyedListAppend {list key value} {
  lappend list [list $key $value]
  return $list
}
##########################################################################
# proc keyedListSearch {list keyName}
#    Retrieve the first element that matches $keyName
# Arguments
#    list:    The keyed list
#    keyName: The name of a key
proc keyedListSearch {list keyName} {
  set pos [lsearch $list "$keyName*"]
  return [lindex [lindex $list $pos] 1]
}
##########################################################################
# proc keyedListRetrieve {list keyName}
#    Retrieve all elements that match a key
# Arguments
#    list:    The keyed list
#    keyName: The name of key to retrieve
#
```

```
proc keyedListRetrieve {list keyName} {
  set start 0
  set pos [lsearch [lrange $list $start end] "${keyName}*"]
  while {$pos >= 0} {
    lappend locations [expr $pos + $start]
    set start [expr $pos + $start + 1]
    set pos [lsearch [lrange $list $start end] "${keyName}*"]
  }
  foreach l $locations {
    lappend rtn [lindex [lindex $list $l] 1]
  }
  return $rtn
}
```

Using the Keyed Pair Procedures

```
# Define a simple e-mail header
set header {
Return-Path: <root@firewall.example.com;
Received: from firewall.example.com
Received: from mailserver.example.com
Received: from workstation.noucorp.com
Date: Tue, 6 Aug 2002 04:13:38 -0400
Message-Id: <200208060813.g768DcP30231>
From: root@firewall.example.com (Cron Daemon)
To: root@firewall.workstation.com
Subject: Daily Report
}

# Initialize a keyed list
set keyedList ""

# Parse the e-mail header into the keyed list.
#    The first ":" marks the key and value for each line.
#
# Note that [split $line :] won't work because of lines
#    with times+amps.
foreach line [split $header \n] {
  set p [string first : $line]
  if {$p < 0} {continue}
  set key [string range $line 0 [expr {$p - 1}]]
  set value [string range $line [expr {$p + 2}] end]
  set keyedList [keyedListAppend $keyedList $key $value]
}

# Extract some data from the keyed list
puts "Mail is from: [keyedListSearch $keyedList From]"
```

```
puts "Mail passed through these systems in this order:"
foreach r [keyedListRetrieve $keyedList Received] {
  puts " [lindex $r 1]"
}
```

Script Output

```
Mail is from: root@firewall.example.com (Cron Daemon)
Mail passed through these systems in this order:
  firewall.example.com
  mailserver.example.com
  workstation.noucorp.com
```

For most lists, this technique works well. However, the time for the `lsearch` command to find an entry increases linearly with the position of the item in the list. Lists longer than 1,000 entries become noticeably sluggish. The pairing of data to a key value can also be done with dicts or associative arrays. Dicts and arrays use hash tables instead of a linear search to find a key, so the speed does not degrade as more records are added.

6.2 USING THE DICT

The `dict` command was added to Tcl in version 8.5. Conceptually, the dict is very similar to the keyed list. Internally, it uses many of the same code constructs as the associative array, making it both fast and efficient.

The keyed list example shown above parsed an email header. This example can be easily reworked to use a dict instead of a keyed list, as shown below.

Example 3

```
# Define a simple e-mail header
set header {
Return-Path: <root@firewall.example.com;
Received: from firewall.example.com
Received: from mailserver.example.com
Received: from workstation.noucorp.com
Date: Tue, 6 Aug 2002 04:13:38 -0400
Message-Id: <200208060813.g768DcP30231>
From: root@firewall.example.com (Cron Daemon)
To: root@firewall.workstation.com
Subject: Daily Report
}

# Parse the e-mail header into the keyed list.
#    The first ":" marks the key and value for each line.
```

```
#
# Note that [split $line :] won't work because of lines
#    with timestamps.
foreach line [split $header \n] {
  set p [string first : $line]
  if {$p < 0} {continue}
  set key [string range $line 0 [expr {$p - 1}]]
  set value [string range $line [expr {$p + 2}] end]
  dict lappend keyedList $key $value
}

# Extract some data from the keyed list
puts "Mail is from: [dict get $keyedList From]"
puts "Mail passed through these systems in this order:"
foreach r [dict get $keyedList Received] {
  puts " [lindex $r 1]"
}
```

Script Output

```
Mail is from: root@firewall.example.com (Cron Daemon)
Mail passed through these systems in this order:
  firewall.example.com
  mailserver.example.com
  workstation.noucorp.com
```

With this short dataset, the `dict` version of the example runs in about 60% of the time that it takes the keyed list version to run. As the data set gets larger, the improved speed of the dict becomes more significant.

6.2.1 Grouping Related Values

Many languages support a data structure for grouping information. In "C" this is the `struct`, and in Pascal it's called a `record`. Arrays of structs are commonly used to hold collections of related information. For example, within the operating system, memory segments and file descriptors are maintained as arrays of structs each of which defines a specific area of memory or open file.

The `dict` command can provide this functionality in Tcl.

C Structure

```
struct {
  int value;
  char desc[80];
} var;
var.value = 1;
strcpy(var.desc,"One");
```

Tcl Dict

```
set var [dict create \
  value 1 desc "One" ]
```

An enumerated array of structures can also be created with the `dict` command like this:

Array of C Structures

```
struct {
  int value;
  char desc[80];
} var[5];

var[0].value = 1;
strcpy(var[0].desc, "One");

var[1].value = 2;
strcpy(var[1].desc, "Two");
```

Tcl Dict

```
set var [dict create \
  0 [dict create \
    value 1 desc "One" ] \
  1 [dict create \
    value 2 desc "Two" ]
]
```

New values can be added to an existing dict with the `dict set` command or the `dict lappend` or `dict append` commands as shown below.

Example 4

Adding Elements to a Dict

```
# Define a dict with 2 elements
set var [dict create  \
    0 [dict create value 1 desc "One"]  \
    1 [dict create value 2 desc "Two"] \
    ]
# Add a new element 2
dict set var 2 [dict create val 3 desc three]

# Add element 3 using lappend
dict lappend var 3 val 4 desc Four

# Add element 4 using append
dict append var 4 [dict create val 5 desc Five]

# Print the value for element 3
puts "The value for element 3 is [dict get $var 3 val]"
```

Script Output

```
The value for element 3 is 4
```

Using numbers to distinguish elements in a dictionary works, but may require a second layer of mapping concepts to values. Unlike a "C" struct or Pascal record, the key for a dict element can be textual and thus have information content other than just the element position.

The next example shows how a set of data can be represented in "C" and with a Tcl dict.

Here is a table of gemstone color and hardness.

Agate	Brown	7.0
Amethyst	Purple	7.0
Aquamarine	Blue	7.5

The next example is a "C" program to store and display this data.

Example 5
Saving Data in an Array of "C" Structs

```
#include <stdio.h>
#include <strings.h>

main () {
  struct {
    char name[20];
    char color[10];
    float hardness;
  } gems[3];

  int i;

  strcpy(gems[0].name, "Agate");
  strcpy(gems[1].name, "Amethyst");
  strcpy(gems[2].name, "Aquamarine");

  strcpy(gems[0].color, "Brown");
  strcpy(gems[1].color, "Purple");
  strcpy(gems[2].color, "Blue");

  gems[0].hardness = 7.0;
  gems[1].hardness = 7.0;
  gems[2].hardness = 7.5;

  for (i = 0; i < 3; i++) {
    printf("%s is colored %s with %.1f hardness\n", \
        gems[i].name, gems[i].color, gems[i].hardness);
  }
}
```

When compiled and run this program generates the following output:

```
Agate is colored Brown with 7.0 hardness
Amethyst is colored Purple with 7.0 hardness
Aquamarine is colored Blue with 7.5 hardness
```

An array of structs has no obvious relationship between the key to the data (the numeric index) and the contents. This makes it difficult to maintain this type of code. It's not obvious that the hardness value of 7.5 is connected to the name value Aquamarine unless you look at the index values.

The next example shows how this data can be represented in a dict. Notice that the name, color and hardness are all kept together, making it easier to see the data relation. This example generates the same output as the previous "C" struct example.

Example 6

Saving Data in a Nested Dict

```
set gems [dict create \
  Agate [dict create color Brown  hardness 7.0] \
  Amethyst [dict create color Purple  hardness 7.0] \
  Aquamarine [dict create color Blue  hardness 7.5] \
]

foreach key [dict keys $gems] {
  puts "$key is colored [dict get $gems $key color] with \
      [dict get $gems $key hardness] hardness"
}
```

Unlike a "C" struct, the dictionary is a dynamic data structure. More gemstones can be added without needing to change the `gem` array declaration and recompile. You can even add new fields to the Tcl dictionary without redefining the data structure.

The next example adds the family of the gemstone to the dictionary and then displays all the characteristics and values of all the gems. Notice that using the `dict get` to extract the keys and values for each gemstone makes the output section of the code generic. The code will continue to work when fields are added or deleted.

Example 7

```
dict lappend gems Agate family Quartz
dict lappend gems Amethyst family Quartz
dict lappend gems Aquamarine family Beryl

foreach key [dict keys $gems] {
  set str "$key"
  foreach {character value} [dict get $gems $key] {
    append str " $character is $value"
  }
  puts "$str."
}
```

Script Output

```
Agate color is Brown hardness is 7.0 family is Quartz.
Amethyst color is Purple hardness is 7.0 family is Quartz.
Aquamarine color is Blue hardness is 7.5 family is Beryl.
```

6.3 USING THE ASSOCIATIVE ARRAY

Like the keyed list and the dict, the associative array links a data value to a key. The differences are that the associative array does not put the data in a given order and the syntax for using an associative array is simpler than the dict or keyed list.

The Tcl associative array can be used just like a C or Fortran array by setting the indices to numeric values. If you are familiar with Fortran or Basic programming, you might be familiar with coding constructs such as the following.

C Arrays

```
int values[5];
char desc[5][80];
values[0] = 1;
strcpy(desc[0], "First");
```

Tcl Array

```
set values(0) 1;
set desc(0) "First"
```

When programming in Tcl, you can link the value and description together more efficiently by using a nonnumeric index in the associative array.

```
set value("First") 1;
```

Data that consist of multiple items that need to be grouped together are frequently collected in composite data constructs such as a C struct or a Pascal record. These constructs allow the programmer to group related data elements into a single data entity, instead of several entities. The data elements within a struct or record can be manipulated individually.

Grouping information in a struct or record is conceptually a naming convention which the compiler enforces for you. When you define the structure, you name the members and define what amount of storage space they will require. Once this is done, the algorithm developer generally does not need to worry about the internal memory arrangements. The data could be stored anywhere in memory, as long as a program can reference it by name. You can group data in an associative array variable by using different indices to indicate the different items being stored in that associative array, which is conceptually the same as a structure.

C Structure

```
struct {
  int value;
  char desc[80];
} var;
var.value = 1;
strcpy(var.desc,"First");
```

Tcl Array

```
set var(value) 1
set var(desc) "First"
```

It may not be immediately obvious, but the Tcl variable `var` groups the description and value together just as a struct would do. Another common C data construct is the array of structs. Again, so far as your algorithm is concerned, this is primarily a naming convention. By treating the associative array index as a list of fields, separated by some obvious character (in the following example, a period is used), this functionality is available in Tcl.

Array of C Structures

```
struct {
  int value;
  char desc[80];
} var[5];
var[0].value = 1;
strcpy(var[0].desc, "First");
var[1].value = 2;
strcpy(var[1].desc, "Second");
```

Tcl Array

```
set var(0.value) 1
set var(0.desc) "First"
set var(1.value) 2
set var(1.desc) "Second"
```

You can create naming conventions to group data in Tcl, but the Tcl interpreter does not enforce adherence to any naming convention. You can enforce a convention by writing procedures that will hide the conventions from people using a package.

6.4 TREES IN TCL

You do not need to use a binary tree in Tcl for data access speed. The dict and associative array provide fast access to data. However, sometimes the underlying data is best represented as a tree. A tree is the best way to represent a set of data that subdivides into smaller and smaller subsets. For example, a file system is a single large disk divided into directories, which are further divided into subdirectories and files. A tree can represent a set of data that has inherent order (with possible branches), such as the steps in an algorithm.

Tcl does not provide a binary tree as a built-in data type; however, the `dict` data structure can be nested to create a tree view of a data set. Since the underlying data structure of a dict is a hash table, not an actual tree, there is no need to balance the tree.

The example below recursively searches a folder and then returns the folder's contents as a nested dictionary. Each folder is a key and the associated value is a dict of the folder's contents.

Example 8

File System as Dict

```
proc fileDict {parent} {
  set dct {}
  foreach fl [glob –nocomplain $parent/*] {
    if {[file type $fl] eq "directory"} {
      dict set dct $fl [fileDict $fl]
    } else {
```

```
      dict set dict $fl {}
    }
  }
  return $dict
}
```

To test this, we need a small test folder, which can be created in Linux as shown below.

Example 9

Create a Set of Folders and Files

```
$> mkdir /tmp/a
$> mkdir /tmp/a/b
$> mkdir /tmp/a/c
$> echo X > /tmp/a/a.txt
$> echo X > /tmp/a/b/ab.txt
$> echo X > /tmp/a/c/ac.txt

$> tclsh fileDict.tcl /tmp/a
```

Script Output

```
/tmp/a/a.txt {} /tmp/a/b {/tmp/a/b/ab.txt {}}
    /tmp/a/c {/tmp/a/c/ac.txt {}}
```

It's easier to visualize a tree if it's printed in a pretty format with indentation to denote the nesting, rather than as a raw dictionary. Tcl does not include a pretty-printer for dictionaries, but it's not difficult to create one.

Example 10

A Nested Dict Pretty-Printer

```
proc showDict {dct indent} {
  # If dict keys fails, this is not a dictionary.
  # Print it and return.
  if {[catch {dict keys $dct}]} {
    puts "[string repeat " " $indent]$dct"
    return
  }

  # Step through the keys in this dictionary and recurse
  foreach k [dict keys $dct] {
    puts "[string repeat " " $indent]$k"
```

```
        set v [dict get $dct $k]
        showDict $v [expr {$indent+2}]
    }
}
```

Using this procedure with the output from the previous test shows the indenting like this:

Script Output

```
/tmp/a/a.txt
/tmp/a/b
  /tmp/a/b/ab.txt
/tmp/a/c
  /tmp/a/c/ac.txt
```

6.5 TCL AND SQL

The data structures discussed in this chapter are suitable for organizing temporary data within an application or even for small-to-medium sized data sets saved on a disk. You should use a database engine for large or complex persistent data sets.

Tcl has supported many database engines since version 7 including Oracle, Sybase, Postgres, MySQL, ODBC and Sqlite. Each of these databases required a special extension and added a slightly different set of new commands for accessing the underlying database.

Release 8.6 of Tcl includes the `tdbc` (Tcl DataBase Connection) package which provides a single interface to many database engines.

The `dict` data structure is very well suited for working with SQL databases and is used with `tdbc` to pass values to and from a database engine.

The next sections will introduce enough SQL to understand the examples and the Tcl `tdbc` package. If you need to know SQL, there are many books that will provide more details.

If you are already familiar with SQL, you can skip the next section.

6.5.1 SQL Basics

The data in an SQL database is contained in tables. A table can be conceptualized as a spreadsheet. Each column along the top is a field like “First Name”, “Last Name”, etc.

Each row, reading down, is a single element in the table with values for the fields.

A table of authors might look like this:

ID	First	Last
0	Clif	Flynt
1	Ken	Jones
2	Mark	Twain

There are several commands in the SQL specification to define and access the tables. The commands are not case sensitive, but it's common to capitalize the commands to distinguish them from the surrounding data values.

You define and create a table with the `CREATE TABLE` command.

Syntax: `CREATE TABLE` *`tableName ( field1, field2 );`*

Create and define the fields in a table.

`tableName`	The name of the table being defined.
`field*`	A list of the fields and field descriptors.

The descriptor values must describe the type for the data and may define other parameters. The type of data may be one of several types including:

`integer`	integer value
`text`	string value
`varchar`	variable length string

Other parameters may include:

`primary key`	Marks this field as the primary identifier for a row.
`auto_increment`	Marks this field to be incremented to a new value with each insertion.
`not null`	Marks this field to not accept a null value.
`default` *`value`*	Defines the value to place in this field if it is otherwise undefined.

The command below defines and creates an SQL table. This table will hold a collection of authors. It has text fields to hold the first and last names, and an integer id that will hold a unique value for each record in this table. The `AUTO_INCREMENT` parameter tells the database engine to automatically assign the next higher value whenever new data is inserted into this table.

```
CREATE TABLE author (
  id INTEGER PRIMARY KEY AUTO_INCREMENT,
  first TEXT,
  last TEXT
);
```

Data is added to a table with the `INSERT` command.

Syntax: INSERT INTO *tableName (columns)* VALUES *(values)*

Insert values into selected columns of a table.

tableName	The name of the table to receive the data.
columns	A comma delimited list of columns to receive data in the order in which the data will be presented.
values	A comma delimited list of values in the order defined in the *columns* field..

The next example shows how rows can be added to the author table. Notice the single quotes around the names. SQL requires single quotes to define a string. You insert a single quote into a string with a pair of single quotes.

```
INSERT INTO author (first, last) VALUES ('Clif', 'Flynt');
INSERT INTO author (last, first) VALUES ('Jones', 'Ken');
```

Rows are retrieved from a table with the SELECT command.

Syntax: SELECT *fields* FROM *tableName* WHERE *test*

Select data in the defined fields one or more rows from a table which match a test.

fields	A comma delimited list of the columns to select data from.
tableName	Name of the table to select data from.
test	One or more tests to apply to each row to determine whether or not to select it.

The *test* in a SELECT statement is a boolean expression using the operators shown below. A set of tests can be negated with the NOT keyword and multiple tests can be joined with AND or OR.

columnName = *value*	Contents of a column exactly match a value.
columnName like *value*	Contents of a column match a wildcard value. The wildcard symbol is a % symbol.
columnName is NULL	The data in this column was never defined.

The next example shows some selections and what they will return using the previously defined table and data.

Example 11

Select All Fields, All Rows

```
SELECT * FROM author;
```

```
1, Clif, Flynt
2, Ken, Jones
```

Select Only Last Name, All Rows

```
Select last FROM author;

Flynt
Jones
```

Select First and Last Name Where Last Name Includes the Letter y

```
SELECT first, last FROM author WHERE last like '%y%';

Clif,  Flynt
```

One feature of relational databases is that a field can point to a record in another table. This means that data only has to be in the database once in order to be referenced many times.

The next example creates a book table that references the authors in the author table.

Example 12

```
CREATE TABLE book (
  id INTEGER PRIMARY KEY AUTO_INCREMENT,
  authorID INTEGER REFERENCES author,
  title TEXT
);
```

We can populate the book table using the id value from the author table as shown below:

```
INSERT INTO book (authorID, title) \
    VALUES (1, 'Tcl/Tk: A Developer"s Guide');
);
```

6.5.2 Using `tdbc`

The `tdbc` package is included with the 8.6 release of Tcl. It provides a database-engine neutral view of SQL databases. The `tdbc` package supports all the commands that the underlying database engine supports with either line-by-line or bulk data returns.

This section will describe how to use the `tdbc` package and show how to use the `dict` commands to interact with it.

The basic program flow for using `tdbc` looks like this:

1. Load `tdbc` packages.
2. Create a new command to interact with a database engine.
3. Interact with the database using the new command.
4. Close the new SQL command when all interactions are complete.

There are several ways to interact with the database including single SQL commands, loops, precompiled SQL commands and variable substitutions. These techniques will be covered working from the simple techniques to the more complex.

In order to use the `tdbc` package you must first load the packages. Your code must include both `tdbc` and one or more of the `tdbc` driver packages. The driver packages provide the database neutral interface to the database engines.

The `package include` commands will look like this:

```
# Load the base tdbc package
package require tdbc

# Include the sqlite driver
package require tdbc::sqlite3

# Include the mysql driver
package require tdbc::mysql
```

Once the packages are loaded, the next step is to create a new command to interact with the databases.

Syntax: `tdbc::`*`driver`*`::connection create` *`name -key value`*

Create a connection between the Tcl script and the database engine.

`driver`	The type of database to be connected to. Options are `sqlite3`, `mysql`, `oracle`, etc.
`name`	The name for the new command that is created for this connection.
`-key value`	Keys and values specific to a database engine.

The code to create a connection resembles the next example. The first line creates a new command named `sqliteDB` that connects to an sqlite3 database named `sqliteDB.sql`. The second command creates a new command named `mysqlDB` that connects to the mySql database `testDB` as user `user` with a password of `password`.

```
# Create an sqlite3 database connection.
tdbc::sqlite3::connection create sqliteDB sqliteDB.sql

# Create a mySql database connection

tdbc::mysql::connection create mysqlDB –db testDB \
    –user user –password password
```

The new database connection command has several subcommands. These subcommands include commands to manipulate the data in the tables as well as commands to examine and modify the connection to the database and to access the database metadata such as the names of tables, foreign keys, primary keys and constraints.

These subcommands are supported in tdbc version 1.0. Not all of the introspection commands are supported for all database drivers.

`allrows`	Evaluate an SQL statement and return all rows that are selected.
`foreach`	Evaluate an SQL statement and evaluate a Tcl script for each row.
`prepare`	Prepare an SQL command for future use.
`preparecall`	Prepare a call to a stored procedure.
`statements`	Returns a list of statements prepared with `prepare` or `preparecall`.
`resultsets`	Returns a list of resultsets created by evaluating a prepared command.
`begintransaction`	Start a transaction.
`commit`	Commit (accept) all the updates done during a transaction.
`rollback`	Rollback (discard) any modifications done during a transaction.
`transaction`	A shorthand for collecting a set of Tcl commands into a transaction. If the Tcl command set fails, the actions are rolled back, else they are committed.
`configure`	Return or modify the connection options.
`foreignkeys`	Returns a dictionary with information about a table's foreign keys.
`primarykeys`	Returns a dictionary with information about the table including the table's primary key.
`tables`	Returns a list of the tables that exist in the database.
`columns`	Returns a list of the columns in a table.
`close`	Close the connection to the database.

Manipulating Data

The `allrows` and `foreach` subcommands can be used to interact directly with the database. The `prepare` and `execute` subcommands are used to prepare and delay execution of the SQL statement. If your application needs to repeat the same SQL command multiple times, it is better to prepare the command and save it until it's needed.

The `allrows` command will evaluate an SQL command. If the command is a `SELECT` command, it will return all the rows that are selected. You can also use the `allrows` subcommand for commands that do not generate any return values.

Syntax: *dbCmd* `allrows` ?`-as` *format? SQL ?dict?*

Evaluates an SQL command and returns the results in the requested format.

dbCmd	The database command created with the `tdbc::*::create` command.
format	The format for the returned data. Options are `dict` or `list`.
SQL	An SQL statement to be evaluated.
dict	A dictionary that maps a set of variable names to values.

The next example creates the author and book tables and then inserts two authors.

Example 13

Using* `db` *Command to Create and Populate Tables

```
dbBook allrows {
   CREATE TABLE author (
     id INTEGER PRIMARY KEY AUTO_INCREMENT,
     first TEXT,
     last TEXT);

   CREATE TABLE book (
     id INTEGER PRIMARY KEY AUTO_INCREMENT,
     authorID INTEGER REFERENCES author,
     title TEXT);
}

foreach {f l} {Clif Flynt Ken Jones} {
  dbBook allrows \
      "INSERT INTO author (first,last) VALUES ('$f', '$l')"
}
```

The `allrows` subcommand will return all results as a set of dictionaries or as a set of lists. The default is to return the results as a set of dicts. Each entity returned is a separate dictionary.

The next example shows the output from using the `allrows` command to extract the contents of the author table as dicts and lists.

Example 14

Display Database Returns as Dicts and Lists

```
puts "As dicts: [dbBook allrows \
    {SELECT first, last FROM author}]"
puts "As lists: [dbBook allrows –as lists \
    {SELECT first, last FROM author}]"
```

Script Output

```
As dicts: {first Clif last Flynt} {first Ken last Jones}
As lists: {Clif Flynt} {Ken Jones}
```

The list format return does not distinguish between empty and NULL. The `dict` format will return an empty value for fields in which there is an empty string but will not include either key or value for fields that are not defined. If your application treats an empty field the same as an undefined field, you may use the `list` format. If your application needs to distinguish between empty contents in a field or a NULL (undefined) value in the field, your code must use the `dict` format.

The next example shows the differences when a field contains an empty string or is undefined. The author Saki had no first name (Saki was a single-name pseudonym), so it is undefined. The dict format return has no `first` key, while the list format return shows an empty string. Mark Twain used first and last names, but the first name is defined as an empty string in this example. Both the dict and list return show this as an empty string.

Example 15

Empty and Undefined Fields

```
dbBook allrows \
    "INSERT INTO author last VALUES ('Saki')"
dbBook allrows \
    "INSERT INTO author (first, last) VALUES ('', 'Twain')"

puts "As Dicts: [dbBook allrows \
    {SELECT first, last FROM author WHERE last like '%a%';}]"

puts "As lists: [dbBook allrows –as lists\
    {SELECT first, last FROM author WHERE last like '%a%';}]"
```

Script Output

```
As Dicts: {last Saki} {first {} last Twain}
As lists: {{} Saki} {{} Twain}
```

The `allrows` command returns all of the results of a search in a single list or dict. This can be convenient for small datasets, but may use too many machine resources if the returned dataset is large (say, a few terabytes). If the dataset will be large, or if you intend to process each line, it's better to use the `foreach` subcommand to iterate through the rows.

Syntax: *dbCmd* `foreach ?-as` *format? varname SQL ?dict?* `script`

Evaluates an SQL command and returns the results in the requested format.

dbCmd	The database command created with the `tdbc::*::create` command.
format	The format for the returned data. Options are `dict` or `list`.
varName	The name of a variable to be hold each row.

SQL	An SQL statement to be evaluated.
script	A Tcl script to be evaluated for each row returned.
dict	A dictionary that maps a set of variable names to values.

Example 16

Using the `tdbc foreach`

```
dbBook foreach row {SELECT * FROM author} {
  puts "Dict $row"
}
dbBook foreach -as lists row {SELECT * FROM author} {
  puts "List $row"
}
```

Script Output

```
Dict id 1 first Clif last Flynt
Dict id 2 first Ken last Jones
List 1 Clif Flynt
List 2 Ken Jones
```

6.5.3 Using Referenced Tables

In order to insert rows into the book table we need to know the author id for the book. The simple (and sometimes best) way to do this is to query the database for the relation tables as new rows are inserted, as shown in the next example.

Example 17

Inserting Rows with References

```
foreach {last title} {
  Flynt {Tcl/Tk: A Developer''s Guide}
  Jones {Practical Programming in Tcl/Tk} } {
  set authorID [dbBook allrows -as lists \
      "SELECT id FROM author WHERE last='$last'"]
  dbBook allrows "INSERT INTO book (authorID, title) \
      VALUES ($authorID, '$title')"
}
```

The list format return can be used to initialize an associative array. Depending on the size of your data set and use frequency it's often faster to cache values in an associative array rather than hit the database for every lookup. This technique is particularly useful when there are many records in one table that have references to another smaller table and you need to map the reference to values in the referenced table.

The next example is similar to the previous one, except that it uses an associative array instead of accessing the database for each row to be inserted. The `lookup` associative array uses the author's last name as the index, and the `authorID` value as the array value. This allows the application to map from author last name to `authorID` without a database call.

Example 18

Using an Associative Array to Cache References

```
dbBook foreach –as lists xx \
    "SELECT last,id FROM author" {
    array set lookup $xx
}

foreach {last title} {
  Flynt {Tcl/Tk: A Developer''s Guide}
  Jones {Practical Programming in Tcl/Tk} } {
  dbBook allrows "INSERT INTO book (authorID, title) \
      VALUES ($lookup($last), '$title')"
}
```

If a statement will be used many times, it is better to use the `prepare` subcommand to pre-process the SQL statement. The `prepare` subcommand returns the name of a new command that supports several subcommands including `allrows`, `foreach` and `execute`.

Syntax: *dbCmd* `prepare` *SQL*

Prepare an SQL command for future evaluation.

dbCmd The database command created with the `tdbc::*::create` command.

SQL An SQL statement to be evaluated. This may include variables for substitution.

The `allrows` and `foreach` commands are similar to the previously discussed subcommands. The main difference is that the connection subcommands require an SQL statement to evaluate while the prepared commands have the SQL already embedded. The SQL command can include variables which will be substituted when the SQL is evaluated.

Syntax: *dbCmd* `allrows` *?-as format? ?dict?*

Evaluates an SQL command and returns the results in the requested format.

dbCmd The database command created with the `tdbc::*::create` command.

format The format for the returned data. Options are `dict` or `list`.

?dict? An optional dictionary that sets values for variables in the SQL command.

A variable is defined in the SQL command by starting the variable name with the colon (:) character. When the SQL command is evaluated the variables inside the SQL command are substituted. The values can be defined by supplying a dictionary of names and values. If no dictionary is supplied and there are Tcl variables with the same name as the SQL variables, the value of the Tcl variables is substituted into the SQL command.

The next example demonstrates the `prepare` command, the prepared command's `execute` subcommand and variable substitution. These three features are used to populate the author table. A new command is created to select values from the author table and the `foreach` command is used to extract lines from that table.

Example 19

```
package require tdbc

# Include the sqlite driver
package require tdbc::sqlite3

# Create an sqlite3 database connection.
tdbc::sqlite3::connection create dbBook sqliteDB.sql

dbBook allrows {
   CREATE TABLE author (
     id INTEGER PRIMARY KEY AUTO_INCREMENT,
     first TEXT,
     last TEXT);
}

# Prepare a command for inserting into author table
set authorInsert [dbBook prepare {
  INSERT INTO author (first, last) VALUES (:first, :last);
  } ]

foreach first {Clif Ken Mark} last {Flynt Jones Twain} {
  $authorInsert execute
}


# Prepare a command to select rows from the author table
# pattern is defined in a dict.
set authorSearch [dbBook prepare {
  SELECT * from author WHERE last like  :pattern;
} ]

$authorSearch foreach rowDict {pattern %n%} {
  puts "rowDict: $rowDict"
}
```

Script Output

```
rowDict: id 1 first Clif last Flynt
rowDict: id 2 first Ken last Jones
rowDict: id 3 first Mark last Twain
```

Introspection into Databases

The `tdbc` package provides tools to examine a database's meta-data as well as the data in the tables. The introspection commands include the `configure`, `foreignkeys`, `primarykeys`, `tables` and `columns`.

The `configure` command provides access to database connection options. Database engines support different sets of configuration options. The `configure` command returns a dictionary that lists the supported options and their values.

Syntax: *dbCmd* `configure` *?-parameter? ?value?*

Returns or modifies configurable options for a database connection.

dbCmd	A database connection command defined with `tdbc::`*driver* `connection`
-parameter	A configurable option for this database. If no -parameter is provided, a dictionary of all appropriate configurable parameters and their values is returned. If an *-parameter* is provided, but no value, the value for just this parameter is returned. If both *-parameter* and *value* are provided, the value for this parameter is modified.
value	The new value for this parameter.

This example shows the return value for connections to an SQLite database and a MySql database. SQLite is a single connection database, so it does not need a user and password to define a connection. MySql supports multiple connections, and thus needs a user and password to create a connection. These differences (and others) are shown in the `configure` output.

Example 20

Configure Options

```
# Create an sqlite3 database connection.
tdbc::sqlite3::connection create sqliteDB sqliteDB.sql

# Create a mySql database connection

tdbc::mysql::connection create mysqlDB –db testDB \
    –user user –password password

# Report the sqlite options
puts "SQLite: [sqliteDB configure]"
```

```
# Report the mysql options

puts "MySql: [mysqlDB configure]"
```

Script Output

```
SQLite: -encoding utf-8 -isolation serializable
  -readonly 0 -timeout 0

MySql:  -compress 0 -database testDB -encoding utf-8
  -host localhost -interactive 0 -isolation repeatableread
  -password {} -port 3306 -readonly 0
  -socket /var/run/mysqld/mysqld.sock -ssl_ca {}
  -ssl_capath {} -ssl_cert {} -ssl_cipher {} -ssl_key {}
  -timeout 28800 -user user@localhost
```

The `tables` and `columns` provide introspection into the database schema. The values returned by these commands vary depending on the underlying database program and the Tcl driver. If your application needs to be able to examine the database it's connected to (for instance, to generate input forms automatically), you will need to check the exact return values for the database and drivers you are using.

Syntax: *dbCmd* `tables` *?pattern?*

Returns a dictionary providing information about one or more tables.

dbCmd	A database connection command defined with `tdbc::`*driver* `connection`
pattern	A pattern to select which table's information will be returned. If no pattern, all tables are selected.

The next example shows the returns for SQLite and MySql databases using the author and book tables that we've defined in previous examples.

Example 21

Tables

```
# Create an sqlite3 database connection.
tdbc::sqlite3::connection create sqliteDB sqliteDB.sql

# Create a mySql database connection

tdbc::mysql::connection create mysqlDB -db testDB \
    -user user -password password

puts "SQLite Tables: [sqliteDB tables]"
puts "MySql Tables: [mysqlDB tables]"
```

Script Output

```
SQLite Tables:
 author {type table name author tbl_name author rootpage 2
   sql {CREATE TABLE author (
     id INTEGER PRIMARY KEY AUTO_INCREMENT,
     first TEXT,
     last TEXT)
   }
 }
 book {type table name book tbl_name book rootpage 3
   sql {CREATE TABLE book (
     id INTEGER PRIMARY KEY AUTO_INCREMENT,
     authorID INTEGER REFERENCES AUTHOR,
     title TEXT)
   }
}

MySql Tables: author {} book {}
```

The `tables` command returns information about one or more tables. The `columns` command returns internal data about the columns in a specific table.

Syntax: *dbCmd* `columns` *table ?pattern?*

Returns a dictionary providing information about one or more fields in a table. The return is a nested dictionary in which the key is the name of a field and the value is another dictionary of parameters and values.

dbCmd	A database connection command defined with `tdbc::`*driver* `connection`
table	The table to return column information for.
pattern	A pattern to select which column's information will be returned. If no pattern, all columns in the table are selected.

Example 22

Columns

```
puts "SQLite: \n[sqliteDB columns author]"

puts "MySql: \n[mysqlDB columns author]"
```

Script Output

```
SQLite:
id {cid 0 name id type integer notnull 0 pk 1 precision 0
    scale 0 nullable 1}
```

```
first {cid 1 name first type text notnull 0 pk 0 precision
    0 scale 0 nullable 1}
last {cid 2 name last type text notnull 0 pk 0
    precision 0 scale 0 nullable 1}


MySql:
id {name id type integer precision 11 scale 0
    nullable 0}
first {name first type text precision 65535 scale 0
    nullable 1}
last {name last type text precision 65535 scale 0
    nullable 1}
```

The `primarykeys` and `foreignkeys` return information about the primary keys and foreign keys (references) in one or more tables. These commands return a dictionary that describes the parameters and values that the underlying database and driver support.

The example below shows the return from the `primarykeys` command when sent to a MySql connection with two databases that contain a table named `author`.

Example 23

Columns Use

```
puts [mysqlDB primarykeys author]
```

Script Output

```
{tableSchema Library1 tableName author
  constraintSchema Library1 constraintName PRIMARY
  columnName id ordinalPosition 1}
{tableSchema testDB tableName author
  constraintSchema testDB constraintName PRIMARY
  columnName id ordinalPosition 1}
```

The `primarykeys`, `tables` and `columns` subcommands can be used to make a single generic entry procedure for databases. The next example shows a generic prompt/response type interaction to query a user for fields and data to enter.

If the `AUTO_INCREMENT` parameter is set, the value of the primary key is set by the database engine, rather than being defined in the `INSERT` command. This example uses the `primarykeys` command to determine the primary key and only query for input on the non-primary key fields.

When a user provides a value, the name of the field and the new value are appended to lists to be inserted into the SQL command and values are set in the dictionary to be passed to the `allrows` command.

Example 24

```
##########################################################################
# proc promptResponse {dbCmd table}--
#    Query a user for values to insert into a table
# Arguments
#   dbCmd - A command as returned from a tdbc::connection call
#   table - The name of the table to recieve a new row
#
# Results
#   The database is modified by having a new row inserted.
#
proc promptResponse {dbCmd table} {

  # Find the primary key for this table
  foreach d [$dbCmd primarykeys $table] {
    # If this dictionary references the requested table,
    #  get the name of the primary key field.
    if {![dict exists $d tableName] ||
        ([dict get $d tableName] eq $table)} {
      set primary [string tolower [dict get $d columnName]]
    }
  }

  # Step through the filed names and dictionaries
  #  returned by the columns command.

  foreach {field dct} [$dbCmd columns $table] {
    # If this field is not the primary key
    # query the user for a value

    if {[string tolower $field] ne $primary} {

      # extract the type and name from the dictionary
      set type [dict get $dct type]
      set name [dict get $dct name]

      # Query the user and accept the user input
      puts -nonewline "$field ($type): "
      set input [gets stdin]
      flush stdout

      if {$input ne ""} {
        # Append the name of this field list of fields
        lappend fields "$name"

    # Append the name to a list of variables
```

```
      lappend vars ":$name"

      # set a dictionary key/value pair
      # to map field name to a value
      dict set data $name $input
        }
      }
    }

    # Join the lists into strings for the SQL command.
    set fields [join $fields ,]
    set vars [join $vars ,]

    # Show the command that will be invoked to INSERT
    # data into the table.

    puts "Insert Command is:"
    puts "$dbCmd allrows {INSERT INTO $table ($fields) "
    puts "    VALUES ($vars)} {$data}"

    # Insert the data into the database.
    $dbCmd allrows "INSERT INTO $table ($fields) VALUES ($vars)" $data
}
```

Script Output

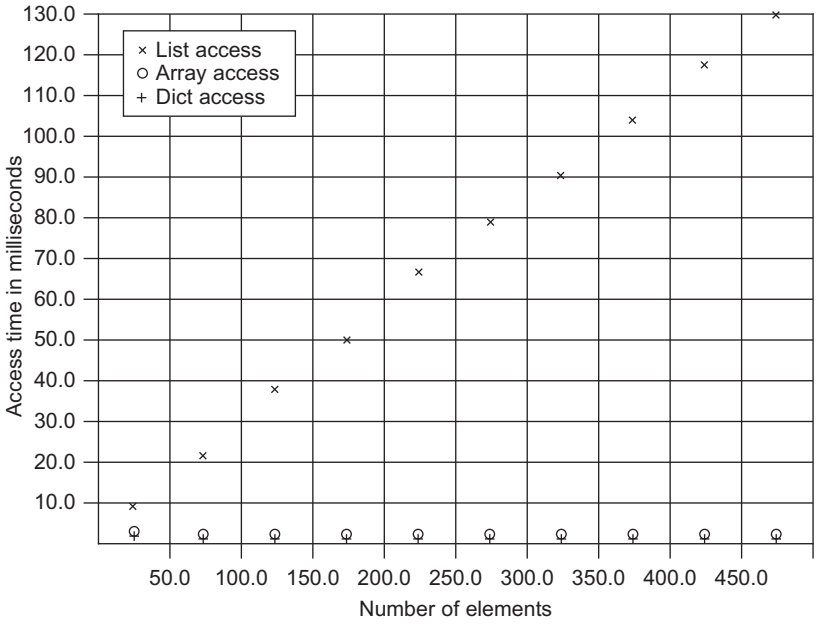
```
first (text): Clif
last (text): Flynt
Insert Command is:
mysqlDB allrows {INSERT INTO author (first,last)
    VALUES (:first,:last)} {first Clif last Flynt}
```

6.6 PERFORMANCE

So, how do lists, dicts and associative arrays compare when it comes to accessing a particular piece of data? The following example shows the results of plotting the list, dict and array access times.

Note that the time to access the last element in a list increases linearly with the length of the list. In Tcl 8.3, the list-handling code was rewritten to improve the performance, but even with that performance improvement the access speed for dict and array elements is much faster and does not degrade as the number of elements increases.

Note that the performance between `lsearch` and an array access is not significantly different until you exceed 100 elements in the list. If your lists are shorter than 100 elements, you can use whichever data construct fits your data best.



The next examples show the code that was used to generate the accompanying graph. The Tcl `time` command returns clock time (not CPU time) that it takes a command to run. This can be influenced by other background tasks that might interrupt the Tcl interpreter and extend the clock time. Background tasks may make a set of code take longer, but will never make code run faster. This is the reason for running the test multiple times and taking the minimum time.

Example 25

List Element Access Time

```
for {set i 0} {$i < 500} {incr i} {
  set x "abcd.$i.efg"
  lappend lst $x
  set min [lindex [time {lsearch $lst $x} 100] 0]
  for {set j 0} {$j < 10} {incr j} {
    set tmp [lindex [time {lsearch $lst $x} 100] 0]
    if {$tmp < $min} {set min $tmp}
  }
  puts "$i $min"
}
```

Array Element Access Time

```
for {set i 0} {$i < 500} {incr i} {
  set x "abcd.$i.efg"
```

```
    set arr($i) $x
    set min [lindex [time {set y $arr($i)} 100] 0]
    for {set j 0} {$j < 10} {incr j} {
      set tmp [lindex [time {set y $arr($i)} 100] 0]
      if {$tmp < $min} {set min $tmp}
    }
    puts "$i $min"
  }
```

Dict Element Access Time

```
  for {set i 0} {$i < 500} {incr i} {
    set x "abcd.$i.efg"
    dict set dct $i $x
    set min [lindex [time {dict get $dct $i} 100] 0]
    for {set j 0} {$j < 10} {incr j} {
      set tmp [lindex [time {dict get $dct $i} 100] 0]
      if {$tmp < $min} {set min $tmp}
    }
    puts "$i $min"
  }
```

Note that although accessing a known index in an associative array is very fast, using the `array names` command builds a list of array indices, and extracts the list of names that match a pattern. This operation becomes slower as the number of indices increases. If you need to deal with applications that have thousands of nodes, it may be better to use multiple arrays rather than indices with multiple fields.

6.7 BOTTOM LINE

This chapter has demonstrated several ways to use Tcl lists, dicts and associative arrays with data read from a file, in-memory data and data from an SQL database.

- Lists can be used to organize information as position-oriented data or as key/value pairs.
- Dicts organize data as ordered key/value pairs.
- Indices in an array and keys in a dict must be unique.
- Dicts can be nested to manipulate tree structured data.
- Naming conventions can be used with associative array indices to provide the same functionality as structures and arrays of structures.
- A variable can contain the name of another variable, providing the functionality of a pointer.
- The `catch` command is used to catch an error condition without causing a script to abort processing.
  Syntax: `catch` *`script ?varName?`*

- The `file` command provides access to the file system.
  Syntax: `file` *`type pathName`*
  Syntax: `file` *`nativename pathName`*
  Syntax: `file` *`delete pathName`*
  Syntax: `file` *`exists pathName`*
  Syntax: `file` *`isdirectory pathName`*
  Syntax: `file` *`isfile pathName`*
- The `glob` command returns directory entries that match a particular pattern.
  Syntax: `glob` *`?-nocomplain? ?--? pattern ?pattern?`*
- The `lreplace` command replaces one or more list elements with 0 or more new elements.
  Syntax: `lreplace` *`list first last ?element element ...?`*
- The `join` command will convert a list into a string, using an optional character as the element separator.
  Syntax: `join` *`?joinString?`*
- Accessing an array element is frequently faster than using `lsearch` to find a list element.

6.8 PROBLEMS

The following numbering convention is used in all Problem sections:

Number Range	Description of Problems
100–199	Short comprehension problems review material covered in the chapter. They can be answered in a few words or a 1–5-line script. These problems should each take under a minute to answer.
200–299	These quick exercises require some thought or information beyond that covered in the chapter. They may require reading a man page or making a web search. A short script of 1–50 lines should fulfill the exercises, which may take 10–20 minutes each to complete.
300–399	Long exercises may require reading other material or writing a few hundred lines of code. These exercises may take several hours to complete.

- *100* Given a large amount of data, which is likely to be faster: using `lsearch` to search a list or indexing the data in an associative array?
- *101* What Tcl command would convert a Tcl list into a string with commas between the list elements?
- *102* What associative array indices would provide a data relationship similar to the following?

```
struct {
   char *title;
   char *author;
   float price;
} books[3];
```

- *103* What will be the first Tcl command in the error stack generated by the Tcl command `error "Illegal value"`?
- *104* The code fragment `set result [expr $numerator / $divisor]` will fail if the numerator or divisor is an illegal value. Write a code fragment to divide one value by another without generating an error. If the numerator or divisor is an illegal value, set `result` to the phrase `Not-A-Number`.
- *105* What Tcl command will report the type of file (a normal file or directory, for example) given a file name?
- *200* Given a set of data in which each line follows the form `Name:UserName:Password`, write a `foreach` command that will split a line into the variables `name`, `user`, and `passwd`.
- *201* Given the following

```
{
{ {Name {John Doe}} {UserName johnd} {Password JohnPwd} }
{ {UserName jdoe}   {Name {Jane Doe}} {Password JanePwd} }
{ {Name {John Smith}} {Password JohnPwd} {UserName johns} }
{ {UserName jonnjonzz} {Password manhunter} {Name {John Jones}} }
}
```

  a. What combination of `lsearch` and `lindex` commands would find the password for John Jones?
  b. Write a short script that will list users with identical passwords.
  c. Will you get the record with John Doe's password using an `lsearch` pattern of `John*`?
  d. Write a short script that would change John Smith's user name to `jsmith`.

- *300* The kitchen in an automated restaurant might receive orders as patrons select items from a menu via a format such as the following.

```
{{Table 2} {burger} {ketchup mustard}}
   {{Table 3} {drink} {medium}}
   {{Table 2} {fries} {large}}
   {{Table 1} {BLT}  {no mayo}}
   {{Table 3} {Complete} {} }
   {{Table 1} {drink} {small}}
   {{Table 1} {Complete} {} }
```

  Write a script that will accept data in a format such as this, collecting the items ordered at a table and reporting a table's order when the `Complete` message is received. After reporting an order, it should be ready to start assembling a new order for that table.

- *301* Write a script that will accept multiple lines in the form "author, title", and will create a list author, title, author, title…

```
  Clif Flynt, Tcl/Tk: A Developer's Guide
Richard Stevens, TCP/IP Illustrated
Donald Knuth, The Art of Computer Programming: Vol 1
Donald Knuth, The Art of Computer Programming: Vol 2
Donald Knuth, The Art of Computer Programming: Vol 3
```

```
John Ousterhout, Tcl and the Tk Toolkit
Richard Stevens, Unix Network Programming
```

Convert the list to an associative array that would allow you to get lists of books by an author.

- *302* Write a "Safe Math" procedure that will accept a mathematical expression, and evaluate it and return the result without generating an error. If any of the values in the math expression are illegal, return the phrase "Illegal Expression" instead of a numeric answer.

CHAPTER

Procedure Techniques

7

One key to writing modular code is dividing large programs into smaller subroutines. Tcl supports the common programming concept of subroutines—procedures that accept a given number of arguments and return one or more values. Tcl also supports procedures with variable numbers of arguments, and procedures with arguments that have default values.

The Tcl interpreter allows scripts to rename procedures and create new procedures while the script is running. When a new procedure is created by another procedure, the new procedure is defined in the global scope and is not deleted when the procedure that created it returns.

Previous chapters introduced some of these capabilities. This chapter expands on that discussion with more details and examples, including the following.

- Defining the arguments to procedures
- Renaming and deleting procedures
- Examining a procedure's body and arguments
- Performing Tcl variable and command substitutions on a string
- Constructing and evaluating command lines within a script
- Turning a set of procedures and data into an object

7.1 ARGUMENTS TO PROCEDURES

When the `proc` command is invoked to create a new procedure, the new procedure is defined with a name, a list of arguments, and a body to evaluate when the procedure is invoked. When a procedure is called, the Tcl interpreter counts the arguments to confirm that there are as many arguments in the procedure call as there were in the procedure definition. If a procedure is called with too few arguments, the Tcl interpreter generates an error message resembling this:

```
no value given for parameter "arg1" to "myProcedure"
```

If a procedure is called with too many arguments, the Tcl interpreter generates an error message resembling this:

```
called "myProc" with too many arguments
```

This runtime checking helps you avoid the silent errors that occur when you modify a procedure to take a new argument and miss changing one of the procedure calls. However, there are times when you do not know the number of arguments (as with the `expr` command) or want an argument to be optional, with a default value when the argument is not present (as 1 is the default increment value for

Tcl/Tk: A Developer's Guide. DOI: 10.1016/B978-0-12-384717-1.00007-5

the `incr` command). You can easily define a procedure to handle a variable number of arguments or define a default value for an argument in Tcl.

7.1.1 Variable Number of Arguments to a Procedure

You can define a procedure that takes a variable number of arguments by making the final argument in the argument list the word `args`. When this procedure is called with more arguments than expected, the Tcl interpreter will concatenate the arguments that were not assigned to declared variables into a list and assign that list to the variable `args`, instead of generating a `too many arguments` error.

Note that `args` must be the last argument in the argument list to get the excess arguments assigned to it. If there are other arguments after the `args` argument, `args` is treated as a normal argument. In the following example, the procedure `showArgs` requires at least one argument. If there are more arguments, they will be placed in the variable `args`.

Example 1

Script Example

```
# A proc that accepts a variable number of args
proc showArgs {first args} {
    puts "first: $first"
    puts "args: $args"
}
# Example Script
puts "Called showArgs with one arg"
showArgs oneArgument
puts "\nCalled showArgs with two args"
showArgs oneArgument twoArgument
puts "\nCalled showArgs with three args"
showArgs oneArgument twoArgument threeArgument
```

Script Output

```
Called showArgs with one arg
first: oneArgument
args:

Called showArgs with two args
first: oneArgument
args: twoArgument

Called showArgs with three args
first: oneArgument
args: twoArgument threeArgument
```

7.1.2 Default Values for Procedure Arguments

The technique for setting a default value for an argument is to define the argument as a list; the first element is the argument name, and the second is the default value. When the arguments are defined

as a list and a procedure is called with too few arguments, the Tcl interpreter will substitute the default value for the missing arguments, instead of generating a `no value given for parameter` error.

Example 2

Script Example

```
# A proc that expects at least one arg, and has defaults for 2
proc showDefaults {arg1 {numberArg 0} {stringArg {default val}}} {
   puts "arg1: $arg1"
   puts "numberArg: $numberArg"
   puts "stringArg: $stringArg"
}

# Example Script
puts "\nCalled showDefaults with one argument"
showDefaults firstArgument
puts "\nCalled showDefaults with two arguments"
showDefaults firstArgument 3
puts "\nCalled showDefaults with three arguments"
showDefaults firstArgument 3 "testing"
```

Script Output

```
Called showDefaults with one argument
arg1: firstArgument
numberArg: 0
stringArg: default val

Called showDefaults with two arguments
arg1: firstArgument
numberArg: 3
stringArg: default val
cback nCalled showDefaults with three arguments
arg1: firstArgument
numberArg: 3
stringArg: testing
```

The procedure `showDefaults` must be called with at least one argument. If only one argument is supplied, `numberArg` will be defined as 0, and `stringArg` will be defined as `default val`.

Note that the order of the arguments when a procedure is invoked must be the same as the order when the procedure was defined. The Tcl interpreter will assign the values in the order in which the variable names appear in the procedure definition. For example, you cannot call procedure `showDefaults` with arguments for `arg1` and `stringArg`, but must use the default for `numberArg`. The second value in the procedure call is assigned to the second variable in the procedure definition.

You cannot create a procedure that has an argument with a default before an argument without a default. If you created a procedure such as the following,

```
proc badProc {{argWithDefault dflt} argWithOutDefault} {...}
```

and called it with a single argument, it would be impossible for the Tcl interpreter to guess for which variable that argument was intended. Tcl would assign the value to the first variable in the argument list, and the error return would resemble the following.

```
% badProc aa
no value given for parameter "argWithOutDefault" to "badProc"
```

7.2 RENAMING OR DELETING COMMANDS

The `proc` command will create a new procedure. Sometimes you may also need to rename or delete a procedure. For example, if you need to use two sets of Tcl code that both have a `processData` procedure, you can load one package, rename the `processData` procedure to `package1processData`, and then load the second package. (A better solution is to use a name space, as described in Chapter 8.)

If you have had to deal with name collisions in libraries and DLLs before, you will appreciate this ability.

The `rename` command lets you change the name of a command or procedure. You can delete a procedure by renaming it to an empty string.

Syntax: `rename` *`oldName ?newName?`*

Rename a procedure.

`oldName`	The current name of the procedure.
`?newName?`	The new name for the procedure. If this is an empty string, the procedure is deleted.

The following example shows the procedure `alpha` renamed to beta and then deleted.

Example 3

Script Example

```
proc alpha {} {
  return "This is the alpha proc"
}
# Example Script
puts "Invocation of procedure alpha: [alpha]"
rename alpha beta
catch alpha rtn
puts "Invocation of alpha after rename: $rtn"
puts "Invocation of procedure beta: [beta]"
rename beta ""
beta
```

Script Output

```
Invocation of procedure alpha: This is the alpha proc
Invocation of alpha after rename: invalid command name "alpha"
Invocation of procedure beta: This is the alpha proc
invalid command name "beta"
```

7.3 GETTING INFORMATION ABOUT PROCEDURES

The Tcl interpreter is a very introspective interpreter. A script can get information about most aspects of the interpreter while the script is running. The `info` command that was introduced in Chapter 6 provides information about the interpreter. These four `info` subcommands will return the names of all commands known to the Tcl interpreter, and can return more information about procedures that have been defined by Tcl scripts using the `proc` command.

`info commands` *`pattern`*	Return a list of commands with names that match a pattern. This includes both Tcl procedures and commands defined by compiled C code.
`info procs` *`pattern`*	Return a list of procedures with names that match a pattern. This will return only the names of procedures defined with a `proc` command, not those defined using compiled C code.
`info body` *`procName`*	Return the body of a procedure. This is only valid for Tcl commands defined as a `proc`.
`info args` *`procName`*	Return the arguments of a procedure. This is only valid for Tcl commands defined as a `proc`.

The `proc` subcommand will list the available procedures in an interpreter. This includes many of the Tcl and Tk built-in commands that are implemented as Tcl scripts, but not the commands that are implemented in "C" code.

The `commands` subcommand will list the available commands that match a `glob` pattern. This includes both procedures and commands implemented in "C" code. This can be used to confirm that an expected set of code has been loaded.

Syntax: `info procs` *`pattern`*

Syntax: `info commands` *`pattern`*

Returns a list of command or procedure names that match the pattern. If no command names match the pattern, an empty string is returned.

`pattern` A `glob` pattern to attempt to match.

Example 4

```
# Check to see if md5 command is defined. If not, load a
#  Tcl version
```

```
if {[string match [info commands md5] ""]} {
  source "md5.tcl"
}
```

The `info body` command will return the body of a procedure. This can be used to generate and modify procedures at runtime or for distributed applications to exchange procedures.

Syntax: `info body` *`pattern`*

Returns the body of a procedure.

`procName` The procedure from which the body will be returned.

Example 5

Script Example

```
proc example {one two} {
  puts "This is example"
}
# Display the body of the proc
puts "The example procedure body is:\n [info body example]"
```

Script Output

```
The example procedure body is:
 puts "This is example"
```

The `info args` command returns the argument list of a procedure. This is useful when debugging code that has generated its own procedures.

Syntax: `info args` *`procName`*

Returns a procedure's argument list.

`procName` The procedure from which the arguments will be returned.

Example 6

Script Example

```
puts "The example proc has [llength [info args example]] arguments"
```

Script Output

```
The example proc has 2 arguments
```

The next example shows how you can check that a procedure exists and create a slightly different procedure from it.

Example 7

Script Example

```
 # Define a simple procedure.
proc alpha {args} {
  puts "proc alpha is called with these arguments: $args"
}

# Confirm that the procedure exists
if {[info commands alpha] != ""} {
  puts "There is a procedure named alpha"
}

# Get the argument list and body from the alpha procedure
set alphaArgs [info args alpha]
set alphaBody [info body alpha]

# Change the word "alpha" in the procedure body to "beta"
regsub "alpha" $alphaBody "beta" betaBody

# Create a new procedure "beta" that will display its arguments.
proc beta $alphaArgs $betaBody

# Run the two procedures to show their behavior
alpha How about
beta them Cubs.
```

Script Output

```
There is a procedure named alpha
proc alpha is called with these arguments: How about
proc beta is called with these arguments: them Cubs.
```

7.4 SUBSTITUTION AND EVALUATION OF STRINGS

Section 3.2 discussed how the Tcl interpreter evaluates a script: the interpreter examines a line, performs a pass of command and variable substitutions, and then evaluates the resulting line. A Tcl script can access these interpreter functions to perform substitutions on a string or even evaluate a string as a command. This is one of the unusual strengths in Tcl programming. Most interpreters do not provide access to their parsing and evaluation sections to the program being interpreted. The two commands that provide access to the interpreter are `subst` and `eval`.

7.4.1 Performing Variable Substitution on a String

The `set` command returns the current value of a variable as well as assigning a value. A common use for this is to assign and test a variable in one pass, as shown in the following.

```
while {[set len [string length $password]] < 8} {
  puts "$len is not long enough. Use 8 letters"
  set password [gets stdin]
}
```

This capability is also useful when you have a variable that contains the name of another variable, and you need the value from the second variable, as shown in the following.

```
% set a 1
% set b a
% puts "The value of $b is [set $b]"
The value of a is 1
```

If you need to perform more complex substitutions, you can use the `subst` command. The `subst` command performs a single pass of variable and command substitutions on a string and returns the modified string. This is the first phase of a command being evaluated, but the actual evaluation of the command does not happen. If the string includes a square-bracketed command, the command within the brackets will be evaluated as part of this substitution.

Syntax: `subst` *`string`*

Perform a substitution pass upon a string. Do not evaluate the results.

`string` The string upon which to perform substitutions.

The `subst` command can be used when you need to replace a variable with its content but do not want to evaluate it as a command. The previous example could be written as follows, using `subst`.

```
% set a 1
% set b a
% puts [subst "The value of $b is $$b"]
The value of a is 1
```

In this example, the `$$b` is replaced by `$a` in the usual round of substitutions, and then `$a` is replaced by 1 by the `subst` command. In this case, you can obtain the same result with either `set` or `subst`. As the string becomes more complex, the `subst` command becomes a better option, particularly when combined with the `eval` command, as discussed in the following section.

7.4.2 Evaluating a String as a Tcl Command

The `eval` command concatenates its arguments into a string and then evaluates that string as if it were text in a script file. The `eval` command allows a script to create its own command lines and evaluate them.

You can use `eval` to write data-driven programs that effectively write themselves based on the available data. You can also use the `eval` command to write agent-style programs, where a task on one machine sends a program to a task on another machine to execute. These techniques (and the security considerations) are described in later chapters.

Syntax: `eval` *`arg ?args?`*

Evaluate the arguments as a Tcl script.

`arg ?args?` These arguments will be concatenated into a command line and evaluated.

Example 8

Script Example

```
set cmd(0) {set a 1}
set cmd(1) {puts "start value of A is: $a"}
set cmd(2) {incr a 3}
set cmd(3) {puts "end value of A is: $a"}
for {set i 0} {$i < 4} {incr i} {
  eval $cmd($i)
}
```

Script Output

```
start value of A is: 1
end value of A is: 4
```

Because the arguments to `eval` are concatenated, the command that is evaluated will lose one level of grouping. Discarding a level of grouping is a common use of the `eval` command.

The next example demonstrates some of the tricks involved in removing the grouping for some sections of a command while maintaining the grouping in others.

In Tcl 8.5, a new operator was added to Tcl to remove the grouping. The three character {`*`} operator is much simpler to use than collections of extra lists and `eval` commands, as shown in the last of these examples. One advantage of the {`*`} operator over `eval` is that it allows selection of individual variables to be expanded, while the `eval` command expands all the variables, which may require adding layers of grouping to some variables.

For example, in the next example, the `regexpArgs` variable has three options for the `regexp` command. If this command is invoked as `regexp $regexpArgs`, the three options are presented to the `regexp` command as a single argument, and the `regexp` command generates an error, since it does not support a "`-line -nocase --`",-option.

The `eval` command can be used to separate the list of options into three separate options, as the `regexp` command requires. However, we do not want to separate the string the regular expression is being compared to into separate words. When using `eval` to split one set arguments you must also add a layer of grouping to elements you do not want to split.

- The following generates a runtime error.

```
set regexpArgs {-line -nocase --}
set str "This is a test"
set exp {is.*}
regexp $regexpArgs $exp $str m1
# After Substitution
# regexp {-line -nocase --} {is.*} {This is a test} m1
```

- The following is legal, but the string is also split into separate arguments.

```
eval regexp $regexpArgs $exp $str m1
# After Substitution:
# regexp -line -nocase -- is.* This is a test m1
```

- The following is legal code, which works as expected.
```
eval regexp $regexpArgs $exp {$str} m1
# After Substitution:
# regexp -line -nocase -- is.* {This is a test} m1
```
- The following is legal code, which works as expected and is preferred style using `eval`.
```
eval regexp $regexpArgs $exp [list $str] m1
# After Substitution:
# regexp -line -nocase -- is.* {This is a test} m1
```
- The following is legal code, which works as expected. This is the preferred style for Tcl 8.5 and newer.
```
regexp {*}$regexpArgs $exp $str m1
# After Expansion:
# regexp -line -nocase -- is.* {This is a test} m1
```

Example 9

Script Example

```
set regexpArgs {—line —nocase ——}
set str "This is a test"
set exp {is.*}
set m1 ""
set fail [catch {regexp $regexpArgs $exp $str m1} message]
if {$fail} {
   puts "regexp failed: $message"
}
set rtn [regexp {*}$regexpArgs $exp $str m1]
puts "Second regexp returns: $rtn"
puts "Matched: $m1"
```

Script Output

```
regexp failed: bad switch "-line -nocase --": must be -all,
   -about, -indices, -inline, -expanded, -line, -linestop,
   -lineanchor, -nocase, -start, or --
Second regexp returns: 1
Matched: is is a test
```

7.5 WORKING WITH GLOBAL AND LOCAL SCOPES

The Tcl variable scope rules provide a single global scope, and private local scopes for each procedure being evaluated. This facility makes it easy to write robust, modular programs. However, some applications require scripts being evaluated in one local scope to have access to another scope. The `upvar` and `uplevel` commands allow procedures to interact with higher-level scopes.

This section discusses scopes and the upvar and uplevel commands in more detail than previously, and shows how to use the uplevel and upvar commands. This discussion is expanded upon in Chapter 8, which discusses the namespace command. For now, a namespace is a technique for encapsulating procedures and variables in a named, private scope.

7.5.1 Global and Local Scope

The global scope is the primary scope in a Tcl script. All Tcl commands and procedures that are not defined within a namespace are maintained in this scope. All namespaces and procedures can access commands and variables maintained in the global scope.

When a procedure is evaluated, it creates a local scope. Variables are created in this scope as necessary, and are destroyed when the procedure returns. The variables used within a procedure are visible to other procedures called from that procedure, but not to procedures outside the current call stack.

Any variable defined outside of the procedures or identified with the global command is maintained in the global scope. Variables maintained in the global scope persist until either the script exits or they are explicitly destroyed with the unset command.

These variables can be accessed from any other scope by declaring the variable to exist in the global scope with the global command. Note that the global command must be evaluated before that variable name is used in the local scope. The Tcl interpreter will generate an error if you try to declare a variable to be global after using it in a local scope.

```
set globalVar "I'm global"
proc goodProc {} {
  global globalVar
  # The next line prints out
  # "The globalVar contains: I'm global"
  puts "The globalVar contains: $globalVar"
}
proc badProc {} {
  set globalVar "This defines 'globalVar' in the local scope"
  # The next line causes an error
  global globalVar
}
```

Each time a procedure invokes another procedure, another local scope is created. These nested procedure calls can be viewed as a stack, with the global scope at the top and each successive procedure call stacked below the previous ones.

A procedure can access variables within the global scope or within the scope of the procedures that invoked it via the upvar and uplevel commands. The upvar command will link a local variable to one in a previous (higher) stack scope.

Syntax: upvar *?level? varName1 localName1 ?Name2? ?localName2?*

Example 10

Script Example

```
proc top {topArg} {
 set localArg [expr $topArg+1]
```

```
  puts "Before calling bottom localArg is: $localArg"
  bottom localArg
  puts "After calling bottom, localArg is: $localArg"
}

proc bottom {bottomArg} {
  upvar $bottomArg arg
  puts "bottom is passed $bottomArg with a value of $arg"
  incr arg
}

top 2
```

Script Output

```
Before calling bottom localArg is: 3
bottom is passed localArg with a value of 3
After calling bottom, localArg is: 4
```

The `uplevel` command will concatenate its arguments and evaluate the resulting command line in a higher level scope. The `uplevel` is like `eval`, except that it evaluates the command in a different scope instead of the current scope.

The main use for the `uplevel` command is to implement program flow control structures such as `for`, `while`, and `if`. Using the `uplevel` command as a macro facility to change variables in a calling scope (as done in the next example) is a bad idea that leads to hard-coded variable names and code that is difficult to maintain. When you need to modify a variable that does not exist in the current scope, you should pass the variable name and use `upvar` in your procedure rather than `uplevel`.

The following example shows a set of procedures (`stack1`, `stack2`, and `stack3`) that call each other and then access and modify variables in the scope of the procedures that called them. All of these `stack` procedures have a local variable named `x`. Each is a separate variable. Note that procedure `stack1` cannot access the variables in the scope of procedure `stack2`, although `stack2` can access variables in the scope of `stack1`.

Example 11

Script Example

```
#  Create procedure stack1 with a local variable x.
#   display the value of x, call stack2, and redisplay the
#   value of x

proc stack1 {} {
  set x 1;
```

```
  puts "X in stack1 starts as $x"
  stack2
  puts "X in stack1 ends as $x"
  puts ""
}

#  Create procedure stack2 with a local variable x.
#   display the value of x, call stack3, and redisplay the
#   value of x

proc stack2 {} {
  set x 2;
  puts "X in stack2 starts as $x"
  stack3
  puts "X in stack2 ends as $x"
}

#  Create procedure stack3 with a local variable x.
#  display the value of x,
#  display the value of x in the scope of procedures that
#  invoked stack3 using relative call stack level.
#  Add 10 to the value of x in the proc that called stack3
#    (stack2)
#  Add 100 to the value of x in the proc that called stack2
#    (stack1)
#  Add 200 to the value of x in the global scope.
#  display the value of x using absolute call stack level.

proc stack3 {} {
  set x 3;
  puts "X in stack3 starts as $x"
  puts ""

  #  display the value of x at stack levels relative to the
  #  current level.
  for {set i 1} {$i <= 3} {incr i} {
    upvar $i x localX
    puts "X at upvar $i is $localX"
  }

  puts "\nx is being modified from procedure stack3"
  #  Evaluate a command in the scope of procedures above the
  #  current call level.
  uplevel 1 {incr x 10}
  uplevel 2 {incr x 100}
  uplevel #0 {incr x 200}
  puts ""
```

```
  # display the value of x at absolute stack levels
  for {set i 0} {$i < 3} {incr i} {
    upvar #$i x localX
    puts "X at upvar #$i is $localX"
  }
  puts ""
}

#  Example Script
set x 0;
puts "X in global scope is $x"
stack1
puts "X in global scope ends as $x"
```

Script Output

```
X in global scope is 0
X in stack1 starts as 1
X in stack2 starts as 2
X in stack3 starts as 3

X at upvar 1 is 2
X at upvar 2 is 1
X at upvar 3 is 0

x is being modified from procedure stack3

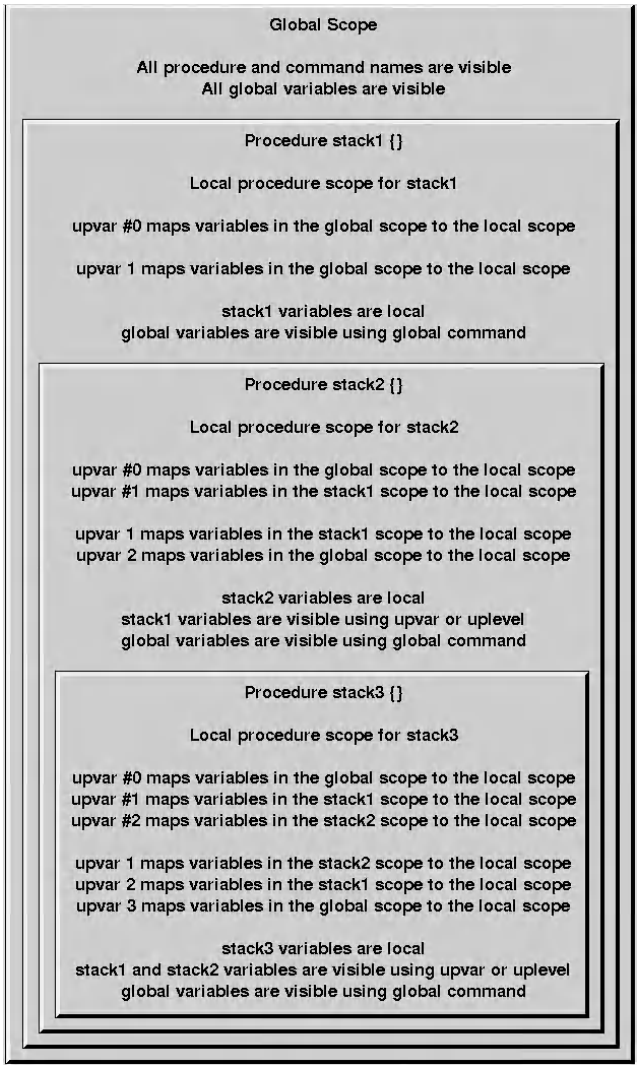
X at upvar #0 is 200
X at upvar #1 is 101
X at upvar #2 is 12

X in stack2 ends as 12
X in stack1 ends as 101

X in global scope ends as 200
```

The scopes in the preceding example resemble the diagram that follows, in which the procedure `stack3` is being evaluated. Each local procedure scope is nested within the scope of the procedures that called it.

When the `uplevel 1 {incr x 10}` command is evaluated, it causes the string `incr x 10` to be evaluated one scope higherthan the current `stack3` scope, which is the `stack2` scope. The `uplevel #0 {incr x 200}` command is evaluated at absolute scope level 0, or the global scope. The evaluation level for a command `uplevel #1` would be the first level down the call stack, `stack1` in this example. This example is not a recommended technique for using the command. It is intended only to demonstrate how `uplevel` works.



The preceding example demonstrates how procedures can access all levels above them in the call stack but not levels below. The `global` command works well if you have a single global variable that a procedure will manipulate. If your application requires several variables to describe the system's state, the best technique is to use a single associative array and multiple indices to hold the different values.

If your application has multiple entities that each have their own state, you may use a different associative array to describe each entity's state and use the `upvar` command to map the appropriate array into a procedure.

The next example shows a simple two-person game with the players' positions kept in separate global variables. The `move` procedure may be invoked with the name of either variable, which it maps to a local variable called `player` and makes a move for that player.

Example 12

Script Example

```
set player1(position) 0
set player2(position) 0

proc move {playerName} {
  upvar #0 $playerName player
  #  Move the piece a random number of spaces
  #    between 0 and 9.
  set move [expr int(rand() * 10)]
  incr player(position) $move
}

while {($player1(position) < 20) & ($player2(position) < 20)} {
   move player1
   move player2
   puts "\nCurrent Positions:"
   puts "  1: $player1(position)"
   puts "  2: $player2(position)"
}

if {$player1(position) > $player2(position)} {
    puts "Player 1 wins!"
} else {
     puts "Player 2 wins!"
}
```

Script Output

```
Current Positions:
  1: 8
  2: 3

Current Positions:
  1: 11
  2: 11

Current Positions:
  1: 14
  2: 17
```

```
Current Positions:
  1: 19
  2: 25
Player 2 wins!
```

7.6 MAKING A TCL OBJECT

The C++/Java model of object-oriented programming is the most common type of OO programming, but it's not the only OO model. Tcl supports several styles of OO programming from `[incr Tcl]` which is very much like `C++` to functional or personal styles of implementing OO concepts.

This section describes how you can perform simple object-style programming in pure Tcl. This discussion and the discussion of namespaces in the next chapter deals with a subset of a complete object programming environment. The `[incr Tcl]` extension and `TclOO` package support full object-oriented programming. The `[incr Tcl]` extension is discussed briefly later, and the `TclOO` package will be discussed in more detail in the next chapters.

Chapter 3 mentioned that Tcl keeps separate hash tables for commands, associative array indices, and variables, and that the first word in a Tcl command string must be a command name. These features mean that any name can be defined as both a variable name and a procedure name. The interpreter will know which is meant by the position of the name in the command line.

This section discusses techniques for using the same label as a variable name or array index and a procedure name. In this example, the procedure name is the same as the name of a variable in the global-scope.

This lets us implement the object-oriented programming concept of having methods attached to a data object. This implementation of an object is different from the implementation of a C++ or Java but the concept of data and method related is similar.

7.6.1 An Object Example

For a simple example, the following code creates variables with the name of a common fruit and then creates a procedure with the same name that tests whether or not its argument is a valid color for this fruit.

This example uses the `info level` command to discover the name of the procedure at run time and map the variable with that name into the local scope.

Example 13

Script Example

```
# Define a set of fruit names and colors
set fruitList [list {apples {red yellow green}} \
                    {bananas yellow} \
                    {grapes {green purple}}]
```

```
foreach fruitDefinition $fruitList {
  # 1) Extract the name and possible colors from the
  #    fruit definition.

  lassign $fruitDefinition fruitName fruitColors

  # 2) Create a global variable named for the fruit, with
  #    the fruit colors as a value

  set $fruitName $fruitColors

  # 3) Define a procedure with the name of the fruit
  #    being checked. The default value for "name" is
  #    also the name of the fruit, which is also the name
  #    of the global variable with the list of fruit colors.

  proc $fruitName {color} {
    set name [lindex [info level 0] 0]
    upvar #0 $name fruit
    if {[lsearch $fruit $color] >= 0} {
        return "Correct, $name can be $color"
    } else {
        return "No, $name are $fruit"
    }
  }
}

# 4) Loop through the fruits, and ask the user for a color.
#     Read the input from the keyboard.
#     Evaluate the appropriate function to check for correctness.

foreach fruit [list apples bananas grapes ] {
  puts "What color are $fruit?"
  gets stdin answer
  puts [$fruit $answer]
}
```

Script Output

```
 What color are apples?    # User types red
Correct, apples can be red
What color are bananas?    # User types red
No, bananas are yellow
What color are grapes?    # User types red
No, grapes are green purple
```

The first procedure defined resembles this:

```
proc apples {color} {
  set name [lindex [info level 0] 0]
  upvar #0 $name fruit
  if {[lsearch $fruit $color] >= 0} {
    return "Correct, $name can be $color"
  } else {
    return "No, $name are $fruit"
  }
}
```

When this procedure is invoked as `apples red`, the name of the procedure `apples` is assigned to the variable `name` and the `upvar #0 $name fruit` maps the global variable `apples` to the local variable `fruit`.

The code in the previous example can be wrapped into a `createFruitObject` procedure to create both the global variable and procedure. The procedure created in `createFruitObject` is created in the global scope, even though the `proc` command is in a procedure.

Data can be represented in several ways in Tcl. To demonstrate the variety of data styles, the `fruitList` is defined as a keyed list instead of a list of lists in the next example.

Example 14

```
proc createFruitObject {name colors} {
  global $name
  set $name $colors
  proc $name {color} {
    set name [lindex [info level 0] 0]
    upvar #0 $name fruit
    if {[lsearch $fruit $color] >= 0} {
        return "Correct, $name can be $color"
    } else {
        return "No, $name are $fruit"
    }
  }
}

set fruitList {apples {red yellow green}
               bananas yellow
               grapes {green purple}}

foreach {fruit colors} $fruitList {
  createFruitObject $fruit $colors
}

foreach fruit [list apples bananas grapes ] {
  puts "What color are $fruit?"
```

```
  gets stdin answer
  puts [$fruit $answer]
}
```

A functional language type of an object method can be created by embedding the data into the procedure, instead of using the name of an external array as the default argument.

```
proc createFruitObject {fruitName fruitColors} {
  proc $fruitName [list color [list colors $fruitColors]] {
    set name [lindex [info level 0] 0]
    if {[lsearch $colors $color] >= 0} {
        return "Correct, $name can be $color"
    } else {
        return "No, $name are $colors"
    }
  }
}

# Create new fruit commands
foreach {name colors} {apples {red yellow green}
                       bananas yellow
                       grapes {green purple}} {
  createFruitObject $name $colors
}
```

7.7 BOTTOM LINE

- A procedure can be defined as taking an undefined number of arguments by placing the `args` argument last in the argument list.
- A procedure argument with a default value is defined by declaring the argument as a list. The second list element is the default value.
- When a Tcl procedure is evaluated, it creates a local scope for variables. This local scope stacks below the scope of the code that invoked the procedure.
- A Tcl procedure can access all of the scopes above it in the procedure call stack.
- Tcl procedures can be constructed and evaluated in a running program.
- Data items can be treated as objects by creating a procedure with the same name as the data item and using that procedure to manipulate the data item.
- The `rename` command renames or removes a command or procedure.
  Syntax: `rename` *`oldName ?newName?`*
- The `subst` command performs a pass of command and variable substitutions upon a string.
  Syntax: `subst` *`string`*
- The `eval` command will evaluate a set of arguments as a Tcl command.
  Syntax: `eval` *`arg ?args?`*
- The `info` subcommands that report information about procedures include the following.

Syntax: `info procs pattern`
Return a list of procedures that are visible in the current scope and match a pattern.
Syntax: `info commands pattern`
Return a list of commands that are visible in the current scope and match a pattern.
Syntax: `info args procName`
Return the arguments of a procedure.
Syntax: `info body procName`
Return the body of a procedure.

- The `global` command declares that a variable exists in the global scope.
  Syntax: `global varName1 ?varName2...varNameN?`
- Simple objects can be created by using the same word for both a variable name (or associative array index) and the procedure that will manipulate that data.
- Macintosh users may need to prepend file names with a colon to allow Tcl to recognize file names with embedded forward slashes.

7.8 PROBLEMS

The following numbering convention is used in all Problem sections:

Number Range	Description of Problems
100–199	Short comprehension problems review material covered in the chapter. They can be answered in a few words or a 1–5 line script. These problems should each take under a minute to answer.
200–299	These quick exercises require some thought or information beyond that covered in the chapter. They may require reading a man page or making a web search. A short script of 1–50 lines should fulfill the exercises, which may take 10–20 minutes each to complete.
300–399	Long exercises may require reading other material or writing a few hundred lines of code. These exercises may take several hours to complete.

- *100* What would be the arguments portion of a `proc` command that duplicated the behavior of the Tcl `format` command?
- *101* What would be the arguments portion of a `proc` command that duplicated the behavior of the Tcl `incr` command?
- *102* After a procedure has returned, can you access any of the local variables that were used in that procedure?
- *103* If `procA` invokes `procB`, can `procB` local variables be accessed from `procA`?
- *104* If `procA` invokes `procB`, can `procA` local variables be accessed from `procB`?
- *105* Can you use the `rename` command to rename Tcl commands, such as `while` or `for`?

- *106* Under what circumstances would the `subst` command be preferable to using the `set` command?
- *107* What Tcl command will return the argument list for a procedure?
- *108* What Tcl command will define a procedure named `foo`
  a. with a single required argument?
  b. with a single optional argument with a default value of 2?
  c. that accepts zero or more arguments?
  d. that accepts two or more arguments?
  e. that has one required argument, one optional argument with a default value of 2, and may accept more arguments?
- *109* What Tcl command could be used to determine if a procedure has been defined?
- *200* Write a procedure that will accept one or more numeric arguments and return their sum.
- *201* Write a procedure that will accept zero or more numeric arguments and return the sum if there are multiple arguments, or a 0 if there were no arguments.
- *202* Write a procedure that will duplicate the functionality of the `incr` command using the `upvar` command.
- *203* Write a procedure that will duplicate the functionality of the `incr` command using the `uplevel` command.
- *300* Write a procedure that will rename the following Tcl commands to new commands. Write a short script to test the new commands.

```
if -> if,like
for -> so
expr -> fuzzyNumbers
```

- *301* Write a script that will display all permutations of the values of a list of variable names, as suggested by the following.

```
set a 1
set b 2
set list {a b}
showPermutations $list
...
```

  with output:

```
1 1
1 2
2 1
2 2
```

- *302* Given a procedure `report {data} {...}` that uses `puts` to print a report to the screen, write a short script to create a new procedure `reportFile {data outputChannel}` that will send an identical report to an open Tcl channel.

- *303* Write a short script that will compare the bodies of all visible procedures and generate a list of procedures with identical bodies.
- *304* Write a procedure for the `tree.tcl` module described in Chapter 6 that will return a list of a node siblings (the other child nodes to this node parent). Add a new method to the `treeObjectProc` described in this chapter to evaluate the `treeGetSibling` procedure.
- *305* Write a procedure that will create simple objects. The procedure should accept a variable name, value, and body to evaluate when the variable's procedure is invoked.
- *306* Write two procedures, as follows.

  Syntax: `class` *`className body`*
  Adds a new class name to a a collection of known classes and associates the *body* with that class.

  Syntax: `new` *`className value`*

  Creates a variable with a unique name in the global scope, and assigns *value* to it. Creates a new procedure with the same name and single argument `args`, and uses the *body* that was defined with the `class` command as the body for the procedure.

  When complete, code such as follows should function.

```
class test {return "args are: $args"}
set x [new test 22]
puts ‘‘Value of test: [set $x]’’
puts ‘‘Results of test: [$x a b c]’’
```

  The script output would look as follows.

Script Output

```
Value of test: 22
Results of test: args are: a b c
```

CHAPTER

Namespaces, Packages and Modules

8

The `namespace` and `package` commands implement two different concepts that work together to make it easier to write reusable, modular, easily maintained code. The `namespace` command provides encapsulation support for developing modular code. The `package` command and modules provide tools for organizing collections of code into libraries.

The `namespace` command collects persistent data and procedure names in a private scope where they will not interact with other data or procedure names. This lets you load new procedures without cluttering the global space (avoiding name collisions) and protects private data from unintentional corruption.

The `package` command groups a set of procedures that may be in separate files into a single logical entity. Other scripts can then declare which packages they will need and what versions of those packages are acceptable. The Tcl interpreter will find the directories where the packages are located, determine what other packages are required, and load them when they are needed. The `package` command can load both Tcl script files and binary shared libraries or DLLs.

The original `package` implementation used special files in each directory to list the available packages and versions in that directory. As the number of available packages increases, this caused a speed issue when applications start.

Since many packages are contained in a single file the `package` command was extended to also search for files that end with a .tm suffix, which are known as *modules*. The name of the module file encodes the package name and revision number. This allows single-file packages to be found by only examining directories, not opening and searching files.

This chapter discusses the following.

- The namespace scope.
- Encapsulating Tcl procedures and data in namespaces.
- Nesting one namespace within another.
- Modularizing Tcl scripts into packages.
- Creating a Tcl module.
- Assembling a namespaced library within a package.
- Guidelines for writing modules with relative namespace paths.

8.1 NAMESPACES AND SCOPING RULES

Chapter 7 discussed the Tcl global and procedure variable scopes. This section expands on that discussion and introduces the `namespace` and `variable` commands. These commands allow the Tcl programmer to create private areas within the program in which procedure and variable names will not conflict with other names.

Tcl/Tk: A Developer's Guide. DOI: 10.1016/B978-0-12-384717-1.00008-7

8.1.1 Namespace Scope

Namespaces provide encapsulation similar to that provided by C++ and other object-oriented languages. Namespace scopes have some of the characteristics of the global scope and some characteristics of a procedure local scope. A namespace can be viewed as a global scope within a scope. Namespaces are similar to the global scope in that:

- Procedures created at any procedure scope within a namespace are visible at the top level of the namespace.
- Variables created in a namespace scope (outside a local procedure scope) are persistent and will be retained between executions of code within the namespace.
- Variables created in a namespace scope (outside a local procedure scope) can be accessed by any procedure being evaluated within that namespace.
- While a procedure defined within a namespace is being evaluated, it creates a local scope within that namespace, not within the global namespace.

Namespaces are similar to local procedure scopes in that:

- Code being evaluated within a namespace can access variables and procedures defined in the global space.
- All namespaces are contained within the global scope.
- Namespaces can nest within each other.

Namespaces also have the following unique features.

- A namespace can declare procedure names to be exportable. A script can import these procedure names into both the global and other namespace scopes.
- A nested namespace can keep procedures and variables hidden from higher-level namespaces.

8.1.2 Namespace Naming Rules

A namespace can contain other namespaces, creating a tree structure similar to a filesystem. The namespace convention is similar to the filesystem naming convention.

- Instead of separating entities with slashes (/ or \), namespace entities are separated with double colons (::).
- The global scope is the equivalent of a filesystem "/" directory. It is identified as "::".
- Namespace identifiers that start with a double colon (::) are absolute identifiers and are resolved from the global namespace.

  An entity identified as `::foo::bar::baz` represents an entity named `baz`, in the `bar` namespace, which was created within the `foo` namespace, which was created in the global scope.
- Namespace identifiers that do not start with a double colon are relative, and are resolved from the current namespace.

  An entity identified as `bar::baz` represents an entity named `baz` that is a member of the namespace `bar`, which was created in the current namespace. The current namespace may be the global namespace (::) or a child namespace.

The following diagram shows the scopes in a script that contains two namespaces (`example` and `demo`), each of which contains procedures named `proc1` and `proc2`. Because the procedures are in separate namespaces, they are different procedures. If a script tried to define two `proc1` procedures at the global level, the second definition would overwrite the first.

In this example, `::example::proc1` and `::example::proc2` are procedures that are both called independently, whereas `::demo::proc2` is called from `::demo::proc1`. The `::demo::proc2` is displayed within `::demo::proc1` to show that the procedure local scopes nest within a namespace just as they nested within the `stack` example in the previous chapter.

Example 1

Script Example

```
# Define namespace example with two independent procedures.
namespace eval example {
  proc proc1 {} {puts "proc1"}
  proc proc2 {} {puts "proc2"}
}

# Define namespace demo with a procedure that invokes
# another procedure within the namespace.
namespace eval demo {
  proc proc1 {} {proc2}
  proc proc2 {} {puts "proc2"}
}
```

8.1.3 Accessing Namespace Entities

In C++, Java, or other strongly object-oriented languages, private data is completely private and other objects cannot access it. In Tcl, the namespace encapsulation is advisory. An entity (procedure or data) in a namespace can always be accessed if your script knows the full path to that entity.

A script can publish the procedures within a namespace it considers public with the `namespace export` command. Other scripts can import procedure names from the namespace into their local scope with the `namespace import` command. These commands are discussed later in this section. To avoid potential namespace collisions, your scripts should access namespace procedures by their full identifier, instead of importing the procedures into the global scope.

- Using the namespace identifier for an entity makes it easier to figure out what package a procedure or data originated from.
- Using the namespace identifier removes the possibility of name collisions when you load a new package with the same procedure and data names.

8.1.4 Why Use Namespaces?

If you always access namespace members by their full path, and namespaces do not provide truly private data, why should you use a namespace instead of having your own naming convention?

- The namespace naming conventions are enforced by the interpreter. This provides a consistent naming convention across packages, making it easier to merge multiple packages to get the functionality you need.
- Namespaces nest. Scripts being evaluated in one namespace can create a nested namespace within that namespace and access the new namespace by a relative name.
- Namespaces conceal their internal structure from accidental collisions with other scripts and packages. Data and procedures named with a naming convention exist in the global scope.
- A set of code within a namespace can be loaded multiple times in different namespaces without interfering with other instantiations of the namespace.

8.1.5 The `namespace` and `variable` Commands

The `namespace` command has many subcommands, but most Tcl applications will only use the `eval`, `export`, and `import` commands. The `children` command is discussed in this chapter, and although not required for using namespaces it provides some information that is useful in determining what namespaces exist within which scopes. The `namespace scope` and `namespace current` commands are useful when namespace procedures may be invoked by an event handler. These subcommands are discussed later.

The `namespace eval` command evaluates a script within a namespace. The namespace is created, if it does not already exist. The script evaluated by the `namespace eval` command can define procedures and variables within the namespace.

Syntax: `namespace eval` *`namespaceID arg1 ?argN...?`*

Create a namespace, and evaluate the script `arg` in that scope. If more than one `arg` is

present, the arguments are concatenated into a single script to be evaluated.

namespaceID The identifying name for this namespace.

*arg** The script or scripts to evaluate within namespace *namespaceID*.

Example 2

```
# Create a namespace named 'demo'
namespace eval demo {
  proc PrivateProc {} {
    # Do stuff
    ...
  }
  proc publicProc {} {
    # Do other stuff
    ...
  }
}
```

Once a namespace has been created, new procedures can be added to the namespace either by defining them within another `namespace eval` command or naming them for the namespace they occur in, as follows.

```
namespace eval demo {}
proc demo::newProc {} {
  # Do stuff
  ...
}
```

Note that `namespace eval` must be evaluated before defining a procedure within the namespace. Namespaces are only created by the `namespace eval` command.

It is often necessary to permit certain procedures in one namespace to be imported into other scopes. You will probably want to allow the procedures that provide the application programmer interface (API) to your package to be imported into other namespaces but not allow the importation of internal procedures.

The `namespace export` command defines the procedures within one namespace that can be imported to other scopes. By convention, procedures that will be exported are given names starting with a lowercase letter, whereas procedures for internal use have names starting with capital letters.

Syntax: `namespace export` *pattern1 ?patternN...?*

Export members of the current namespace that match the patterns. Exported procedure names can be imported into other scopes. The patterns follow `glob` rules.

*pattern** Patterns that represent procedure names and data names to be exported.

Example 3

```
# Create a namespace named 'demo'
namespace eval demo {
  namespace export publicProc publicAPI
  proc PrivateProc {} {
     # Do stuff
     ...
  }
  proc publicProc {} {
     # Do other stuff
     ...
  }
}
proc demo::publicAPI {} {...}
```

The `namespace import` command imports a procedure from one namespace into the current namespace. When a procedure that was defined within a namespace is imported into the global namespace, it becomes visible to all scopes and namespaces.

Syntax: `namespace import` *`?-force? ?pattern1 patternN...?`*

Imports variable and procedure names that match a `pattern`.

`-force` If this option is set, an import command will overwrite existing commands with new ones from the `pattern` namespace. Otherwise, `namespace import` will return an error if a new command has the same name as an existing command.

`pattern*` The patterns to import. The pattern must include the `namespaceID` of the namespace from which items are being imported. There will be more details on naming rules in the next section.

Example 4

```
# Create a namespace named 'demo'
namespace eval demo {
  namespace export publicProc publicAPI
  proc PrivateProc {} {
     #  Do stuff
     ...
  }
  proc publicProc {} {
     #  Do other stuff}\\
     ...
  }
}
proc demo::publicAPI {} {...}
```

```
# import all public procedures.
namespace import demo::pub*
```

When naming procedures that may be imported into other namespaces, it is a good rule to avoid names that may have collisions. In particular, avoid names that already exist as core Tcl commands. For example, a script that imports a modified version of `set` will develop some difficult-to-debug problems.

Importing procedures using a `glob` pattern can be fragile. If a namespace imports from multiple namespaces, you can get unexpected collisions. It is usually better to explicitly name the procedures you need to import.

For example, if namespace A imports all exported procedures from namespaces B and C, and B and C both import procedures from namespace D, there will be an import collision with the B::D:: procedures and C::D:: procedure names.

The Tcl interpreter generates an error when a script attempts to import a procedure that is already defined in the current namespace. If the script should redefine the procedures, you can use the `-force` flag with import to force the interpreter to import over existing procedures.

The `namespace children` returns a list of the namespaces that are visible from a scope.

Syntax: `namespace children` *`?namespaceID? ?pattern?`*

Returns a list of the namespaces that exist within `namespaceID`. (If `namespaceID` is not defined, the current namespace is used.)

`?namespaceID?` The namespace scope from which the list of namespaces will be returned. If this argument is not present, the list of namespaces visible from the current namespace will be returned.

`?pattern?` Return only namespaces that match a `glob` pattern. If this argument is not present, all namespaces are returned.

Example 5

```
if {[lsearch ::demo:: [namespace children]] == -1} {
    #  The demo namespace was not found
    #  Create a namespace named 'demo'
    namespace eval demo {
      ...
    }
  }
```

The `variable` command allows you to define persistent data within a namespace. It declares that a variable exists and may also define an initial value for the variable. The variable command can be used within a procedure or in the non-procedure namespace scope.

When used inside a procedure, the `variable` command is similar to the `global` command. It tells the interpreter to map a variable with this name in the outer namespace scope into the procedure local scope.

When used outside a procedure, the variable command is similar to a global scope assignment.

A variable defined with the `variable` command is equivalent to a variable defined in the `global` scope, except that the variable name is visible only within the scope of the namespace. These variables are easily accessed by the procedures in the namespace, but are not visible from the global namespace. The variables declared with the `variable` command are not destroyed when the namespace scope is exited.

Note that the syntax for the `variable` command is different from the `global` command. The `variable` command supports setting an initial value for a variable, whereas the `global` command does not.

Syntax: `variable` *`varName ?value? ?varNameN? ?valueN?`*

Declare a variable to exist within the current namespace. The arguments are pairs of name and value combinations.

`varName` The name of a variable.

`?value?` An optional value for the variable.

Example 6

```
# Create a namespace named 'demo'
namespace eval demo {
  #  name1 has no initial value
  variable name1
  #  name2 and name3 are initialized
  variable name2 initial2 name3 initial3
}
```

8.1.6 Creating and Populating a Namespace

The `namespace eval` command creates a new namespace. Because all arguments with a `namespace eval` command are evaluated as a Tcl script, any valid command can be used in the argument list. This includes procedure definitions, setting variables, creating graphics widgets, and so on.

A namespace can be populated with procedures and data either in a single `namespace eval` command, in multiple invocations of the `namespace eval` command or by creating procedures named with a full namespace path. The following example shows a namespace procedure that provides a unique number by incrementing a counter. The `uniqueNumber` namespace contains the counter variable, `staticVar`. The `getUnique` procedure, which is also defined within the `uniqueNumber` namespace, can access this variable easily, but code outside the `uniqueNumber` namespace cannot.

Example 7

Script Example

```
# Create a namespace.
namespace eval uniqueNumber {
    # staticVar is a variable that will be retained between
```

```
    # evaluations. This declaration defines the variable
    # and its initial value.
    variable staticVar 0;

    # allow getUnique to be imported into other scopes
    namespace export getUnique

    # return a unique number by incrementing staticVar
    proc getUnique {} {
      # This declaration of staticVar is the equivalent of a
      # global — if it were not here, then a staticVar
      # in the local procedure scope would be created.
      variable staticVar;
      return [incr staticVar];
      }
    }

# Example Script
# Display the currently visible namespaces:
puts "Visible namespaces from the global scope are:"
puts " [namespace children]\n"

# Display "get*" commands that are visible in the global
# scope before import
puts "Before import, global scope has these \"get*\" commands:"
puts "    [info commands get*]\n"

# Import all exported members of the namespace uniqueNumber
namespace import ::uniqueNumber::*

# Display "get*" commands that are visible in the global
# scope after importing
puts "After import, global scope has these \"get*\" commands:"
puts "    [info commands get*] \n"

# Run getUnique a couple times to prove it works
puts "first Unique val: [getUnique]"
puts "second Unique val: [getUnique]"

# Display the current value of the staticVar variable
puts "staticVar: [namespace eval uniqueNumber {set staticVar}]"
puts "staticVar: $uniqueNumber::staticVar"

# The next line generates an error condition because
#   staticVar does not exist in the global scope.

puts "staticVar is: $staticVar"
```

Script Output

```
Visible namespaces from the global scope are:
 ::uniqueNumber ::platform ::oo ::tcl

Before import, global scope has these "get*" commands:
    gets

After import, global scope has these "get*" commands:
    gets getUnique

first Unique val: 1
second Unique val: 2
staticVar: 2
staticVar: 2
can't read " staticVar": no such variable
    while executing
"puts "staticVar is: $staticVar""
```

There are a few points to note in this example.

- The procedure `getUnique` can be declared as an exported name before the procedure is defined.
- After the `uniqueNumber` namespace is defined, there are four namespaces visible from the global scope: the `::uniqueNumber` namespace, the `::tcl` namespace, the `::oo` (Object Oriented) namespace and the `::platform` (platform information). The `::tcl`namespace is always present when the Tcl interpreter is running. The `::oo` and `::platform` namespaces are present in Tcl version 8.6. Other namespaces may also be present, depending on what packages have been loaded.
- The `namespace import ::uniqueNumber::*` command imports all exported entities from `::uniqueNumber` into the global scope.
- The value of the `staticVar` variable can be accessed via the `namespace eval`. It can also be accessed as `::uniqueNumber::staticVar`.
- The `staticVar` variable is initialized by the line
  `variable staticVar 0`
  when the namespace is created. This code is roughly equivalent to placing
  `set staticVar 0`
  within the arguments to `namespace eval`, which would cause the variable to be defined at the top scope of the `uniqueNumber` namespace.
- Using the `variable varName initValue` construct is safer than initializing variables with the `set` command. When the Tcl interpreter searches for a variable to assign the value to, it looks first in the local namespace, then in the global namespace. If the variable exists in the current namespace, the value is assigned to the variable in the current namespace. If the variable does not exist in the current namespace, but does exist in the global namespace, the global variable will be assigned the value. If the variable exists in neither namespace, it is created in the current namespace. The `variable varName initValue` will always initialize a variable within the namespace.

8.1.7 Namespace Nesting

Tcl namespaces can be nested within one another. This provides lightweight equivalents of the object-oriented concepts of inheritance (the is-a relationship) and aggregation (the has-a relationship). Note that even using namespaces, pure Tcl is not quite a *real* object-oriented language. The `[incr Tcl]` and `TclOO` extensions use namespaces to support the complete set of object-oriented functionality, and smaller packages such as `stooop` and `snit` provide frameworks for lighter-weight (and fewer-featured) object-style programming. The `[incr Tcl]` extension is discussed in Chapter 17.

A namespace can be used to create an object. It can inherit functionality and data from other namespaces by nesting those namespaces and importing procedures into itself. Procedures can be imported from any namespace, regardless of where the namespace is located in the hierarchy. If the namespace being imported from only includes procedures, your script can create a single copy of that namespace in the global scope, and import from there to as many objects as it creates.

However, if the namespace being imported from also includes variables, you need a copy of the namespace to hold a separate copy of the variables for each object your script creates. In this case, it is simplest to nest the namespaces, rather than keep multiple copies at the global scope.

Note that using namespaces to implement inheritance is implemented in the opposite way as C++ or Java-style inheritance. In C++ and Java, a child class inherits functionality down from a parent, whereas in Tcl a primary namespace can inherit functionality up from a nested namespace.

If two or more namespaces need functionality that exists in a third namespace, there are a couple of options. They can create a shared copy of the third namespace in the scope of the first two namespaces, or each can create a copy of the third namespace nested within its own namespace.

One advantage of nesting the third namespace is that it creates a unique copy of the persistent data defined within the third namespace. If a namespace is shared, that data is also shared.

Note that whereas procedures can be imported from any namespace, regardless of parent/child relationship, variables cannot be imported. If your design requires separate copies of data, you must have separate copies of the namespace. The namespace copies can be placed at any position in the namespace hierarchy, but it is simplest to inherit the functionality from child namespaces.

The next example shows the `uniqueNumber` namespace being shared by two separate namespaces (`Package1` and `Package2`). Since the counter is shared between the two namespaces, the unique numbers are never duplicated. The example after this shows how two namespaces can have the `uniqueNumber` namespace nested within them so that each namespace gets a full set of unique numbers with no gaps.

Example 8

Script Example

```
# These commands create a uniqueNumber namespace in the scope of
# the script that invokes it.
namespace eval uniqueNumber {
  variable staticVal 0
  namespace export getNext;
  proc getNext {} {
     variable staticVal
     return [incr staticVal]
  }
}
```

```
# Create the unique1 namespace,
# Create a counter namespace within the unique1 namespace
# The Package1 namespace includes a procedure to return unique
# numbers

namespace eval unique1 {
  namespace import ::uniqueNumber::getNext
  proc example {} {
    return "unique1::example: [getNext]"
  }
}

# Create the unique2 namespace,
# Create a counter namespace within the unique2 namespace
# The unique2 namespace includes a procedure to report unique
# numbers

namespace eval unique2 {
  namespace import ::uniqueNumber::getNext
  proc example {} {
    return "unique2::example: [getNext]"
  }
}

# Example Script
puts "unique1 example returns: [::unique1::example]"
puts "invoking unique1::getNext directly returns: \
  [unique1::getNext]"
puts "unique1 example returns: [::unique1::example]"
puts ""
puts "unique2 example returns: [::unique2::example]"
puts "unique2 example returns: [::unique2::example]"
puts "unique2 example returns: [::unique2::example]"
```

Script Output

```
unique1 example returns: unique1::example: 1
invoking unique1::getNext directly returns: 2
unique1 example returns: unique1::example: 3

unique2 example returns: unique2::example: 4
unique2 example returns: unique2::example: 5
unique2 example returns: unique2::example: 6
```

In the next example, the `uniqueNumber` namespace is created inside the `unique1` and `unique2` namespaces. There are two `staticVal` variables—`unique1::uniqueNumber::staticVal` and `unique2::uniqueNumber::staticVal`.

Example 9

Script Example

```
# This string of commands creates a uniqueNumber namespace
#  in the scope of the script that invokes it.

proc createUnique {} {
  uplevel 1 {
    namespace eval uniqueNumber {
      variable staticVal 0
      namespace export getNext;
      proc getNext {} {
         variable staticVal
         return [incr staticVal]
      }
    }
    namespace import uniqueNumber::getNext
  }
}

# Create the unique1 namespace,
# Create a uniqueNumber namespace within the unique1 namespace
# The Package1 namespace includes a procedure to return unique
# numbers

namespace eval unique1 {
  createUnique
  proc example {} {
    return "unique1::example: [getNext]"
  }
}

# Create the unique2 namespace,
# Create a uniqueNumber namespace within the unique2 namespace
# The unique2 namespace includes a procedure to report unique
# numbers

namespace eval unique2 {
  createUnique
  proc example {} {
    return "unique2::example: [getNext]"
  }
}

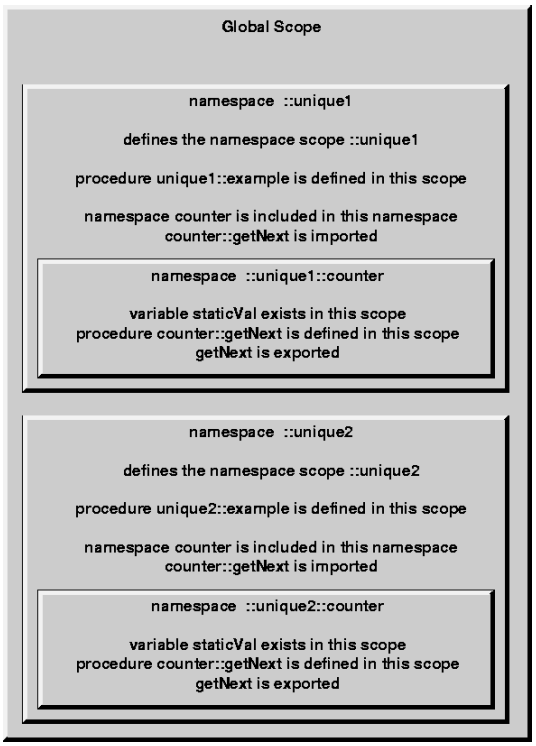
# Example Script
puts "unique1 example returns: [::unique1::example]"
puts "invoking unique1::getNext directly returns: \
  [unique1::getNext]"
puts "unique1 example returns: [::unique1::example]"
```

```
puts ""
puts "unique2 example returns: [::unique2::example]"
puts "unique2 example returns: [::unique2::example]"
puts "unique2 example returns: [::unique2::example]"
```

Script Output

```
unique1 example returns: unique1::example: 1
invoking unique1::getNext directly returns: 2
unique1 example returns: unique1::example: 3
unique2 example returns: unique2::example: 1
unique2 example returns: unique2::example: 2
unique2 example returns: unique2::example: 3
```

Note that the `createUnique` process uses the `uplevel` to force the `namespace eval` to happen at the same scope as the code that invoked `createUnique`. By default, a procedure will be evaluated in the scope in which it was defined. Since the `createUnique` procedure is created in the global scope, it will by default create the `getNext` namespace as a child namespace of the global space. By using the `uplevel` command, we force the evaluation to take place in the scope of the calling script: within the package namespace. The following illustration shows what the namespaces look like after evaluating the previous example.



Namespaces can be used to implement aggregation by including multiple copies of another namespace (complete with data) in child namespaces. For example, a simple stack can be implemented as follows.

```
set stackDef {
  variable stack {}
  proc push {value} {
    variable stack
    lappend stack $value
  }
  proc pop {} {
    variable stack
    set rtn [lindex $stack end]
    set stack [lrange $stack 0 end-1]
    return $rtn
  }

  proc peek {{pos end}} {
    variable stack
    return [lindex $stack $pos]
  }

  proc size {} {
    variable stack
    return [llength $stack]
  }

}
```

The Tower of Hanoi puzzle is 3 posts with rings of various sizes on them. At the beginning, all of the rings are on the left most stack, with the largest at the bottom and the smallest on the top. The object of the puzzle is to move the rings to another post. The trick is that you can only place a smaller ring on top of a larger ring.

The example below uses the `stackDef` string to create three stacks within the `Hanoi` namespace to implement a text-only version of the Tower of Hanoi puzzle. The rings are represented by a number from 1 to 4, which is the size of the ring.

A move is made by popping a value off the top of one stack and pushing it onto the top of another stack.

The stacks are displayed using the `format` and `string repeat` commands.

Example 10
Script Example

```
namespace eval Hanoi {
  namespace eval left $stackDef
  namespace eval center $stackDef
  namespace eval right $stackDef
```

```
##################################################
# proc moveRing {from to}--
#   Move the last element of the "from" stack
#   to the end of the "to" stack.
#
# Arguments
#   from: The name of the stack to move from.
#   to: The name of the stack to move to.
# Results
#   The "from" and "to" stacks are modified.
proc moveRing {from to} {
    ${to}::push [${from}::pop]
}
proc show {} {
  puts ""
  for {set p 4} {$p >= 0} {incr p -1} {
    set out ""
    foreach stack {left center right} {
      set ring [${stack}::peek $p]
      if {$ring ne ""} {
        set l [format %5s [string repeat - $ring]]
        set r [format %-5s [string repeat - $ring]]
      } else {
        set l [format %5s " "]
        set r $l
      }
      append out [format %s%s%s $l "||" $r]
    }
    puts $out
  }
  puts ""
}

proc start {} {
  variable left
  # Fill the left stack
  for {set i 4} {$i > 0} {incr i -1} {
    left::push $i
  }
}

proc done {} {
  if {[right::size] == 4} {
    return 1
  } else {
    return 0
  }
}
```

```
}
Hanoi::start

array set abbr {l left r right c center}
while {![Hanoi::done]} {
  Hanoi::show
  puts -nonewline "From (lcr): "
  flush stdout
  set from [gets stdin]
  puts -nonewline "To (lcr): "
  flush stdout
  set to [gets stdin]
  Hanoi::moveRing $abbr($from) $abbr($to)
}
```

Script Output

```
     ||        ||       ||
    -||-       ||       ||
   --||--      ||       ||
  ---||---     ||       ||
 ----||----    ||       ||

From (lcr): l   <== User Types l
To (lcr): c     <== User Types c

     ||        ||       ||
     ||        ||       ||
   --||--      ||       ||
  ---||---     ||       ||
 ----||----   -||-      ||

From (lcr): l  <== User Types l
To (lcr): r    <== User Types r

     ||        ||       ||
     ||        ||       ||
     ||        ||       ||
  ---||---     ||       ||
 ----||----   -||-    --||--

From (lcr): c  <== User Types c
To (lcr): r    <== User Types r

     ||        ||       ||
     ||        ||       ||
     ||        ||       ||
  ---||---     ||      -||-
 ----||----    ||     --||--
```

8.1.8 Namespace Ensembles

The `namespace ensemble` command makes it easier to write commands that mimic the "C" layer commands that have subcommands, and it provides a tool for better pure-Tcl object-style programming. We'll examine the Tower of Hanoi again in the discussion of TclOO.

One problem with the code above is that the procedures in the `stackDef` are duplicated in each of the left, center and right namespaces. In this example, the procedures are small and there are only three of them. In other applications the procedures might be large, or there might be many of them, and the duplication could become a machine resource issue.

Another problem is that the code is slightly ugly with the need to escape variable names and include all the double colons.

Tcl 8.5 introduced the `namespace ensemble` command to namespaces. This command makes it easy to treat a namespace like a procedure and procedures within a namespace as if they were subcommands for that command. The ensemble commands can be invoked like regular Tcl commands, without the `::` separators.

For example, instead of `left::push` we can write `left push`.

The `namespace ensemble` command has a few sub commands. The one that creates a new ensemble command is `namespace ensemble create`.

Syntax: `namespace ensemble create -map ensName script`

Create an ensemble command to map a command name in the current scope to different command.

`ensName` The ensemble name of the command.

`script` The script to evaluate when this ensemble command is invoked.

The previous code can be rewritten to use an ensemble as shown below. In the new code, the `stack` variable exists within the `left`, `center` and `right` namespaces, rather than being embedded within that namespace. The commands exist in the shared `stackCmds` namespace, rather than being repeated in each post's namespace.

The next example also uses the `namespace current` command to identify the appropriate `stack` list. The `namespace current` command returns the fully qualified name of the namespace scope in which code is being evaluated. This is useful when code within a namespace is registered for callback operation using the `trace`, `fileevent`, etc. commands or if it is used as a callback from a Tk Widget (which will be discussed in Chapter 11).

Syntax: `namespace current`

Returns the fully qualified name of the current namespace.

Example 11

Script Example

```
namespace eval foo {
  namespace eval bar {
    namespace eval and {
      namespace eval grill {
      }
```

```
    }
  }
}

namespace eval ::foo::bar::and {namespace eval grill {
    puts "Current namespace is [namespace current]"
    }
}
Current namespace is ::foo::bar::and::grill
```

Notice that the code below doesn't need extra curlie braces to delimit the variable names. The normal Tcl spacing is sufficient.

```
namespace eval stackCmds {
    proc push {name val} {
        upvar $name stack
        lappend stack $val
    }

    proc peek {name {pos end}} {
        upvar $name stack
        return [lindex $stack $pos]
    }

    proc size {name} {
        upvar $name stack
        return [llength $stack]
    }

    proc pop {name} {
        upvar $name stack
        set rtn [lindex $stack end]
        set stack [lrange $stack 0 end-1]
        return $rtn
        }
}

set stackDef {
  # Stack variable is maintained in each stack
  variable stack {}

  # Commands are taken from stackCmd namespace
  namespace ensemble create -map [list \
     peek "::stackCmds::peek [namespace current]::stack" \
     size "::stackCmds::size [namespace current]::stack" \
     push "::stackCmds::push [namespace current]::stack" \
     pop  "::stackCmds::pop  [namespace current]::stack"]
}
```

```
namespace eval Hanoi {
  namespace eval left $stackDef
  namespace eval center $stackDef
  namespace eval right $stackDef

  ################################################
  # proc moveRing {from to}--
  #   Move the last element of the "from" stack
  #   to the end of the "to" stack.
  #
  # Arguments
  #   from: The name of the stack to move from.
  #   to: The name of the stack to move to.
  # Results
  #   The "from" and "to" stacks are modified.
  proc moveRing {from to} {
      $to push [$from pop]
  }

  proc show {} {
    puts ""
    for {set p 4} {$p >= 0} {incr p -1} {
      set out ""
      foreach stack {left center right} {
        set ring [$stack peek $p]
        if {$ring ne ""} {
          set l [format %5s [string repeat - $ring]]
          set r [format %-5s [string repeat - $ring]]
        } else {
          set l [format %5s " "]
          set r $l
        }
        append out [format %s%s%s $l "||" $r]
      }
      puts $out
    }
    puts ""
  }

  proc start {} {
    variable left
    # Fill the left stack
    for {set i 4} {$i > 0} {incr i -1} {
      left push $i
    }
  }

  proc done {} {
    if {[right size] == 4} {
      return 1
```

```
    } else {
      return 0
    }
  }
}
Hanoi::start

array set abbr {l left r right c center}
while {![Hanoi::done]} {
  Hanoi::show
  puts -nonewline "From (lcr): "
  flush stdout
  set from [gets stdin]
  puts -nonewline "To (lcr): "
  flush stdout
  set to [gets stdin]
  Hanoi::moveRing $abbr($from) $abbr($to)
}
```

This section described creating nested namespaces by using a procedure or using a script string. These techniques require the child namespace code to be resident in the script that uses it. The next section discusses how packages can be used to find and load appropriate scripts from other files, and then describes how to nest namespaces contained within a package.

8.2 PACKAGES

The `namespace` command allows you to assemble related information and procedures within a private area. The `package` commands allow you to group a set of procedures that may be in multiple files and identify them with a single name. The `namespace` command provides encapsulation functionality, whereas the `package` command provides library functionality. This section describes how to turn a set of procedures into a package other scripts can load easily.

People frequently refer to any collection of procedures and variables that work together to perform related functions as a package. A *real* Tcl package is a collection of procedures that can be indexed and loaded easily with the `package` command. The actual code may be identified using the `package` commands discussed first, or the `module` naming convention that will be discussed later.

The `package provide` command defines a set of procedures that can be a part of a package identified by the package name and a revision number. These procedures can be indexed and can be loaded automatically when they are needed by another script. The Tcl `package` command provides a framework for the following.

- Finding and loading the code modules a script requires.
- Tracking the version numbers of packages and loading the proper version.
- Defining whether the file to be loaded is a script file (discussed here) or shared library/DLL (discussed in Chapter 15).
- A single file package may be identified by including the file in the `module` search path and using a file name that conforms to the `module` naming convention instead of using the `package` commands to create the package.

8.2.1 How Packages Work

A Tcl package includes an index file that lists the procedures and commands defined in the package. The Tcl interpreter resolves unknown procedures by searching the index files in the directories listed in the global variable `auto_path` for required packages. The `auto_path` variable is defined in the `init.tcl` script, which is loaded when a Tcl or Tk interpreter is started.

This section describes creating the index files and adding your package directories to the list of places the interpreter will search. Note that when you create an index for a package that has procedures defined within a namespace, only the procedure names listed in a `namespace export` command will be indexed.

8.2.2 Internal Details: Files and Variables Used with Packages

The following files and global variables are used to find and load a package.

`pkgIndex.tcl` file

These files contain a list of procedures defined by the packages in the directory with this file. The `pkgIndex.tcl` files are created with the `pkg_mkIndex` command.

`auto_path` global variable

The `auto_path` variable contains a list of the directories that should be searched for package index files.

The `auto_path` variable is defined in the `init.tcl` script, which is loaded when a `tclsh` interpreter is started.

On UNIX systems, `init.tcl` will probably be found in `/usr/local/lib/tcl8.6`, `/opt/ActiveTcl8.6/lib` or some variant, depending on your installation and the revision number of your `tclsh`.

On Windows systems, `init.tcl` is stored in `\Program Files\Tcl\lib\tcl8.4`, or some variant, depending again on your version of Tcl and the base directory in which you installed the Tcl interpreter.

On Macintoshes running OS8 or earlier, the `init.tcl` file may be stored in `Tcl/Tk Folder 8.4:Tcl:library`, again depending on installation options.

On Mac OS/X, the `init.tcl` file may be stored in `/Library/Frameworks/Tcl.framework/Versions/ 8.6/Resources/Scripts/init.tcl`, again depending on installation options.

When the Tcl interpreter is trying to find a package to fulfill a `package require` command, it will search all of the directories listed in the `auto_path` variable, and all of the children in those directories, but not second-level subdirectories of the directories listed. This makes it possible to place a single directory in the `auto_path` list, and use separate directories under that for each supported package.

8.2.3 Package Commands

The package functionality is implemented with several commands. These commands convert a simple script into a package. The pkg_mkIndex command creates a package index file. It is evaluated when a package is installed rather than when a script is being evaluated.

The pkg_mkIndex command scans the files identified by the patterns for package provide commands. It creates a file (pkgIndex.tcl) in the current directory. The pkgIndex.tcl file describes the commands defined in the package, and how to load the package when a script requires one of these commands.

Syntax: pkg_mkIndex *?-option? dir pattern ?pattern?*

Creates an index of available packages. This index is searched by package require when it is determining where to find the packages.

?-option? An option to fine-tune the behavior of the command. The option set has been evolving, and you should check the documentation on your system for details.

dir The directory in which the packages being indexed reside.

*pattern** Glob-style patterns that define the files that will be indexed.

When the pkg_mkIndex command has finished evaluating the files that match the pattern arguments, it creates a file named pkgIndex.tcl in the dir directory. The pkgIndex.tcl file contains the following.

- The names of the packages defined in the files that matched the patterns.
- The version level of these packages.
- The name of the command to use to load the package.
- Optionally, the names of the procedures defined in those packages.

The pkg_mkIndex will overwrite an existing pkgIndex.tcl file. If you are developing multiple packages in a directory, you will need to enter the name of each file every time you update the index. In this case, it may become simpler to create a two-line script that lists all files, as follows.

```
#!/usr/local/bin/tclsh
pkg_mkIndex [pwd] file1.tcl file2.tcl file3.tcl
```

The package provide command defines the name and version of the package that includes these procedures. This command makes the procedures defined within the file available to other scripts.

Syntax: package provide *packageName ?version?*

Declares that procedures in this script module are part of the package packageName. Optionally declares which version of packageName this file represents.

packageName The name of the package that will be defined in this script.

?version? The version number of this package.

The pkg_mkIndex command looks for a package provide command in a file. It uses the package name and version information to generate an entry in the pkgIndex.tcl file with the name and version of the package and a list of procedures defined in this file.

The package provide command tells the Tcl interpreter that all of the procedures in the file are members of a package. You can use multiple files to construct your package, provided there is a

`package provide` command in each file. Note that if you have multiple `package provide` commands with different *packageName* or *version* arguments in a source file, the `pkg_mkIndex` will not be able to generate an index file, and may generate an error.

```
#  Declare this file part of myPackage
package provide myPackage 1.0
```

The `package require` command declares that this script may use procedures defined in a particular package. Scripts that require procedures defined in other files will use this command.

Syntax: `package require` *?-exact? packageName ?versionNum?*

Informs the Tcl interpreter that this package may be needed during the execution of this script. The Tcl interpreter will attempt to find the package and be prepared to load it when required.

-exact	If this flag is present, `versionNum` must also be present. The Tcl interpreter will load only that version of the package.
packageName	The name of the package to load.
?versionNum?	The version number to load. If this parameter is provided, but `-exact` is not present, the interpreter will allow newer versions of the package to be loaded, provided they have the same major version number (see Section 8.2.4 for details of version numbering). If this parameter is not present, any version of `packageName` may be loaded.

Example 12

```
# This program needs to load myPackage
package require myPackage 1.0
```

The `package require` command checks the `pkgIndex.tcl` files in the search path (defined in `auto_path`) and selects the best match to the version number requested. If the `-exact` flag is used, it will select an exact match to that version number. If `versionNum` is defined and the `-exact` is not set, the largest minor revision number greater than `versionNum` will be selected. If `versionNum` is not defined, the highest available version will be selected. If an acceptable revision cannot be found, `package require` will generate an error.

If the Tcl interpreter can locate an appropriate package, it will load the required files as necessary to resolve the procedures defined in the package. Older Tcl versions (prior to 8.2) deferred loading packages until the procedures were needed. Since 8.2 the default behavior is to load packages immediately. The older, Just-In-Time style of loading can still be used by including the `-lazy` option to the `pkg_mkIndex` command when you create the `pkgIndex.tcl` file.

The packages loaded by `package require` are loaded into the global namespace.

8.2.4 Version Numbers

The `package` command has some notions about how version numbers are defined. The rules for version numbers are as follows.

- Version numbers are one or two positive integers separated by periods (e.g., `1.2` or `75.19`).
- The first integer in the string is the major revision number. As a general rule, this is changed only when the package undergoes a major modification. Within a major revision the API should be constant and application code should behave the same with later minor revision numbers. Between major revisions, there may be changes such that code that worked with one major revision will not work with another.
- The second integer is the minor revision number. This corresponds to an intermediate release. You can expect that bugs will be fixed, performance enhanced, and new features added, but code that worked with a previous minor revision should work with later minor revisions.
- The Tcl interpreter compares revision numbers integer by integer. Revision 2.0 is more recent than revision 1.99.

8.2.5 Package Cookbook

This section describes how to create and use a Tcl package. The next section will show a more detailed example.

Creating a Package

1. Create the Tcl source file or files. You can split your package procedures across several files if you wish.
2. Add the command
   `package provide` *`packageName versionNumber`*
   to the beginning of each of these Tcl source files.
3. Invoke `tclsh`. If you have several revisions of Tcl installed on your system, be certain to invoke the correct `tclsh`. The `package` command was introduced with revision 7.5 and has been modified slightly in successive versions of Tcl.
4. At the `%` prompt, type
   `pkg_mkIndex` *`directory fileNamePattern ?Pattern2...?`*
   You may include multiple file names in this command. The `pkg_mkIndex` command will create a new file named `pkgIndex.tcl`, with information about the files that were listed in the `pkg_mkIndex` command.

In Tcl version 8.2 and more recent, the default option is to create a `pkgIndex.tcl` file that loads packages immediately. Older versions of Tcl created `pkgIndex.tcl` files that would defer loading until a procedure was needed. This behavior can be duplicated in newer interpreters with the `-lazy` option to `pkg_mkIndex`.

Using a Tcl Package

1. If the package is not located in one of the Tcl search directories listed by default in the `auto_path` variable, you must add the directory that contains the package's `pkgIndex.tcl` file to your search list. This can be done with one of the following techniques.
   A. Add a line resembling
   ```
   lappend auto_path /usr/project/packageDir
   ```
   to the beginning of your Tcl script.

   B. Set the environment variable `TCLLIBPATH` to include your package directory. The environment variable need not include the path to the default Tcl library. Note that `TCLLIBPATH` is a Tcl-style whitespace-delimited list, rather than the shell-style colon-delimited list.
   C. Add your package directory to the `auto_path` definition in the `init.tcl` file. This file is located in the Tcl library directory. The location of that directory is an installation option. Modifying the `init.tcl` script will require continuing the nonstandard modification every time the Tcl interpreters are updated. This can make a system fragile.
2. Add the line

   ```
   package require packageName ?versionNumber?
   ```

   to load the package into the global namespace.

8.3 TCL MODULES

The package system is very versatile. A package can include one or more files. A file in a package can be either Tcl code or a loadable compiled library.

This versatility comes at a cost. In order to find a package the Tcl interpreter must look through every directory and subdirectory for `pkg_index.tcl` files, open, and scan them for appropriate packages.

Many packages can fit into a single file. The `module` system uses a file naming convention to identify a module, saving a few steps over the package system.

To be used as a module a package must conform to a few conventions:

1. A module must be a single file with a filename in the format `NAME-VERSION.tm`. The `NAME` section must start with a letter, after which it may be alphanumeric. The `VERSION` section must be digits or a period.
2. A module must include a `package provide` command.
3. The package provided in the `package provide` command must match the `NAME` portion of the filename.
4. The version provided in the `package provide` command must match the `VERSION` portion of the filename.

A simple module would resemble this:

```
$> cat testmod-1.2.tm
package provide testmod 1.2
proc testmodCmd {} {
  puts "testmod is loaded OK"
}
```

The module system makes use of namespaces, which were not part of Tcl when the `package` system was developed. Rather than use a global variable `auto_path` to define the directories to be searched, the module system keeps a hidden variable and provides an API to query and modify the list of folder to be examined for modules.

Syntax: `::tcl::tm::path add` *path*

Add a directory to the beginning of the list of directories to be searched for a module.

path A full or relative path to the directory to be added to the list.

The `::tcl::tm::path add` command enforces that the directories to be searched are not duplicated and are not children of other folders in the search path.

Your script should not need to examine the search path or remove directories from the search path. The commands to examine and reduce the search list can be useful during debugging.

Syntax: `::tcl::tm::path list`

Returns the list of directories to be searched for a module.

Example 13

Script Example

```
# Add the current folder to the search path if it
# is not already in the set of directories to be searched.
if {[lsearch [::tcl::tm::path list] [pwd]] < 0} {
  ::tcl::tm::path add [pwd]
}
```

A path can be removed from the list with the `::tcl::tm::path remove` command. This can be useful if you have a test version of a module and you want to be certain to load your new module, not the ones in your normal search path.

Syntax: `::tcl::tm::path remove` *path*

Remove a directory from the list of directories to be searched for a module.

path A full or relative path to the directory to be removed to the list.

8.4 NAMESPACES AND PACKAGES

Namespaces and packages provide different features to the Tcl developer, with namespaces providing encapsulation support and packages providing modularization/library support. A package can be written with or without using a namespace. (In fact, packages were added to Tcl a couple years before namespace support was added.) The three namespace options when developing a package are:

- *Use no namespaces.* The package can be loaded into the global scope or merged into a script-defined namespace. This option is suitable for small special-purpose packages that will not conflict with other packages, or are intended to always be merged into another namespace. This is not a recommended technique, but may be appropriate under some circumstances.
- *Require the package be created in a given namespace.* The package uses an absolute name for the namespace in the `namespace eval` command, and uses absolute names to define procedures used within the namespace. This is appropriate for packages that add new language features (for

example, communications protocols, database connections, and http support) that do not include data structures that are specific to a given instantiation.

- *Allow the package to be created in an arbitrary namespace.* The package uses a relative name for the namespace in the `namespace eval` command, and uses relative names to define procedures used within the namespace. This is appropriate for code modules that contain state information that is specific to a given instantiation of the namespace. Complex data structures (such as trees and stacks or GUI objects that maintain state information) are examples of this type of design.

Note that nesting namespaces requires that code be duplicated as well as data. If you have an application that creates many objects, you may run into memory constraints. In that case, you may need to separate procedures and variables into separate namespaces in order to have nested copies of the namespaces with data, and import procedures from a single copy of the code namespace. There are a few techniques that can be used to ensure a script will be namespace neutral.

- *Use only relative namespace identifiers.* Namespace names that start with a double colon (`::`) are absolute and are rendered from the global scope. Namespaces that start with a letter are relative names and are resolved from the current namespace.
- *Define procedures within the `namespace eval`, or with relative namespace names.*

```
namespace eval bad {
    variable badVar
}
proc ::bad::badProc {} {
    variable badVar
    set badVar "Don't Do This"
}
namespace eval good {
    variable goodVar
    proc goodProc1 {} {
      variable goodVar
      set goodVar "OK"
    }
}
proc good::goodProc2 {} {
    variable goodVar
    set goodVar "Also OK"
}
```

- Use *`namespace current` or a relative name to identify the current namespace*. The `namespace current` command is discussed in Chapters 11 and 14.
- Use the `variable` command to map namespace variables into procedures within the namespace rather than using namespace pathnames for variables.

```
namespace eval bad {
  variable badVar
  proc badProc {} {
```

```
        set ::bad::badVar "Don't Do This"
    }
}
namespace eval good {
    variable goodVar
    proc goodProc {
        variable goodVar
        set goodVar OK
    }
}
```

The `package require` command will load packages into the toplevel (::) namespace. If you wish to force a package to be loaded into a child namespace, you must work around the normal behavior. The package system uses the `auto_index` global array to hold the names of procedures defined in required packages so that they can be loaded automatically when needed. This can be used to work around the normal behavior of the package system as described below.

1. Your package must export at least one procedure in the package's namespace scope.
2. Create a `pkgIndex.tcl` using the `pkg_mkIndex` command with the -lazy option.
3. Within the application that requires a package, evaluate the command that Tcl has saved in `::auto_index(`*`procedureName`*`)`.

An example is shown below:

```
# cat tstPkg.tcl
package provide tstPkg 1.0
namespace eval tstPkg {
    namespace export sampleProc
    proc sampleProc {} {puts "[namespace current]"}
}


# tclsh
pkg_mkIndex -lazy . tstPkg.tcl

# cat useTstPkg.tcl
lappend auto_path .
package require tstPkg
namespace eval tstInChild {
  eval $::auto_index(::tstPkg::sampleProc)
}
tstInChild::tstPkg::sampleProc

# OUTPUT from tclsh useTstPkg.tcl

tstInChild::tstPkg
```

8.5 HANOI WITH A STACK NAMESPACE AND PACKAGE

The previous version of the Tower of Hanoi puzzle code included a set of stackCmds that implemented the three stacks of rings. A stack is a primitive data structure that's useful for many types of processing, including parsing XML and HTML.

A set of stack procedures is the type of tool that should be generalized and put into a package where other applications can access it.

The next example shows the namespace with `stackCmds` from the previous Tower of Hanoi example converted to a package. The stack operations, `push`, `pop`, etc. remain as they were in the previous example. The `stackDef` string of commands to create ensemble commands is moved into the namespace and a new command is added to create a stack.

The `createStack` command provides a level of implementation hiding. In the previous version of the stack, the Tower of Hanoi code had intimate knowledge of the stack implementation. Since the stack and puzzle code were designed to run from a single file, this was appropriate.

When the stack is made generally available to other applications, it's better to hide the internal details from an application developer.

Notice that the `namespace ensemble create` command that maps `::stackCmds::createStack` to `stackCmds createStack` is not within a procedure. This is code that's evaluated when the stack code is loaded by a `package require stackCmds` command.

Example 14

Stack Package

```
package provide stackCmds 1.0

namespace eval stackCmds {

  # Define a set of commands that will be used to create an
  # ensemble version of the stack.
  variable stackDef {
    # stack variable is maintained in each stack
    variable stack {}

    # Commands are taken from stackCmd namespace
    namespace ensemble create -map [list \
       peek "::stackCmds::peek [namespace current]::stack" \
       size "::stackCmds::size [namespace current]::stack" \
       push "::stackCmds::push [namespace current]::stack" \
       pop  "::stackCmds::pop  [namespace current]::stack"]
  }

  # Make the createStack procedure an ensemble command

  namespace ensemble create -map {
       createStack "::stackCmds::createStack"
       }

  proc createStack {stackName} {
```

```
    variable stackDef
    uplevel 1 [list namespace eval $stackName $stackDef]
  }

  proc push {name val} {
    upvar $name stack
    lappend stack $val
  }

  proc peek {name {pos end}} {
    upvar $name stack
    return [lindex $stack $pos]
  }

  proc size {name} {
    upvar $name stack
    return [llength $stack]
  }

  proc pop {name} {
    upvar $name stack
    set rtn [lindex $stack end]
    set stack [lrange $stack 0 end-1]
    return $rtn
  }

}
```

This package is wholly contained in a single file and is suitable for being treated like a Tcl module. Saving the code in a file named `stackCmds-1.0.tm` allows a `package require` command to find the package without creating a `pkg_index.tcl` file.

Only a few lines of the previous Tower of Hanoi code need to be modified. The differences in the first few lines are shown below.

The current working directory is not normally checked for packages. In this example, the package is in the same directory as the mainline Tower of Hanoi code. The first new line adds the current directory to the module search path.

The `stackCmds` and `stackDef` code have been moved to the package and replaced by the `package require` command.

The code to create the three posts is replaced by calls to the `stackCmds createStack` namespace ensemble commands.

Example 15

Tower of Hanoi Changes

```
::tcl::tm::path add .
package require stackCmds
```

```
namespace eval Hanoi {
  stackCmds createStack left
  stackCmds createStack center
  stackCmds createStack right
...
```

8.6 CONVENTIONS AND CAVEATS

It is a common convention in Tcl programming to use the same name for a package and the namespace that holds the code and data to implement the package's functionality. The `namespace export` commands allow the entry points to be accessed (and `namespace imported`) by other programs.

The `pkg_mkIndex` command will put only exported procedure names into the `pkgIndex.tcl` file. All procedures that can be called from code outside the package should be exported. Internal procedures should not be exported.

If a namespace has internal variables that need to be initialized, define them using the `variable` command rather than the `set` command. This avoids potential confusion or limiting the use of namespaces that you might want to allow to be nested.

All entry points should be visible from the top level of a namespace. If you implement a package as a set of nested namespaces and support direct access to the nested namespaces API, your top level namespace should import those procedures into the top level namespace. This technique allows the package developer to rework the package architecture without breaking application code that would otherwise have to understand the package's internal structure.

A procedure in a higher-level namespace can be invoked from a nested namespace as well as accessing procedures in a nested namespace from the higher level namespace. However, the procedure will need to be defined with a complete path. Tcl does not search up a tree of nested namespaces to find a procedure name. When Tcl evaluates a command, it tries first to evaluate the command in the requested namespace. If that fails, it attempts to evaluate the command in the global namespace.

When `info commands` is evaluated in a nested namespace the output includes the commands in the current namespace scope and the global scope. It does not report commands in the higher level scopes. The `info commands` command reports only the procedures visible in the scope where the command is evaluated.

8.7 BOTTOM LINE

- Tcl namespaces can be used to hide procedures and data from the global scope.
- One namespace can be nested within another.
- The `namespace` commands manipulate the namespaces.
- The `namespace eval` command evaluates its arguments within a particular namespace.
  Syntax: `namespace eval` *`namespaceID arg ?args?`*
- The `namespace export` command makes its arguments visible outside the current namespace scope.
  Syntax: `namespace export` *`pattern ?patterns?`*

- The `namespace import` will make the procedures within a namespace local to the current namespace scope.
  Syntax: `namespace import ?-force? ?pattern?`
- The `namespace children` command will list the children of a given namespace.
  Syntax: `namespace children ?scope? ?pattern?`
- The `namespace ensemble` command allows scripts to be invoked as *`namespace command`* instead of *`namespace::command`*.
  Syntax: `namespace ensemble create -map {cmdName script}`
- The `variable` command declares a variable to be static within a namespace. The variable is not destroyed when procedures in the namespace scope complete processing.
  Syntax: `variable varName ?value? ... ?varNameN? ?valueN?`
- Tcl libraries (packages) can be built from one or more files of Tcl scripts.
- A module is a package that exists in a single file with a filename that conforms to the pattern *`NAME-REVISION.tm`*.
- The `package` commands provide support for declaring what revision level of a package is provided and what revision level is required.
- The `package` commands provide methods for manipulating packages.
- The `package provide` command declares the name of a package being defined in a source file.
  Syntax: `package provide packageName ?version?`
- The `pkg_mkIndex` command creates the `pkgIndex.tcl` file that lists the procedures defined in appropriate files.
  Syntax: `pkg_mkIndex dir pattern ?pattern?`
- The `package requires` command declares what packages a script may require.
  Syntax: `package require ?-exact? packageName ?versionNum?`

8.8 PROBLEMS

The following numbering convention is used in all Problem sections:

Number Range	Description of Problems
100–199	Short comprehension problems review material covered in the chapter. They can be answered in a few words or a 1–5-line script. These problems should each take under a minute to answer.
200–299	These quick exercises require some thought or information beyond that covered in the chapter. They may require reading a man page or making a web search. A short script of 1–50 lines should fulfill the exercises, which may take 10–20 minutes each to complete.
300–399	Long exercises may require reading other material or writing a few hundred lines of code. These exercises may take several hours to complete.

- *100* What Tcl commands provide encapsulation functionality?
- *101* What Tcl commands can be used to build an index of available procedures?

- *102* What Tcl commands support building modular programs?
- *103* Can a Tcl namespace be used in a package?
- *104* Can a Tcl namespace be nested within another namespace?
- *105* What Tcl command is used to create a new namespace?
- *106* Can more than one file be included in a package?
- *107* What Tcl command will build a package index file?
- *108* Can procedure names be imported from one namespace to another?
- *109* Can a directory contain files that define multiple packages?
- *110* Can a directory contain files that comprise multiple versions of a package?
- *111* Can more than one file be included in a module?
- *200* A FIFO queue can be implemented with a Tcl list. Create a `queue` namespace in which the Tcl list is a namespace variable. Implement `push`, `pop`, `peek`, and `size` procedures within the namespace.
- *201* Make a queue package from the script developed in problem 200. Use the `namespace ensemble` command to allow the methods to be invoked as `queue push`, etc. instead of `queue::push`.
- *202* Given this code fragment:

```
namespace eval pizza {
variable toppingList
variable size medium
variable style deep-dish
}
```

  Add procedures to
  a. Add toppings to a pizza.
  b. Set a pizza size.
  c. Set a pizza style.
  d. Report the size, style, and toppings on a pizza.
  e. Report the price of a pizza. (Define a base price and a price per topping. Ignore style and size.)
- *203* Using the queue package from exercise 201, write a simulation of a single-queue check-out counter (such as a small grocery store). Each element pushed onto the queue will be an integer representing the time that this customer takes to be serviced. At each time interval, there will be a random chance that a new customer will be added to the queue. A good set of values for testing is
  a. Shortest Time: 2 sec
  b. Longest Time: 10 sec
  c. Customer Probability: .1
  Use a `for` loop to iterate through 1000 iterations, with each iteration representing 1 second of time in the simulation. At each iteration, record the size of the queue. Report the number of customers and the queue sizes.
- *204* Expand the check-out counter simulation from exercise 203 to report how long each customer stands in line. You can do this by adding a second queue for customer wait times. Push the iteration when a customer is created onto the wait queue, and pop it when a customer is popped off the customer queue.

- *301* Expand the check-out counter simulation in exercise 204 to use 3 queues. Assume that a new customer always goes to the shortest queue.
- *302* Modify the check-out counter simulation in exercise 301 to use 3 servers, but a single queue.
- *303* Collect output from running the simulations built in exercise 301 and 302 and compare the wait time for N servers with N queues vs. N servers and 1 queue.
- *304* Using the Tower of Hanoi board described in Section 8.1.7, write a script that will solve the puzzle. Information about the Tower of Hanoi puzzle is available at:
  www.dcs.napier.ac.uk/a.cumming/hanoi/
  and *www.cut-the-knot.com/recurrence/hanoi.shtml*.

CHAPTER

Basic Object-Oriented Programming in Tcl

9

Over the years, Tcl has suffered from a surplus of object-oriented programming systems without ever having an object-oriented system integrated into the language. With Tcl8.6, the TclOO package has become part of the core language. TclOO can be used with Tcl8.5 as a loadable package.

A few of the object-oriented packages that led to TclOO include:

- `[incr tcl]`

  This was the first object-oriented system for Tcl. Like C++, it has been a standard for many years. The `[incr tcl]` extension duplicates the C++ model of classes with single and multiple inheritance, friend classes, private, protected and public methods and both class and object variables.

  This is the most popular OO extension of Tcl.

  Initially, `[incr tcl]` required changes to the Tcl kernel. After the addition of the *namespace* command, it became a loadable Tcl extension. The latest version of `[incr tcl]` is built using the 8.6 TclOO package which this chapter introduces.
- SNIT

  Snit (*Snit Is not Incr Tcl*) uses composition and delegation to define class relationships instead of the inheritance model used by C++ and Java. It is a pure-Tcl OO system (no compiled modules) and has been used to develop some megawidget libraries.
- MIT Otcl

  A dynamic OO model that uses concepts from CLOS, Smalltalk and Self.
- XOTcl

  The MIT Otcl project was extended and is now maintained as XOTcl (eXtended Otcl). XOTcl has a very rich set of functionality. It supports dynamic object-oriented programming with super classes for inheritance and mixins (at both class and object level) to further define sets of functionality. It supports assertions and filters to control code execution as well as other programming constructs.
- TclOO

  TclOO was developed by Donal Fellows as a bare-bones OO system that could be used within Tcl to construct other OO systems.

  TclOO differs from other object-oriented packages in several ways:

  - TclOO is not intended to be a complete OO programming language. It is intended to be a base set of OO functionality that can be used to build a full-featured OO language. Despite this, it is very powerful, and useful for complex, real-world OO projects.

Tcl/Tk: A Developer's Guide. DOI: 10.1016/B978-0-12-384717-1.00009-9

- TclOO was designed with support for the Tcl concepts of dynamic programming and introspection built in. Many of the features developed for XOTcl were incorporated into TclOO.
- TclOO is part of the Tcl core, not an add-on package. It is always available to an application without requiring that a package be installed.

This chapter will discuss how to use TclOO as a basic OO programming environment. The next chapter will discuss implementing more advanced features including class variables, class methods, introspection, runtime modifications of classes and individual object modification.

9.1 CREATING A TCLOO CLASS AND OBJECT

The TclOO commands are contained within the `::oo::` namespace. This namespace contains several commands. Basic OO style programming only uses the `oo::class` command, which is discussed in this chapter. The next chapters will cover the other `oo::` commands.

The `::oo::class create` command is used to create a new class. When a new class is created a new command with the same name as the class is also created to interact with the class. This is the same mechanism used by Tk to create commands to interact with widgets.

Syntax: `::oo::class create` *`name script`*

Define a new class

`name`	The name of the class being defined
`script`	A Tcl script using TclOO commands to create a new class

The script that defines a class is a normal Tcl script with the addition of a few commands that are specific to the class definition. The specific commands include:

- `method`

  The `method` command creates a procedure (method) that can be evaluated within an object's scope.

  Syntax: `method` *`name arguments body`*

  Define an object method

`name`	The name of the method being defined
`arguments`	The argument list for the method. All the usual procedure argument formats (default values, multiple arguments) are acceptable.
`body`	A Tcl script to be evaluated when this method is invoked

- `constructor`

  This creates a constructor for the class. This procedure will be called automatically whenever an object of this class is created. This is the place to do initializations, define super classes, etc. The arguments to the constructor can be defined in all the styles supported by the `proc` command.

Syntax: `constructor` *`arguments body`*

Initialize an object

`arguments` The argument list for the method. All the usual procedure argument formats (default values, multiple arguments) are acceptable.

`body` A Tcl script to be evaluated when this method is invoked

- `destructor`

  The destructor method is invoked whenever an object is destroyed. If a class constructor allocates a scarce resource (for instance, a channel), it should be released in the destructor. Inherited objects will be destroyed automatically, but any aggregation objects created by the constructor or other methods should be destroyed in the destructor.

  Syntax: `destructor` *`body`*

  Cleanup when an object is destroyed.

  `body` A Tcl script to be evaluated when this object is destroyed.

- `variable`

  This defines an object variable. The TclOO `variable` command differs from the `namespace variable` command in these ways:

  - Variables within a class definition are not initialized using the `variable` command. They are initialized within the constructor.
  - The `variable` command can accept multiple variables on a single line, similar to the `global` command.
  - In the 8.6 version of TclOO a `variable` command will replace any variables defined by a previous `variable` command with the set of variables defined by the current `variable` command. This behavior is different from the `namespace variable` or `global` commands that simply add more variables to the list of variables. This behavior may change by the time 8.6 is out of beta.

  Syntax: `variable` *`variableName1 ?variableName2 ...?`*

  Declares the variables for an object.

  `variableName...` The list of variables.

- `filter`

  A filter method is evaluated before other object methods. A filter can be used to perform data validation, add debugging output, or other tasks that would otherwise need to be added to each method. A filter passes control to the originally requested method with the `next` command which will be discussed in more detail in the *Method Chaining* section. Evaluating the `next` command is similar to invoking a procedure. Control is passed to another method and then returns to the command after the `next` command.

  Any method in a class can be added to the filter stack. When an object's method is invoked, the filter methods are invoked first. At each step, a filter method may evaluate a `return`, breaking the chain, or evaluate a `next` command, passing control to the next method in the chain. Evaluating the `next` command in the last method of a chain of filters passes control to the method that was originally requested.

The `next` command can be evaluated at any point in a filter method. For example, a filter might validate inputs and then pass control to the requested method with the `next` command, or it could pass control to the requested method first and then modify the returned values.

Syntax: `filter` *`methodName`*

Adds a method to the list of filters.

`methodName` Name of a method to use as a filter.

When a class is created with the `oo::class create` command, the Tcl interpreter creates a new command with the same name as the class. This new command has three subcommands:

`create` Create a new object with a given name. The name of the new object is returned.

`new` Create a new object with a computer-generated name. The name of the new object is returned.

`destroy` Destroy the class and all objects of this class.

Class methods can be invoked as

`objectName methodName arguments`

The following example defines a stack datastructure class. The stack is implemented as a Tcl list contained in an object variable named `stackVar` with two methods to access the list. The `push` method appends a new element to the end of the stack, and the `pop` method returns the last element from the stack.

This script performs these actions:

- a variable is created with the `variable` command.
- a `push` method is created.
- a `pop` method is created.

Example 1

Creating a Class

```
# Define the stack class
#   stackVar:  Contains the list of items in the stack

::oo::class create stack {
  variable stackVar   ;# stackVar is a per–object variable.

  # The method to push data onto the top of a stack
  #
  # Arguments
  #   value     A value to push onto the stack

  method push {value} {
    lappend stackVar $value
  }

  # The method to pop data off the top of a stack
```

```
    method pop {} {
      set rtn [lindex $stackVar end]
      set stackVar [lrange $stackVar 0 end-1]
      return $rtn
    }
}
```

The next example shows how the `stack` class can be used. A new `stack` object named `st` is created, two strings are pushed onto the stack, and the last value onto the stack is popped off and displayed.

Example 2

Using a Class

```
# Create a new object named 'st'
stack create st

# Push values onto the 'st' stack
st push "value 1"
st push "value 2"

# Pop and display
puts [st pop]
```

Script Output

```
value 2
```

9.1.1 Constructor and Destructor

Many classes need some initialization when they are created. Any class that creates a resource in the constructor will probably need a destructor to release the resource when the class is destroyed. The resource could be an I/O channel, memory resources or other TclOO objects.

Constructors and destructors are created in the class definition script. The definition of a constructor or destructor is similar to a procedure definition. A class may only have a single constructor or destructor. The constructor and destructor syntax resembles the `method` syntax except that no name is defined for the constructor or destructor.

Syntax: `constructor` *`arguments body`*

Constructor initializes an object when it is created.

`arguments`	The argument list for the method. All the usual procedure argument formats (default values, multiple arguments) are acceptable.
`body`	A Tcl script to be evaluated when this method is invoked

Syntax: `destructor` *`body`*

Destructor cleans up when an object is destroyed.

`body` A Tcl script to be evaluated when this method is invoked

This example shows how to create a class that writes to a file. The constructor accepts a filename to open, the `write` method sends data to the file, and the destructor will close the file.

Example 3
A Class with Constructor and Destructor

```
oo::class create logFile {
  variable channel

  # invoked when an object is created
  # Arguments
  #   name      The pathname of the log file to open
  constructor {name} {
    set channel [open $name w]
  }

  # invoked when an object is destroyed
  destructor {
    close $channel
  }

  # A method to send data to a channel
  # Arguments
  #   data      The text to write to the log file.
  method write {data} {
    puts $channel $data
  }
}
```

The next example shows how to use this class. When you create an object with a defined name (using the `create` subcommand) you provide the constructor arguments after the name of the new object, as is done with the Captain's Log. When you create an object with an automatically generated name (using the `new` comamand), you only provide the constructor arguments, as is shown for the medical log.

Example 4
Using a Class with Constructor and Destructor

```
logFile create captLog ./captLog.txt
captLog write "Captain's Log,Stardate [clock seconds]"
```

```
captLog write "I wish something exciting would happen."
captLog destroy

set medicalLog [logFile new ./medLog.txt]
$medicalLog write "The captain is showing signs of boredom"
$medicalLog write "Initiating Kobyashi Maru scenario."
$medicalLog destroy
```

9.1.2 Methods

A method is similar to a procedure with a few differences:

- The method names are subcommands and do not show up with `info commands` or `info procs`. The name of a TclOO object is a command and will be displayed by `info commands`. Classes and objects can be examined with the `info class` and `info object` commands, which are discussed in the next chapter.
- A method has access to all the variables associated with the object as well as global scope variables.

TclOO methods behave similar to procedures in these respects:

- A TclOO class may have as many methods as it needs. The only limit is system resources.
- An object's method may be registered as the callback script to be evaluated when an event occurs via the `after` or `fileevent` commands, or as a widget callback.
- An object may invoke its own methods, or the methods from other objects within a method.

Method Naming Convention

Any word that is valid for a Tcl command or procedure name can be used within a class as a method name. You can even re-use global scope command and procedure names as method names. Re-using command or procedure names as method names is not recommended—it will work, but is prone to unexplained bugs if you accidentally invoke the command instead of the method.

The *`Tcl Engineering Guide`* recommends that procedures within a namespace use a naming convention in which procedures that start with a lower-case letter are part of the public API and may be exported, while names that start with an upper-case letter are for internal use only.

The TclOO package adheres to this convention.

The next example shows methods defined with one publicly available method (`show`) and one internal method (`HiddenShow`).

Notice that the error message reports that there are two available methods, `show` and `destroy`. TclOO automatically creates a `destroy` method for all objects.

Example 5

Defining and Using Methods

```
oo::class create methods {
  variable var1 var2
```

```
  constructor {val1 val2} {
    set var1 $val1
    set var2 $val2
  }

  method show {} {
    puts "Show: VAR1: $var1 VAR2: $var2"
  }

  method HiddenShow {} {
    puts "HiddenShow: VAR1: $var1 VAR2: $var2"
  }
}
methods create m 1 2
m show
m HiddenShow
```

Script Output

```
Show: VAR1: 1 VAR2: 2
unknown method "HiddenShow": must be destroy or show
```

Invoking Methods from Within Methods

Any existing command, procedure or object method can be evaluated within the body of a method. As you would expect, if you attempt to evaluate a procedure or an object method that has not yet been defined, the Tcl interpreter will generate an error.

Note that methods are not commands. When an object is instantiated, the object name is used as the name of a command, and the methods are subcommands within that command. This is expanded from the `namespace ensemble` command discussed in Chapter 8.

You can invoke methods for other objects with the name of the object and the method name, just as you would evaluate an object's method from the global or a procedure scope.

In order to evaluate a method within the current object, TclOO provides the virtual command `my`. Each object has a `my` command that is the current object.

In the next example a class is created with three methods to show evaluating the objects methods.

`showValue`	Displays the value of the object variable.
`external`	Invokes the `showValue` method of another object.
`internal`	Invokes the `showValue` method of the current object.

Example 6

Evaluating Object Methods

```
oo::class create hasMethods {
  variable value
  constructor {val} {
    set value $val
```

```
  }
  method showValue {} {
    puts "Value is: $value"
  }
  method external {objName} {
    $objName showValue
  }
  method internal {} {
    my showValue
  }
}

set ob1 [hasMethods new 1]
set ob2 [hasMethods new 2]

$ob1 external $ob2
$ob1 internal
```

Script Output

```
Value is: 2
Value is: 1
```

If your code tries to invoke a method with just the method name, it will either fail (because the command is undefined) or it will evaluate a command or procedure defined in the global scope. It is good policy to not overload the names of global scope commands within a class.

The next example shows the unexpected consequences of overusing a command that already exists in the Tcl interpreter.

Example 7

Evaluating Object Methods

```
oo::class create methodDemo {
  variable a b c
  constructor {} {
    set a 1
    set b 2
    set c 3
  }
  method puts {var} {
    puts "THE VALUE of $var IS: [set $var]"
  }
  method bad_showContents {} {
    puts a
    puts b
    puts c
  }
```

```
  method good_showContents {} {
    my puts a
    my puts b
    my puts c
  }
}
methodDemo create demo
puts " BAD:"
demo bad_showContents
puts "GOOD:"
demo good_showContents
```

Script Output

```
 BAD:
a
b
c
 GOOD:
THE VALUE of a IS: 1
THE VALUE of b IS: 2
THE VALUE of c IS: 3
```

Registering Methods for Callbacks

Several Tcl/Tk mechanisms are implemented via a callback. Some examples are the `after` command, `fileevents` and commands associated with a Tk button. These commands need to include the object as well as the method in order to be evaluated in the correct scope.

The Tcl namespace package includes the `namespace current` command in order to register a namespace procedure with a callback. The `namespace current` command returns the current namespace, which can be combined with a procedure name to provide a complete path to the procedure.

TclOO has a `self` command that returns the name of the current object. This name can be combined with the method name and any arguments to register a callback.

This example shows the `after` command being used both inside and outside an object. Notice that the `after10` method uses the `self` command while the global scope uses the name of the object. If you try using this code in an interactive test, run it inside the wish shell, rather than tclsh, in order for the event loop to be evaluated.

Example 8

Registering Object Methods as Callbacks

```
oo::class create delayed {
  variable x
  constructor {val} {
    set x $val
  }
  method show {} {
```

```
    puts "SHOW: x is $x at [clock seconds]"
  }
  method after10 {} {
    after 10000 [list [self] show]
  }
}

set a [delayed new 2]
set b [delayed new 4]
$a after10
after 5000 [list $b show]
```

Script Output

```
SHOW: x is 4 at 1295761965
SHOW: x is 2 at 1295761970
```

Your code may use either `my` or `[self]` when invoking object methods from within an object method. Only `[self]` can be used when registering a callback, since the `my` command only exists within the scope where code is currently being evaluated.

9.1.3 Inheritance

One of the driving forces in object-oriented design and programming is the ability to re-use and extend code. The term for adding new functionality from another class is *inheritance*. The many object oriented languages all have slightly different models for inheriting variables and methods from other classes.

TclOO supports single inheritance, multiple inheritance and mixins as methods for merging the functionality of multiple classes into a single class. There are different reasons for using each style of inheritance, and slightly different behaviors.

Single inheritance is suitable when you have several classes that will share some functionality but have additional parameters or methods that must be implemented differently. For example, a business might have a base class *personnel* with derived classes for salaried staff and hourly staff.

Multiple inheritance is suitable when the derived classes have permanent parameters and methods from other classes. In the business example, an hourly union worker might inherit from both the hourly worker and union classes.

Mixins are frequently a better choice than multiple inheritance. A mixin is most appropriate when a class has special features that are orthogonal to the inheritance lineage. For instance, a photo of the employee is not related to employee's paycheck and might be part of a biographical mixin.

The next chapter will discuss how mixins can be added on a per-object basis, which makes it easy to build custom objects without needing a large, pre-defined class hierarchy that covers all possible combinations. Using per-object mixins we could add individual employee skills, adding welding methods to one employee and accounting methods to another.

Method Chaining

In order to create a uniform API, multiple classes will need to use the same name for a method. For example, a method in a derived class might pre-process data into a standard form and then pass control to a method with the same name in a superclass.

It's quite common for the same method name to be used in a class, one or more superclasses, and even one or more mixins. This can create a problem in determining which methods will be invoked and which order they'll be invoked in.

TclOO constructs a *method chain* to define the order in which methods will be invoked. Within a method, control can be transferred to the next method in the stack with the `next` command.

Syntax: `next` *`args`*

Invoke the next method in a hierarchy.

`args` a list of arguments to be passed to the next method.

For simple class hierarchies the method chain is built in the order shown below. This will be expanded in the next chapter after more complex hierarchies are discussed. If a method can appear in the chain more than once (perhaps a class has been included both as a super and a mixin), it will be added at the latest valid position.

Filter Methods If multiple filters are associated with a class, they appear in the order of the class hierarchy.

Mixin Methods A method from a mixin is evaluated next. If there are multiple mixins, they are added in the order they are defined in the list of mixins.

Class Methods A method defined in the class description text is evaluated next.

Superclass Methods Methods from the superclasses are added in the order that the superclasses were defined. If the superclasses include more superclasses, those methods will be added before proceeding to the next superclass.

The `next` command can define the arguments being passed to the next method. It always invokes the next method in the method chain. It can't skip methods and cannot specify which overridden method to invoke (the next chapters will discuss doing tricky stuff with the method chain.)

The return from `next` is whatever value the first method in the chain returns. If the return value from an interior method needs to be returned, the code will have to propagate that value.

The `next` command can be used at any point in a method, filter, constructor or destructor. This gives the script author the ability to both rework arguments before passing them to a superclass and rework results before they are returned from the object.

Inheritance

The next few examples will develop a small class hierarchy to define a character in a fantasy game. A character in a fantasy game has a name and a few attributes like how well it can defend itself, how well it attacks and how many life points it has.

This base class can be modified in several ways. The values of the parameters can be changed permanently based on the character's race (human, elf, dwarf, etc.) and occupation (knight, wizard, healer, etc.). The character's abilities can also be modified temporarily when the character uses artifacts such as magic armor and weapons.

Most of the methods in these classes have `puts` commands to make it easy to show the call hierarchy as they are invoked.

Example 9

Base Character Class

```
# Define a base character class with the default values
# for defensive strength, attack strength and life points

oo::class create character {
  variable state

  # Constructor assigns base values

  constructor {name} {
    puts "character constructor"
    array set state {defense 2 attack 3 hitpoints 5}
    set State{name} $name
  }

  # Display the values for debugging

  method show {} {
    parray State
  }

  # Return whether an attack is larger than the character's
  # defensive ability

  method defense {attackStrength} {
    puts "Final Attack is: $attackStrength"
    if {$attackStrength > $State(defense)} {
      return " $attackStrength is larger than $State(defense),\
           $State(name) is Hit"
    } else {
      return " $attackStrength is not larger than $State(defense),\
          $State(name) is Missed"
   }
  }
}
```

Single Inheritance

Single inheritance is the simplest of the inheritance models. This is used when you have a class that has basic characteristics and you need to create more classes that have all the basic characteristics and some specific characteristics.

One way of defining characters in a Fantasy boardgame is to start with a base set of parameters, and then modify them depending on the character's class (Warrior, Mage, etc.)

The next example uses the `character` class as a basis for a warrior class. A `warrior` character has higher attack and defense skills, so these values are modified in the constructor.

In the `warrior` constructor, the attack and defense are modified after the `next` command is invoked. This allows the `character` class to set the initial values before they are changed.

Also notice that the `name` parameter must be provided to the `character` class using the `next` command, since the name is assigned in that class.

Example 10

Single Inheritance

```
source character.tcl

# Define a warrior that has better attack and defense
# skills because of its class

oo::class create warrior {
  superclass character
  variable State
  constructor {name} {
    puts "warrior constructor"
    next $name
    incr State(defense) 2
    incr State(attack) 2
  }
}
    puts "Create object"
    warrior create elmer siegfried

    puts ""
    puts "Show object"
    elmer show

    puts ""
    puts "Attack value 8 against a warrior"
    puts [elmer defense 8]
```

Script Output

```
Create object
Warrior constructor
Character constructor

Show object
State(attack)    = 4
State(defense)   = 4
State(hitpoints) = 5
```

```
State(name)      = Siegfried

Attack value 8 against a warrior
Final Attack is: 8
 8 is larger than 4, Siegfried is Hit
```

Multiple Inheritance

Some class hierarchies will require that a class inherit characteristics from more than one parent class. This can be done by using multiple parent classes as arguments to the `superclass`. Do not use multiple `superclass` commands—as with the TclOO variable command, the latter commands overwrite the previous ones. (This behavior may change when Tcl8.6 is fully released.)

Multiple inheritance should only be used if there is a reason in the class logic to use multiple inheritance. With languages like C++, multiple inheritance is used to add functionality that might not be part of the logical class relationships. For example, a C++ class might use multiple inheritance to include both calculation methods and GUI graphing methods. In TclOO, adding features that are orthogonal to the class logic should be done with the `mixin` command, which will be discussed in the next section.

A fantasy character can have characteristics modified based on both race and occupation. For this example, our hero has better defense and attack values because he is a warrior, and more hitpoints because he is a human.

In this example, each constructor displays when it is invoked. The constructors are placed in the call tree in the order that they appear in the `super` command. The `next` command steps to the next procedure in the call tree.

Example 11

Multiple Inheritance

```
source character.tcl

oo::class create warrior {
  variable State
  constructor {name} {
    puts "Warrior constructor"
    next $name
    incr State(defense) 2
    incr State(attack) 2
  }
}

oo::class create human {
  variable State
  constructor {name} {
    puts "Human constructor"
    next $name
```

```
    incr State(hitpoints) 2
  }
}

oo::class create humanwarrior {
  superclass human warrior character
  variable State
  constructor {name} {
    puts "Human Warrior constructor"
    next $name
  }
}

puts "Creating a human warrior"
humanwarrior create elmer Sigfried

puts ""
puts "The character's attributes are:"
elmer show

puts ""
puts "Attack value 8 with no armor"
puts [elmer defense 8]
```

Script Output

```
Creating a human warrior
Human Warrior constructor
Human constructor
Warrior constructor
Character constructor

The character's attributes are:
State(attack)    = 4
State(defense)   = 4
State(hitpoints) = 7
State(name)      = Sigfried

Attack value 8 with no armor
Final Attack is: 8
 8 is larger than 4, Sigfried is Hit
```

Using Mixins

Mixins are useful for parameters and methods that are orthogonal to the class logic or are likely to change. For instance, a mixin is a better choice than multiple inheritance for adding a GUI to a class that does data analysis.

Our fantasy hero will need weapons and armor. These could be added by having the class inherit a dagger and shield, but possessions are different from base attributes like strength. Using inheritance for this makes the logic of defining the character muddy.

The next example shows how we can use mixins to add a dagger and shield. For this example, we're only worrying about our hero's ability to defend himself. A real class would include an attack method as well.

One feature of mixins is that they are designed to be added and removed at creation time or runtime. The next chapter will show how to handle the character finding better armor and weapons.

Example 12

Multiple Inheritance

```
source character.tcl

oo::class create warrior {
  variable State
  constructor {name} {
    puts "Warrior constructor"
    next $name
    incr State(defense) 2
    incr State(attack) 2
  }
}

oo::class create human {
  variable State
  constructor {name} {
    puts "Human constructor"
    next $name
    incr State(hitpoints) 2
  }
}

oo::class create shield {
  method defense {attackStrength} {
    puts "Shield reduces attack by 2"
    return [next [expr {$attackStrength - 2}]]
  }
}

oo::class create dagger {
  method defense {attackStrength} {
    puts "Dagger reduces attack by 1"
    return [next [expr {$attackStrength - 1}]]
  }
}
```

```
oo::class create humanwarrior {
  superclass human warrior character
  mixin shield dagger
  variable State
  constructor {name} {
    puts "Human Warrior constructor"
    next $name
  }
}

humanwarrior create elmer Sigfried

puts ""
puts "The characters attributes are:"
elmer show

puts ""
puts "Attack value 8 against human warrior with dagger and shield"
puts [elmer defense 8]
```

Script Output

```
Human Warrior constructor
Human constructor
Warrior constructor
Character constructor

The characters attributes are:
State(attack)    = 4
State(defense)   = 4
State(hitpoints) = 7
State(name)      = Sigfried

Attack value 8 against human warrior with dagger and shield
Shield reduces attack by 2
Dagger reduces attack by 1
Final Attack is: 5
 5 is larger than 4, Sigfried is Hit
```

Aggregation

The superclass and mixin define the *is-a* relationship that objects can have—a warrior *is a* character. An object can also have a *has-a* relationship to other objects. For instance, our character may have one or more treasures.

The *has-a* relationship can be implemented as a Tcl list of object identifiers. Object methods can be invoked by methods in other objects just as the methods are invoked from mainline code.

The next example shows a character with an `acquire` method. The character's possessions are a list of treasure objects (`oo::Obj6` and `oo::Obj7`). The `acquire` method simply appends the new object to the list of possessions.

The `netWorth` method steps through the objects, invokes each object's `getValue` method, and calculates the total value of the treasure.

Example 13
Multiple Inheritance

```
# Define the character class
oo::class create character {
  variable State

  # This character has a name and a list of possessions
  constructor {nm} {
    set State(name) $nm
    set State(possessions) {}
  }

  # Display the contents of the State variable
  method show {} {
    parray State
  }

  # Acquire a new item by appending it to
  # the list of possessions
  method acquire {item} {
    lappend State(possessions) $item
  }

  # Return the total value of the treasures
  method netWorth {} {
    set total 0
    foreach item $State(possessions) {
      incr total [$item getValue]
    }
    return $total
  }
}

# Define a treasure class
oo::class create treasure {

  # Treasures have a name and a value
  variable name value

  constructor {nm val} {
```

```
    set name $nm
    set value $val
  }

  # Return the value of this treasure
  method getValue {} {
    return $value
  }
}

# Create a character and create two treasures
# that the character has acquired.
character create daffy Allmine
daffy acquire [treasure new ruby 100]
daffy acquire [treasure new diamond 200]

puts "Daffy's State is:"
daffy show
puts "Allmine has treasure worth: [daffy netWorth]"
```

Script Output

```
Daffy's State is:
State(name)        = Allmine
State(possessions) = ::oo::Obj6 ::oo::Obj7
Allmine has treasure worth: 300
```

9.1.4 Filters

There are some actions that need to be evaluated in each of a class's methods. These actions may include input validation, logging, displaying debugging information, etc. The `filter` command lets you define one or more methods to be called before evaluating an object's methods. The filter can either abort processing with a `return` command, or pass control to the next method with the `next` command.

A filter is invoked before a method or destructor is evaluated, but is not evaluated before a constructor.

A filter is defined as part of the `oo::class create` command, similar to the way that superclasses and variables are defined.

Syntax: `filter` *`methodName`*

Adds a method to the list of filters.

`methodName` Name of a method to use as a filter.

The next example shows how to use a filter to test inputs before calling an object's methods. The `rectangle` class has no constructor, just a method to calculate a rectangle's area and the filter. The filter will abort processing and return an area of 0 if the height or width of a rectangle are less than 0.

Example 14

Filter to Test Inputs

```
oo::class create rectangle {
  filter isPositive

  # Check that height and width are both positive.
  method isPositive {ht wd} {
    if {($ht < 0) || ($wd < 0)} {
      return 0
    } else {
      next $ht $wd
    }
  }

  # Calculate the area of a rectangle of given height and width.
  method area {ht wd} {
    return [expr {$ht * $wd}]
  }
}

rectangle create r1
puts "Area of a 2 x 3 rectangle is: [r1 area 2 3]"
puts "Area of a 2 x -3 rectangle is: [r1 area 2 -3]"
```

Script Output

```
Area of a 2 x 3 rectangle is: 6
Area of a 2 x -3 rectangle is: 0
```

A filter method does not need to be a defined part of a class. The method may be defined in a mixin or as part of a superclass. When debugging a problem, sometimes it's useful to display information whenever any method is invoked.

The next example is a small `debug` class with a single method to display the object name, method being evaluated and the arguments. The `showCall` method uses the `info` command to display the contents. The filter is being evaluated as part of invoking the object's method, so the `info level` command uses a relative position of 0 in the procedure stack to show the information.

Example 15

Filter to Provide Debug Info

```
oo::class create debug {
  method showCall {args} {
    puts "--- Debugging: [info level 0]"
    next {*}$args
  }
}
```

The `debug` class can be mixed into another class to provide debugging information while the code is being developed, and then removed once the code is perfect.

The next example shows the previous `humanwarrior` class with the debugging information added. Whenever one of the `elmer` methods is invoked, the `showCall` method displays the object, method name and arguments.

Example 16
Using a Mixin Filter

```
source character.tcl
source debug.tcl

oo::class create warrior {
  # Mix in the debug class and add the filter
  mixin debug
  filter showCall

  variable State
  constructor {name} {
    puts "Warrior constructor"
    next $name
    incr State(defense) 2
    incr State(attack) 2
  }
}

oo::class create human {
  variable State
  constructor {name} {
    puts "Human constructor"
    next $name
    incr State(hitpoints) 2
  }
}

oo::class create humanwarrior {
  superclass human warrior character
  variable State
  constructor {name} {
    puts "Human Warrior constructor"
    next $name
  }
}

humanwarrior create elmer Sigfried

elmer show
```

```
puts "Attack value 8 against human warrior"
puts [elmer defense 8]
```

Script Output

```
Human Warrior constructor
Human constructor
Warrior constructor
Character constructor
—— Debugging: elmer show
State(attack)    = 4
State(defense)   = 4
State(hitpoints) = 7
State(name)      = Sigfried
Attack value 8 against human warrior
—— Debugging: elmer defense 8
Final Attack is: 8
 8 is larger than 4, Sigfried is Hit
```

Using the filter for debugging like this is better than adding a lot of code to each method to track a class's behavior, but it would be nicer if this class could be mixed in at run time, or on a single object. These techniques will be discussed in the next chapter.

9.2 BOTTOM LINE

- Tcl supports several object systems. The most popular have been `[incr tcl]`, XOTcl and SNIT.
- The TclOO package is built in with Tcl 8.6 and newer.
- The TclOO package is available as a loadable package for Tcl 8.5.
- The TclOO package is designed to be a minimal OO package that can also be used as a basis for creating more complex OO packages.
- TclOO supports single inheritance, multiple inheritance, mixins and method chaining.
- The TclOO commands are created in the `::oo::` namespace.
- A new class is defined with the `::oo::class create` command.

  Syntax: `::oo::class create` *`name script`*

  Define a new class

  `name` The name of the class being defined

  `script` A Tcl script using TclOO commands to create a new class

- A class may contain a constructor, a destructor and multiple variables, methods, superclasses and mixin classes.
- The `next` command passes control to the next method in a method chain.
- A filter method will be invoked before other object methods. The filter may abort processing the method or pass control to other methods in the call chain. The filter may modify inputs before passing control to the next layer and may modify return values before returning control to the calling code.

9.3 PROBLEMS

The following numbering convention is used in all Problem sections:

Number Range	Description of Problems
100–199	Short comprehension problems review material covered in the chapter. They can be answered in a few words or a 1–5-line script. These problems should each take under a minute to answer.
200–299	These quick exercises require some thought or information beyond that covered in the chapter. They may require reading a man page or making a web search. A short script of 1–50 lines should fulfill the exercises, which may take 10–20 minutes each to complete.
300–399	Long exercises may require reading other material or writing a few hundred lines of code. These exercises may take several hours to complete.

- *100* What Tcl command will define a new class?
- *101* How many constructors can a class possess?
- *102* Can a Tcl class have more than one method?
- *103* What does the `next` command return?
- *104* When is a constructor called?
- *105* What command is used to inherit characteristics from another class?
- *106* What command should be used to inherit methods that are not part of a class's logical construction (for instance, a set of debugging methods)?
- *107* What command can be added to a class definition to preprocess arguments before any method is evaluated?
- *200* Expand the `stack` class described in this chapter by adding methods for peek and size.
- *201* Define a card class for playing cards. The class should have a parameter for suit (clubs, spades, etc.), name (Ace, King, Queen, etc.) and value (Ace might be 14, King would be 13, Queen would be 12, etc.).
- *202* Add a method to the `card` class from exercise 201 to return a card's rank.
- *203* Add a method to the `card` class from exercise 202 to compare the current to the value of a card object passed as an argument. Using this method to compare a King and Ace might look like this:

```
    set card1 [card create Ace Spades]
    set card2 [card create King Diamonds]
    $card1 compare $card2
  Larger
```

```
    $card2 compare $card1
Smaller
```

- *204* A Pizza namespace was created in Chapter 8, exercise 202. Write a TclOO class with the same variables and methods.
- *300* Define a deck of cards class that inherits from the `stack` class in exercise 201 and the `card` class from 204. The `deck` class will need a destructor to destroy the cards when a deck is destroyed.
- *301* Add a `shuffle` method to the deck of cards class created for exercise 300.
- *302* Write an application to play a game of war against the computer. This will need a deck of cards class and four stack classes (one for the played cards and one for unplayed cards for both the user and the computer). Use the `compare` method from exercise 203 to determine which player wins each round and then push the cards onto the appropriate stack.
- *303* Add an `attack` method to the `humanwarrior` class described in this chapter. There will need to be attack methods in the base character class and the weapon class. The final value returned should be calculated from the character's base attack value, the character's weapon and a random value.
- *304* Add a method to calculate the damage inflicted by a weapon. This should be added to a weapon class.
- *305* Add a method to accept the damage inflicted by a weapon. This should be added to the character class.

CHAPTER

Advanced Object-Oriented Programming in Tcl

10

There are many styles of object programming languages, ranging from the dynamically configured Smalltalk-like languages to the rigidly defined Java and C++ style languages.

One goal of the TclOO package is to allow the programmer to use whatever features they need to develop their application and to provide object-oriented programming support in a form that's consistent with the rest of Tcl, being both dynamic and introspective.

TclOO can be used like C++ or Java with a set of classes and inheritance defined during a project's design phase. You can also use TclOO in a dynamic manner, with classes and objects being defined while an application is running. TclOO classes and objects can be modified at runtime: new methods can be added or modified, variables can be added or deleted, mixins can be added or removed, and even inheritance can be modified.

This is very different from the C++/Java view of OO programming as unchanging class structures and methods. The dynamic nature of the TclOO classes and objects gives the developer powerful tools to develop applications that run in changing environments.

The flip side of the dynamic nature of TclOO is the introspective nature. As with other Tcl applications, you can examine an object to discover its class, superclasses, mixins, variables, etc.

Tcl's introspection support is a different technique than the black-box design paradigm of the C++ and Java style languages. You can program using the paradigm that an object's state is hidden, or you can use the Tcl introspection tools to examine an object's state.

This is a powerful tool during debugging and also makes it easy to write generic functions to serialize a class or object in order to completely save and restore an application's state.

The dynamic and introspective nature of TclOO are powerful tools. Using these tools can make complex tasks easier and can also allow a careless programmer to write code that is difficult to maintain. When functionality can be created easily with traditional OO methods, it's best to stick to the minimal set of functionality. However, when the going gets tough, TclOO has the tools to deal with the unusual programming environments.

The commands that implement these features are:

`oo::define`	Define a feature for all members of a class.
`oo::objdefine`	Define a feature for a single object.
`info class`	Return information about a class.
`info object`	Return information about a single object.

The `oo::class` commands introduced in the previous chapter provide the basic inheritance and mixin styles of object-oriented programming. This chapter introduces more commands that provide the developer with more capabilities and dig deeper into the underlying structure of TclOO.

Tcl/Tk: A Developer's Guide. DOI: 10.1016/B978-0-12-384717-1.00010-5

10.1 MODIFYING CLASSES AND OBJECTS

The dynamic nature of TclOO means that a Tcl application can adapt to a changing environment without the developer needing to know all the possible permutations at design time.

We can use the ability to redefine a class's behavior to model real world systems better. For instance, a simulation of elevators in an office needs to define what an elevator does when it has no active request. The default behavior for an idle elevator might be to wait at the last floor it was sent to until there is a request from another floor. During the morning rush, you might want an idle elevator to return to the main floor to be immediately available for folks coming to work.

Rather than clutter an `idle` method with information about time of day, we can use the `oo::define` command to change the `idle` method depending on the time of day.

We can use the `oo::objdefine` command to modify the behavior of a single object, rather than all objects of a class. We could use this to make one elevator an express that only runs from the main floor to the floor where most people work during the morning and afternoon rush times.

10.1.1 Modifying Classes

The `oo::define` command defines features that will be common to all members of a class. Most class attributes are defined using the `oo::class` command. The `oo::class` command invokes the `oo::define` command to fulfill these parameter requests.

The `oo::define` command is also available at the script level, which allows an application to customize a class or define features like static class methods that are not supported by the `oo::class` command.

The `oo::define` command supports all of the features that can be defined in an `oo::class create` script. This provides the potential for reworking a class at runtime, as well as defining it during the design phase of a project.

When `oo::define` is used to change a feature of a class that has objects, all the existing objects immediately gain the new feature and all new objects are built with the new features.

Like other Tcl commands, the `oo::define` command supports several sub-commands. It also supports two argument conventions as shown in the next table.

- a single subcommand and required arguments.
- a script which might contain several subcommands to evaluate

Syntax: `oo::define` *`className script`*
`oo::define` *`className ?subcommand ?arg1 arg2 ...?`*
Define a feature of a class.

`className`	The name of the class to be modified.
`script`	A script with commands to modify the class. This may modify several features.
`subcommand`	A subcommand that defines a single feature to be modified.
`?arg1 arg2 ...?`	The arguments that a subcommand requires.

All of the commands to modify a class are supported in both the `script` and `subcommand` version of the `oo::define` command.

Modifying Methods

- Creating or modifying a method
  A new method can be added to an existing class or a method body can be changed with the `method` command. After this command is evaluated, all existing objects of this class as well as all newly created objects will contain the new method.

  Syntax: `oo::define` *`className`* `method` *`methodName arguments script`*

  Define a new method for all members of a class.

`methodName`	The name of the method to be added.
`arguments`	A list of method arguments, following the normal Tcl procedure argument format.
`script`	The body to be evaluated when the method is invoked.

```
# Create an addition class
oo::class create addition {
}

# Add a method to the addition class
oo::define addition method add {a b} {
  return [expr {$a + $b}]
}
```

- Adding a filter to a class
  A filter method is one that will be called before other class methods. This can be used to check inputs before passing control to the requested method with the `next` command or to modify results after the `next` command has been evaluated. A filter can also be used as a debugging tool to display information when methods are invoked or to capture method calls during early development when the methods are not yet implemented.

  A filter method can be used to redirect processing if a system has a systemic failure. For example, if a connection to a remote database server fails, a filter can be used to redirect calls to the database object methods to create a journal that can be used to update the database when the connectivity is restored.

  A permanent filter to check method inputs should be defined in the `oo::class create` definition. A temporary filter for use during development, debugging or to handle exception states can be defined with the `filter` subcommand.

  A filter must be a method in a class that's included in the class hierarchy. It can be a method in a mixin or inherited class. A filter cannot be a normal Tcl procedure. If you attempt to add an illegal method to the filter list it will not be added.

  Syntax: `oo::define` *`className`* `filter` *`methodName ?methodName?`*

  Add one or more methods to the list of filters for all members of a class.

`methodName`	The name of the method to be added to the filter list.

```
# Define a new method to show information about a method call
oo::define addition method show {args} {
    puts "--- [info level 0]"
    next {*}$args
}

# Show method information before all method calls
# to addition objects
oo::define addition filter show
```

- Forwarding one method to another
  In order to migrate code from one set of libraries to another it's sometimes easier to rework the library, rather than rework all the original code. You can use the `forward` subcommand to create an alias for a method by mapping one method name to another or even to a normal Tcl procedure.

  Forwarding a method is a one-way street. When a method call is forwarded the Tcl call stack is modified to show the location that has been forwarded to, not the original method call. There is no information available within the forwarded procedure to redirect evaluation back to the original method.

  Syntax: `oo::define` *`className`* `forward` *`newName actualCmd ?args?`*

  Forward an invocation of `newName` to `actualCmd` with optional arguments added before the arguments provided to `newName`.

`newName`	The name of a new method for this class.
`actualCmd`	The command to evaluate when the `newName` method is invoked.
`?args?`	Arguments which will be placed before any arguments in the invocation of `newName`.

```
# Add a forward to a global scope procedure
oo::define addition forward print puts

# Add a 'sum' method by re-using the 'add' method.
oo::define addition forward sum my add
```

- Renaming a method
  If you are using class libraries from several sources you may run into method name conflicts. TclOO creates a method chain of methods in superclasses and mixins by name. If multiple classes have methods with the same name all these methods will be added to the method chain. If you need to remove a method from the chain, you can rename it.

  Syntax: `oo::define` *`className`* `renamemethod` *`oldName newName`*

  Renames a method.

`oldName`	The current name for this method.
`newName`	The new name for this method.

```
# Rename the sum2 method to twice
```

```
oo::define addition renamemethod sum2 twice

puts "Renamed to twice: [addObj twice 1 2]\n"
```

Renaming a method will not modify code that relies on the old method name. This includes filters and forwards as well as code inside a method that invokes another method.

The example below will fail.

Example 1

Script Example

```
# Add a 'sum' method by re—using the 'add' method.
oo::define addition forward sum my add

# Rename the add method
oo::define addition renamemethod add oldAdd

addObj sum 3 4
```

Script Output

```
# Error: unknown method "add".
```

- Deleting a method

  If you have added a method for temporary use, you will need to delete it when the need is gone.

  The `deletemethod` subcommand removes a method from a class and all of that class's objects. If the method is referenced in other evaluation lists (`filter` or `forward`), it is removed from those lists also.

  Syntax: `oo::define` *`className`* `deletemethod` *`methodName ?methodName?`*

  Delete a method from all members of a class.

  `methodName` The name of the method to be deleted.

```
# Done with debugging now, delete filter method
oo::define addition deletemethod show
```

- Private and public methods

  The preferred coding style for using Tcl namespaces is to name public procedures with lowercase letters and private procedures with uppercase letters. This is just a convention with the Tcl namespace. TclOO enforces this convention.

  Like most restrictions in Tcl, we can work around it.

  The `export` and `unexport` commands can be used to export a method that starts with an uppercase letter, or to unexport one that starts with a lowercase letter. You can use these commands within a class definition as you would use `public` and `private` in C++ or Java, or you can modify the class at runtime with the `oo::define` command.

Both exported and unexported methods are available to derived classes and when a class with a private method is mixed into another class.

Syntax: `oo::define` *`className`* `export` *`methodName ?methodName?`*

Make a method available at the global scope.

`methodName` The name of the method to be made public.

Syntax: `oo::define` *`className`* `unexport` *`methodName ?methodName?`*

Make a method unavailable at the global scope.

`methodName` The name of the method to be made private.

Example 2

Script Example

```
oo::class create hasPrivateMethod {
  method PrivateMethod {args} {
    puts "PrivateMethod $args"
  }
}

oo::class create usesPrivateMethod {
  superclass hasPrivateMethod

  method usePrivateMethod {args} {
    my PrivateMethod "From usesPrivateMethod $args"
  }
}

hasPrivateMethod create private1
usesPrivateMethod create use1

puts "\n—— default state"
use1 usePrivateMethod "from global scope"
catch {private1 PrivateMethod "from global"} rtn
puts "Attempt to access PrivateMethod returns:\n $rtn"

puts "\n—— after export"
oo::define hasPrivateMethod export PrivateMethod
private1 PrivateMethod "from global after export"

puts "\n—— after unexport"
oo::define hasPrivateMethod unexport PrivateMethod
catch {private1 PrivateMethod "from global after unexport"} rtn
puts "Attempt to access PrivateMethod returns:\n $rtn"
use1 usePrivateMethod "from global scope"
```

Script Output

```
—— default state
PrivateMethod {From usesPrivateMethod {from global scope}}
Attempt to access PrivateMethod returns:
 unknown method "PrivateMethod": must be destroy

—— after export
PrivateMethod {from global after export}

—— after unexport
Attempt to access PrivateMethod returns:
 unknown method "PrivateMethod": must be destroy
PrivateMethod {From usesPrivateMethod {from global scope}}
```

The example below demonstrates the commands that create and modify methods. The `addition` class is created with no methods. Methods are added as they are needed with the `oo::define` *className* `method` command.

Example 3

Using `method, filter, forward` ***and*** `deletemethod`

```
# Create an addition class
oo::class create addition {
}

# Add a method to the addition class
oo::define addition method add {a b} {
  return [expr {$a + $b}]
}

addition create addObj

puts "No Filter"
puts "[addObj add 1 2]\n"

# Define a new method to show information about a method call
oo::define addition method show {args} {
    puts "—— [info level 0]"
    next {*}$args
    }


# Show method information before all method calls
# to addition objects
oo::define addition filter show
```

```
puts "With Filter"
puts "[addObj add 1 2]\n"

# Create a global scope procedure for doubling a pair of numbers
proc double {a b} {
  return [expr {$a+$a+$b+$b}]
}

# Add a forward to a global scope procedure
oo::define addition forward sum2 double

puts "double: [addObj sum2 1 2]\n"

# Add a 'sum' method by re-using the 'add' method.
oo::define addition forward sum my add

puts "Using sum: [addObj sum 1 2]\n"

# Rename the sum2 method to twice
oo::define addition renamemethod sum2 twice

puts "Renamed to twice: [addObj twice 1 2]\n"

# Done with debugging now, delete filter method
oo::define addition deletemethod show

puts "No More Filter"
puts "[addObj add 1 2]\n"
```

Script Output

```
No Filter
3

With Filter
---- addObj add 1 2
3

---- addObj sum2 1 2
double: 6

---- addObj sum 1 2
---- my add 1 2
Using sum: 3

---- addObj twice 1 2
Renamed to twice: 6
```

```
No More Filter
3
```

10.1.2 Modifying Inheritance

Superclasses and mixins can be defined at runtime as well as inside the `oo::class create` command script. The `superclass` and `mixin` subcommands create (or recreate) the list of classes associated with this class, thus they will replace any previously defined superclasses or mixins. If you need to add a superclass or mixin without removing the current classes you can use the `info class` command (discussed later in this chapter) to find out what classes are currently defined.

Adding a Superclass

When a superclass is added to a class, the original class methods will be called first. These methods must use the `next` command to transfer control to a method with the same name in the super class. If there are multiple superclasses, the `next` command will transfer control to the next class in the order that they are defined.

Syntax: `oo::define` *`className`* `superclass` *`className ?className?`*

Assign one or more classes to the list of superclasses.

`className` The name of the class (or classes) to assign as the list of superclasses.

Mixing in a New Class

When mixins are added to a class the mixin methods are called first and these methods must include a `next` command to transfer control to the base method with the same name.

Syntax: `oo::define` *`className`* `mixin` *`className ?className?`*

Assign one or more classes to the list of mixin classes.

`className` The name of the class (or classes) to assign as the list of mixin classes.

The next example shows using both superclasses and mixin classes. When both mixins and superclasses exist, the control goes first to the mixin, then to the base class, then to superclasses. If your code uses a `next` command when there is no method in the chain, Tcl will throw an error. You can use the `catch` command to invoke `next` when you don't know whether or not a method will be the last method in a chain.

Example 4

Script Example

```
oo::class create base {
  method show {args} {
    puts "base show args: $args"
    catch {next "base::show args: $args"}
```

```
  }
}

oo::class create addSuper {
  method show {args} {
    puts "addSuper show — $args"
    catch {next "addSuper::show args: $args"}
  }
}

oo::class create addMixin {
  method show {args} {
    puts "addMixin show — $args"
    catch {next "addMixin::show args: $args"}
  }
}

base create base1
puts "\nbase class"
base1 show "no super"

puts "\nbase class with addSuper"
oo::define base superclass addSuper
base1 show "with addSuper "

puts "\nbase class with addSuper and addMixin"
oo::define base mixin addMixin
base1 show "with super and mixin"
```

Script Output

```
base class
base show args: {no super}

base class with addSuper
base show args: {with addSuper }
addSuper show — {base::show args: {with addSuper }}

base class with addSuper and addMixin
addMixin show — {with super and mixin}
base show args: {addMixin::show args: {with super and mixin}}
addSuper show — {base::show args: \
    {addMixin::show args: {with super and mixin}}}
```

10.1.3 Modifying Class, Constructor, Variables and Destructor

A class can be fully defined with the `oo::class create` command, or you can use the `oo::class create` command to create an empty class and add the parameters using the `oo::define` command.

Defining a class dynamically can be useful when there are several similar classes to be created, or when your code creates on-demand classes to react to the external environment.

If a constructor is added with the `oo::define` command, it behaves exactly like a constructor that is created within an `oo::class create` script.

The `constructor` subcommand requires a set of arguments and a constructor body script.

Syntax: `oo::define` *`className`* `constructor` *`args script`*

Define a constructor for a class.

`args`	The argument list for the constructor.
`script`	The script to evaluate when an object of this class is created.

A destructor can be added much like the constructor, except that the destructor does not require any arguments. Note that the destructor will not return a value.

Syntax: `oo::define` *`className`* `destructor` *`script`*

Define a destructor for a class.

`script`	The script to evaluate when an object is destroyed.

Like the `mixin` and `superclass` subcommands, the `variable` command will replace the variable list with a new list. You can add a new variable to a class using the `info` command discussed later in this chapter.

Syntax: `oo::define` *`className`* `variable` *`variableName1 variableName2 ...`*

Set the variables for a class.

`variableName*`	A list of variables for this class.

The next example shows three classes being defined dynamically with the `oo::class` and `oo::define` commands. This script defines three character basic classes for a fantasy game: the warrior, mage and cleric. The initial hitpoints for a character of each class is hardcoded into the constructor, while the name of the character is passed when a character is created.

Example 5

Script Example

```
foreach type {warrior mage cleric} \
        hits {8 4 6} {
  oo::class create $type
  oo::define $type variable hitpoints myName
  oo::define $type constructor {name} \
      "set hitpoints $hits; set myName \$name"
  oo::define $type method display {} {return "$myName has $hitpoints"}
```

```
  oo::define $type destructor {puts "$myName go bye—bye"}
}

set w1 [warrior new Sigfried]
set m1 [mage new Brunhilda]
puts "[$w1 display] hitpoints"
puts "[$m1 display] hitpoints"
$m1 destroy
```

Script Output

```
Sigfried has 8 hitpoints
Brunhilda has 4 hitpoints
Brunhilda go bye—bye
```

10.1.4 Static Methods and Variables I

The common use of object-oriented programming is to assign parameters and methods to objects and work with the objects. In some circumstances, however, it's appropriate to assign a parameter or method to the class itself, not the objects instantiated from that class.

The `self` subcommand performs an action on the class, rather than modifying the class definition and objects created from the class. Other techniques for creating static variables and methods are discussed later in this chapter.

Syntax: `oo::define` *`className`* `self` *`action arg1 ...`*

Perform an action upon a class.

`action` A command to evaluate upon the class.

`args*` The arguments associated with the action.

The next example shows a class that can have a limited number of objects created from it. The count of how many objects have been created is maintained in the class. A new method to create objects on this class is also attached to the class, not to the objects created from the class.

Example 6

Script Example

```
# Define a class named limited.
oo::class create limited {
  constructor {} {
    puts "Creating new limited class"
  }
  method show {} {
    puts "This is an object of the limited class"
  }
}
```

```
# Define a variable count associated with the class,
# not objects of this class
oo::define limited self variable count

# Define a method to modify the count variable.
oo::define limited self method setCount {{val {}}} {
  if {$val ne ""} {
    set count $val
  }
  return $count
}

# Define a method to create objects of this class.
# This invokes the usual 'new' command to create objects.

oo::define limited self method make {} {
  puts "Count starts at: $count"
  if {$count > 0} {
    incr count -1
    puts "New count: $count"
    return [limited new]
  } else {
    error "Exceeded available object count"
  }
}

# Only allow one limited object to be created.
limited setCount 1

# Confirm that an object can be created.
set l1 [limited make]
$l1 show

# Confirm that the second attempt fails
set fail [catch {set l2 [limited make]} rtn]
if {$fail} {
  puts "Create failed: $rtn"
}
```

Script Output

```
Count starts at: 1
New count: 0
Creating new limited class
This is an object of the limited class
Count starts at: 0
Create failed: Exceeded available object count
```

10.2 MODIFYING OBJECTS

The `oo::objdefine` command can be used to modify individual objects as well as all the objects in a class. The syntax is the same as the `oo::define` command except that the name of a class is replaced with the name of an object.

Syntax: `oo::objdefine` *`objectName script`*
`oo::objdefine` *`objectName ?subcommand ?arg1 arg2 ...?`*
Define a feature of an individual object.

`className`	The name of the object to be modified.
`script`	A script with commands to modify the object. This may modify several features.
`subcommand`	A subcommand that defines a single feature to be modified.
`?arg1 arg2 ...?`	The arguments that a subcommand requires.

Most of the subcommands supported by the `oo::define` command are also supported by the `oo::objdefine` command.

These subcommands are supported by both `oo::define` and `oo::objdefine`:

`method`	`method` *`methodName args script`*
`filter`	`filter` *`methodName ?methodName?`*
`forward`	`forward` *`newName actualCmd ?args?`*
`renamemethod`	`renamemethod` *`oldName newName`*
`deletemethod`	`deletemethod` *`methodName ?methodName?`*
`export`	`export` *`methodName ?methodName?`*
`unexport`	`unexport` *`methodName ?methodName?`*
`mixin`	`mixin` *`className ?className?`*
`variable`	`variable` *`variableName1 variableName2 ...`*

The `constructor`, `destructor`, `self`, and `superclass` commands are only supported with `oo::define`, not `oo::objdefine`.

10.2.1 Changing an Object's Class

In most object-oriented programming systems, once an object of a given class is created, it is a member of that class until it is destroyed.

If the capabilities of the object change, it can be better to change the class, rather than encode all of the possible behaviors in a single class's methods.

The `class` subcommand of the `oo::objdefine` command changes an object from one class to another. The constructor is not called when an object changes class, but all of the old methods and variables are replaced by the methods and variables of the new class.

In some fantasy games (for example, the classic `hack`), when a character dies, a corpse is left behind. The corpse still has a name and possessions, but can no longer attack or defend itself.

The next example shows a Fantasy hero in which we have a `character` base class, and two derived classes: `liveCharacter` and `deadCharacter`. The character's name and possessions are stored in variables in the base class. The character's attack strength is determined by the `attackStrength` method in the derived classes. The `deadCharacter` class adds a new method `takePossession` to allow a player to remove items from a dead character, but not from a live character.

Example 7

Script Example

```
# Create a base class with variables for name and
# a list of possessions, and a general method for accessing
# the contents of class variables.

oo::class create character {
  variable possessions name

  constructor {nm args} {
    set name $nm
    set possessions $args
  }

  # Return the value of an object's variable
  # NOTE: No error checking.  Variable must be valid.
  method get {id} {
    return [set $id]
  }
}

# Create a class for live characters.
# Derives from character.
# Includes a method for attacking
oo::class create liveCharacter {
  superclass character

  # Pass control to superclass constructor
  constructor {args} {
    next {*}$args
  }

  # Return attack strength
  method attackStrength {} {
    return 8
  }
```

```
}

# Create a class for dead characters.
# There is no constructor
# The attackStrength method returns a 0 for attack.

oo::class create deadCharacter {
  superclass character
  variable possessions

  method attackStrength {} {
    return 0
  }

  method takePossession {} {
    set taken [lindex $possessions 0]
    set possessions [lrange $possessions 1 end]
    return $taken
  }
}

set char1 [liveCharacter new Sigfried Spear Tarnhelm]

puts "[$char1 get name] possesses [$char1 get possessions]"
puts "[$char1 get name] attacks with strength of \
    [$char1 attackStrength]"

puts "After dieing,"
oo::objdefine $char1 class deadCharacter
puts "[$char1 get name] possesses [$char1 get possessions]"
puts "[$char1 get name] attacks with strength of \
    [$char1 attackStrength]"
puts "Took [$char1 takePossession] from [$char1 get name]"
puts "Took [$char1 takePossession] from [$char1 get name]"
```

Script Output

```
Sigfried possesses Spear Tarnhelm
Sigfried attacks with strength of 8
After dieing,
Sigfried possesses Spear Tarnhelm
Sigfried attacks with strength of 0
Took Spear from Sigfried
Took Tarnhelm from Sigfried
```

10.2.2 Defining Per-object Mixins

Adding a mixin to the class definition is useful for features that will be permanent and are shared by all members of a class. Some features may be object specific (for example, an employee's skills) or may not be permanent (for example, an employee's security status).

You can add a mixin to an individual object with the `mixin` subcommand. The object specific `mixin` subcommand has the same behavior as the class version of the command, except that a mixin is added to a single object.

An object has a two lists of mixin classes. One list is the mixins assigned to the class and the other is the mixins assigned to an individual object.

This example shows a fantasy character obtaining a spear and magic helmet as mixins. The magic helmet is mixed into the warrior class in the class definition. The spear is mixed into the object. The object mixin methods are invoked before the class mixin methods.

Example 8

Script Example

```
oo::class create magicHelmet {
  method defense {attackStrength} {
    puts "Magic Helmet Reduces attack by 2"
    return [next [expr {$attackStrength - 2}]]
  }
}

oo::class create spear {
    method defense {attackStrength} {
    puts "Spear reduces attack by 2"
    return [next [expr {$attackStrength - 2}]]
  }
}

oo::class create warrior {
  mixin magicHelmet

  constructor {} {
    variable State
    array set State {defense 4 attack 4 hitpoints 5}
  }

  method defense {attackStrength} {
    variable State
    puts "Final Attack is: $attackStrength"
    if {$attackStrength > $State(defense)} {
      return "Hit"
    } else {
      return "Missed"
    }
```

```
    }
}


warrior create elmer

puts "With only a magic helmet"
puts "With a magic helmet elmer is [elmer defense 8]"

puts "\n Add a spear to the elmer object"
oo::objdefine elmer mixin spear
puts "With spear and magic helmet elmer is [elmer defense 8]"
```

Script Output

```
With only a magic helmet
Magic Helmet Reduces attack by 2
Final Attack is: 6
With a magic helmet elmer is Hit

Add a spear to the elmer object
Spear reduces attack by 2
Magic Helmet Reduces attack by 2
Final Attack is: 4
With spear and magic helmet elmer is Missed
```

10.2.3 Adding a Method to an Object

Most methods are defined within the `oo::class create` script or with the `oo::define` command. These methods are common to all members of a class.

We can create an object-specific method with the `method` subcommand. This method may have a unique name, or may override the name of an existing method defined for all objects in the class. If the name is overriding an existing class-defined method, the object method will be called first and may pass control to the class method with the `next` command.

In the case of overriding the class-defined method with an object-specific method, both methods exist in the *method chain*. The object-specific method will be invoked first and may either return (breaking the chain), or can pass control to the rest of the method chain with the `next` command.

In the next example a new `defense` method is added to the `elmer` object.

10.3 EXAMINING CLASSES AND OBJECTS

One difference between Tcl and most other languages is the amount of introspection available to the script writer. Information that needs to be hard-coded in other languages can be determined at run-time in Tcl. This makes it possible to write generic routines that adapt to new conditions, rather than coding information about the application into the application.

The `info` command is the tool that provides information into the configuration of the Tcl interpreter.

The basic form of the info command for classes and objects is:

Syntax: `info class` *`subcommand className args`*
`info object` *`subcommand objectName args`*

Return information about a class or object.

`subcommand`	A subcommand describing what info should be returned.
`className`	The name of a class to examine.
`objectName`	The name of an object to examine.
`args`	Other arguments that may be required (name of a method, etc.).

The return value from `info class` and `info object` may not be the same. The `class` commands return information that is common to all members of a class. The `object` commands return information for only a single object. When separate lists are maintained for object and class (as is done for mixins), the commands return the value specific to a given list, not common to both lists.

The previous example has a `::magicHelmet` mixin attached to the `warrior` class and a `::spear` mixin attached to the `elmer` object.

The `info class mixin warrior` command returns `::magicHelmet` while `info object mixin elmer` returns `::spear`.

10.3.1 Evaluation of Chains

When one class acquires properties from another class, either by inheritance or with a mixin, control is passed from one class to another. In order to pass control from one class to another, TclOO creates a chain of elements to evaluate when a user requests an action. There is a chain of constructors when an object is created, a chain of methods when a method is invoked, and a chain of destructors when an object is destroyed.

The method chain is constructed in the same order for constructors, destructors and methods. The following list shows the order in which items will be evaluated. Note that the second step only applies to methods and destructors since a constructor is evaluated before an object exists.

If a method can appear in the chain more than once (perhaps a class has been included both as a super and a mixin), it will be added at the latest valid position.

Filter Methods If multiple filters are associated with a class, they appear in the order of the class hierarchy.

Object Methods A method defined for this object (with `oo::objdefine` method) is evaluated next.

Object Mixin Methods Method defined in the mixin list associated with the object, from first to last in the order of the list of mixins. Per-object mixins are defined with the `oo::objdefine` command, as described above.

Class Mixin Methods Methods defined in the mixin list associated with the class, from first to last in the order of the list of mixins. These mixins are defined with the `oo::class create` command or an `oo::define` command.

Class Method The method in the current class. This might be defined in the `oo::class create` script or with the `oo::define` command.

Superclass Methods The method defined in superclass list for the class, from first to last in the list of superclasses. If a superclass has one or more mixins defined, those mixins will appear before the superclass.

The `next` command evaluates the next method in the chain and returns to the method that invoked `next`. The `next` command can come early in a procedure or late. Your procedure can modify the parameters that will be passed to the next method in the chain, or modify the values after the next method has returned.

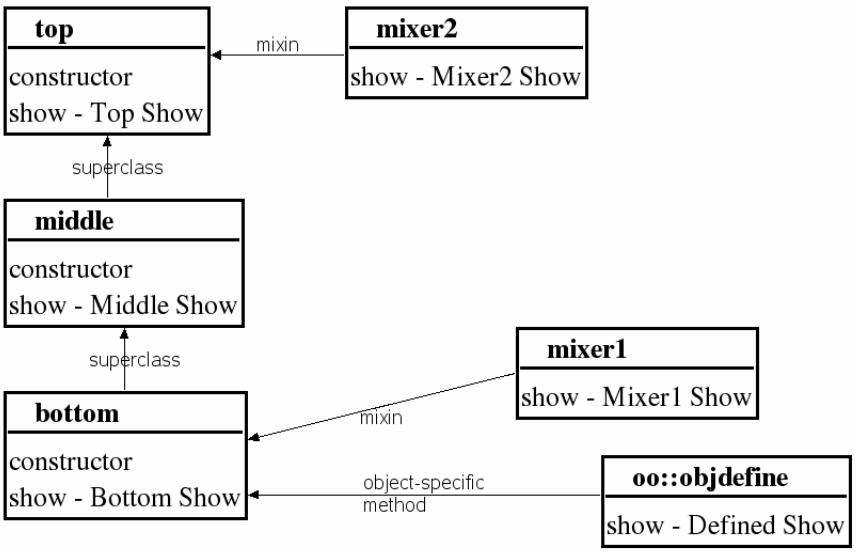
If there is no `next` command in a method, the evaluation will stop with the current method and not proceed to other elements of the chain.

Including a `catch next` command in constructors, methods and destructors is one way to allow a class to be reused for other applications. This is particularly true of classes that are expected to be used as mixins to massage values before other processing.

The example below shows a set of classes that use `superclass` and `mixin` inherit functionality. The constructors and methods are dummies that display when they are invoked in order to show the chain.

The class hierarchy is shown in the following illustration. The `bottom`, `middle` and `top` classes inherit from each other using the `superclass` command, while `mixer1` and `mixer2` are inherited using the `mixin` command. Each class has a `show` method which displays the class that contains it.

Constructor Method and Destructor Chaining



The following example uses this class hierarchy to create an object, showing order in which the constructors are chained together. It then invokes the `show` method, again showing how the chain is constructed. Finally, a new `show` method is added to the `test` object to show that the object's method is placed first in the chain.

Example 9

Script Example

```
oo::class create mixer1 {
  constructor {} {
    puts "Mixer1 Constructor"
    catch {next}
  }
  method show {} {
    puts "Mixer1 Show"
    catch {next}
  }
}

oo::class create mixer2 {
  constructor {} {
    puts "Mixer2 Constructor"
    catch {next}
  }
  method show {} {
    puts "Mixer2 Show"
    catch {next}
  }
}

oo::class create top {
  mixin mixer2
  constructor {} {
    puts "Top Constructor"
    catch {next}
  }
  method show {} {
    puts "Top Show"
    catch {next}
  }
}

oo::class create middle {
  superclass top;
  constructor {} {
    puts "Middle Constructor"
    catch {next}
```

```
  }
  method show {} {
    puts "Middle Show"
    catch {next}
  }
}

oo::class create bottom {
  superclass middle;
  mixin mixer1;
  constructor {} {
    puts "Bottom Constructor"
    catch {next}
  }
  method show {} {
    puts "Bottom Show"
    catch {next}
  }
}


puts "Order of constructors"
bottom create test

puts "\nOrder of methods being invoked"
test show

puts "\nOrder of methods after oo::objdefine method"
oo::objdefine test method show {} {
    puts "Defined Show";
    catch {next}
}
test show
```

Script Output

```
Order of constructors
Mixer1 Constructor
Bottom Constructor
Middle Constructor
Mixer2 Constructor
Top Constructor

Order of methods being invoked
Mixer1 Show
Bottom Show
Middle Show
Mixer2 Show
Top Show
```

```
Order of methods after oo::objdefine method
Defined Show
Mixer1 Show
Bottom Show
Middle Show
Mixer2 Show
Top Show
```

Including the `next` command in all constructors, methods and destructors ensures that control will propagate through the chain in the order that TclOO defines.

This is not always what you want to occur.

The `nextto` command will send control to a specific element of the chain. The `nextto` command can be used to invoke functions in any desired order.

Instead of using `next` to step through the functions, your code can use the `nextto` command to invoke functions in a the order you want them invoked. The downside to this technique is that it requires that the first method invoked knows the rest of the call hierarchy. If an object acquires a mixin, that mixin's method will be the first method on the list and will either need to know the rest of the hierarchy or use the `next` command to pass control into the class-specific methods.

Syntax: `nextto` *`className args`*

Invoke the method in the defined class.

`className`	The name of a class with a method of the same name as the method calling `nextto`.
`args`	Any arguments required by the method in the defined class.

You can mix `next` and `nextto` commands within a class structure and even within a single method. Note that invoking `next` or `nextto` more than once can cause the underlying functions to be evaluated multiple times.

In the previous example the `mixer1` constructor is the first to be invoked. If that constructor is modified to use `nextto` instead of `next` some constructors can be skipped. In this example, the constructor for `bottom` (the class of the object being created) is skipped.

Example 10

Script Example

```
oo::class create mixer1 {
  constructor {} {
    puts "Mixer1 Constructor"
    catch {nextto middle}
  }
```

Script Output

```
Order of constructors
Mixer1 Constructor
```

```
Middle Constructor
Mixer2 Constructor
Top Constructor
```

Unless you have a tightly defined class hierarchy, you will want to examine the call chain before doing very much with `next` and `nextto`. You can determine the chain for a constructor, method or destructor with the `info class call` or `info object call` commands.

Syntax: `info class call` *`className methodName`*
`info object call` *`objectName methodName`*
Return a list describing the call chain for the given method.

`className`	The name of the class associated with this method.
`objectName`	The name of the object associated with this method.
`methodName`	The name of the method to be queried.

The `call` subcommand returns a list of lists. Each list element consists of four fields:

1. General type of element. This will be one of
   - `method`: This is a normal method.
   - `filter` This is a method attached as a filter.
   - `unknown` If the method is attached by some other technique.
2. The name of the method.
3. The class this method is associated with. If the method is attached directly to an object, this will be the word `object`.
4. Specific type of method—this is the value that is returned by the `methodtype` subcommand, which will be discussed later in this chapter.

Adding these commands to the end of the previous example produces the following output. Notice that the `object call` output is almost identical to the `class call` output, except that it shows the show method that was attached to the object as well as the normal class-defined methods.

Example 11

Script Example

```
puts "CLASS CALL"
foreach link [info class call bottom show] {
  puts $link
}
puts "\nOBJECT CALL"
foreach link [info object call test show] {
  puts $link
}
```

Script Output

```
CLASS CALL
method show ::mixer1 method
method show ::bottom method
method show ::middle method
method show ::mixer2 method
method show ::top method

OBJECT CALL
method show object method
method show ::mixer1 method
method show ::bottom method
method show ::middle method
method show ::mixer2 method
method show ::top method
```

If a filter is added to a class, it will be called first, regardless of where it appears. If there are multiple filters, they are called in the order of the class hierarchy.

The next example adds a couple of filters to the code above and shows the new `call` output:

Example 12

Script Example

```
# Define a new method in class 'top'
oo::define top method testfilter {args} {
    puts "top filter";
    next
}

# Define a new method in class 'middle'
oo::define middle method testfilter {args} {
    puts "middle filter";
    next
}

# Add the new methods as filters
oo::define top filter testfilter
oo::define middle filter testfilter
```

Script Output

```
OBJECT CALL after adding filter
filter testfilter ::middle method
filter testfilter ::top method
method show object method
method show ::mixer1 method
method show ::bottom method
```

```
method show ::middle method
method show ::mixer2 method
method show ::top method
```

If a class has forwarded methods, the `class` command only shows the chain up to the forwarded method.

Adding these lines to the previous example demonstrates how the filters are seen immediately, but the last element to be shown in the chain is the forwarded method.

Example 13

Script Example

```
oo::define bottom forward testforward my show

puts "\nCLASS CALL for testforward"
foreach link [info class call bottom testforward] {
  puts $link
}
```

Script Output

```
CLASS CALL for testforward
filter testfilter ::middle method
filter testfilter ::top method
method testforward ::bottom forward
```

The rest of the information about a forwarded method (like, what it's been forwarded to) can be obtained. This will be discussed in the next section.

10.3.2 Examining Methods

If you are examining a class or object, you may need to examine the methods, method arguments and method bodies as well as the execution chain. This facility is useful for writing general-purpose, class-driven serialization functionality, or for constructing classes on the fly with a clone-and-modify function.

The `info` commands that give you information about methods are implemented for both classes and objects. The subcommands are listed below.

info class methods	Returns a list of methods.
info class methodtype	Returns the type of method.
info class definition	Returns full information about a method.
info class constructor	Returns the arguments and body of a constructor.
info class destructor	Returns the body of a destructor.

info class forward	Returns the script that this command is forwarded to.
info class filterss	Returns a list of filters.

One place to start examining a class's or object's methods is just to get a list of the methods. The `info class methods` and `info object methods` commands behave slightly differently.

Syntax: `info class methods` *`className`*

Return a list of methods associated with a class.

`className`	The class with methods to be returned.
`?-all?`	By default, `info class methods` returns only the methods defined in this class. When the `-all` flag is added methods associated with the superclasses and mixins are also returned.

Syntax: `info object methods` *`objectName ?-all?`*

Return a list of methods associated with a object.

`objectName`	The object with methods to be returned.
`?-all?`	By default, `info object methods` returns only the methods defined for the object using the `oo::objdefine method` command. When the `-all` flag is added methods associated with the class and mixins are also returned.

The next examples revisit the class for a fantasy role-playing character. The class has methods for `show` and `defense`. After a character is created, we add an object-specific method to cast lightning (`cast`).

The `info class methods` call displays the methods defined for the class `show` and `defense`. The first `info object methods` call only shows the `cast` method. When the `-all` flag is added, it displays the class methods and also the `destroy` method for deleting the object.

Example 14

Script Example

```
# Define a character class with minimal methods.

oo::class create character {
  variable State
  constructor {nm} {
    array set State {defense 2 attack 2 hitpoints 5}
    set State(name) $nm
  }
  destructor {
    puts "$State(name) is gone"
  }
  method show {args} {
    parray State
  }
```

```
    method defense {attackStrength} {
      if {$attackStrength > $State(defense)} {
        return "Hit"
      } else {
        return "Miss"
      }
    }
}

# Create a character named Sigfried and
# give him a sing method

character create elmer Sigfried

oo::objdefine elmer method cast {} {
    puts "Lightning Flash.  Thunder Roar."
}

puts "The methods for the character class are:"
puts "  [info class methods character]"

puts "The methods for the object elmer are:"
puts "  [info object methods elmer]"

puts "All the methods for the object elmer are:"
puts "  [info object methods elmer -all]"
```

Script Output

```
The methods for the character class are:
  show defense
The methods for the object elmer are:
  cast
All the methods for the object elmer are:
  cast defense destroy show
```

We can quickly add an attack method to the class by creating a simple procedure and adding it to the class with the `oo::define character forward` command.

The new code and a command to view the methods are shown in the next example. The new method `attack` is shown in the list of class methods.

Example 15

Script Example

```
oo::define character forward attack attackStrength

proc attackStrength {} {
```

```
  return 8
}

puts "After adding forward, the methods for the character class are:"
puts "  [info class methods character]"
```

Script Output

```
After adding forward, the methods for the character class are:
  attack show defense
```

We can add a filter method to test that input is a valid integer. This filter replaces invalid input with a 0.

The new code to define a method and then add it as a filter looks like this:

Example 16

Script Example

```
oo::define character method inputTest {val} {
    if {![string is integer] $val} {
      set val 0
    }
    next $val
  }

oo::define character filter inputTest

puts "After adding filter, the methods for the class are:"
puts "  [info class methods character]"
```

Script Output

```
After adding filter, the methods for the class are:
  inputTest attack show defense
```

The output from this example shows that `inputTest` is added to the list of methods associated with the class, but doesn't tell us whether this is a filter, a forward or a normal method.

The `info class methodtype` command reports whether a method is a regular method or a forward. A filter is a normal method and isn't distinguished. We can get a list of a class's filters with the `filters` subcommand.

Using the filters command to get a list of the filters is shown below.

Example 17

Script Example

```
puts "The filters defined for the character class are:"
puts "  [info class filters character]"
```

Script Output

```
The filters defined for the character class are:
  inputTest
```

We can get information about the arguments and body of a method with the `definition` and `forward` subcommands. These subcommands each return a two-element list. The first element will be a list of the method's arguments and the second element is the body of the method.

These commands are sufficient to construct a procedure which will serialize a class and generate output suitable for rebuilding the class.

The example below is a procedure that will accept the name of a class and will generate the code to reconstruct the class. This can be used to save or view a simple class definition. The next section discusses how to handle classes with superclasses and mixins.

Note that the `info` commands do not distinguish between methods added in the `oo::class create` script and methods added with an `oo::define` command. The `oo::class create` command invokes the `oo::define` command to add methods, thus there is no distinction between methods added with either command.

Example 18

Script Example

```
proc defineClass {className} {
  puts "oo::class create class $className {"

  foreach m [info class methods character] {
    switch [info class methodtype character $m] {
       forward {
         set forwardTo [info class forward character $m]
         append cmds "\n# Method $m is forwarded to $forwardTo\n"
         append cmds "oo::define $className forward $m $forwardTo\n"
         set procArgs [info args $forwardTo]
         set procBody [info body $forwardTo]]
         append cmds [list proc $m $procArgs $procBody]
       }
       method {
         puts "\n# Definition of method $m"
         puts "method $m [info class definition character $m]"
      }
    }
  }
  puts "}"

  puts "\n# These filters are defined for $className"
  foreach f [info class filters $className] {
    puts "oo::define character filter $f"
```

```
  }

  puts "\n # These forwards are defined for $className"
  puts $cmds

}

puts "Full definition" defineClass character
```

Script Output

```
oo::class create class character {

# Definition of method inputTest
method inputTest val {
    if {![string is integer] $val} {
      set val 0
    }
    next $val
  }

# Definition of method show
method show args {
    parray State
  }

# Definition of method defense
method defense attackStrength {
    if {$attackStrength > $State(defense)} {
      return "Hit"
    } else {
      return "Miss"
    }
  }
}

# These filters are defined for character
oo::define character filter inputTest

 # These forwards are defined for character

# Method attack is forwarded to attackStrength
oo::define character forward attack attackStrength
proc attack {} {
  return 8
}
```

10.3.3 Examining Inheritance

The dynamic nature of TclOO's way of handling superclasses and mixins can lead to unexpected combinations of classes. Fortunately, the `info` command has tools to untangle a web of classes and make sense of them.

The subcommands that provide information about the class hierarchy are:

`info class superclasses`	Returns a list of superclasses.
`info class subclasses`	Returns a list of subclasses.
`info class mixins`	Returns a list of mixins attached to a class.
`info object mixins`	Returns a list of mixins attached to an object.

The `superclasses` and `subclasses` subcommands let you trace what classes are derived from each other.

Syntax: `info class superclasses` *`className`*

Returns a list of superclasses.

`className`	The name of the class to return superclasses for.

If you need to know what classes are derived from a given class, you can use the `subclasses` subcommand.

Syntax: `info class subclasses` *`className ?pattern?`*

Returns a list of classes derived from this class that match an optional pattern.

`className`	The name of the class to return superclasses for.
`pattern`	An optional glob-pattern to limit the number of derived classes to return.

The next example shows the character class and a warrior class that's derived from it. The `info class` commands show the superclass and subclass relationships.

Example 19

Script Example

```
# The base character class
oo::class create character {
  variable State
  constructor {nm} {
    array set State {defense 2 attack 2 hitpoints 5}
    set State(name) $nm
  }

  method defense {attackStrength} {
    if {$attackStrength > $State(defense)} {
      return "Hit"
    } else {
```

```
      return "Miss"
    }
  }
}


# A warrior gets better attack and defense skills
oo::class create warrior {
  superclass character

  variable State
  constructor {name} {
    puts "Warrior constructor"
    next $name
    incr State(defense) 2
    incr State(attack) 2
  }
}


puts "These classes superclass from character"
puts "  [info class subclass character]"

puts "The superclass for warrior is"
puts "  [info class superclass warrior]"
```

Script Output

```
These classes superclass from character
  ::warrior
The superclass for warrior is
  ::character
```

10.3.4 Getting a List of Base Classes

The TclOO class hierarchy is a singly-rooted hierarchy. At the base is the `oo::object` class. All classes are subclasses of the `oo::object` class from which they inherit the default `destroy` method.

A dynamic TclOO application might define new classes while it's running. We can use the `info class subclasses` command with the `oo::object` class to get a list of all the defined classes as shown in the next example. Only the top level classes that have been defined show up as subclasses of `oo::object`. The `subclasses` subcommand does not recurse into the class hierarchy.

Example 20

Script Example

```
info class subclasses oo::object
```

Script Output

```
::character
```

If our hero should find a magic helmet to wear it should be added as a mixin to the object for our hero, not to the warrior class. The `info class mixin` and `info object mixin` commands return different values in the next example.

Syntax: `info class mixins` *`className`*
`info object mixins` *`className`*

Returns a list of classes mixed into the named class or object.

`className` The name of the class to return superclasses for.

The next example adds a magicHelmet class, creates a hero, and uses the `mixin` command to give him the helmet. The `info class mixin` command returns an empty list, while the `info object mixin` command shows that elmer has a magic helmet.

Example 21

Script Example

```
# The base character class
oo::class create character {
  variable State
  constructor {nm} {
    array set State {defense 2 attack 2 hitpoints 5}
    set State(name) $nm
  }

  method defense {attackStrength} {
    if {$attackStrength > $State(defense)} {
      return "Hit"
    } else {
      return "Miss"
    }
  }
}


# A warrior gets better attack and defense skills
oo::class create warrior {
  superclass character

  variable State
  constructor {name} {
    puts "Warrior constructor"
    next $name
```

```
    incr State(defense) 2
    incr State(attack) 2
  }
}



oo::class create magicHelmet {
  method defense {attackStrength} {
    puts "Magic Helmet Reduces attack by 2"
    return [next [expr {$attackStrength – 2}]]
  }
}

warrior create elmer Sigfried
oo::objdefine elmer mixin magicHelmet

puts "Class Mixins: [info class mixins warrior]"
puts "Elmer's Mixins: [info object mixins elmer]"
```

Script Output

```
Warrior constructor
Class Mixins:
Elmer's Mixins: ::magicHelmet
```

10.4 EXAMINING OBJECTS

In order to save an application's state, we need to record that state of the objects, as well as the class hierarchy. There are more `info` commands that report information about the objects.

The first step in getting information about an object is to find out what classes and objects exist. The `info class subclasses` command does not recurse into the class hierarchy, but this command can be wrapped with a recursive procedure to return the complete list of defined classes. The next example is a recursive procedure that returns a list of all the classes that are defined.

Example 22

Script Example

```
proc findClasses {parent  }  {
  set childClasses [info class subclasses $parent]
  if {$childClasses eq ""} {
    return
  } else {
    set children $childClasses
```

```
    foreach cl $childClasses {
      # To protect from looping, confirm that
      # each class is only included once.
      foreach gch [findClasses $cl] {
        if {[lsearch $children $gch] < 0} {
          lappend children $gch
        }
      }
    }
    return $children
  }
}
```

Given a list of classes, the next step is to learn what objects have been instantiated from each class. The `info class instances` command will return the names of objects instantiated from a class. On the flip side, the `info object class` command will return the class of an object.

The previously discussed commands can be used to extract the method, superclass and mixin information for an object. Getting the contents of variables requires a bit more work.

One technique is to create generic methods to assign and retrieve values from object variables. A new method to examine and modify variables associated with the warrior object `elmer` is shown in the next example. This type of a procedure can be useful in any class. The ability to access an object's variables breaks the black-box concept of object-oriented programming, but it's often preferable to having set/get routines for each variable that might need to be accessed outside a class.

The `variable` command must be used in this method to map the object variables into the local scope because this method is designed as a mixin, not part of the class definition defined with the `oo::class create` command.

Example 23

Script Example

```
oo::objdefine elmer method config {name args} {
      if {[set p [string first "(" $name]] > 0} {
        incr p -1
        set varName [string range $name 0 $p]
      } else {
        set varName $name
      }
      variable $varName

      if {$args ne ""} {
        set $name {*}$args
      }
        return [set $name]
    }
```

```
puts [elmer config State(attack)]
puts [elmer config State(attack) 6]
```

Script Output

```
4
6
```

A routine like this can be used to examine an object's variables and to create a serialized image of an object. The list of variables defined for an object is returned by the `info object vars` command.

Syntax: `info object vars` *?pattern?*

Return an object's variables as a list.

pattern Return only the variables with names that match this pattern.

Example 24

Script Example

```
puts [info object vars elmer]
```

Script Output

```
State
```

Another technique for examining variables is to use the `namespace` commands. The basis of the TclOO package is the Tcl namespace. Each class definition and each object is contained in a separate namespace. When you create a new object with the `new` command (not the `create` command) the return value for that object is the namespace. If you know the object's namespace, you can use the namespace commands to examine and recreate it.

If an object was created with the `create` command, the return value is the name defined in your script, not a namespace path. The `info object namespace` command will return the namespace associated with an object whether the object was created with `new` or `create`.

Syntax: `info object namespace` *objectName*

Return the namespace for an object.

objectName The name of the object.

The next example shows an object serialization procedure that uses the previously discussed `findClasses` procedure. This procedure will return a set of commands to rebuild all the objects in a system. Note that while this generic procedure will work for many class hierarchies, it's not guaranteed to work for all class hierarchies. The Tcler's wiki (`http://wiki.tcl.tk`) has discussion of other techniques for serializing objects.

The `info class instances` command will return all objects that have instances of a class—this includes objects that use a class as a mixin. For instance, since our hero `elmer` has a mixin of `magicHelmet`, the command `info instances magicHelmet` will return `elmer` as an object instantiated from this class.

The test to confirm that `info object class $obj` returns the same value as the class used in `info class instances` reduces the list of objects to only objects that are truly members of a class.

Objects can be created with a `new` or `create` command. When the `new` command is used, the name of the object is the same as the name of the namespace that contains the object's variables. The test to see if the object name is the same as the namespace name determines which command should be used to recreate an object.

Example 25

Script Example

```
proc findClasses {parent  }  {
  set childClasses [info class subclasses $parent]
  if {$childClasses eq ""} {
    return
  } else {
    set children $childClasses
    foreach cl $childClasses {
      # To protect from looping, confirm that
      # each class is only included once.
      foreach gch [findClasses $cl] {
        if {[lsearch $children $gch] < 0} {
          lappend children $gch
        }
      }
    }
    return $children
  }
}


proc serializeAllClasses {} {
  foreach cl [findClasses oo::object] {
    set comment "# Rebuilding members of class $cl"
    foreach obj [info class instances $cl] {

      # Check that an object is truly a member of this
      # class.
      # A mixin class will report objects that include
      # the mixin.
      if {[info object class $obj] eq $cl} {

        set ns [info object namespace $obj]
```

```
        # Check to see if this object was created with
        # a 'new' or 'create' command, to rebuild correctly.
        if {$ns eq $obj} {
          set cmd "new "
        } else {
          set cmd "create [string trim $obj ::]"
        }
        append cmds "$comment\n"
        append cmds "set obj \[$cl $cmd {} \]\n"
        append cmds "set ns \[info object namespace \$obj]\n"
        foreach var [info object vars $obj] {
          if {[catch {array get ${ns}::$var} arrayDef]} {
            append cmds \
            "set \${ns}::$var [list [set ${ns}::$var]]\n"
          } else {
            append cmds \
            "array set \${ns}::$var [list $arrayDef]\n"
          }
        }
      }
      set comment ""
    }
  }
  return $cmds
}

puts [serializeAllClasses]
```

Script Output

```
set obj [::warrior create elmer {} ]
set ns [info object namespace $obj]
array set ${ns}::State {attack 4 name Sigfried \
    hitpoints 5 defense 4}
```

10.5 USING TCLOO WITH CALLBACKS

Several Tcl commands including `fileevent`, `after`, and various Tk widgets perform callbacks when some event occurs. In order to register a callback within an object's method and have the callback be directed to a method in the same object, the code within that method needs to be able to introspect the current object.

The `info` commands provide information about classes and methods, but they aren't always appropriate for use within a method. The `self` command allows a method to introspect itself. The `self`

command has several subcommands. The man page for `self` describes all of the subcommands. To register an object and method for a callback, we need the `self object` command.

Syntax: `self object`
Return the name of the object in which code is being evaluated.

If the `self` command is used with no subcommand, it defaults to the `self object` behavior.

This example shows how the `self` command can be used with the `after` command to create a callback that will be invoked at least once. If the `repeat` variable is true it will nag the user every second.

Example 26

Script Example

```
oo::class create timer {
  variable interval repeat
  constructor {int rpt} {
    set interval $int
    set repeat $rpt
    puts "Registering [self] for a $interval ms callback"
    after $interval \
        [list [self] alert "$interval milliseconds have elapsed"]
  }
  method alert {msg} {
    puts $msg
    if {$repeat} {
      after 1000 \
          [list [self] alert "You've missed the alert!"]
    }
  }
}

timer create twoSeconds 2000 1

vwait forever
```

Script Output

```
Registering ::twoSeconds for a 2000 ms callback
2000 milliseconds have elapsed
You've missed the alert!
You've missed the alert!
...
```

The `trace` command will evaluate a callback when a variable is modified. In order to trace an object's variable with the `trace` command, the trace command needs the full namespace path to the variable.

The `my varname` command returns the rooted namespace path for an object variable. This value can be supplied to the `trace` command just as you would otherwise provide the name of a variable in the current scope.

Syntax: `my varname` *`variableName`*

Returns the full namespace path to a named variable.

`variableName` The simple variable name.

The callback for the `trace` command can be another class method or a global scope procedure. The trace callback is performed in a procedure scope below the level where the assignment occurs; thus the variable name can always be mapped into the callback's scope with the `upvar` command.

If the callback procedure is a method within the class, then the code that registers the callback must use the `self` command to provide the object context to invoke.

The example below shows a class with two variables, `var1` and `var2`. Each variable has a trace attached to it. The `var1` callback is a global scope procedure, while the `var2` callback is a method within the `withTrace` class.

Example 27

Script Example

```
oo::class create withTrace {
  variable var1 var2
  constructor {} {
    set var1 2
    set var2 2
    trace add variable [my varname var1] write \
        showVar
    trace add variable [my varname var2] write \
        [list [self] showVar]
    puts "Full path for var1 is: [my varname var1]"
  }
  method decrVar1 {} {
    incr var1 -1
  }
  method decrVar2 {} {
    incr var2 -2
  }
  method showVar {name index operation} {
    upvar $name vv
    puts "Method shows new value for $name is $vv"
  }
}

proc showVar {name index operation} {
  upvar $name vv
  puts "Global proc shows new value for $name is $vv"
}
```

```
withTrace create test
test decrVar1
test decrVar2
```

Script Output

```
Full path for var1 is: ::oo::Obj4::var1
Global proc shows new value for var1 is 1
Method shows new value for var2 is 0
```

The `self` and `my varname` commands are used to link objects with Tk widgets. These uses will be discussed in more detail in the next Tk chapters, but here's a brief overview.

The `self` command is used to link a callback to a Tk widget that needs context for the callback. Using an object method and the argument to the `-command` option of the `button` widget requires the `self` command to provide this context.

The `my varname` command must be used to link a Tk widget with a variable using the widget's `-textvariable` option.

10.6 ADDING NEW FUNCTIONALITY TO TCLOO

TclOO is a sufficiently complete object-oriented system to be useful, but the original intent was that TclOO would be the framework for building more complex and complete OO systems.

This section will discuss some of the hooks for adding new commands and features to TclOO.

10.6.1 Static Variables II

A *class* or *static* variable is a variable that is attached to the class, rather than the objects. For example, if the objects of a class use a scarce resource and you need to control the number of instantiated objects you might use a class variable to keep track of the number of objects that have been created in that class.

A class variable can be implemented as a link (using `upvar` from a variable in the object's namespace to a variable in the class definition namespace). To do this we need to determine the namespace where the class is defined and the namespace of the object being created.

The code in the next example creates a new `classvar` definition that can be added to a constructor (as shown in the `limitedItem` example). Since the `classvar` command is creating `upvar` links in an object's namespace, the command needs to be evaluated inside the constructor, after a new namespace has been created, rather than as a standalone command in the class definition script.

The `classvar` procedure could be placed in any namespace (even the global scope). The `::oo::Helpers` namespace is one which TclOO maintains. A procedure placed in this namespace is available to all methods and definition scripts.

The first few lines of the `classvar` procedure introduce some new commands and concepts.

The `self` command provides information about the object that contains the code being evaluated. The `self class` command returns the name of the class being created.

Syntax: `self class`
Return the name of the class.

When the Tcl interpreter evaluates a set of code, it evaluates that code in the namespace in which the code was originally defined. The `classvar` procedure is evaluated in the `::oo::Helpers` namespace.

The `uplevel` command allows commands to be evaluated in the scope of the calling procedure.

If the `classvar` procedure tried to determine its class using `self class` without the `upvar`, the `self` command would be evaluated in the `::oo::Helpers` namespace, not in the new object's method namespace. The `self` command is only valid when evaluated within a method.

Once the name of the class is determined, the `info object namespace` command can be used to determine the class definition namespace.

Similarly, the `namespace current` command needs to be evaluated in the calling scope, not the `classvar` procedure's scope.

A fantasy game needs some magical items, but not too many. The next example shows the `classvar` procedure and a class that uses a class variable to keep track of how many items of this class have been created and will only allow two items to be made.

Example 28

Script Example

```
proc ::oo::Helpers::classvar {args} {
   # Get reference to class's namespace
   set nsCl [info object namespace [uplevel 1 {self class}]]
   set nsObj [uplevel 1 {namespace current}]
   # Link variables into local (caller's) scope
   foreach v $args {
     uplevel "my variable $v"
     upvar #0 ${nsCl}::$v ${nsObj}::$v
   }
}

oo::class create limitedItem {
  constructor {} {
      classvar quantity
      # Initialize quantity the first time a limitedItem is created
      if {![info exists quantity]} {
        set quantity 1
      }
      if {$quantity > 2} {
        error "Too many magical items"
      }
      incr quantity
  }
}

set magicRings ""
```

```
foreach ring {diamond ruby amethyst emerald garnet gold silver} {
  if {[expr {rand()}] > .5} {
    set fail [catch {limitedItem create $ring} rtn]
    if {!$fail} {
      lappend magicRings $rtn
    } else {
      puts "FAILED: $rtn"
    }
  }
}
puts "Defined $magicRings rings"
```

Script Output

```
FAILED: Too many magical items
FAILED: Too many magical items
FAILED: Too many magical items
Defined ::diamond ::ruby rings
```

10.6.2 Static Methods II

Like a class variable, a *class* (or *static*) method is one that is attached to the class definition, rather than to objects created from the class definition. Class methods are required to manipulate class variables and can be used to create factory methods if a class needs functionality that's not easily handled with a normal constructor.

As with the class variable, creating a class method requires determining the namespace in which the class definition is created. When this is known the `forward` command can be used to link the new command to the command created within the class definition namespace.

The `oo::Helpers` namespace is a holder for new commands that can be evaluated within a constructor or destructor. We can also create new commands in the `oo::define` namespace to make them available to code in the class definition, but outside a method body.

The next example shows the class variable example from the previous example, and adds a class method that will reduce the value of the `quantity` variable, allowing a new `limitedItem` to be created.

Example 29

Script Example

```
catch {limitedItem destroy}

 proc ::oo::Helpers::classvar {args} {
    # Get reference to class s namespace
    set nsCl [info object namespace [uplevel 1 {self class}]]
    set nsObj [uplevel 1 {namespace current}]
    # Link variables into local (caller's) scope
    foreach v $args {
```

```
        uplevel "my variable $v"
        upvar #0 ${nsCl}::$v ${nsObj}::$v
      }
}

proc ::oo::define::classmethod {name {args ""} {body ""}} {
   # Create the method on the class if the caller gave
   # arguments and body
   if {[llength [info level 0]] == 4} {
        uplevel 1 [list self method $name $args $body]
   }
   # Get the name of the class being defined
   set cls [lindex [info level -1] 1]
   # Make connection to private class "my" command by
   # forwarding
   uplevel forward $name [info object namespace $cls]::my $name
}

oo::class create limitedItem {
  constructor {} {
      classvar quantity
      # Initialize quantity the first time a limitedItem is created
      if {![info exists quantity]} {
        set quantity 1
      }
      if {$quantity > 2} {
        error "Too many magical items"
      }
      incr quantity
      puts "Constructor [my varname quantity] - $quantity"
  }
    classmethod decreaseCount {} {
      variable quantity
      puts "Original quantity value is: $quantity"
      incr quantity -1
      puts "New quantity value is: $quantity"
    }
}

limitedItem new
limitedItem decreaseCount
```

Script Output

```
Constructor ::oo::Obj3::quantity - 2
Original quantity value is: 2
New quantity value is: 1
```

10.6.3 Aggregated Objects That Modify the Possessor

Some subclasses have a permanent effect on the parent class, while others may only affect the parent class when their methods are evaluated. A subclass with a permanent effect can be merged into the parent class with a `superclass` or `mixin` relationship. An object that is only active when a method is invoked might be added as an aggregation.

Our fantasy hero might possess some potions. These will have an effect when they are drunk, but do not affect the character's attributes at other times.

As described in the previous chapter, an aggregation can be implemented as a list of objects within an object.

The `uplevel` and `upvar` commands can be used by methods of aggregated objects to modify variables in the parent's object.

Note that an aggregated object must have knowledge of the class that holds it if it is to directly modify variables in that class with `upvar` or `uplevel`. An alternative to this is to have the aggregated object return identifiers to the parent class and let the parent class apply them.

The next example shows a character than can acquire potions and use them and a class of potions. Three techniques are demonstrated, using `upvar`, `uplevel` and returning values that can be used by the character class to modify the State.

This character class has several methods:

`acquire`	Add an object to the list of possessions.
`FindItem`	Find an item by name and return the object identifier.
`use`	Invoke an object's `use` method.
`drink`	Invoke an object's `drink` method.
`apply`	Use the object's `apply` method to get the index and value and use these to modify that State parameter.

The potion objects have a name to identify the potion, the name of a State array index that they modify and the amount of that modification.

The potion class has four methods:

`getName`	Returns the identifier name.
`use`	Applies the potion using the `uplevel` command.
`drink`	Applies the potion using the `upvar` command.
`apply`	Returns the index and value for the parent class to use to modify the State.

Example 30

Script Example

```
# Define a class that can acquire and
# use potions

oo::class create character {
  variable State
```

```
constructor {nm} {
  array set State {defense 2 attack 2 hitpoints 5}
  set State(name) $nm
  set State(possessions) {}
}

# Return the value of an element in the State
method getParameter {param} {
  return $State($param)
}

# Add an object to the list of possessions
method acquire {item} {
  lappend State(possessions) $item
}

# Internal method to return an object with a
# name that matches the requested name

method FindItem {name} {
  foreach item $State(possessions) {
    if {[$item getName] eq $name} {
      return $item
    }
  }
}

# Invoke the requested items's use method
method use {name} {
  set item [my FindItem $name]
  $item use
}

# Invoke the requested items's drink method
method drink {name} {
  set item [my FindItem $name]
  $item drink State
}

# Invoke the requested items's apply method
method apply {name} {
  set item [my FindItem $name]
  lassign [$item apply] index value
  incr State($index) $value
}
```

```
}

# Define a potion class
oo::class create potion {

  # name:      Name of the potion.
  # attribute: The character's attribute that is affected.
  # modifier:  The amount of effect on that attribute.

  variable name attribute modifier

  constructor {nm attr mod} {
    set name $nm
    set attribute $attr
    set modifier $mod
  }

  # Return a name
  method getName {} {
    return $name
  }

  # Apply a potion using the uplevel command
  method use {} {
    uplevel [list incr State($attribute) $modifier]
  }

  # Apply a potion using the upvar command
  method drink {varName} {
    upvar $varName localVar
    incr localVar($attribute) $modifier
  }

  # Return a potion's attribute and modifier
  method apply {} {
    return [list $attribute $modifier]
  }
}

character create elmer Siegfried
elmer acquire [potion new healing hitpoints 10]
elmer acquire [potion new strength attack 2]
elmer acquire [potion new dexterity defense 3]

puts "Before healing potion: [elmer getParameter hitpoints]"
elmer use healing
puts "After healing potion: [elmer getParameter hitpoints]"
```

```
puts ""
puts "Before strength potion: [elmer getParameter attack]"
elmer drink strength
puts "After strength potion: [elmer getParameter attack]"

puts ""
puts "Before dexterity potion: [elmer getParameter defense]"
elmer apply dexterity
puts "After dexterity potion: [elmer getParameter defense]"
```

Script Output

```
Before healing potion: 5
After healing potion: 15

Before strength potion: 2
After strength potion: 4

Before dexterity potion: 2
After dexterity potion: 5
```

10.6.4 Objects That Grow and Change

Most objects have a static set of behavior throughout their lifetime. Other objects can be modeled better as having changed behaviors. For example, you might want to change the behavior of a Software Change Request when it evolves from being *proposed*, to *accepted*, to *implemented*, *tested* and finally *released*.

The change of behavior can be accomplished by changing the object's mixin class and thus changing the methods. The mixin can be changed in the mainline code using the `oo::objdefine` command, or within an object's methods using the `oo::objdefine` command and the `self` command to identify the object.

The next example uses this technique to model an insect that spends four days as a caterpillar, two days as a pupae and finally emerges as a butterfly, lives for 3 days and perishes.

Example 31

Script Example

```
oo::class create bug {
  variable days;
  constructor {} {
    # Variable days will not exist unless initialized.
    set days 4
    oo::objdefine [self] mixin larvae
  }
```

```
  method day {} {
    [self] draw
    if {[incr days -1] <= 0} {
      [self] nextPhase
    }
  }
}

oo::class create butterfly {
  method draw {} {
    # This method can access variable 'days' in parent class.
    variable days;
    puts "I am a butterfly - $days days left in this state"
  }
  method nextPhase {} {
    puts "My season in the sun is over"
  }
}

oo::class create pupae {
  # All methods can access variable 'days' in parent class.
  variable days;
  method draw {} {
    puts "I am a pupae - $days days left in this state"
  }
  method nextPhase {} {
    set days 3
    oo::objdefine [self] mixin butterfly
  }
}

oo::class create larvae {
  # All methods can access variable 'days' in parent class.
  variable days;
  method draw {} {
    puts "I am a larvae - $days days left in this state"
  }
  method nextPhase {} {
    set days 2
    oo::objdefine [self] mixin pupae
  }
}



bug create myBug
```

```
for {set i 0} {$i < 9} {incr i} {
  myBug day
}
```

Script Output

```
I am a larvae – 4 days left in this state
I am a larvae – 3 days left in this state
I am a larvae – 2 days left in this state
I am a larvae – 1 days left in this state
I am a pupae – 2 days left in this state
I am a pupae – 1 days left in this state
I am a butterfly – 3 days left in this state
I am a butterfly – 2 days left in this state
I am a butterfly – 1 days left in this state
My season in the sun is over
```

10.7 BOTTOM LINE

- TclOO can be used for rigidly defined class hierarchies or dynamically modified class environments.
- TclOO supports introspection into class inheritance and mixins, method argument and body, and the underlying namespace implementation of the TclOO package.
- The `oo::define` and `oo:objdefine` commands let you modify a class or object at runtime.
- The `info class` and `info object` commands provide information about a class or object.
- The `oo::define` and `oo::objdefine` commands can accept either a single command and options, or a Tcl script with multiple commands and options.
- When `oo::define` is used to modify a class all new and existing objects created from that class are modified.
- When `oo::objdefine` is used to modify an object, only that object is modified.
- Methods can be modified with the `method`, `filter`, `forward delete`, `export` and `rename` subcommands.
- Class hierarchy can be modified with the `superclass` and `mixin` subcommands.
- Variables can be modified with the `variable` subcommand.
- Static (or class) methods and variables can be created for TclOO classes using the `self` command or the `oo::Helpers` namespace or a `oo::define::`*NAME* procedure.
- The `nextto` command can be used to pass control directly to a given class's method.
- The `info class` command will provide information about a class.
- The `info object` command will provide information about an object.
- Information about methods and method chains is provided by the `methods`, `methodtype`, `definition`, `filters`, `forward` and `call` subcommands.

- A class hierarchy can be serialized using the info commands to create a set of commands to recreate the class structure.
- An object can be introspected to create a set of commands to recreate the object.
- A variable can be linked to a callback (like the `trace`) command with the `my varname` command.
- A method can be linked to a callback (like an `after` event) with the `self` command.
- The `self class` command returns the name of the class which contains the command.

10.8 PROBLEMS

The following numbering convention is used in all Problem sections:

Number Range	Description of Problems
100–199	Short comprehension problems review material covered in the chapter. They can be answered in a few words or a 1-5-line script. These problems should each take under a minute to answer.
200–299	These quick exercises require some thought or information beyond that covered in the chapter. They may require reading a man page or making a web search. A short script of 1–50 lines should fulfill the exercises, which may take 10–20 minutes each to complete.
300–399	Long exercises may require reading other material or writing a few hundred lines of code. These exercises may take several hours to complete.

- *100* What command will modify a TclOO class?
- *101* What command will modify a TclOO object?
- *102* What command returns the list of a classes superclasses?
- *103* What command adds a mixin to an object?
- *104* If a class has a superclass and needs to use the superclasses constructor, what command is used to transfer evaluation to the superclass constructor?
  Where is this command placed?
- *105* Is a filter method evaluated before or after a non-filter method call?
- *106* What command returns a list of an object's variables?
- *107* What command adds a mixin to all of a class's objects?
- *108* What command will add a method to a single object?
- *109* What command will add a new method to all the objects of a particular class?

- *200* What types of inheritance are better served with a superclass?
- *201* What types of inheritance are better served with a mixin?
- *202* When is it appropriate to modify an object's class?
- *203* Write a class for an elevator. It should have methods to `moveToFloor`, `openDoor`, `closeDoor` and `requestFloor`, and variables to track the requested floors, the current floor, the state of the door and whether or not the elevator is moving.
- *204* Write a base class with a variable named `State` and a method named `display` which will display the State contents. Create a derived class that has methods to modify the `State` variable. Then create two separate objects and confirm that they each have their own State variables.
- *205* For the previous exercise, extract the base class's `display` method and put it into a separate class. Mix this class into the derived class and confirm that everything is working correctly.
- *206* Use the `oo::objdefine` command to add a filter to the previous class that will only allow the `State` variable to accept integer values.
- *207* Write a class that has methods to perform addition or subtraction. Add a filter to limit the inputs and returns to positive integers. Trying to subtract 10 from 2 should return `0` instead of `-8`
- *208* Modify the `character` class in section 10.6.3 so that a potion is destroyed after being consumed.
- *300* Write a class that opens a file in write mode and has a method that will add a timestamp to a message and then write the timestamp and message to that file. Add a method to this class that will write a *heartbeat* message to the file every 10 seconds. Create an object from this class and confirm that requested messages and heartbeat messages all end up in the file.
- *301* Modify the class described in exercise 203 to use the `trace` command to confirm that no illegal floors are added to the list of requested floors. Any number less than 0, or the number 13 are illegal floors. If an illegal floor is requested, remove it from the list of requested floors.
- *302* Create a class with a class variable that counts the number of objects of this class that have been created. The `destructor` will reduce this count. Add a class method to report the count. Create and destroy objects to confirm that the count is correct.
- *303* Create two classes, `author` and `book`. The `author` class will have a single variable `name`. The `book` class will have two variables `title` and `author`. The `author` variable will contain the name of the object that has the author's name.
- *304* Create a class named `library` that uses the aggregation technique to contain several books. Add methods so that a book can be searched for by author or title.

CHAPTER

11 Introduction to Tk Graphics

Everyone knows the fun part about computer programming is the graphics. The Tk graphics package lets a Tcl programmer enjoy this fun too. Tk is a package of graphics widgets that provides the tools to build complete graphics applications. Tk supports the usual GUI widgets (such as buttons and menus), complex widgets (such as color and file selectors), and data display widgets (such as an editable text window and an interactive drawing canvas).

The user interaction widgets include buttons, menus, scrollbars, sliders, pop-up messages, and text entry widgets. A script can display either text or graphics with the text and canvas widgets. Tk provides three algorithms, for controlling the layout of a display, and a widget for grouping widgets.

Finally, if none of the standard widgets do what you want, the Tk package supports low-level tools at both the script and C API levels to build your own graphical widgets. You can create either simple standalone widgets (similar to those provided by Tcl), or you can combine the simple widgets into complex widgets, sometimes called *compound widgets*, or *megawidgets*.

The Tk widgets are very configurable, with good default values for most settings. For many widgets, you can set your own background colors, foreground colors, and border widths. Some classes of widgets have special-purpose options such as font, line color, and behavior when selected. This chapter discusses some of the more frequently used options. Consult the on-line manual pages with your installation for a complete list of options supported with your version of Tcl/Tk.

Like most GUI packages, the Tk package is geared toward event-driven programming. If you are already familiar with event-driven programming, skip to Section 11.1. If not, the following paragraphs will give you a quick overview of event-driven programming.

Using traditional programming, your program watches for something to happen and then reacts to it. For instance, user interface code resembles the following.

```
while {![eof stdin]} {
  gets command
  switch command {
    "cmd1" {doCmd1}
    "cmd2" {doCmd2}
    default {unrecognized command}
  }
}
```

The user interface code will wait until a user types in a command and will then evaluate that command. Between commands, the program does nothing. While a command is being evaluated, the user interface is inactive.

Tcl/Tk: A Developer's Guide. DOI: 10.1016/B978-0-12-384717-1.00011-7

With event-driven programming there is an event loop that watches for events, and when an event occurs it invokes the procedure that was defined to handle that event. This event may be a button press, a clock event, or data becoming available on a channel. Whenever the program is not processing a user request, it is watching for events. In this case, the user interface pseudo-code resembles the following.

```
REGISTER exit           TO BE INVOKED UPON exitCondition
REGISTER parseUserInput TO BE INVOKED UPON CarriageReturn
REGISTER processButton  TO BE INVOKED UPON ButtonPress
```

With Tk, the details of registering procedures for events and running an event loop are handled largely behind the scenes. At the time you create a widget, you can register a procedure to be evaluated whenever the widget is selected, and the event loop simply runs whenever there is no other processing going on.

11.1 CREATING A WIDGET

The standard form for the command to create a Tk widget is as follows.

Syntax: `WidgetClass` *`widgetName requiredArguments ?options?`*

`WidgetClass`	A widget type, such as `button`, `label`, `scrollbar`, or `tk_chooseColor`.
`widgetName`	The name for this widget. Must conform to the Tk naming conventions described in the next section.
`requiredArguments`	Some Tk widgets have required arguments.
`?options?`	Tk widgets support a large number of options that define the fonts, colors, actions to be taken when the widget is selected, etc. As with the other command options, these are defined as `-keyword value` pairs.

For example, this line:

```
label .hello –text "Hello, World."
```

will create a label widget named *`.hello`*, with the text `Hello, World`. As part of creating a widget, Tcl creates a command with the same name as the widget. After the widget is created, your script can interact with the widget via this command. This is similar to the way the objects are created with TclOO. Tk widgets support commands for setting and retrieving configuration options, querying the widget for its current state, and other widget-specific commands such as scrolling the view, selecting items, and reporting selections. The `label` in the preceding example will appear in the default window. When the `wish` interpreter starts, it creates a default window for graphics named ".".

The various Tk window creation commands (`button`, `label`, etc.) allocate machine resources and create the internal structures for a window, but do not display the window. In order to display a widget, a script needs to describe where and how the widget is to be displayed. This is done with a *geometry manager*.

Tk supports three geometry manager commands: `place`, `pack` and `grid`. The `grid` command is one of the easiest to use.

This command will display the `.hello` label:

```
grid .hello
```

11.2 CONVENTIONS

There are a few conventions for widgets supported by Tk. These conventions include naming conventions for widgets and colors, and the conventions for describing screen locations, sizes, and distances.

11.2.1 Widget Naming Conventions

The Tk graphics widgets are named in a tree fashion, similar to a file system or the naming convention for namespaces. Instead of the slash used to separate file names, widget and window names are separated by periods. Thus, the root window is named ".", and a widget created in the root window could be named `.widget1`.

A widget or window name must start with a period and must be followed by a label. The label may start with a lowercase letter, digit, or punctuation mark (except a period). After the first character, other characters may be uppercase or lowercase letters, numbers, or punctuation marks (except periods). It is recommended that you use a lowercase letter to start the label.

Some widgets can contain other widgets. In that case, the widget is identified by the complete name from the top dot to the referenced widget. Tk widgets must be named by absolute window path, not relative. Thus, if widget `one` contains widget `two`, which contains widget `three`, you would access the last widget as `.one.two.three`.

A widget path name must be unique. You can have multiple widgets named `.widget1` if they are contained in different widgets (e.g., `.main.widget1` and `.subwin.widget1` are two different widgets).

11.2.2 Color Naming Conventions

Colors may be declared by name (red, green, lavender, and so on) or with a hexadecimal representation of the red/green/blue intensities. The hexadecimal representation starts with the # character, followed by 3, 6, 9, or 12 hexadecimal digits. The number of digits used to define a color must be a multiple of 3. The number will be split into three hexadecimal values, with an equal number of digits in each value, and assigned to the red, green, and blue color intensities in that order. The intensities range from 0 (black) to 0xF, 0xFF, 0xFFF, or 0xFFFF (full brightness), depending on the number of digits used to define the colors. Thus, `#f00` (bright red, no green, no blue) creates deep red, `#aa02dd` (medium red, dim green, medium blue) creates purple, and `#ffffeeee0000` (bright red, bright green, no blue) creates a golden yellow.

11.2.3 Dimension Conventions

The size or location of a Tk object is given as a number followed by an optional unit identifier. The numeric value is maintained as a floating-point value. Even pixels can be described as fractions. If there is no unit identifier, the numeric value defaults to pixels. You can describe a size or location in

inches, millimeters, centimeters, or points (1/72 of an inch) by using the unit identifiers shown in the following examples.

Unit Identifier	Meaning
`15.3`	15.3 pixels
`1.5i`	1-1/2 inches
`10m`	10 millimeters (1 cm)
`1.3c`	1.3 centimeters (13 mm)
`90p`	90 points (1-1/4 inches)

11.3 COMMON OPTIONS

The Tk widgets support many display and action options. Fortunately, these options have reasonable default values associated with them. Thus, you do not need to define every option for every widget you use.

The following are some common options supported by many Tk widgets. They are described here, rather than with each widget that supports these options. Widget-specific options are defined under individual widget discussions. The complete list of options and descriptions is found in the Tcl/Tk on-line documentation under *options.*

`-background` *color*	The background color for a widget.
`-borderwidth` *width*	The width of the border to be drawn around widgets with 3D effects.
`-font` *fontDescriptor*	The font to use for widgets that display text. Fonts are further discussed in Chapter 12, in regard to the `canvas` widget.
`-foreground` *color*	The foreground color for a widget. This is the color in which text will be drawn in a text widget or on a label. It accepts the same color names and descriptions as `-background`.
`-height` *number*	The requested height of the widget in pixels.
`-highlightbackground` *color*	The color rectangle to draw around a widget when the widget does not have input focus.
`-highlightcolor` *color*	The color rectangle to draw around a widget when the widget has input focus.
`-padx` *number* `-pady` *number*	The `-padx` and `-pady` options request extra space (in pixels) to be placed around the widgets when they are arranged in another widget or in the main window.
`-relief` *condition*	The 3D relief for this widget: `condition` may be `raised`, `sunken`, `flat`, `ridge`, `solid`, or `groove`.

`-text` *text*	The text to display in this widget.
`-textvariable` *varName*	The name of a variable to associate with this widget. The content of the variable will reflect the content of the widget. For example, the `textvariable` associated with an `entry` widget will contain the characters typed into the `entry` widget.
`-width` *number*	The requested width of the widget in pixels.

11.4 DETERMINING AND SETTING OPTIONS

The value of an option can be set when a widget is created, or it can be queried and modified after the widget is created using the `cget` and `configure` commands. The `cget` subcommand will return the current value of a widget option.

Syntax: *widgetName* `cget` *option*

Return the value of a widget option.

widgetName	The name of this widget.
`cget`	Return the value of a single configuration option.
option	The name of the option to return the value of.

Example 1

Script Example

```
button .exit_button –text "QUIT" –command exit
puts "The exit button text is: [.exit_button cget –text]"
puts "The exit button text color is: [.exit_button cget –foreground]"
```

Script Output

```
The exit button text is: QUIT
The exit button text color is: Black
```

The `configure` subcommand will return the value of a single configuration option, return all configuration options available for a widget, or allow you to set one or more configuration options.

Syntax: *widgetName* `configure` *?opt1? ?val1? ... ?optN? ?valN?*

widgetName	The widget being set/queried.
`configure`	Return or set configuration values.
?opt?*	The first option to set/query.
?val?*	An optional value to assign to this option.

If `configure` is evaluated with a single option, it returns a list consisting of the option name, the name and class that can be used to define this option in the windowing system resource file, the default value for the option, and the current value for the option.

Example 2

Script Example

```
button .exit_button –text "QUIT" –command exit
puts "The exit button text is: [.exit_button configure –text]"
puts "The exit button text color is:"
puts "    [.exit_button configure –foreground]
```

Script Output

```
The exit button text is: –text text Text {} QUIT
The exit button text color is:
    –foreground foreground Foreground Black Black
```

If `configure` is evaluated with no options, a list of lists of available option names and values is returned.

Example 3

Script Example

```
button .exit_button -text "QUIT" -command exit
puts [.exit_button configure]
```

Script Output

```
{–activebackground activeBackground Foreground #ececec #ececec}
{–activeforeground activeForeground Background Black Black}
{–anchor anchor Anchor center center}
{–background background Background #d9d9d9 #d9d9d9}
{–bd –borderwidth}
{–bg –background}
{–bitmap bitmap Bitmap {} {}}
{–borderwidth borderWidth BorderWidth 2 2}
{–command command Command {} exit}
{–cursor cursor Cursor {} {}}
{–default default Default disabled disabled}
{–disabledforeground disabledForeground DisabledForeground #a3a3a3 #a3a3a3}
{–fg –foreground}
{–font font Font {Helvetica –12 bold} {Helvetica –12 bold}
{–foreground foreground Foreground Black Black}
{–height height Height 0 0}
{–highlightbackground highlightBackground HighlightBackground #d9d9d9 #d9d9d9}
{–highlightcolor highlightColor HighlightColor Black Black}
{–highlightthickness highlightThickness HighlightThickness 1 1}
{–image image Image {} {}}
{–justify justify Justify center center}
{–padx padX Pad 3m 3m}
```

```
{-pady padY Pad 1m 1m}
{-relief relief Relief raised raised}
{-state state State normal normal}
{-takefocus takeFocus TakeFocus {} {}}
{-text text Text {} QUIT}
{-textvariable textVariable Variable {} {}}
{-underline underline Underline -1 -1}
{-width width Width 0 0}
{-wraplength wrapLength WrapLength 0 0
```

If `configure` is evaluated with `option value` pairs, it will set the options to the defined values. The following example creates a button that uses the `configure` command to change its label when it is clicked. The `-command` option and `grid` command are discussed in detail later in this chapter.

Example 4

Script Example

```
set clickButton [button .b1 -text "Please Click Me" \
    -command {.b1 configure -text "I've been Clicked!"}]
grid $clickButton
```

Script Output

11.5 THE BASIC WIDGETS

These basic widgets, as follows, are supported in the Tk 8.0 and later distributions.

`button`	A clickable button that may evaluate a command when it is selected.
`radiobutton`	A set of on/off buttons and labels, one of which may be selected.
`checkbutton`	A set of on/off buttons and labels, many of which may be selected.
`menubutton`	A button that displays a scrolldown menu when clicked.
`menu`	A holder for menu items. This is attached to a `menubutton` .
`listbox`	Creates a widget that displays a list of options, one or more of which may be selected. The listbox may be scrolled.
`entry`	A widget that can be used to accept a single line of textual input.
`label`	A widget with a single line of text in it.
`message`	A widget that may contain multiple lines of text.
`text`	A widget for displaying and optionally editing large bodies of text.
`canvas`	A drawing widget for displaying graphics and images.

`scale`	A slider widget that can call a procedure when the selected value changes.
`scrollbar`	Attaches a scrollbar to widgets that support a scrollbar. Will call a procedure when the scrollbar is modified.
`frame`	A container widget to hold other widgets.
`labelframe`	A container widget with an optional border and label to hold other widgets.
`toplevel`	A window with all borders and decorations supplied by the Window manager.

Release 8.5 added these basic widgets:

`panedwindow`	Resizable windows that can hold other widgets.
`spinbox`	A widget to select one element from a list by cycling through the list.

11.6 INTRODUCING WIDGETS: `label`, `button`, AND `entry`

The `label`, `button`, and `entry` widgets are the easiest widgets to use. The `label` widget simply displays a line of text, the `button` widget evaluates a Tcl script when it is selected, and the `entry` widget accepts user input.

All Tcl widget creation commands return the name of the widget they create. A good coding technique is to save that name in a variable and access the widget through that variable rather than hard-coding the widget names in your code.

11.6.1 The `label` Widget

Syntax: `label` *`labelName ?option1? ?option2? ...`*

Create a `label` widget.

`labelName`	The name for this widget.
`option`	Valid options for `label` include:

`-font` *`fontDescriptor`*	Defines the font to use for this display. Font descriptors are discussed in the next chapter.
`-textvariable` *`varName`*	The name of a variable that contains the display text.
`-text` *`displayText`*	Text to display. If `-textvariable` is also used, the variable will be set to this value when the widget is created.

Note that the options can be defined in any order in the following example.

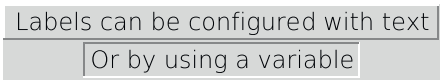
Example 5

Script Example

```
set txt [label .la –relief raised –text \
    "Labels can be configured with text"]
grid $txt
set var [label .lb –textvariable label2Var –relief sunken]
```

```
grid $var
set label2Var "Or by using a variable"
```

Script Output



11.6.2 The `button` Widget

The `button` widget will evaluate a script when a user clicks it with the mouse. The script may be any set of Tcl commands, including procedure calls. By default, a script attached to a widget will be evaluated in the global scope. The scope in which a widget's `-command` script will be evaluated can be modified with the `namespace current`, `namespace code` or the TclOO `self` command.

Syntax: `button` *`buttonName ?option1? ?option2? ...`*

Create a `button` widget.

`buttonName`	The name to be assigned to the widget.
`?options?`	Valid options for `button` include:
`-font` *`fontDescr`*	Defines the font to use for this button. Font descriptors are discussed in the next chapter.
`-command` *`script`*	A script to evaluate when the button is clicked.
`-text` *`displayTxt`*	The text that will appear in this button. A newline character (`\n`) can be embedded in this text to create multi-line buttons.

The example shows two sets of code, one in global scope and one in a namespace. Each of these generate the same GUI with the same behavior. Note the use of `namespace current`, `namespace code` and the `-textvariable` option in the second code example.

Example 6

Script Example (Global Scope)

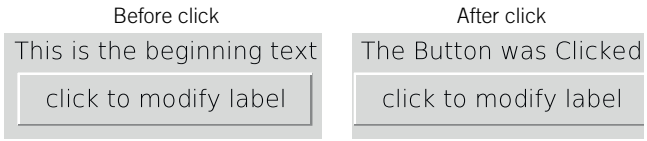
```
set myLabel [label .l1 -text "This is the beginning text"]
set myButton [button .b1 -text "click to modify label"\
    -command \
    "$myLabel configure -text {The Button was Clicked}"]
grid $myLabel
grid $myButton
```

Script Example (Within a namespace)

```
namespace eval demo {
  variable labelText "This is the beginning text"
  set myLabel [label .l1 \
      -textvariable [namespace current]::labelText]
```

```
  set myButton [button .b1 -text "click to modify label"\
      -command [namespace code \
          {set labelText {The Button was Clicked}}]]
  grid $myLabel
  grid $myButton
}
```

Script Output



11.6.3 The `entry` Widget

The `entry` widget allows a user to enter a string of data. This data can be longer than will fit in the widget's displayed area. The widget will automatically scroll to display the last character typed and can be scrolled back and forth with the arrow keys or by attaching the widget to a scrollbar (scrollbars are discussed later in this chapter). The `entry` widget can be configured to reflect its content in a variable, or your script can query the widget for its content.

Syntax: `entry` *`entryName ?options?`*

Create an `entry` widget.

`entryName`	The name for this widget.
`?options?`	Valid options for `label` include:
`-font` *`fontDescriptor`*	Defines the font to use for this display. Font descriptors are discussed in Section 10.4.5.
`-textvariable` *`VarName`*	The variable named here will be set to the value in the entry widget.
`-justify` *`side`*	Justify the input data to the left margin of the widget (for entering textual data) or the right margin (for entering numeric data). Values for this option are `right` and `left`.

Example 7

Script Example

```
set input [entry .e1 -textvariable inputval]
set action [button .b1 -text "Convert to UPPERCASE" \
    -command {set inputval [string toupper $inputval]} ]
grid $input
grid $action
```

Script Output

Note that the button command in the Section 11.6.2 example was enclosed within quotes, whereas the command in the Section 11.6.3 example was enclosed within curly braces. In Section 11.6.2 we want the variable `theLabel` to be replaced by the actual name of the widget when the `button` creation command is being evaluated. The command being bound to the button is:

```
.la configure -text {The Button was Clicked}
```

If our script changes the content of the variable `theLabel` after creating the button, the command will still configure the original widget it was linked to (.la), not the window now associated with `$thelabel`. In Section 11.6.3, we do not want the substitution to occur when the button is created; we want the substitution to occur when the button is clicked. When the command is placed within brackets, the command bound to the button is:

```
set inputval [string toupper $inputval]
```

The variable `$inputval` will be replaced by the content of the `inputval` variable when the button is selected. If the command in Section 11.6.3 were enclosed in quotes, the command bound to the button would have been evaluated, `$inputval` would be replaced by the current value (an empty string), and the command `[string toupper ""]` would be evaluated and replaced by an empty string. The command bound to the button would be:

```
set inputval ""
```

In this case, clicking the button would cause the entry field to be cleared. The script associated with a button can be arbitrarily long and complex. As a rule of thumb, if there are more than three or four commands in the script, or if you are mixing variables that need to be substituted at widget creation time and evaluation time, it is best to create a procedure for the script, and invoke that procedure with the button `-command` option. For example, you can make an application support multiple languages by using an associative array for a translation table.

Example 8

Script Example

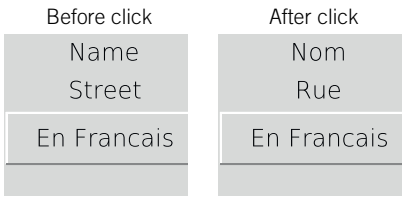
```
array set english {Nom Name Rue Street}
array set french {Name Nom Street Rue}

grid [label .name -text Name]
grid [label .street -text Street]

button .translate -text "En Francais" -command {
  foreach w {.name .street} {
    $w configure -text $french([$w cget -text])
```

```
    }
  }
  grid .translate
```

Script Output



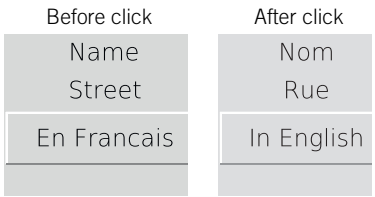
You can also make the button change text and command when it is clicked, to translate back to English, but the command starts to get unwieldy.

Example 9

Script Example

```
array set english {Nom Name Rue Street "In English" "En Francais"}
array set french {Name Nom Street Rue "En Francais" "In English"}
grid [label .name -text Name]
grid [label .street -text Street]
button .translate -text "En Francais" -command {
  if {[string match [.translate cget -text] "En Francais"]} {
    foreach w {.name .street .translate} {
      $w configure -text $french([$w cget -text])
    }
  } else {
    foreach w {.name .street .translate} {
      $w configure -text $english([$w cget -text])
    }
  }
}
grid .translate
```

Script Output



Using a `procedure` instead of coding the translation in-line simplifies the code and makes the application more maintainable.

Example 10

Script Example

```
proc translate {widgetList request} {
  if {[string match "En Francais" $request]} {
    upvar #0 french table
  } else {
    upvar #0 english table
  }

  foreach w $widgetList {
    $w configure –text $table([$w cget –text])
  }
}
array set english {Nom Name Rue Street}
array set french {Name Nom Street Rue}
grid [label .name –text Name]
grid [label .street –text Street]
button .convert –text "En Francais" –command \
    {translate {.convert .name .street} [.convert cget –text]}
grid .convert
```

11.6.4 Using Namespaces or TclOO with Widgets

The label attached to a widget with the `-textvariable` option or the script attached to a button with the `-command` option is mapped to items in the global scope. If the variable exists in a namespace or TclOO object, or the script should be evaluated within a namespace or as an object's method, you need to add some extra information to let the interpreter know where the script to be evaluated exists.

Using Namespace Scope with a Widget

Tcl provides two hooks for accessing code within a namespace.

- `namespace current` Returns the fully qualified path to the current namespace.
- `namespace code` Wraps a script so that it will be evaluated in the current namespace.

It's easiest to link a `-textvariable` to a label or a procdure to a button with the `namespace current` command.

Syntax: `namespace current`
Returns the fully qualified name of the current namespace.

Example 11

Script Example

```
namespace eval buttonSpace {
  proc makeButton {} {
```

```
    button .b –text "Button" \
        –command [namespace current]::buttonProc
    grid .b
  }
  proc buttonProc {} {
    puts "You clicked button .b"
  }
}

buttonSpace::makeButton
```

Script Output


```
You clicked button .b
```

If you need to attach a more complex script to a button's `-command` option, the `namespace code` command may be more appropriate. As a rule, if the script attached to a button is complex, it should probably be a procedure.

In the next example, the command to modify the label's textvariable is evaluated in the `buttonSpace` namespace, but not within any procedure.

Syntax: `namespace code` *`script`*

Wrap a script so that it will be evaluated in the current namespace.

`script` A script to be evaluated in the current namespace at some future time.

Example 12

Script Example

```
namespace eval guiDemo {

  variable varName "Original Value"

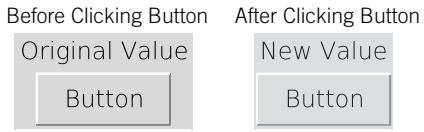
  proc makeButton {} {
    label .l –textvar [namespace current]::varName

    button .b –text "Button" \
        –command [namespace code {set varName "New Value"}]

    grid .l
    grid .b
  }
}

guiDemo::makeButton
```

Script Output



Using TclOO with a Widget

TclOO supports commands similar to the `namespace` commands to attach an object variable or method to a Tk widget. The two commands are:

- `my varname` Returns the fully qualified path to a variable.
- `self` Returns the name of the the current object.

The `my varname` command is used to link an object's variable to a Tk widget with the `textvariable` command. This is similar to the `namespace current` command.

Syntax: `my varname` *`variableName`*

Returns the full namespace path to a named variable.

`variableName` The simple variable name.

Objects don't normally need to know how they are named outside the object. An object can access other methods with the `my` *`methodName`* construct. However, in order to register a callback, the object needs to be able to register its name with the callback.

The `self` command returns the name of the current object. This is used to invoke a method in the current object without needing to know the name of the object. The `self` command is used to register an object method using `-command` option for a button.

The next example shows how to map an object variable to a label and an object method to a button.

Example 13

Script Example

```
oo::class create guiDemo {
  variable objectVar

  constructor {} {
    set objectVar "Before button is clicked"
    label .l –textvar [namespace current]::objectVar

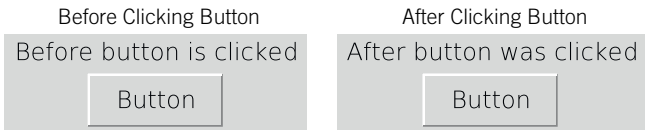
    button .b –text "Button" \
        –command "[self] changeObjectVar"

    grid .l
    grid .b
  }
  method changeObjectVar {} {
    set objectVar "After button was clicked"
```

```
    }
}

guiDemo new
```

Script Output



11.7 APPLICATION LAYOUT: GEOMETRY MANAGERS AND CONTAINER WIDGETS

When Tk creates a new widget, it allocates machine resources and creates the internal structures for a window, but does not display the widget. The widget is displayed when a *geometry manager* command is invoked for the widget.

Tk supports three geometry manager commands.

`place`	Low-level control with support for placing windows at specific locations.
`pack`	High-level control that places windows in the largest open space.
`grid`	High-level control that places windows in a grid arrangement.

Simple applications can be created using just the top level windows and a geometry manager. More complex (i.e., most) applications require some level of grouping windows.

Tk supports five container widgets that can be used to group windows.

`toplevel`	A new toplevel window that can contain other widgets.
`frame`	A simple container that holds other widgets within a toplevel window.
`labelframe`	A container window with optional label and outline that holds other widgets within a toplevel window.
`panedwindow`	A set of container windows that can be resized to show more or less of the windows they contain.
`ttk::notebook`	A tabbed notebook container widget.

11.7.1 Container Widgets: `frame, labelframe, panedwindow`

You frequently need to group a set of widgets when you are designing a display. For instance, you will probably want all buttons placed near each other, and all results displayed as a group.

The Tk container widgets range from the basic `frame` that holds other widgets to the `toplevel` that provides a complete new application level window.

You can place any widget (including more container widgets) into any of these container widgets, nesting the window hierarchy as deeply as the application requires.

The `frame` Widget

The basic Tk widget for grouping other widgets is `frame`. The `frame` widget is a rectangular container widget that groups other widgets. You can define the height and width of a `frame` or let it automatically size itself to fit the content.

Syntax: `frame` *`frameName ?options?`*

Create a `frame` widget.

`frameName` The name for the `frame` being created.

`?options?` The options for a `frame` include:

`-height` *`numPixels`*	Height (using units, as described in Section 11.2.3).
`-width` *`numPixels`*	Width (using units, as described in Section 11.2.3).
`-background` *`color`*	The color of the background (see Section 11.2.2).
`-relief` *`value`*	Defines how to draw the widget edges. Can make the `frame` look raised, sunken, outlined, or flat. The *value* may be one of `sunken`, `raised`, `ridge`, or `flat`. The default is `flat`, to display no borders.
`-borderwidth` *`width`*	Sets the width of the decorative borders. The `width` value may be any valid size value (as described in Section 11.2.3).

A display can be divided into frames by functionality. For example, an application that interfaces with a database would have an area for the user to enter query fields, an area with the database displays, an area of buttons to generate searches, and so on. This could be broken down into three primary frames: `.entryFrame`, `.displayFrame`, and `.buttonFrame`. Within each of these frames would be the other widgets, with names such as `.entryFrame.userName`, `.displayFrame.securityRating`, and `.buttonFrame.search`.

The next example shows a common application layout with a basic information frame at the top, a set of action buttons, a main area with the application and a bottom status line. The frame names are assigned to variables, and the variable names are used to create widgets within the frames. This technique makes it easier to pick up sections of a GUI and rearrange them.

Example 14

Script Example

```
set infoFrame [frame .info]
set buttonFrame [frame .buttons]
set mainFrame [frame .main -relief solid -borderwidth 2]
set statusFrame [frame .status]

set w [label $infoFrame.name -text "Sample Application" \
    -font {arial 16 bold}]
grid $w
set w [label $infoFrame.version -text "Revision 1.0"]
grid $w
```

```
foreach buttonName {File Edit Help} {
  lappend buttonList [button $buttonFrame.b_$buttonName \
      -text $buttonName -command "perform_$buttonName"]
}

grid {*}$buttonList

foreach id {Name Street City State} {
  set w1 [label $mainFrame.l_$id -text $id:]
  set w2 [entry $mainFrame.e_$id -textvariable State($id) \
      -background white]
  grid $w1 $w2
}

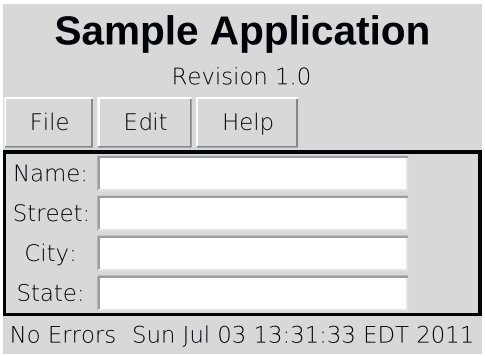
set w1 [label $statusFrame.status -textvariable status]
set w2 [label $statusFrame.time -textvariable time]

grid $w1 $w2

set status "No Errors"
set time [clock format [clock seconds]]

grid $infoFrame
grid $buttonFrame -sticky w
grid $mainFrame -sticky ew
grid $statusFrame
```

Script Output



The previous example uses the `-relief` option to make the main frame stand out from the other frames. You can modify the background color and the outline for frames, but there is very little else that you can do to affect the appearance of an application with just a frame. The `labelframe`, `panedwindow` and `ttk::notebook` widgets allow more control over the applications appearance.

The `labelframe` Widget

The `labelframe` widget provides the functionality of the `frame` widget with a border and a label. The labelframe can be used to hold a single widget, or a set of widgets.

Syntax: `labelframe` *`frameName ?options?`*

Create a `labelframe` widget.

`frameName` The name for the `labelframe` being created.

`?options?` The options for a `labelframe` include:

`-background` *`color`*	A background color, or an empty string. If the background is an empty string the windows *behind* the frame can show through.
`-text` *`string`*	Text to display as this widget's label.
`-labelanchor` *`location`*	Where to display the label. May be one of: `nw`, `n`, `ne`, `e`, `se`, `s`, `sw`, or `w`.

The next example revisits the previous example, except that instead of label/entry pairs for the input fields it uses some labelframes.

Example 15

Script Example

```
set infoFrame [frame .info]
set buttonFrame [frame .buttons]
set mainFrame [frame .main –relief solid –borderwidth 2]
set statusFrame [frame .status]

set w [label $infoFrame.name –text "Sample Application" \
    –font {arial 16 bold}]
grid $w
set w [label $infoFrame.version –text "Revision 1.0"]
grid $w

foreach buttonName {File Edit Help} {
  lappend buttonList [button $buttonFrame.b_$buttonName \
      –text $buttonName –command "perform_$buttonName"]
}

grid {*}$buttonList

foreach id {Name Street City State} {
  set w [labelframe $mainFrame.l_$id –text $id:]
  set w2 [entry $w.e_$id –textvariable State($id) \
      –background white ]
  grid $w
  grid $w2
```

```
}

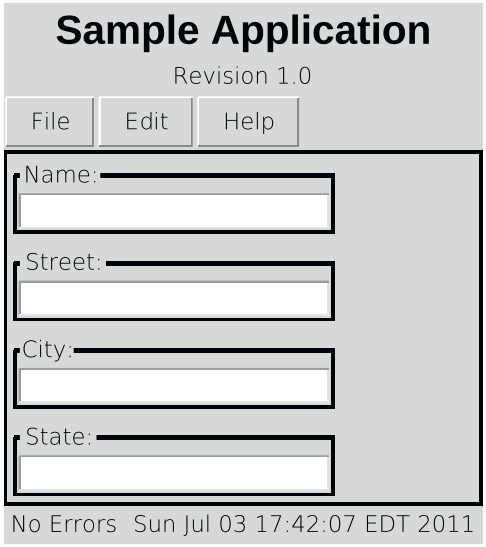
set w1 [label $statusFrame.status -textvariable status]
set w2 [label $statusFrame.time -textvariable time]

grid $w1 $w2

set status "No Errors"
set time [clock format [clock seconds]]

grid $infoFrame
grid $buttonFrame -sticky w
grid $mainFrame -sticky ew
grid $statusFrame
```

Script Output



The `ttk::notebook` Widget

Some applications have multiple windows in which the individual window is a self-contained entity. The tabbed browsers are a common example of this.

The `notebook` widget was introduced with the Themed ToolKit widgets in the 8.5 release. The ttk widgets provide a more native look and feel on Mac and Windows platforms and add several new widgets to Tk. Unlike the standard Tk widgets, the ttk widgets are developed in a private `ttk` namespace.

Syntax: `ttk::notebook` *`widgetName ?-option value?`*

Creates a notebook widget which can contain multiple tabs.

`widgetName` The name of this widget following the normal Tk naming conventions.

?-option? Options to control how the notebook appears.

?-height? If greater than 0, this specifies the height of the notebook page area. The default is to use the largest height of windows that have been added to the notebook.

When you create a notebook, Tk also creates a new command with the same name as the notebook. You can use this command to control the notebook. The notebook command supports two subcommands:

`add` Add a new tab as the last tab in the set of tabs.

`insert` Add a new tab at a specific location in the set of tabs.

Syntax: *notebookName* `add` *window ?-option value?*
notebookName `insert` *position window ?-option value?*
Add a new tab to the set of tabs.

window The window to add to the notebook.

position For insert, this is the location in the list of tabs to add the new tab.

-option Options to control how the tab is added. There are several options including:

?-text? The text to display in the tab.

?-image? An image to display in the tab.

?-sticky? One or more of `n`, `e`, `s` or `w`. This specifies how the new window should be placed within the notebook page. The default is to center the new window.

To use the notebook:

1. Create a notebook widget with the `ttk::notebook` command.
2. Create a frame to hold the widgets that will be shown in a tab.
3. Add the frame to the notebook, supplying some text to display in the tab.
4. Create widgets within the frame that was added to the notebook.

If a notebook page will only have a single widget displayed (perhaps a text or canvas widget), you can skip creating the frame. For notebook pages with multiple widgets, it's best to provide a `frame` or `labelframe` to hold them.

The next example shows building the `mainFrame` portion of the previous example for more information. The nested lists are used to create the three tabs and the widgets inside the notebook pages. Rather than try to create human-readable names for the widgets, the `unique` value is incremented each time a new window is created. This technique is useful when a GUI is being constructed from a set of data, rather than coding each window name individually.

Example 16

Script Example

```
# Define a unique number to create an undefined
# number of unique widget names
```

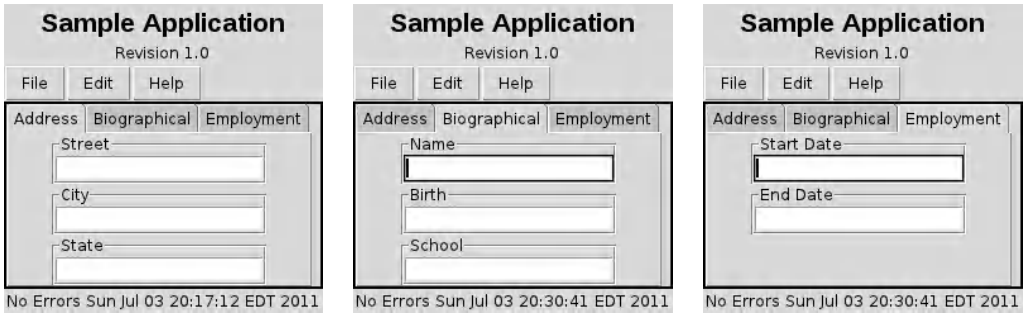
```
set unique 0

# Create the notebook and grid it in the main frame
set note [::ttk::notebook $mainFrame.n]
grid $note

# Nested lists for the 3 tabs and the elements within
#  each tab.

foreach txt      {Address Biographical Employment}  \
        elements { {Street City State}
                   {Name  {Birth} {School}}
                   {{Start Date} {End Date}} } {
  set w [frame $note.f_[incr unique]]
  foreach el $elements {
    set w1 [labelframe $w.lf_[incr unique] -text $el]
    set w2 [entry $w1.e_[incr unique] -textvar State($el)]
    grid $w1
    grid $w2
  }
  $note add $w -text $txt -sticky n
}
```

Script Output



The `panedwindow` Widget

A common GUI style is to have one or more windows that can be resized at the expense of other windows. This is particularly common with applications that have a large canvas or text widget and another window of information about the primary window. The TclTutor application on the website has three resizeable text windows to allow the user to decide how much screen real estate they want to devote to the lesson text, example or example output.

The `panedwindow` widget is a container widget that can show multiple windows with a space between each main window for a *resize* sash. When the mouse passes over the resize area the cursor is modified to show that resizing is active. When the cursor shows resizing is active, a left-mouse-drag operation will resize the two windows adjacent to the sash area.

As with the `ttk::notebook` widget, the widgets added to the `panedwindow` widget can be frames containing more widgets or single widgets such as a text widget, canvas or listbox.

Syntax: `panedwindow` *`windowName ?-option value?`*

Create a `panedwindow` widget that can hold multiple widgets.

`-option` The `panedwindow` widget supports several options including:

- `-orient` May be `vertical` or `horizontal`. The default is to arrange the widgets horizontally.
- `-showhandle` If true, a handle is displayed between the resizable windows. Default is false.
- `-handlepad` A numeric value to define how far from the top or left edge of the sash area to draw the handle.

The `panedwindow` has an `add` command for adding windows to the pane. This command defaults to adding a window in the last position, but can also insert a window before or after a currently displayed window.

Syntax: *`panedWindowName`* `add` *`widgetName ?-option value? ?widgetName...?`*

Add one or more widgets to a `panedwindow` widget.

`panedWindowName` The name of the paned window to have new windows added. This is the value returned by the `panedwindow` command.

`-option` The `add` command supports several options including:

- `-after` *`widgetName`* Place the new window after the named window.
- `-before` *`widgetName`* Place the new window before the named window.

The next example uses a `panedwindow` instead of three notebook tabs to display the three input areas. By default, each of the entry areas would be fully displayed, but the application window has been shrunk to show the sash being used to resize the windows.

Example 17

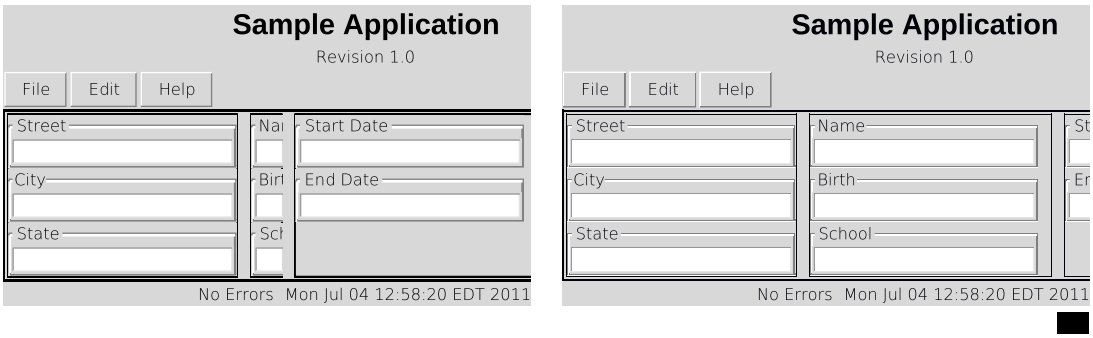
```
# Create the paned window and grid it in the main frame
set pw [panedwindow $mainFrame.n]
grid $pw

# Nested lists for the 3 tabs and the elements within
#  each tab.

foreach txt      {Address Biographical Employment}  \
        elements { {Street City State}
                   {Name  {Birth} {School}}
                   {{Start Date} {End Date}} } {
  set w [frame $pw.f_[incr unique] -relief solid -borderwidth 1]
```

```
    foreach el $elements {
      set w1 [labelframe $w.lf_[incr unique] -text $el]
      set w2 [entry $w1.e_[incr unique] -textvar State($el)]
      grid $w1
      grid $w2
    }
    $pw add $w
  }
```

Script Output



11.7.2 Widget Layout: `place`, `pack`, and `grid`

Stacking one widget atop another is useful for extremely simple examples, but real-world applications need a bit more structure. Tk includes three layout managers (`place`, `pack`, and `grid`) to give you control over your application interface.

The layout managers allow you to describe how the widgets in your application should be arranged. The layout managers use different algorithms for describing how a set of widgets should be arranged. Thus, the options supported by the managers have little overlap.

All of the window managers support the `-in` option, which defines the window in which a widget should be displayed. By default, a widget will be displayed in its immediate parent widget. For example, a button named `.mainButton` would be packed in the root (.) window, whereas a button named `.buttonFrame.controls.offButton` would default to being displayed in the `.controls` widget, which is displayed in the `.buttonFrame` widget.

Using the tree-style naming conventions and default display parent for widgets makes code easier to read. This technique documents the widget hierarchy in the widget name. However, the default display parent can be overridden with the `-in` option if your program requires this control.

The `place` Layout Manager

The `place` command lets you declare precisely where a widget should appear in the display window. This location can be declared as an absolute location or a relative location based on the size of the window. Applications that use few widgets or need precise control are easily programmed with the `place` layout manager, and the `place` layout manager is useful to display a window (like an alert) over other windows.

For most applications, however, the `place` layout manager requires too much programmer overhead and makes an application that doesn't look correct after resizing. The `pack` and `grid` managers are recommended as general-use layout managers.

Syntax: `place` *`widgetName option ?options?`*

Declare the location of a widget in the display.

`widgetName`	The name of the widget being placed.
`option`	The `place` command requires at least one X/Y pair of these placement options:
`-x` *`xLocation`*	
`-y` *`yLocation`*	An absolute location in pixels.
`-relx` *`xFraction`*	
`-rely` *`yFraction`*	A relative location given as a fraction of the distance across the window.
`-in` *`windowName`*	A window to hold this widget. The window *`windowName`* must be either the immediate parent or lower in the window hierarchy than the window being placed. That is, for example, Window `.frame1.frame2.label` can be placed `-in .frame1.frame2` or `frame1.frame2.frame3.frame4`, but not `.frame1`.

The following example uses the `place` command to build a simple application that calculates a 15% sales tax on purchases. As you can see, there is a lot of overhead in placing the widgets. You need a good idea of how large widgets will be, how large your window is, and so on, in order to make a pretty screen.

Example 18

Script Example

```
# Create the "quit" and "calculate" buttons
set quitbutton [button .quitbutton —text "Quit" —command "exit"]

set gobutton [button .gobutton —text "Calculate Sales Tax" \
    —command {set salesTax [format %.2f [expr $userInput * 0.15]]}]

# Create the label prompt, and entry widgets
set input [entry .input —textvariable userInput]
set prompt [label .prompt —text "Base Price:"]

# Create the label and result widgets
set tax [label .tax —text "Tax :"]
set result [label .result —textvariable salesTax —relief raised]

# Set the size of the main window
```

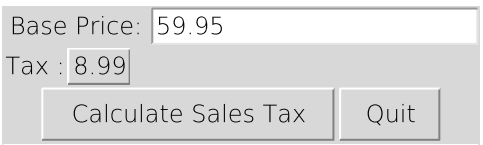
```
. configure –width 250 –height 100

# Place the buttons near the bottom
place $quitbutton –relx .75 –rely .7
place $gobutton –relx .01 –rely .7

# Place the input widget near the top.
place $prompt –x 0 –y 0
place $input –x 75 –y 0

# Place the results widgets in the middle
place $tax –x 0 –y 30
place $result –x 40 –y 30
```

Script Output



The `pack` *Layout Manager*

The `pack` command is quite a bit easier to use than `place`. With the pack command, you declare the positions of widgets relative to each other and let the `pack` command worry about the details.

Syntax: `pack` *`widgetName ?options?`*

Place and display the widget in the display window.

`widgetName` The widget to be displayed.

`?options?` The `pack` command has many options. The following options are the most used:

`-side` *`side`* Declares that this widget should be packed closest to a given side of the parent window. The `side` argument may be one of `top`, `bottom`, `left`, or `right`. The default is `top`.

`-anchor` *`edge`* If the space assigned to this widget is larger than the widget, the widget will be anchored to this edge of the space. The *space* parameter may be one of `n`, `s`, `e`, or `w`.

`-expand` *`boolean`* If set to `1` or `yes`, the widget will be expanded to fill available space in the parent window. The default is `0`: (do not expand to fill the space). The `boolean` argument may be one of `1`, `yes`, `0`, or `no`.

`-fill` *`direction`* Defines whether a widget may expand to fill extra space in its parcel. The default is `none` (do not fill extra space). The `direction` argument may be:

	`none`	Do not fill
	`x`	Fill horizontally
	`y`	Fill vertically
	`both`	Fill both horizontally and vertically
`-padx` *number*	Declares how many pixels to leave as a gap between widgets.	
`-pady` *number*	Declares how many pixels to leave as a gap between widgets.	
`-after` *widgetName*	Pack this widget after (on top of) *widgetName*.	

The packer can be conceptualized as starting with a large open square. As it receives windows to pack, it places them on the requested edge of the remaining open space. For example, if the first window is a label with the `-side left` option set, `pack` would place the left edge of the label against the left side of the empty frame. The left edge of the empty space is now the right edge of the label, even though there may be empty space above and below this widget.

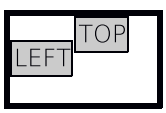
If the next item is to be packed `-side top`, it will be placed above and to the right of the first widget. The following code shows how these two widgets would appear. Note that even though the anchor on the top label is set to `west`, it only goes as far to the west as the east-most edge of the first widget packed.

Example 19

Script Example

```
label .la –background gray80 –text LEFT –relief solid
label .lb –background gray80 –text TOP –relief solid
pack .la –side left
pack .lb –side top –anchor w
. configure –background white –relief solid –borderwidth 3
# wm interacts with the window manager – discussed later
# this command makes the window smaller
wm geometry . 80x50
```

Script Output



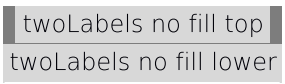
The pack algorithm works by allocating a rectangular parcel of display space and filling it in a given direction. If a widget does not require all available space in a given dimension, the parent widget will "show through" unless the `-expand` or `-fill` option is used.

The following example shows how a label widget will be packed with a frame with various combinations of the `-fill` and `-expand` options being used. The images show the steps as new frames are added to the display.

Example 20

Script Example

```
# Create a root frame with a black background, and pack it.
frame .root –background black
pack .root
# Create a frame with two labels to allocate 2 labels worth of
# vertical space.
# Note that the twoLabels frame shows through where the top
# label doesn't fill.
frame .root.twoLabels –background gray50
label .root.twoLabels.upperLabel –text "twoLabels no fill top"
label .root.twoLabels.lowerLabel –text "twoLabels no fill lower"
pack .root.twoLabels –side left
pack .root.twoLabels.upperLabel –side top
pack .root.twoLabels.lowerLabel –side bottom
```




```
# Create a frame and label with no fill or expand options used.
# Note that the .nofill frame is completely covered by the
# label, and the root frame shows at the top and bottom
frame .root.nofill –background gray50
label .root.nofill.label –text "nofill, noexpand"
pack .root.nofill –side left
pack .root.nofill.label
```


```
# Create a frame and label pair with the –fill option used when
# the frame is packed.
# In this case, the frame fills in the Y dimension to use all
# available space, and the label is packed at the top of
# the frame. The .fill frame shows through below the label.
frame .root.fill\_frame –background gray50
label .root.fill\_frame.label –text "fill frame"
pack .root.fill\_frame –side left –fill y
pack .root.fill\_frame.label
```


```
# Create a label that can fill, while the frame holding it will not.
# In this case, the frame is set to the size required to hold
# the widget, and the widget uses all that space.
frame .root.fill\_label –background gray50
label .root.fill\_label.label –text "fill label"
pack .root.fill\_label –side left
```

```
pack .root.fill\_label.label –fill y
```


```
# Allow both the frame and widget to fill.
# The frame will fill the available space,
# but the label will not expand to fill the frame.
frame .root.fillBoth –background gray50
label .root.fillBoth.label –text "fill label and frame"
pack .root.fillBoth –side left –fill y
pack .root.fillBoth.label –fill y
```


```
# Allow both the frame and widget to expand and fill.
# The –expand option allows the widget to expand into extra
# space.
# The –fill option allows the widget to fill the available
# space.
frame .root.expandFill –background gray50
label .root.expandFill.label –text "expand and fill label and frame"
pack .root.expandFill –side left –fill y –expand y
pack .root.expandFill.label –fill y –expand y
```

The `pack` command is good at arranging widgets that will go in a line or need to fill a space efficiently. It is somewhat less intuitive when you are trying to create a complex display. The solution to this problem is to construct your display out of frames and use the `pack` command to arrange widgets within the frame, and again to arrange the frames within your display. For many applications, grouping widgets within a frame and then using the `pack` command is the easiest way to create the application. The following example shows a fairly common technique of making a frame to hold a `label` prompt and an entry widget.

Example 21

Script Example

```
# Create the frames for the widgets
set buttonFrame [frame .buttons]
set inputFrame [frame .input]
set resultFrame [frame .results]

# Create the widgets
set quitbutton [button .buttons.quitbutton –text "Quit" \
    –command "exit"]
```

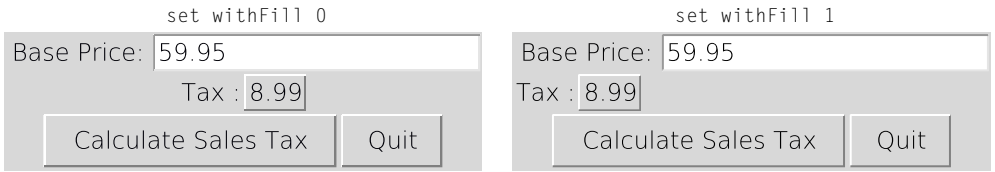
```
set gobutton [button .buttons.gobutton \
    -text "Calculate Sales Tax" \
    -command {set salesTax [format %.2f [expr $userInput * 0.15]]}]
set input [entry $inputFrame.input -textvariable userInput]
set prompt [label $inputFrame.prompt -text "Base Price:"]
set tax [label $resultFrame.tax -text "Tax :"]
set result [label .results.result -textvariable salesTax \
    -relief raised]

# Pack the widgets into their frames.
pack .buttons.quitbutton -side right
pack .buttons.gobutton -side right
pack $input -side right
pack $prompt -side left
pack $tax -side left
pack $result -side left

# Pack the frames into the display window.
pack .buttons -side bottom
pack $inputFrame -side top

# The left example image is created by setting
#  withFill to 0 outside this code snippet.
# The right example image is created by setting
#  withFill to 1 outside this code snippet.
if {$withFill} {
  pack $resultFrame -after $inputFrame -fill x
} else {
  pack $resultFrame -after $inputFrame
}
```

Script Output



Note the way the `-side` option is used in the previous example. The buttons are packed with `-side` set to `right`. The `pack` command will place the first widget with a `-side` option set against the requested edge of the parent frame. Subsequent widgets will be placed as close to the requested side as possible without displacing a widget that previously requested that side of the frame.

The `-fill x` option to the `pack $resultFrame` allows the frame that holds the `$tax` and `$result` widgets to expand to the entire width of the window, as shown in the `set Withfill 1` result. By default, a frame is only as big as it needs to be to contain its child widgets. Without the `-fill` option, the `resultFrame` would be the narrowest of the frames and would be packed in the center of the middle row, as shown in the `set withfill 0` result. With the `-fill` option set to fill

in the X dimension, the frame can expand to be as wide as the window that contains it, and the widgets packed on the left side of the frame will line up with the left edge of the window instead of lining up on the left edge of a small frame. If you want a set of widgets `.a`, `.b`, and `.c` to be lined up from left to right, you can pack them as follows.

```
pack .a –side left
pack .b –side left
pack .c –side left
```

The `grid` *Layout Manager*

Many graphic layouts are conceptually grids. The `grid` layout manager is ideal for these types of applications, since it lets you declare that a widget should appear in a cell at a particular column and row. The Grid manager then determines how large columns and rows need to be to fit the various components.

Syntax: `grid` *`widgetName ?widgetNames? option`*

Place and display the widget in the display window.

`widgetName`	The name of the widget to be displayed.
`?options?`	The `grid` command supports many options. The following is a minimal set.
`-column` *`number`*	The column position for this widget.
`-row` *`number`*	The row for this widget.
`-columnspan` *`number`*	How many columns to use for this widget. Defaults to 1.
`-rowspan` *`number`*	How many rows to use for this widget. Defaults to 1.
`-sticky` *`side`*	Which edge of the cell this widget should "stick" to. Values may be `n`, `s`, `e`, `w`, or a combination of these letters (to stick to multiple sides).

The `-sticky` option lets your script declare that a widget should "stick" to the north (top), south (bottom), east (right), or west (left) edge of the grid in which it is placed. By default, a widget is centered within its cell. Any of the options for the `grid` command can be reset after a widget is packed with the `grid configure` command.

Syntax: `grid configure` *`widgetName -option optionValue`*

Change one or more of the configuration options for a widget previously positioned with the `grid` command.

`widgetName`	The name of the widget to have a configuration option changed.
`-option ?value?`	The option and value to modify (or set).

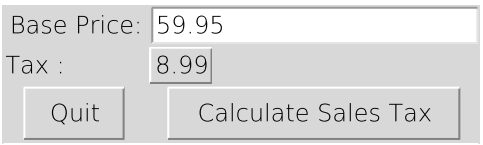
Example 22

Script Example

```
set quitbutton [button .quitbutton –text "Quit" –command "exit"]
set gobutton [button .gobutton –text "Calculate Sales Tax" \
```

```
    -command {set salesTax [format %.2f [expr $userInput * 0.15]]}]
set input [entry .input -textvariable userInput]
set prompt [label .prompt -text "Base Price:"]
set tax [label .tax -text "Tax :"]
set result [label .result -textvariable salesTax -relief raised]
grid $quitbutton $gobutton -row 3
grid $prompt $input -row 1
grid $tax $result -row 2 -sticky w
```

Script Output



Working Together

The different layout managers can be used together to get the screen layout you desire. In the rules defined in the following, `.f1` and `.f2` are frames that contain other widgets.

- Within a single frame, any single layout manager can be used. The following is valid.
  ```
  pack .f1.b1 -side left
  grid .f2.b1 -column 1 -row 3
  ```
- Within a single frame, the `grid` and `pack` cannot be mixed. The following is not valid.
  ```
  pack .f1.b1 -side left
  grid .f1.b2 -column 1 -row 3
  ```
- The `place` can be used with either the `grid` or `pack` command. The following is valid.
  ```
  pack .f1.b1 -side left
  place .f1.label -x 25 -y 10
  grid .f2.b1 -column 1 -row 3
  place .f2.label -xrel .1 -yrel .5
  ```
- Frames can be arranged with either `grid` or `pack`, regardless of which layout manager is used within the frame. The following is valid.
  ```
  pack .f1.b1 -side left
  grid .f2.b1 -column 1 -row 3
  grid .f1 -column 0 -row 0
  grid .f2 -column 1 -row 0
  ```

11.8 SELECTION WIDGETS: `radiobutton`, `checkbutton`, `menu`, AND `listbox`

Allowing the user to select one (or more) items from a list is a common program requirement. The Tk graphics extension provides the selection widgets `radiobutton`, `checkbutton`, `menu`, and `listbox`.

11.8.1 radiobutton and checkbutton

The radiobutton and checkbutton widgets are very similar. The primary difference is that radiobutton will allow the user to select only one of the entries, whereas checkbutton will allow the user to select multiple items from the entries.

radiobutton

The radiobutton widget displays a label with a status indicator next to it. When a radiobutton is selected, the indicator changes color to show which item has been selected, and any previously selected radiobutton is deselected.

Syntax: radiobutton *radioName ?-options value?*

Create a radiobutton widget.

radioName	The name for this radiobutton widget.
-option	Options for the radiobutton include:
-command *script*	A script to evaluate when this button is clicked.
-variable *varName*	The variable defined here will contain the value of this button when the button is selected. If this option is used, the -value must also be used.
-value *value*	The value to assign to the variable.

The -variable and -value options allow the radiobutton widget to be attached to a variable that will be set to a particular value when the button is selected. By using the -command option, you can assign a script that will be evaluated when the button is selected. If the -variable and -value options are also used, the script will be evaluated after the new value has been assigned to the widget variable.

The following example shows how the magic shop in a computerized Fantasy Role Playing game can be modernized. Note the foreach {item cost} $itemList command. This style of the foreach command (using a list of variables instead of a single variable) was introduced in Tcl version 7.4. It is useful when stepping through a list that consists of repeating fields, such as the *name price name price* data in itemList.

Example 23

Script Example

```
# Update the displayed text in a label
proc updateLabel {myLabel item} {
  global price
  $myLabel configure –text \
      "The cost for a potion of $item is $price gold pieces"
}

# Create and display a label
set l [label .l –text "Select a Potion"]
grid $l –column 0 –row 0 –columnspan 3

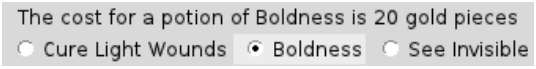
# A list of potions and prices
```

```
set itemList [list "Cure Light Wounds" 16 \
                   "Boldness" 20 \
                   "See Invisible" 60]
set position 0
foreach {item cost} $itemList {
  radiobutton .b_$position –text $item –variable price \
      –value $cost –command [list updateLabel $l $item]
  grid .b_$position –column $position –row 1
  incr position
}
```

Script Output

Before Selection

After Selecting **Boldness**



All of the `radiobutton` widgets in this example share the global variable `price`. The variable assigned to the `-variable` option is used to group `radiobutton` widgets. For example, if you had two sets of `radiobuttons`, one for magic potions and one for magic scrolls, you would need to use two different variable names, such as `potionPrice` and `scrollPrice`.

If you assign each `radiobutton` a different variable, the `radiobuttons` will be considered as separate groups. In this case, all buttons can be selected at once and cannot be deselected, since there would be no other button in their group to select. Note that the variable `price` is declared as a global in the procedure `updateLabel`. The variables attached to Tk widgets default to being in the global scope.

The `updateLabel` procedure is called with an item name and uses the variable `price` to get the price. The `$price` value could be passed to the procedure by defining the `-command` argument as:

```
-command [list updateLabel $l $item $cost]
```

This example used the variable for demonstration purposes. Either technique for getting the data to the procedure can be used.

checkbutton

The `checkbutton` widget allows multiple items to be selected. The `checkbutton` widget displays a label with a square status indicator next to it. When a `checkbutton` is selected, the indicator changes color to show which item has been selected. Any other checkbuttons are not affected.

The `checkbutton` widget supports a `-variable` option, but unlike the case of `radiobutton` you must use a separate variable for each widget, instead of sharing a single variable among the widgets. Using a single variable will cause all buttons to select or deselect at once, instead of allowing you to select one or more buttons.

Syntax: `checkbutton` *`checkName ?options?`*

Create a `checkbutton` widget.

`checkName` The name for this `checkbutton` widget.

`?options?` Valid options for the `checkbutton` widget include:

`-variable` *`varName`*	The variable defined here will contain the value of this button when the button is selected.
`-onvalue` *`selectValue`*	The value to assign to the variable when this button is selected. Defaults to 1.
`-offvalue` *`nselectValue`*	The value to assign to the variable when this button is not selected. Defaults to 0.

Example 24

Script Example

```
# Update the displayed text in a label
proc updateLabel {myLabel item} {
  global price
  set total 0
  foreach potion [array names price] {
    incr total $price($potion)
  }
  $myLabel configure -text "Total cost is $total Gold Pieces"
}

# Create and display a label
set l [label .l -text "Select a Potion"]
grid $l -column 0 -row 0 -columnspan 3

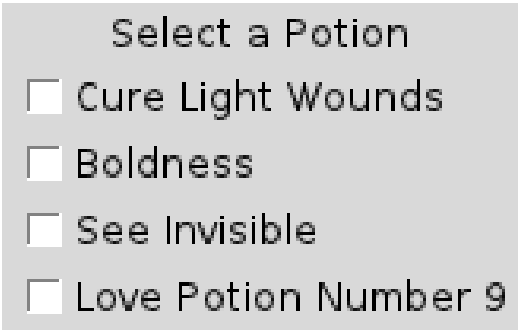
# A list of potions and prices
set itemList [list "Cure Light Wounds" 16 \
    "Boldness" 25 \
    "See Invisible" 60 \
    "Love Potion Number 9" 45 ]

set position 1

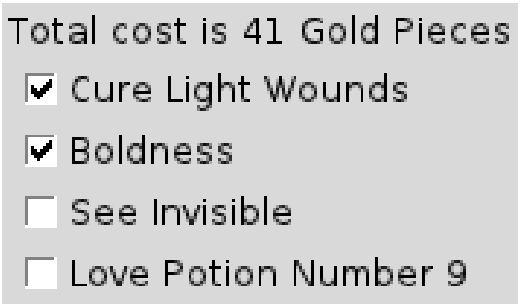
foreach {item cost} $itemList {
  checkbutton .b_$position -text $item \
      -variable price($item) -onvalue $cost -offvalue 0 \
      -command "[list updateLabel $l $item]"
  grid .b_$position -row $position -column 0 -sticky w
  incr position
}
```

Script Output

Before Selection



After Selecting



11.8.2 Pull-down Menus: `menu`, `menubutton`, and Menubars

The Tk `menu` command creates the ubiquitous pull-down menu that we have all become so fond of. A Tk menu is an invisible holder widget that can be attached to a `menubutton` widget or used as a window 's menu bar. The `menubutton` is similar to the `button` widget, described in Section 11.6. The differences include:

- The default relief for a menubutton is flat, whereas a button is raised.
- A menubutton can be attached to a menu instead of a script.

Most of the examples that follow will use the `-relief raised` option to make the menubutton obvious.

Syntax: `menubutton` *`buttonName ?options?`*

Create a `menubutton` widget.

`buttonName`	The name for this `menubutton`.
`?options?`	The `menubutton` supports many options. Some of the more useful are:

`-text` *`displayText`*	The text to display on this button.
`-textvariable` *`varName`*	The variable that contains the text to be displayed.
`-underline` *`charPosition`*	Selects a position for a hot key.
`-menu` *`menuName`*	The name of the `menu` widget associated with this `menubutton`.

The `-text` and `-textvariable` options can be used to control the text displayed on a button. If a `-textvariable` option is declared, the button will display the content of that variable. If both a `-text` and `-textvariable` are defined, the `-textvariable` variable will be initialized to the argument of the `-text` option.

The `-underline` option lets you define a hot key to select a menu item. The argument to the `-underline` option is the position of a character in the displayed text. The character at that position will be underlined. If that character and the Alt key are pressed simultaneously, the `menubutton` will be selected. A `menu` widget is a invisible container widget that holds the menu entries to be displayed.

Syntax: `menu` *`menuName ?options?`*

Create a `menu` widget.

`menuName` The name for this `menu` widget. Note that this name must be a child name to the parent `menubutton`. For example, if the `menubutton` is `.foo.bar`, the `menu` name must resemble `.foo.bar.baz`.

`?options?` The `menu` widget supports several options. A couple that are unique to this widget are:

`-postcommand` *`script`*	A script to evaluate just before a menu is posted.
`-tearoff` *`boolean`*	Allows (or disallows) a `menu` to be removed from the `menubutton` and displayed in a permanent window. This is enabled by default.

Once a `menu` widget has been created, it can be manipulated via several widget subcommands. Those that will be used in these examples are as follows.

Syntax: *`menuName`* `add` *`type ?-option val?`*

Add a new menu entry to a `menu`.

`type` The type for this entry. May be one of:

`separator`	A line that separates one set of menu entries from another.
`cascade`	Defines this entry as one that has another `menu` associated with it, to provide cascading menus.
`checkbutton`	Same as the standalone `checkbutton` widget.
`radiobutton`	Same as the standalone `radiobutton` widget.
`command`	Same as the standalone `button` widget.

`?-option?` A few of the options that will be used in the examples are:

`-command` *`script`*	A script to evaluate when this entry is selected.
`-accelerator` *`string`*	Displays the string to the right of the menu entry as an accelerator. This action must be bound to an event. Binding is covered in Chapter 10.
`-label` *`string`*	The text to display in this menu entry.
`-menu` *`menuName`*	The menu associated with a cascade-type menu element. Valid only for cascading menus.
`-variable` *`varName`*	A variable to be set when this entry is selected.
`-value` *`string`*	The value to set in the associated variable.

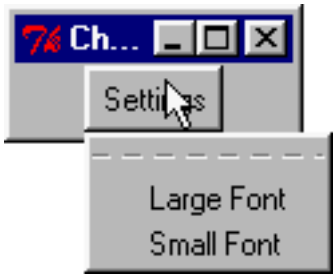
`-underline` *`position`*	The position of the character in the text for this menu item to underline and bind for action with this `menu` entry. This is equivalent to the `-underline` option the `menubutton` supports.

Example 25

Script Example

```
# Create a "Settings" menubutton
menubutton .mb –menu .mb.mnu –text "Settings" \
    –relief raised –background gray70 \
    –menu .mb.mnu
# Add font selectors
.mb.mnu add radiobutton –label "Large Font" \
    –variable font –value {Times 18 bold} }
.mb.mnu add radiobutton –label "Small Font" \
    –variable font –value {Times 8 normal}
grid .mb
```

Script Output



Alternatively, the command *`menuName`* `insert` can be used to insert items into a menu. This command supports the same options as the `add` command. The `insert` command allows your script to define the position to insert this entry before. The `insert` and `delete` commands require an *`index`* option. An index may be specified in one of the following forms.

`number`	The position of the item in the menu. Zero is the topmost entry in the menu.
`end` or `last`	The bottom entry in the menu. If the menu has no entry, this is the same as `none`.
`active`	The entry currently active (selected). If no entry is selected, this is the same as `none`.
`@number`	The entry that contains (or is closest to) the Y coordinate defined by `number`.
`pattern`	The first entry with a label that matches the glob-style `pattern`.

Syntax: *menuName* insert *index type ?options?*

Insert a new entry before position *index*.

index	The position to insert this entry before.
type	Type of menu item to insert. As described for add.
?options?	Options for this menu item. As described for add.

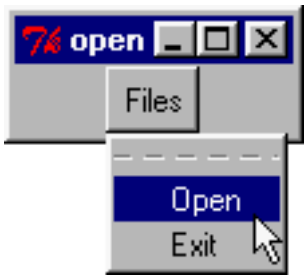
In the following example, the Open selection will always be the first menu entry, and Exit will always be last.

Example 26

Script Example

```
# Create a "Files" menu
menubutton .mb –menu .mb.mnu –text "Files" \
    –relief raised –background gray70
menu .mb.mnu
# Insert open and exit as first and last
#  selections
.mb.mnu insert 0 command –label "Open" \
    –command openFile\\
# .. Other menu entries...
.mb.mnu insert end command –label "Exit" \
    –command quitTask\\
grid .mb
```

Script Output



A non-functional entry can be removed with the delete command or it could be configured as *disabled* with the entryconfigure command

Syntax: *menuName* delete *index1 index2*

Delete menu entries.

index1 index2	Delete the entries between the numeric indices index1 and index2, inclusive.

Syntax: *menuName* `entryconfigure` *index ?-option value?*

If no option, returns a list of all options and values. If no value, returns current setting for the option. If both option and value are specified, modify the option.

index The index to query or modify.

-option An option that can be set when using the `add` or `insert` command.

Example 27

Script Example

```
menubutton .mb –menu .mb.mnu –text "Files" \
    –relief raised –background gray70

menu .mb.mnu
.mb.mnu insert 0 command –label "Open" \
    –command openFile

.mb.mnu add separator

.mb.mnu add command –label "Save" \
    –command saveData

.mb.mnu add command –label "SaveAs" \
    –command saveDataAs
grid .mb

# If in demo mode, disable the Save options

if {$demoMode} {
  .mb.mnu entryconfigure 3 –state disabled
  .mb.mnu entryconfigure 4 –state disabled
}

# ... In file open code.
# Check write permission on file.  If not
#   writable, set 'permission' to 0, else 1.
# ...
# Remove save option if no write permission

if {!$permission} {
  .mb.mnu delete Save
}
```

Script Output

With no permission and in demoMode

With permission and not demoMode

The previous example *knows* that the `Save` and `SaveAs` entries are position 3 and 4. If your code can't know that (perhaps menu entries are being added based on recent activity), it can learn the index of a menu entry based on a pattern with the `index` command.

Syntax: *menuName* `index` *pattern*

Return the numeric index of the menu entry with the label *string*.

pattern An index pattern, as described previously.

Example 28

Script Example

```
menubutton .mb –menu .mb.mnu –text "Files" \
    –relief raised

menu .mb.mnu
.mb.mnu add command –label Open
.mb.mnu add separator

foreach {label cmd} {Save saveCmd AutoSave autoSaveCmd \
                     SaveAs saveAsCmd} {
  .mb.mnu add command –label $label –command $cmd
}

.mb.mnu add separator

.mb.mnu add command –label Exit

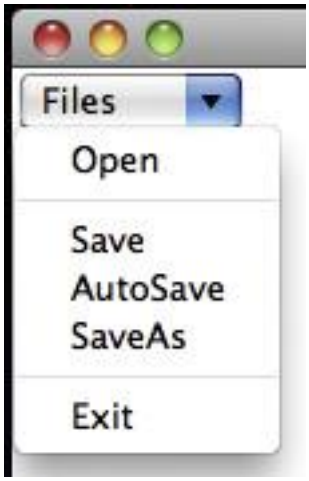
# ... much code.
```

```
# If running in demo mode, remove "Save" menu items'
if {$demoMode} {
  set first [.mb.mnu index Save]
  #  Delete the first separator as well
  incr first -1
  set last [.mb.mnu index SaveAs]
  .mb.mnu delete $first $last
}
grid .mb
```

Script Output

With `demoMode` == 1

With `demoMode` == 0



The following example shows how the various menus are created and how they look in different versions of Tk. In actual fact, you cannot pull down all menus at once. You can, however, use the tear-off strip (select the dotted line at the top of the menus) to create a new window and have multiple windows displayed.

Example 29

Script Example

```
# ------------------------------------------
#  Create a checkbutton menu - Place it on the left
set checkButtonMenu [menubutton .mcheck \
    -text "checkbuttons" -menu .mcheck.mnu]
set checkMenu [menu $checkButtonMenu.mnu]
grid $checkButtonMenu -row 0 -column 0
$checkMenu add checkbutton -label "check 1" \
    -variable checkButton(1) -onvalue 1
```

```
$checkMenu add checkbutton -label "check 2" \
    -variable checkButton(2) -onvalue 2\\

# ------------------------------------------
#  Create a radiobutton menu - Place it in the middle
set radioButtonMenu [menubutton .mradio \
    -text "radiobuttons" -menu .mradio.mnu]
set radioMenu [menu $radioButtonMenu.mnu]
grid $radioButtonMenu -row 0 -column 1
$radioMenu add radiobutton -label "radio 1" \
    -variable radioButton -value 1
$radioMenu add radiobutton -label "radio 2" \
    -variable radioButton -value 2

# ------------------------------------------
#  Create a menu of mixed check, radio, command, cascading and
#   menu separators
set mixButtonMenu [menubutton .mmix -text "mixedbuttons" \
    -menu .mmix.mnu]
set mixMenu [menu $mixButtonMenu.mnu]
grid $mixButtonMenu -row 0 -column 2


# ------------------------------------------
# Two command menu entries\
$mixMenu add command -label "command 1" -command "doStuff 1"
$mixMenu add command -label "command 2" -command "doStuff 2"


# ------------------------------------------
# A separator, and two radiobutton menu entries
$mixMenu add separator
$mixMenu add radiobutton -label "radio 3" \
    -variable radioButton -value 3
$mixMenu add radiobutton -label "radio 4" \
    -variable radioButton -value 4

# ------------------------------------------
#  A separator, and two checkbutton menu entries
$mixMenu add separator
$mixMenu add checkbutton -label "check 3" \
    -variable checkButton(3) -onvalue 3
$mixMenu add checkbutton -label "check 4" \
    -variable checkButton(4) -onvalue 4

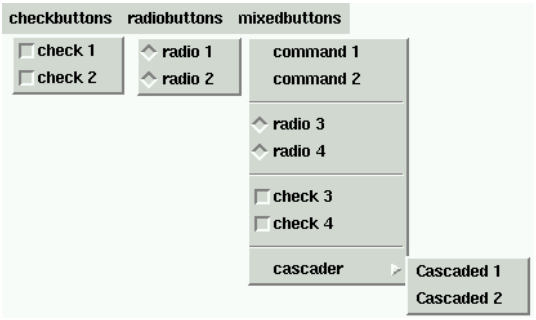
# ------------------------------------------
# A separator, a cascading menu, and two sub menus
```

```
#  within the cascading menu
$mixMenu add separator
$mixMenu add cascade –label "cascader" \
    –menu $mixMenu.cascade
menu $mixMenu.cascade
$mixMenu.cascade add command –label "Cascaded 1"\
    –command "doStuff 3"
$mixMenu.cascade add command –label "Cascaded 2"\
    –command "doStuff 4"
# Define a dummy proc for the command buttons to invoke.
proc doStuff {args} {
  puts "doStuff called with: $args"
}
```

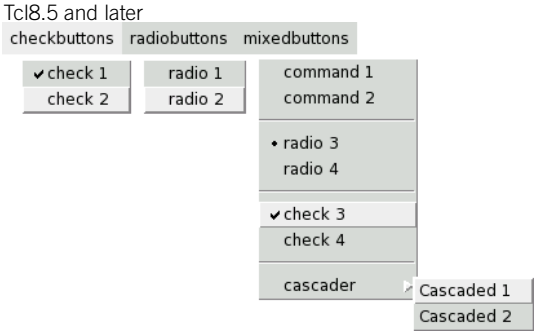
Script Output

The widgets that are displayed will look like whatever is native on the platform where the application is running. These images are for Linux/X11 systems.

Tcl8.4 and earlier



Tcl8.5 and later



Note that the radio buttons in this example all share the variable `radioButton`, even though they are in separate menus. Selecting a radio item from the `mixedbuttons` window deselects an item selected in the `radiobuttons` menu.

Menubars

A menu widget can be attached to a menubutton (as shown previously), can be designated as the menubar in a top-level window (such as the main window), or designated as a window created with the toplevel command (discussed in Section 11.11). The -menu option will designate a menu as a window's menubar.

A menubar will be displayed and implemented in a platform-specific manner. For example, when the script is evaluated on a Macintosh, the menubar of the window with focus will be displayed as the main screen menu. On an MS Windows or X Windows platform, the menubar will be displayed at the top of the window that owns the menubar.

```
# Add a menubar to the main window
. configure –menu .menubar
# Create a new toplevel window with a menubar
toplevel .top –menu .top.mnu
```

Once a menubar has been created, new menu items can be added as with normal menus.

Example 30

Script Example

```
. configure –menu .mbar
# Create the top menubar
menu .mbar

# Add a Files pulldown menu to the menubar
# This appears at the left on all operating systems.
.mbar add cascade –label Files –menu .mbar.files
menu .mbar.files
.mbar.files add command –label Open –command "openFile"

# Help shows up at the right on X11 based systems.
.mbar add cascade –label Help –menu .mbar.help
menu .mbar.help
.mbar.help add command –label About –command "displayAbout"

# Normal menu items appear in the order they are defined.
.mbar add cascade –label Settings –menu .mbar.set
menu .mbar.set
.mbar.set add command –label Fonts –command "setFont"

# The .system menu appears on the icon under Windows systems
.mbar add cascade –label System –menu .mbar.system
menu .mbar.system
.mbar.system add command –label "Windows System"

# The .apple menu appears under the application menu
.mbar add cascade –label Apple –menu .mbar.apple
menu .mbar.apple
```

```
.mbar.apple add command –label "Apple Menu"
.mbar add command –label Run –command "go"
```

Script Output

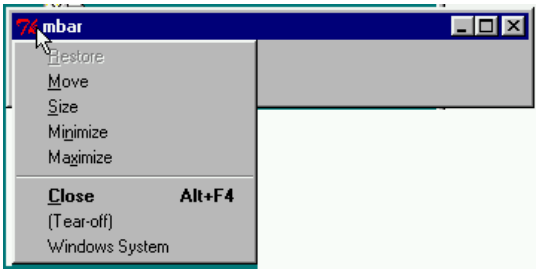
Note that the Help menu is on the far right, even though it was the second menu added. Tcl supports some special naming conventions to access platform-specific conventions. The previous example was run on a Linux platform, so the `.bar.help` menu follows the X Window convention of placing Help on the far right.

Name	Platform	Description
`.menubar.system`	MS Windows	Adds items to the System menu.
`.menubar.help`	X Window	Will make a right-justified Help menu.
`.menubar.help`	Macintosh	Will add entries to the Apple Help menu.
`.menubar.apple`	Macintosh	Items added to this menu will be the first items on the Apple menu.

When the previous example is run on an MS Windows or Macintosh system, the special menus will be displayed, as shown in the following illustration. Note the `(Tear-off)` items in these menus. On X Window based systems, a menu can be turned into a top-level window by clicking the top (dotted) line. This line is referred to as the "Tear-off" line. You can inhibit creating this menu entry by creating the menu with the `-tearoff 0` option.

Under OS/X, the `menubar.apple` entries are added to the application menu.

Under Windows, the `menubar.system` entries are added to the application menu attached to the icon.



11.8.3 Selection Widgets: `listbox`

The `listbox` widget allows the user to select items from a list of items. The `listbox` widget can be configured to select one entry (similar to a radio button) or multiple entries, similar to a checkbutton.

The `listbox` can be queried to return the positions of the selected items. This can be used to index into a list or array of information, or the `listbox` can be queried about the text displayed at that position.

Syntax: `listbox` *`listboxName ?options?`*

Create a `listbox` widget.

`listboxName`	The name for this `listbox`.
`?options?`	The `listbox` widget supports several options. Three useful options are:

`-selectmode` *`style`*	Sets the selection style for this `listbox`. The default mode is `browse`. This option may be set to one of:
`single`	Allows only a single entry to be selected. Whenever an entry is clicked, it is selected, and other selected entries are deselected.
`browse`	Allows only a single entry to be selected. Whenever an entry is clicked, it is selected, and other selected entries are deselected. When the cursor is dragged across entries with the left mouse button depressed, each entry will be selected when the cursor crosses onto it, and deselected when the cursor passes off.
`multiple`	Allows multiple entries to be selected. An entry is selected by clicking it (if not already selected). A selected entry can be deselected by clicking it.

`extended`	Allows a single entry to be selected, or multiple contiguous entries to be selected by dragging the cursor over the entries.
`-exportselection` *`bool`*	If this is set true, the content of the listbox will be exported for other X11 tasks, and only a single listbox selection may be made. If you wish to use multiple `listbox` widgets with different selections, set this option to `FALSE`. This defaults to TRUE.
`-height` *`numLines`*	The height of this listbox in lines.

When a `listbox` widget is created, it is empty. The `listbox` widget supports several commands for manipulating `listbox` content. The following are used in the chapters that follow.

Syntax: *`listboxName`* `insert` *`index element ?elements?`*

Inserts a new element into a `listbox`.

`index`	The position to insert this entry before. The word `end` causes this entry to be added after the last entry in the list.
`element`	A text string to be displayed in that position. This must be a single line. Embedded newline characters are printed as backslash-N, instead of generating a new line.

Example 31

Script Example

```
# Create and display an empty listbox
listbox .l –height 3
grid .l
# Add 3 elements to the listbox
# Note – insert at position 0 makes the display order the
#  opposite of the insertion order.
.l insert 0 first
.l insert 0 second
.l insert 0 third
```

Script Output

Along with inserting elements into a listbox, you might need to delete one or more. The `delete` command will remove elements from a listbox.

Syntax: *listboxName* delete *first ?last?*

Delete entries from a listbox.

first The first entry to delete. If there is no last entry, only this entry will be deleted.

last The last entry to delete. Ranges are deleted inclusively.

Example 32

Script Example

```
# Create and display an empty listbox
listbox .l –height 3
pack .l
# Add 3 elements to the listbox
# Note – insert at end position – order is as expected
.l insert end first
.l insert end second
.l insert end third

# Delete the second listbox entry (count from 0)
.l delete 1
```

Script Output

Elements in a listbox can be selected by clicking on them. The curselection will return the element number that has been selected and the get command can be used to return the contents of a selected element

Syntax: *listboxName* curselection

Returns a list of the indices of the currently selected items in the listbox.

Example 33

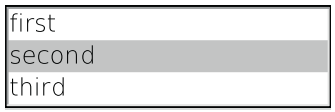
Script Example

```
# Create and display an empty listbox
listbox .l –height 3
grid .l
#  Add 3 elements to the listbox\\
#  Note – insert at end position – order is as expected
.l insert end first
.l insert end second
.l insert end third
```

```
# User selects second item
puts "Selected: [.l curselection]"
```

Script Output

```
Selected: 1
```



Syntax: *listboxName* get *first ?last?*

Returns a list of the text displayed in the range of entries identified by the indices.

first The first entry to return.

last The last last entry to return. If this is not included, only the first entry is returned. The range returned is inclusive.

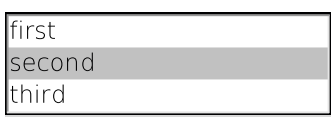
Example 34

Script Example

```
# Create and display an empty listbox
listbox .l –height 3
pack .l
# Add 3 elements to the listbox
# Note – insert at end position – order is as expected
.l insert end first
.l insert end second
.l insert end third
# User selects second item
puts "Selected Text: [.l get [.l curselection]]"
```

Script Output

```
Selected Text: second
```



By default, selecting an element in a listbox will export that selection to the window manager's clipboard. This allows the selection to be cut/pasted into other tasks and also limits the listbox to a single selection. In order to select multiple items from a listbox, you must both disable the export function and enable multiple selection mode as shown in the right hand listbox in the next example. In the next example some selections were made before the graphic and report were created.

Example 35

Script Example

```
# Create the left listbox, defined to allow only a single
# selection
listbox .lSingle –selectmode single –exportselection no
grid .lSingle -row 1 –column 0
.lSingle insert end "top" "middle" "bottom"

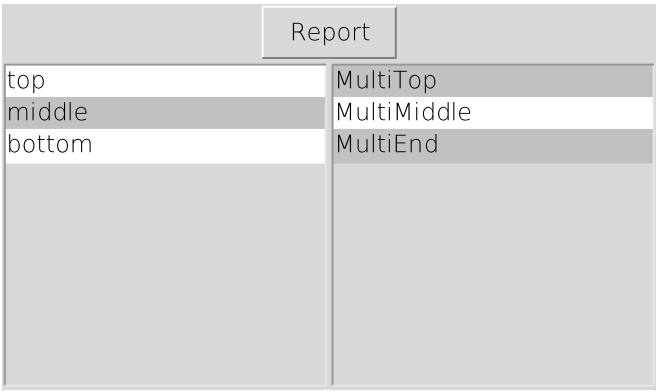
#  Create the right listbox, defined to allow multiple items
#  to be selected.
listbox .lMulti –selectmode multiple –exportselection no
grid .lMulti –row 1 –column 1
.lMulti insert end "MultiTop" "MultiMiddle" "MultiEnd"

#  Create a button to report what's been selected
button .report –text "Report" –command "report"
grid .report –row 0 –column 0 –columnspan 2

#  And a procedure to loop through the listboxes,
#   and display the selected values.
proc report {} {
  foreach widget [list .lSingle .lMulti] {
    set selected [$widget curselection]
      foreach index $selected {
        set str [$widget get $index]
        puts "$widget has index $index selected – $str"
    }
  }
}
```

Script Output

```
.lSingle has index 1 selected - middle
.lMulti has index 0 selected - MultiTop
.lMulti has index 2 selected - MultiEnd
```



11.9 SCROLLBAR

Since the `listbox` does not allow variables or commands to be associated with its selections, it seems less useful than the `button` or `menu` widgets. The `listbox` becomes important when you need to display a large number of selection values and you connect the `listbox` with a `scrollbar` widget.

11.9.1 The Basic `scrollbar`

The `scrollbar` widget allows you to show a portion of a widget's information by using a bar with arrows at each end and a slider in the middle. To change the information displayed, a user clicks the arrows or bar, or drags the slider. At this point, the `scrollbar` informs the associated widget of the change. The associated widget is responsible for displaying the appropriate portion of its data to reflect that change.

Syntax: `scrollbar` *`scrollbarName ?options?`*

Create a `scrollbar` widget.

`scrollbarName`	The name for this `scrollbar`.
`options`	This widget supports several options.
`-command "cmdName ?args?"`	This defines the command to invoke when the state of the `scrollbar` changes. Arguments that define the changed state will be appended to the arguments defined in this option. Most commonly, the *`cmdName`* argument is the name of the widget this `scrollbar` will interact with.
`-orient direction`	Defines the orientation for the scrollbar. The *`direction`* may be `horizontal` or `vertical`. Defaults to `vertical`.
`-troughcolor color`	Defines the color for the trough below the slider. Defaults to the default background color of the frames.

A `scrollbar` interacts with another widget by invoking the defined command whenever the state of the `scrollbar` changes. The widgets that support scrolling (`listbox`, `text`, and `canvas`) have subcommands defined to allow them to be scrolled. The commands that control the behavior of the scrollable widget and the scrollbar are discussed in more detail later in this chapter.

The options that must be used to make a widget scrollable are `-xscrollcommand` and/or `-yscrollcommand`. These are the equivalent of the scrollbar's `-command` option.

Syntax: *`widgetName`* `-xscrollcommand` *`script`*

Syntax: *`widgetName`* `-yscrollcommand` *`script`*

Defines the script to be evaluated when the widget view shifts, so that the scrollbar may reflect the state of the scrollable widget. Information about the change will be appended to this script. In the next example, the command to create the scrollbar includes the following option.

```
—command .box yview
```

This registers the script `.box yview` with the scrollbar. When the scrollbar changes state (someone moves the slider, clicks the bar, and so on), the scrollbar will append information about the change to that script and evaluate it. The `listbox`, `canvas`, and `text` widgets each support a `yview` subcommand that understands the arguments the scrollbar will append. The command to create the listbox includes the following option.

```
-yscrollcommand ".scroll set"
```

This registers the script `.scroll set` with the listbox. When the listbox changes state (for example, when lines are added or deleted), this information will be appended to that script, and the script will then be evaluated. The scrollbar supports a `set` subcommand to be invoked by this script.

The following example shows a scrollbar connected with a listbox.

Note the `-sticky ns` option to the grid command which tells the `grid` layout manager that the ends of the widget should stick to top and bottom of the frame. Without this option, the `scrollbar` would consist of two arrows with a 1-pixel-tall bar, to use the minimal space.

The equivalent `pack` option is `-fill y`, which informs the `pack` layout manager that this widget should expand to fill the available space and that it should expand in the Y direction.

Example 36

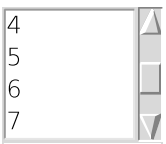
Script Example

```
# Create the scrollbar and listbox.
scrollbar .scroll –command ".box yview"
listbox .box –height 4 –width 8 \
    –yscrollcommand ".scroll set"

# Grid them onto the screen – note –sticky option
grid .box .scroll –sticky ns

# Fill the listbox.
.box insert end 0 1 2 3 4 5 6 7 8 9 10
```

Script Output



11.9.2 `scrollbar` Details

In normal use, the programmer can just set the `-command` option in the scrollbar and the `-yscrollcommand` or `-xscrollcommand` in the widget the scrollbar is controlling, and everything will work as expected. For some applications, though, you need to understand the details of how the scrollbar works. The following is the sequence of events that occurs when a scrollbar is clicked.

1. The user clicks an arrow or bar, or drags a slider.
2. The scrollbar concatenates the registered script (`.box yview`) and information about the change (`moveto .2`) to create a new script to evaluate (`.box yview moveto .2`).
3. The scrollbar evaluates the new script.
4. The widget changes its displayed area to reflect the change.
5. The widget concatenates its registered script and information that describes how it changed to create a new script (`.scroll set .2 .5`).
6. The widget evaluates that script to reconfigure the scrollbar to match the widget.

The information the scrollbar appends to the script will be in one of the formats outlined in the following table.

Command Subset	Description	Action
`scroll ?-?1 unit`	Scroll the displayed widget by one of the smallest units (a line for a listbox or text widget, or a single pixel for a canvas).	Click an arrow.
`scroll ?-?1 page`	Scroll the displayed widget by the displayed area. For example, if four lines of a listbox are displayed, the listbox would scroll by four lines.	Click the bar.
`moveto` *`fraction`*	Set the top of the displayed area to start at the requested percentage. For example, in a 100-line listbox, `.box yview moveto .2` would start the display with line 20.	Drag the slider.

In the preceding example, when the scrollbar is manipulated, a command of the form

```
.box yview scroll 1 unit
```

or

```
.box yview moveto .25
```

would be created by the `scrollbar` widget and then evaluated. The scrollbar's `set` command will modify the size and location of the scrollbar's slider.

Syntax: *`scrollbarName`* `set` *`first last`*

Sets the size and location of the slider.

`first` A fraction representing the beginning of the displayed data in the associated widget (e.g., 0.25) informs the scrollbar that the associated widget starts displaying data at that point (e.g., the 25% point). The scrollbar will place the starting edge of the slider one fourth of the way down the bar.

`end` A fraction representing the end of the displayed data in the associated widget (e.g., 0.75) informs the scrollbar that the associated widget stops displaying data at that point (e.g., the 75% point). The scrollbar will place the ending edge of the slider three fourths of the way down the bar.

Example 37

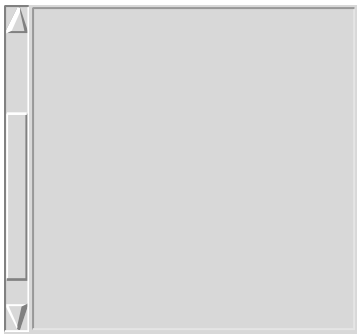
Script Example

```
# Create and grid a scrollbar with no –command option
scrollbar .sb
grid .sb  –row 0 –column 0 –sticky ns

#  Create and grid a listbox (to fill space and expand the
#    scrollbar)
listbox .lb
grid .lb –row 0 –column 1

# The scrollbar slider will start at the 1/3 position,
#   and stop at the 9/10 position.
.sb set .3 .9
```

Script Output



A script can also query a scrollbar to learn the positions of the slider with the `get` subcommand.

Syntax: *scrollbarName* `get`

Returns the current state of the widget. This will be the result of the most recent `set` command.

Example 38

Script Example

```
scrollbar .sb
.sb set .3 .8
puts "Start and end fractions are: [.sb get]"
```

Script Output

```
Start and end fractions are: 0.3 0.8
```

11.9.3 Intercepting `scrollbar` Commands

The next example shows how you can use this knowledge about how the scrollbar works to use a single scrollbar to control two `listbox` widgets. This example uses a previously unmentioned subcommand of the `listbox` widget.

Syntax: *`listboxName`* `size`
Returns the number of entries in a `listbox`.

Example 39

Script Example

```
# Create two listboxes
listbox .leftbox —height 5 —exportselection 0
listbox .rightbox —height 5 —exportselection 0

# And fill them. The right box has twice as many entries as
#  the left.
for {set i 0} {$i < 10} {incr i} {
  .leftbox insert end "Left Line $i"
  .rightbox insert end "Right Line $i"
  .rightbox insert end "Next Right $i"
}

# Display the listboxes.
grid .leftbox —column 0 —row 0
grid .rightbox —column 2 —row 0

# Create the scrollbar, set the initial slider size, and
# display

scrollbar .scroll —command \
  "moveLists .scroll .leftbox .rightbox"

# The right listbox is displaying 5 of 20 lines
.scroll set 0 [expr  5.0 / 20.0]
grid .scroll —column 1 —row 0 —sticky ns

##########################################################################
# proc moveLists {scrollbar listbox1 listbox2 args} ——
#     Controls two listboxes from a single scrollbar
#     Shifts the top displayed entry and slider such that both
#     listboxes start and end together. The list with the most
#     entries will scroll faster.
#
# Arguments
```

```
# scrollbar The name of the scrollbar
# listbox1  The name of one listbox
# listbox2  The name of the other listbox
# args      The arguments appended by the scrollbar widget
#
# Results
#     No valid return.
#     Resets displayed positions of listboxes.
#     Resets size and location of scrollbar slider.

proc moveLists {scrollbar listbox1 listbox2 args} {

  # Get the height for the listboxes — assume
  #     both are the same.

  set height [$listbox2 cget —height]

  # Get the count of entries in each box.
  set size1 [$listbox1 size]
  set size2 [$listbox2 size]
  if {$size1 > $size2} {
     set size ${size1}.0
  } else {
    set size ${size2}.0
  }

  # Get the current scrollbar location
  set scrollPosition [$scrollbar get]
  set startFract [lindex $scrollPosition 0]

  # Calculate the top displayed entry for each listbox
  set top1 [expr int($size1 * $startFract)]
  set top2 [expr int($size2 * $startFract)]

  # Parse the arguments added by the scrollbar widget
  set cmdlst [split $args]

  switch [lindex $cmdlst 0] {
  "scroll" {
    # Parse count and unit from cmdlst
    foreach {sc count unit} $cmdlst {}

    # Determine whether the arrow or the bar was
    # clicked (is the command "scroll 1 unit"
    # or "scroll 1 page")
    if {[string first units $unit] >= 0} {
      set increment [expr 1 * $count];
```

```
    } else {
      set increment [expr $height * $count];
    }

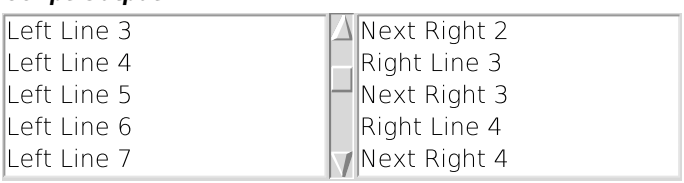
    # Set the new fraction for the top of the list
    set topFract1 [expr ($top1 + $increment)/$size]
    set topFract2 [expr ($top2 + $increment)/$size]
    if {$topFract1 < 0} {set topFract1 0}
    if {$topFract2 < 0} {set topFract2 0}
  }
  "moveto" {
    # Get the fraction of the list to display as top
    set topFract [lindex $cmdlst 1]
    if {$topFract < 0} {set topFract 0}

    # Scale the display to the number of entries in
    # the listbox
    set topFract1 [expr $topFract * ($size1/$size)]
    set topFract2 [expr $topFract * ($size2/$size)]
   }
  }

  # Move the listboxes to their new location
  $listbox1 yview moveto $topFract1
  $listbox2 yview moveto $topFract2

  # Reposition the scrollbar slider
  set topFract [expr ($topFract1 > $topFract2) \
      ? $topFract1 : $topFract2]
  if {$topFract > (1.0 - ($height-1)/$size)} {
    set topFract [expr (1.0 - ($height-1)/$size)]
  }
  set bottomFract [expr $topFract + (($height-1)/$size)]
  $scrollbar set $topFract $bottomFract
}
```

Script Output



Note the calls to `yview` in the previous example. The `yview` and `xview` subcommands set the start location for the data in a `listbox`. The first argument (`scroll` or `moveto`) is used to determine how to interpret the other arguments.

When a scrollbar and a listbox are connected in the usual manner, with a line resembling

```
scrollbar .scroll -command ".box yview"
```

the scrollbar widget will append arguments describing how to modify the data to the arguments supplied in the -command argument, and the new string will be evaluated. The arguments appended will start with either the word scroll or the word moveto. For example, if an arrow were clicked in scrollbar .scroll, the command evaluated would be:

```
.box yview scroll 1 unit
```

The .box procedure would parse the first argument (yview) and evaluate the yview procedure. The yview code would parse the argument scroll 1 unit to determine how the listbox should be modified to reflect scrolling one unit down.

In the previous example, the slider does not behave exactly as described for the default scrollbar procedures. Because we are scrolling lists of two different sizes, the slider size is set to reflect the fraction of data displayed from the larger listbox. The position of the slider reflects the center of the displayed data, rather than the start point. By changing the parameters to $scrollbar set, you can modify that behavior. For instance, you could position the slider to reflect the condition of one listbox and treat the other listbox as a slave.

11.10 THE scale WIDGET

The scale widget allows a user to select a numeric value from within a given range. It creates a bar with a slider, similar to the scrollbar widget, but without arrows at the ends. When the user moves the slider, the scale widget can either evaluate a procedure with the new slider location as an argument or set a defined variable to the new value, or perform both actions.

Syntax: scale *scaleName ?options?*

Create a scale widget.

scaleName The name for this scale widget.

?options?

There are many options for this widget. The minimal set is:

-orient *orientation*	Whether the scale should be drawn horizontally or vertically. *orientation* may be horizontal or vertical. The default orientation is vertical.
-length *numPixels*	The size of this scale. The height for vertical widgets and the width for horizontal widgets. The *height* may be in any valid distance value (as described in Section 11.2.3).
-from *number*	One end of the range to display. This value will be displayed on the left side (for horizontal scale widgets) or top (for vertical scale widgets).
-to *number*	The other end for the range.

`-label` *text*	The label to display with this `scale`.
`-command` *script*	The command to evaluate when the state changes. The new value of the slider will be appended to this string, and the resulting string will be evaluated.
`-variable` *varName*	A variable that will contain the current value of the slider.
`-resolution` *number*	The resolution to use for the scale and slider. Defaults to 1.
`-tickinterval` *number*	The resolution to use for the scale. This does not affect the values returned when the slider is moved.

The next example shows two `scale` widgets being used to display temperatures in Celsius and Fahrenheit scales. You can move either slider and the other slider will change to display the equivalent temperature in the other scale.

Example 40

Script Example

```
# Convert the Celsius temperature to Fahrenheit
proc celsiusTofahren {ctemp} {
  global fahrenheit
  set fahrenheit [expr ($ctemp*1.8) + 32]
}

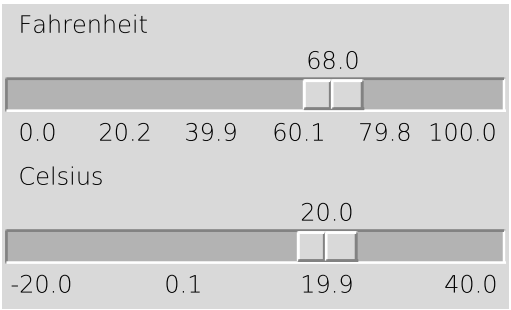
# Convert the Fahrenheit temperature to Celsius
proc fahrenToCelsius {ftemp} {
  global celsius
  set celsius [expr ($ftemp-32)/1.8]
}

# Create a scale for fahrenheit temperatures
set fahrenscale [scale .fht -orient horizontal \
    -from 0 -to 100 -length 250 \
    -resolution .1 -tickinterval 20 -label "Fahrenheit" \
    -variable fahrenheit -command fahrenToCelsius]

# Create a scale for celsius temperatures.
set celscale [scale .cel -orient horizontal \
    -from -20 -to 40 -length 250 \
    -resolution .1 -tickinterval 20 -label "Celsius" \
    -variable celsius -command celsiusTofahren]

# Grid the widgets.
grid $fahrenscale
grid $celscale
```

Script Output



11.11 NEW WINDOWS

When the `wish` interpreter is initialized, it creates a top-level graphics window. This window will be drawn with whatever decorations your display system provides and will expand to fit whatever other widgets are placed within it. If you find you need another, separate window, one can be created with the `toplevel` command.

Syntax: `toplevel` *`windowName ?options?`*

Creates a new top-level window.

`windowName`	The name for the window to create. The name must start with a period and conform to the widget naming conventions described in Section 11.2.1.
`?options?`	Valid options for the `toplevel` widget include:
`-relief` *`value`*	Sets the relief for this window. The `value` may be `raised`, `sunken`, `ridge`, or `flat`. The default is flat.
`-borderwidth` *`size`*	Sets a border to be `size` wide if the `-relief` option is not `flat`. The `size` parameter can be any dimensional value, as described in Section 11.2.3.
`-background` *`color`*	The base color for this widget. The `color` may be any color value, as described in Section 11.2.2.
`-height`	The requested height of this window, in units as described in Section 11.2.3.
`-width`	The requested width of this window, in units as described in Section 11.2.3.

Example 41

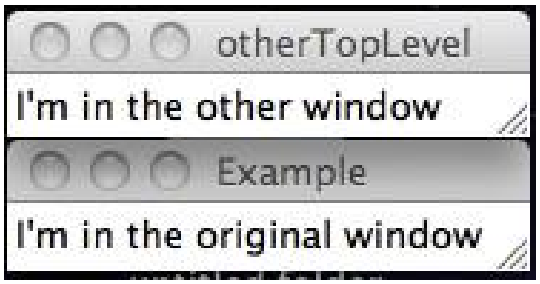
Script Example

```
# Create a label in the original window
label .l —text "I'm in the original window"
```

```
# Create a new window, and a label for it
toplevel .otherTopLevel
label .otherTopLevel.l –text "I'm in the other window"

# Display the labels.
grid .l
grid .otherTopLevel.l
```

Script Output



By default, the window name is shown in the top window decoration. This can be modified with the `wm title` command. For example, this command:

```
wm title. "My Application"
```

will change the name in a main application window. The `wm` command gives the Tk programmer access to the services provided by the Window manager. These services vary slightly between window managers and operating systems. You should read the on-line documentation for the subcommands supported by the `wm` command.

11.12 INTERACTING WITH THE EVENT LOOP

The Tk event loop is processed whenever the interpreter is not evaluating a command or procedure. It is best to write your code to spend as little time as possible within procedures. However, some tasks just plain take a while, and you may need to schedule passes through the event loop while your procedure is running.

A classic error in event-driven GUI programming is to place a command that modifies the display inside a loop but not to force the display to update. (For example, a loop that performs some lengthy calculation might update a completion bar.) If the event loop is not entered, only the final graphic command will be evaluated. The loop will run to completion without modifying the display, and then, suddenly, the display will show the completed image. You can force the Tcl interpreter to evaluate the event loop with the `update` command.

Syntax: `update ?idletasks?`

Process the event loop until all pending events have been processed.

idletasks Do not process user events or errors. Process only pending internal requests, such as updating the display.

The next example shows a small loop to put characters into a label. Without the update in the loop, the task would pause for several seconds and then display the complete string. The update command causes the characters to be displayed one at a time.

Example 42

Script Example

```
# Create the label.
label .l –text "" –width 25
grid .l
# Assign some text.
set str "Tcl makes programming Fun"
# And add new text to the label one character at a time.
for {set i 1} {$i < [string length $str]} {incr i} {
  .l configure –text [string range $str 0 $i]
  update idle
  # Mark time for a second or so.
  #   (Better delay technique described in the next section)
  for {set j 0} {$j < 1000} {incr j} {
    set x [expr $j * .02]
  }
}
```

Script Output

After 5 seconds	After 12 seconds	After 20 seconds
Tcl makes	Tcl makes progr	Tcl makes programming Fun

11.13 SCHEDULING THE FUTURE: after

The previous example uses a busy loop to cause the script to pause between inserting characters into the label. This is a pretty silly waste of CPU time, and Tcl provides a better way to handle this. The after command will perform one of three tasks.

- If invoked with a single numeric argument, it will pause for that many milliseconds.
- If invoked with a numeric argument and a script, it will schedule that script to be run the requested number of milliseconds in the future and continue processing the current script.
- If invoked with a subcommand, it provides support to examine and cancel items from the list of scheduled tasks.

Syntax: after *milliseconds ?script?*

Pause processing of the current script or schedule a script to be processed in the future.

milliseconds	The number of milliseconds to pause the current processing, or the number of seconds in the future to evaluate another script.
script	If this argument is defined, this script will be evaluated after *milliseconds* time has elapsed. The after command will return a unique key to identify this event.

Syntax: `after` *`subcommand option`*

Manipulate the scheduled queue of timer events.

`subcommand` If the second argument is a subcommand name, that subcommand is evaluated. Some of these are discussed in the next section.

The next example shows the `after` command being used instead of the busy loop. Then it shows events being scheduled in the future to remove the characters from the label.

Example 43

Script Example

```
# Create the label.
label .l -text "" -width 40
grid .l

# Assign some text.
set str "Tcl makes programming Fun"

# And add new text to the label one character at a time.
for {set i 1} {$i < [string length $str]} {incr i} {
  .l configure -text [string range $str 0 $i]
  update
  after 1000
}

##################################################
# proc shortenText {widget} --
#  Remove the first character from the displayed string of a
#  widget
# Arguments
#  widget     The name of a widget with a -text option
#
# Results
# Removes the first character from the text string of the
# provided widget if one exists.
#
# Schedules itself to run again if the -text string wasn't
# empty.
proc shortenText {widget} {
  # Get the current text string from the widget
  set txt [$widget cget -text]

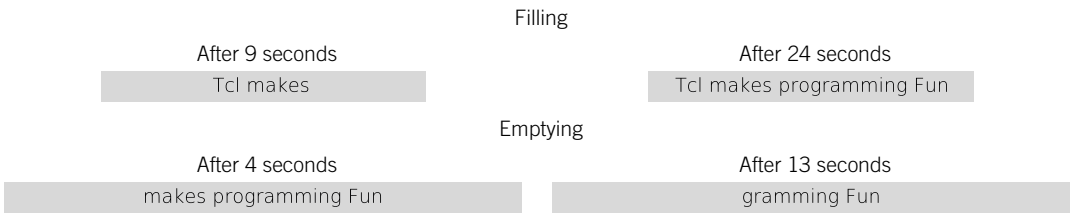
  # If it's empty, we're done.
  if {$txt eq ""} {
    return;
  }
```

```
    # shorten the string
    set txt [string range $txt 1 end]

    # Update the widget
    $widget configure –text $txt

    # And schedule this procedure to be run again in 1 second.
    after 1000 shortenText $widget
}
shortenText .l
```

Script Output



The first loop (filling the widget) is the type of process that occurs immediately to programmers who are used to working in nonevent-driven paradigms. The style of programming demonstrated by the `shortenText` procedure's use of the `after` command is less intuitive but more versatile.

In the first style, the event loop is checked only once per second. If there were an `ABORT` button, there would be a noticeable lag between a button click and the task being aborted. Using the second style, the GUI will respond immediately.

Note that the `shortenText` procedure is not recursive. It does not call itself, it calls the `after` command to schedule itself to be called again.

11.13.1 Canceling the Future

Sometimes, after you have carefully scheduled things, plans change and schedules need to change. When the `after` command is evaluated, it returns a handle for the event that was scheduled. This handle can be used to access this item in the list of scripts scheduled for future evaluation.

Syntax: `after cancel` *`handle`*

Cancel a script that was scheduled to be evaluated.

`handle` The handle for this event that was returned by a previous `after` *`milliseconds script`* command.

We will use the `after cancel` command in the next chapter.

Example 44

```
# Schedule the task to exit in 10 seconds
set exitEvent [after 10000 exit]
button .b -text "Click me to cancel exit!" \
   -command "after cancel $exitEvent"
pack .b
```

If your script needs information about events in the `after` queue, you can query the queue with the `after info` command.

Syntax: `after info` *?handle?*

Returns information about items in the `after` queue.

?handle? If `after info` is invoked with no `handle`, it will return a list of all the handles in the queue. If invoked with a *handle* argument, `after info` will return a list consisting of:

1. The script associated with this item.
2. The word `idle` or `timer`. The word `idle` indicates that this script will be evaluated when the idle loop is next run (no other script requires processing), whereas `timer` indicates that the event is waiting for the timer event when the requested number of milliseconds has elapsed.

Example 45

```
# ... many events added to timer queue
# Delete the exit event from the queue
foreach id [after info] {
  if {[string first exit [after info \$id]] >= 0} {
     after cancel \$id
  }
}
```

11.14 BOTTOM LINE

This chapter has introduced most of the simple Tk widgets. The `text` widget, `canvas` widget, and megawidgets (file boxes, color selectors, and so on) are covered in the next three chapters.

- The Tk primitive widgets provide the tools necessary to write GUI-oriented programs with support for buttons, entry widgets, graphics display widgets, scrolling list boxes, menus, and numeric input.

- Widget names must start with a period followed by a lowercase letter. (See Section 11.2.1.)
- Colors can be declared by name, or red/green/blue intensity values. (See Section 11.2.2.)
- Dimensions and sizes can be declared as pixels, inches, millimeters, or points. (See Section 11.2.3.)
- The default widget configuration options are adequate for most uses. Options can be set when the widget is created with `-option value` pairs, or modified after a widget is created with the *`widgetName`* `configure` command.
  Syntax: *`widgetName`* `configure` *`?opt1? ?val1?... ?optN? ?valN?`*
- The value of a widget's option is returned by the *`widgetName`* `cget` command.
  Syntax: *`widgetName`* `cget` *`option`*
- A label will display a single line of text.
  Syntax: `label` *`labelName ?option1? ?option2?...`*
- A `button` widget will evaluate a script when it is clicked.
  Syntax: `button` *`buttonName ?option1? ?option2?...`*
- The `entry` widget will allow the user to enter text.
  Syntax: `entry` *`entryName ?options?`*
- A `frame` widget can be used to hold other widgets. This makes it easier to design and maintain a GUI.
  Syntax: `frame` *`frameName ?options?`*
- Radio and check buttons let a user select from several options.
  Syntax: `radiobutton` *`radioName ?options?`*
  Syntax: `checkbutton` *`checkName ?options?`*
- A menu contains elements that can be activated or selected. The elements can be commands, radio buttons, check buttons, separators, or cascading menus.
  Syntax: `menu` *`menuName ?options?`*
- A menu can be attached to a `menubutton`, or to a window's `menubar`.
  Syntax: `menubutton` *`buttonName ?options?`*
  `$windowname` `configure -menu` *`$menuName`*
- A listbox displays multiple text elements and allows a user to select one or more of these elements.
  Syntax: `listbox` *`listboxName ?options?`*
- A scrollbar can be connected to a `listbox`, `text`, or `canvas` widget.
  Syntax: `scrollbar` *`scrollbarName ?options?`*
- The `scale` widget lets a user select a numeric value from a range of values.
  Syntax: `scale` *`scaleName ?options?`*
- Tk widgets can be arranged on the display using the `place`, `grid`, or `pack` layout manager.
- New independent windows are created with the `toplevel` command.
  Syntax: `toplevel` *`windowName ?options?`*
- The `update` command can be used to force a pass through the event loop during long computations.
- The `after` command can be used to pause an application or to schedule a script for evaluation in the future.
- The `after cancel` command can be used to cancel a script that was scheduled to be evaluated in the future.

11.15 PROBLEMS

The following numbering convention is used in all Problem sections:

Number Range	Description of Problems
100–199	Short comprehension problems review material covered in the chapter. They can be answered in a few words or a 1-5-line script. These problems should each take under a minute to answer.
200–299	These quick exercises require some thought or information beyond that covered in the chapter. They may require reading a man page or making a web search. A short script of 1–50 lines should fulfill the exercises, which may take 10–20 minutes each to complete.
300–399	Long exercises may require reading other material or writing a few hundred lines of code. These exercises may take several hours to complete.

- *100* Which of the following are valid window names? What is the error in the invalid names?
  a. .b
  b. .Button
  c. .button
  d. .b2
  e. .2b
  f. .buttonFrame.b1
  g. .top.buttons.b1-quit
  h. ..button
  i. .b.1-quit
- *101* What conventions does Tcl use to define colors?
- *102* What conventions does Tcl use to define screen distances?
- *103* What option will set the color of the text in a button?
- *104* What is the difference between the return value of
  ```
  $widget cget -foreground
  ```
  and
  ```
  $widget configure -foreground
  ```
- *105* How many lines of text can be displayed in a `label` widget?
- *106* Can a button's `-command` option contain more than one Tcl command?
- *107* Can you use `place` to display a widget in a frame in which you are also using the `pack` command?
- *108* How many items in a set of `radiobuttons` can be selected simultaneously?
- *109* What types of items can be added to a menu?
- *110* What widgets include scrollbar support by default?
- *111* What is the time unit for the `after` command?
- *112* What command will check for pending events?
- *200* Write a GUI widget with an entry widget, label, and button. When the button is clicked, convert the text in the `entry` widget to pig Latin and display it in the label. The pig Latin rules are:
  - Remove the first letter.

  - If the first letter is a vowel, append *t* to the end of the word.
  - Append the first letter to the end of the word.
  - Append the letters *ay* to the end of the word.
- *201* Create a button and command pair in which each time the button is clicked it will change its background color. The text will inform the user of what the next color will be. The color change can be done with a list of colors, or by using an associative array as a lookup table.
- *202* Write a data entry GUI that will use separate entry widgets to read First Name, Last Name, and Login ID. When the user clicks the Accept button, the data will be merged into proper location in a listbox. Insert new entries alphabetically by last name. Attach a scrollbar to the listbox.
- *203* Add a Save button to the previous exercise. The command associated with this button will open a file and use Tcl commands to insert data into the listbox. Add a Restore button with a command to clear the listbox and load the data using the `source` command. Information in the save file will resemble the following.

```
.listbox insert end "Doe, John: john.doe"
.listbox insert end "Flynt, Clif: clif"
```

- *204* Write a busy-bar application that will use a label to add characters to the text in a `label` widget to indicate a procedure's progress.
- *205* Turn the example in Section 6.4 into a GUI application. Add an entry widget to set the starting directory, a button to start searching the directories, a label to show the directory currently being examined, and a scrolling listbox to display the results. The text in the listbox should have leading spaces to denote which files are within previous directories. You may need to create the listbox with `-font {courier}` to display the leading spaces.
- *206* Write a GUI with two sliders and a label. The label will display the result of dividing the value in the first slider by the value in the second slider.
- *207* Write a procedure that will create a pop-up window using the toplevel command with an information label, and an OK button that will destroy the pop-up when clicked.
- *208* A top-level window can appear anywhere on the screen. Write a procedure named `frame-toplevel` that will accept as a single argument the name of a frame to create. The procedure should create the frame, place it in the center of the parent window, and return the name of the new frame. Change the `toplevel` command in the previous exercise to `frame-toplevel`. What is different between the two implementations?
- *300* Modify the `frame-toplevel` procedure from the previous exercise to accept an undefined number of arguments (for example, `-background yellow -relief raised -borderwidth 3`) and use those arguments to configure the new `frame` widget.
- *301* A Pizza class was created in Problem 9.204. Make a GUI front end that will allow a user to select a single size or style, and multiple toppings with a Done button to create a new Pizza object.
- *302* Add a button to the result of the previous exercise that will generate a description of all Pizza objects in a scrolled listbox within a new top-level window.

CHAPTER

Using the canvas Widget

12

Chapter 11 described many of the Tk widgets that support GUI programming. This chapter discusses the canvas widget, Tcl events, interaction with the window manager, and use of the Tcl image object. You will learn how events can be connected to a widget or an item drawn on a canvas, and how to use a canvas to build your own GUI widgets.

The canvas is the Tk widget drawing surface. You can draw lines, arrows, rectangles, ovals, text, or random polygons in various colors with various fill styles. You can also insert other Tk windows, or images in X-Bitmap, PPM (Portable Pixmap), PGM (Portable GrayMap), or GIF (Graphical Interface Format) format (other formats are supported in various Tk extensions, discussed in Chapter 17). If that is not enough, you will explore writing C code to add new drawing types to the canvas command.

As mentioned in Chapter 11, Tk is event oriented. Whenever a cursor moves or a key is pressed, an event is generated by the windowing system. If the focus for your window manager is on a Tk window, that event is passed to the Tk interpreter, which can then trigger actions connected with graphic items in that window.

The bind command links scripts to the events that occur on graphic items such as widgets, or items displayed in a canvas or text widget. Events include mouse motion, button clicks, key presses, and window manager events such as refresh, iconify, and resize. After an examination of the canvas and bind commands, you will see how to use them to create a specialized button widget, similar to those in Netscape, IE, and other packages.

12.1 OVERVIEW OF THE canvas WIDGET

Your application may create one or more canvas widgets and map them to the display with any of the layout managers pack, grid, or place. The canvas owns all items drawn on it. These items are maintained in a display list and their position in that list determines which items will be drawn on top of others. The order in which items are displayed can be modified.

12.1.1 Identifiers and Tags

When an item is created on a canvas, a unique numeric identifier is returned for that item. The item can always be identified and manipulated by referencing that unique number. You can also attach a tag to an item. A tag is an alphanumeric string that can be used to group canvas items. A single tag can be attached to multiple items, and multiple tags can be attached to a single item.

Tcl/Tk: A Developer's Guide. DOI: 10.1016/B978-0-12-384717-1.00012-8

A set of items that share a tag can be identified as a group by that tag. For example, if you composed a blueprint of a house, you could tag all of the plumbing with `plumbing` and then tag the hot water pipes with `hot`, the cold water pipes with `cold`, and the drains with `drain`. You could highlight all of the plumbing with a command such as the following.

```
$blueprint itemconfigure plumbing -outline green
```

You could highlight just the hot water lines with a command such as the following.

```
$blueprint itemconfigure hot -outline red
```

The following two tags are defined by default in a canvas widget.

`all` Identifies all of the drawable items in a canvas.

`current` Identifies the topmost item selected by the cursor.

12.1.2 Coordinates

The canvas coordinate origin is the upper left-hand corner. Larger `Y` coordinates move down the screen, and larger `X` coordinates move to the right. The coordinates can be declared in pixels, inches, millimeters, centimeters, or points (1/72 of an inch), as described for dimensions in Section 11.2.3.

The coordinates are stored internally using floating-point format, which allows a great deal of flexibility for scaling and positioning items. Note that even pixel locations can be fractional.

12.1.3 Binding

You can cause a particular script to be evaluated when an event associated with a displayed item occurs. For instance, when the cursor passes over a button, an `<Enter>` event is generated by the window manager. The default action for the enter event is to change the button color to denote the button as active. Some widgets, such as the `button` widget, allow you to set the action for a button click event during widget creation with the `-command` option. All widgets can have actions linked to events with the `bind` command, discussed in detail later in the chapter.

The canvas widget takes the concept of binding further and allows you to bind an action to events that occur on each drawable item within the canvas display list. For instance, you can make lines change color when the cursor passes over them, or bind a procedure to a button click on an item.

You can bind actions to displayed items either by tag (to cause an action to occur whenever an item tagged as `plumbing` is clicked, for instance) or by ID (to cause an action when a particular graphic item is accessed). For instance, you could bind a double mouse click event to bringing up a message box that would describe an item. A plumber could then double click on a joint and see a description of the style of fitting to use, where to purchase it, and an expected cost.

12.2 CREATING A `canvas` WIDGET

Creating a canvas is just as easy as creating other Tk widgets. You simply invoke the `canvas` command to create a canvas and then tell one of the layout managers where to display it. As with the other Tk commands, a command will be created with the same name as the `canvas` widget. You will use this command to interact with the canvas, creating items, modifying the configuration, and so on.

Syntax: `canvas` *`canvasName ?options?`*

Create a canvas widget.

`canvasName` The name for this canvas.

`?options?`

Some of the options supported by the canvas widget are:

`-background color`	The color to use for the background of this image. The default color is light gray.
`-closeenough distance`	Defines how close a mouse cursor must be to a displayable item to be considered on that item. The default is 1.0 pixel. This value may be a fraction and must be positive.
`-scrollregion boundingBox`	Defines the size of a `canvas` widget. The bounding box is a list, `left top right bottom`, which defines the total area of this canvas, which may be larger than the displayed area. These coordinates define the area of a `canvas` widget that can be scrolled into view when the canvas is attached to a `scrollbar` widget. This defaults to `0 0 width height`, the size of the displayed `canvas` widget.
`-height size`	The height of the displayed portion of the canvas. If `-scrollregion` is declared larger than this and scrollbars are attached to this canvas, this defines the height of the window into a larger canvas. The `size` parameter may be in pixels, inches, millimeters, and so on.
`-width size`	The width of this `canvas` widget. Again, this may define the size of a window into a larger canvas.
`-xscrollincrement size`	The amount to scroll when scrolling is requested. This can be used when the image you are displaying has a grid nature, and you always want to display an integral number of grid sections and not display half-grid sections. By default, this is a single pixel.
`-yscrollincrement size`	The amount to scroll when scrolling is requested. This can be used when the image you are displaying has a grid nature, and you always want to display an integral number of grid sections and not display half-grid sections. By default, this is a single pixel.

12.3 CREATING DISPLAYABLE canvas ITEMS

As with other Tk widgets, when you create a `canvas` widget you also create a procedure with the same name as the widget. Your script can use this procedure to interact with the canvas to perform actions such as drawing objects, moving objects, reconfiguring height and width, and so on.

The subcommand for drawing objects on a canvas is the `create` subcommand. This subcommand creates a graphic item in a described location. Various options allow you to specify the color, tags, and so on for this item. The `create` subcommand returns the unique number that can be used to identify this drawable item.

The `create` subcommand includes a number of options that are specific to the class of item. Most drawable items support the following options, except when the option is not applicable to that class of drawable. For instance, line width is not applicable to a text item.

`-width width`	The width of line to use to draw the outline of this item.
`-outline color`	The color to draw the outline of this item. If this is an empty string, the outline will not be drawn. The default color is black.
`-fill color`	The color with which to fill the item if the item encloses an area. If this is an empty string (the default), the item will not be filled.
`-stipple bitmap`	The stipple pattern to use if the `-fill` option is being used.
`-tag tagList`	A list of tags to associate with this item.

The items that can be created with the `create` subcommand are examined in the sections that follow.

12.3.1 The Line Item

Syntax: *canvasName* `create line` *x1 y1 x2 y2 ?xn yn? ?options?*

Create a polyline item. The `x` and `y` parameters define the ends of the line segments. Options for the `line` item include the following.

`-arrow end`	Specifies which end of a line should have an arrow drawn on it. The `end` argument may be one of
	`first` The first line coordinate
	`last` The end coordinate
	`both` Both ends
	`none` No arrow (default)
`-fill color`	The color to draw this line.
`-smooth boolean`	If set to `true`, this will cause the described line segments to be rendered with a set of Bezier splines.
`-splinesteps number`	The number of line segments to use for the Bezier splines if `-smooth` is defined true.

12.3.2 The Arc Item

Syntax: `canvasName create arc x1 y1 x2 y2 ?options?`

Create an arc. The parameters `x1 y1 x2 y2` describe a rectangle that would enclose the oval of which this arc is a part. Options to define the arc include the following.

`-start angle` The start location in degrees for this arc. The `angle` parameter is given in degrees. The 0-degree position is at 3:00 (the rightmost edge of the oval) and increases in the counterclockwise direction. The default is 0.

`-extent angle` The number of degrees counterclockwise to extend the arc from the start position. The default is 90 degrees.

`-style styleType` The style of arc to draw. This defines the area that will be filled by the `-fill` option. May be one of:

- `pieslice` The ends of the arc are connected to the center of the oval by two line segments (default).
- `chord` The ends of the arc are connected to each other by a line segment.
- `arc` Draw only the arc itself. Ignore `-fill` options.

`-fill color` The color to fill the arc, if `-style` is not `arc`.

12.3.3 The Rectangle Item

Syntax: *canvasName* `create rectangle x1 y1 x2 y2 ?options?`

Create a rectangle. The parameters `x1 y1 x2 y2` define opposite corners of the rectangle. The rectangle supports the usual `-fill`, `-outline`, `-width`, and `-stipple` options.

12.3.4 The Oval Item

Syntax: *canvasName* `create oval x1 y1 x2 y2 ?options?`

Create an oval. The parameters `x1 y1 x2 y2` define opposite corners of the rectangle that would enclose this oval. The oval supports the usual `-fill`, `-outline`, `-width`, and `-stipple` options.

12.3.5 The Polygon Item

Syntax: *canvasName* `create polygon x1 y1 x2 y2 ... xn yn ?options?`

Create a polygon. The `x` and `y` parameters define the vertexes of the polygon. A final line segment connecting `xn, yn` to `x1, y1` will be added automatically. The polygon item supports the same `-smooth` and `-splinesteps` options as the line item, as well as the usual `-fill`, `-outline`, `-width`, and `-stipple` options.

12.3.6 The Text Item

Syntax: *canvasName* `create text x y ?options?`

Create a `text` item. The `text` item has the following unique options:

`-anchor position` Describes where the text will be positioned relative to the `x` and `y` locations.

The `position` argument may be one of:

`center` Center the text on the position (default).

`n s e w ne se sw nw` One or more sides of the text to place on the position. If a single side is specified, that side will be centered on the x/y location. If two adjacent sides are specified, that corner will be placed on the x/y location. For example, if `position` is defined as `w`, the text will be drawn with the center of the left edge on the specified x/y location. If `position` is defined as `se`, the bottom rightmost corner of the text will be on the x/y location.

`-justify style` If the text item is multi-line (has embedded `newline` characters), this option describes whether the lines should be `right` justified, `left` justified, or `center` justified. The default is `left`.

`-text text` The text to display.

`-font fontDescriptor` The font to use for this text.

`-fill color` The color to use for this text item.

12.3.7 The Bitmap Item

Syntax: *canvasName* `create bitmap x y ?options?`

Create a two-color bitmap image. Options for this item include:

`-anchor position` This option behaves as described for the `text` item.

`-bitmap name` Defines the bitmap to display. May be one of:

`@file.xbm` Name of a file with bitmap data

`bitmapName` Name of one of the predefined bitmaps

The Tcl 8.6 distribution includes the following bitmaps for the PC, Mac, and UNIX platforms.

The Mac platform supports the following additional bitmaps.

12.3.8 The Image Item

Syntax: *canvasName* `create image x1 y1 ?options?`

Create a displayed image item. An image can be created from a GIF, PPM, PGM, or X-Bitmap image. The `image` command is discussed later in the chapter. The `create image` command was introduced several revisions after the `create bitmap` command. The `create image` is similar to the `create bitmap` command but can be used with more image formats and is more easily extended. The `create image` command uses the same `-anchor` option as the `create bitmap`. The image to be displayed is described with the `-image` options, as follows.

`-image imageName` The name of the image to display. This is the identifier handle returned by the `image` command.

12.3.9 An Example

The following example creates some simple graphic items to display a happy face.

Example 1

Script Example

```
# Create the canvas and display it.
canvas .c -height 140 -width 140 -background white
pack .c
# Create a nice big circle, colored gray
.c create oval 7 7 133 133 -outline black -fill gray80 -width 2

# And Two little circles for eyes
.c create oval 39 49 53 63 -outline black -fill black
.c create oval 102 63 88 49 -outline black -fill black
```

```
# A Starfleet insignia nose
.c create polygon 70 67 74 81 69 77 67 81 -fill black

# A big Happy Smile!
.c create arc 21 21 119 119 -start 225 -extent 95 -style arc \
    -outline black -width 3
```

Script Output

12.4 MORE canvas WIDGET SUBCOMMANDS

The create subcommand is only one of the subcommands supported by a canvas widget. This section introduces some of them, with examples of how they can be used.

12.4.1 Modifying an Item

You can modify a displayed item after creating it with the itemconfigure command. This command behaves like the configure command for Tk widgets, except that it takes an extra argument to define the item.

Syntax: *canvasName* itemconfigure *tagOrId ?opt1? ?val1? ?opt2 val2?*

Return or set configuration values.

tagOrId	Either the tag or the ID number of the item to configure.
opt1	The first option to set /query.
?val1?	If present, the value to assign to the previous option.

Example 2

Script Example

```
# Create a canvas with a white background
set canv [canvas .c -height 50 -width 300 -background white]

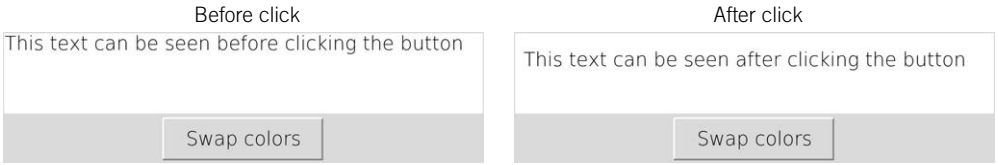
# Create some text colored the default black.
set top [ $canv create text 150 20 \
    -text "This text can be seen before clicking the button"]
```

```
# Create some text colored white.
# It won't show against the background.
set bottom [ $canv create text 150 30 -fill white \
    -text "This text can be seen after clicking the button"]

# Create a button which will use itemconfigure to change the
# colors of the two lines of text.
set colorswap [button .b1 -text "Swap colors" \
    -command "$canv itemconfigure $top -fill white;\
              $canv itemconfigure $bottom -fill black;"]

grid $canv
grid $colorswap
```

Script Output



12.4.2 Changing the Display Coordinates of an Item

The previous happy face example simply displayed some graphic items but did not save the identifiers. The next example creates the happy face and saves the item for one eye to use with the `canvas` widget subcommand `coords`. The `coords` subcommand lets you change the coordinates of a drawable item after the item has been displayed. You can use this to move an item or change an item's shape.

Syntax: *canvasName* `coords` *tagOrId ?x1 y1? ... ?xn yn?*

Return or modify the coordinates of the item.

`tagOrId`	A tag or unique ID that identifies the item.
`?x1 y1?...`	Optional `x` and `y` parameters. If these arguments are absent, the current coordinates are returned. If these are present, they will be assigned as the new coordinates for this item.

Example 3

Script Example

```
#######################################################
# proc wink {canv item }--
#  Close and re-open an 'eye' item.
# Arguments
#  canv    The canvas that includes this item.
#  item    An identifier for the 'eye' to wink
```

```
#
# Results
#  Converts Y coords for the specified item to center, then
#  restores them.
# The item is in its original state when this proc returns.

proc wink {canv item} {
  #
  # Get the coordinates for this item, and split them into
  #  left, bottom, right and top variables.
  set bounding [$canv coords $item]

  set left   [lindex $bounding 0]
  set bottom [lindex $bounding 1]
  set right  [lindex $bounding 2]
  set top    [lindex $bounding 3]

  set halfeye [expr int($top-$bottom)/2]

  # Loop to close the eye
  for {set i 1} {$i < $halfeye} {incr i } {
    $canv coords $item $left [expr $top - $i] $right \
        [expr $bottom + $i];
    update
    after 100
  }

  # Loop to re-open the eye
  for {set i $halfeye} {$i >= 0} {incr i -1} {
    $canv coords $item $left [expr $top - $i] \
        $right [expr $bottom + $i];
    update
    after 100
  }
}

# Create the canvas and display it.
canvas .c -height 140 -width 140 -background white
grid .c

# Create a nice big circle, colored gray
.c create oval 7 7 133 133 -outline black -fill gray80 -width 2

# And Two little circles for eyes
.c create oval 39 49 53 63 -outline black -fill black

set righteye [.c create oval 102 63 88 49 -outline black \
```

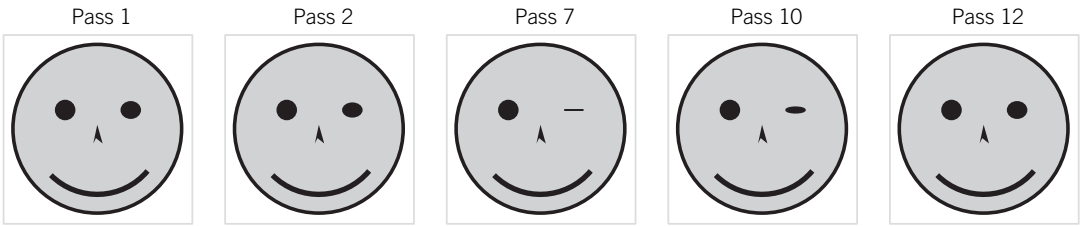
```
    -fill black]

# A Starfleet insignia nose
.c create polygon 70 67 74 81 69 77 67 81 -fill black

# A big Happy Smile!
.c create arc 21 21 119 119 -start 225 -extent 95 -style arc \
    -outline black -width 3

# Now, wink at the folks
wink .c $righteye
```

Script Output



Note that the `coords` subcommand, like most of the `canvas` widget subcommands, will accept either an ID or a tag to identify the item. The `righteye` item could also have been created with the following line.

```
.c create oval 102 63 88 49 -outline black \
    -fill black -tag righteye
```

The `wink` procedure would then be evaluated as follows.

```
wink .c righteye
```

Also notice the `update` command in the `wink` loops. If this were left out, the display would not update between changes. There would be a pause while the new images were calculated, but the display would never change.

12.4.3 Moving an Item

An item can be moved with the `move` subcommand, as well as with the `coords` subcommand. The `move` subcommand supports relative movement, whereas the `coords` command supports absolute positions.

Syntax: *canvasName* `move` *tagOrId xoffset yoffset*

Move an item a defined distance.

`tagOrId`	A tag or unique ID that identifies the item.
`xoffset yoffset`	The values to add to the item's current coordinates to define the new location. Positive values will move the item to the right and lower on the canvas.

For rectangle and oval items, the coordinates define the opposite corners and thus define the rectangle that would cover the item. In this case, it is easy to use the `coords` return to find the edges of the item.

If you have a multi-pointed polygon such as a star, finding the edges is a bit more difficult. The `canvas` widget subcommand `bbox` returns the coordinates of the top left and bottom right corners of a box that would enclose the item.

Syntax: *canvasName* bbox *tagOrId*

Return the coordinates of a box sized to enclose a single item or multiple items with the same tag.

`tagOrId` A tag or unique ID that identifies the item. If a tag is used and multiple items share that tag, the return is the bounding box that would cover all items with that tag.

The next example creates a star and bounces it around within a rectangle. Whenever the bounding box around the star hits an edge of the rectangle, the speed of the star is decreased and the direction is reversed.

Example 4

Script Example

```
# Create the canvas and display it
canvas .c -height 150 -width 150
grid .c

# Create an outline for the bouncing star's boundary
.c create rectangle 3 3 147 147 -width 3 -outline black \
-fill white

# Create a star item
.c create polygon 1 15 3 10 0 5 5 5 8 0 10 5 16 5 12 10 \
   14 15 8 12 -fill black -outline black -tag star

# Set the initial velocity of the star
set xoff 2
set yoff 3

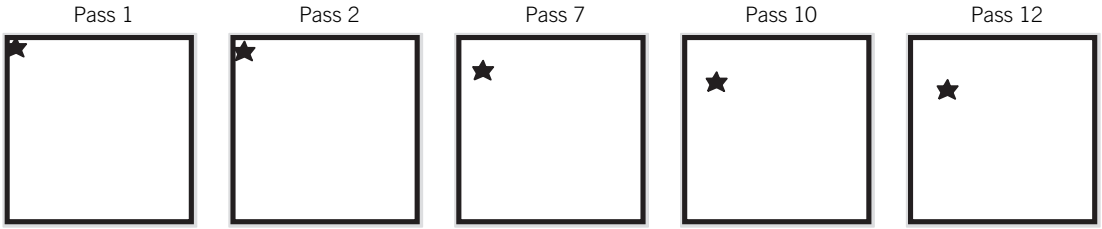
# Move the item forever
while {1} {
  set box [.c bbox star]
  set left [lindex $box 0]
  set top [lindex $box 1]
  set right [lindex $box 2]
  set bottom [lindex $box 3]

  # If the star has reached the left margin, heading left,
  # or the right margin, heading right, make it bounce.
  # Reduce the velocity to 90% in the bounce direction to
```

```
    # account for elasticity losses, and reverse the X
    # component of the velocity
    if {(($xoff < 0) && ($left <= 3)) ||
        (($xoff > 0) && ($right >= 147))} {
    set xoff [expr -.9 * $xoff]
    }

    # The same for the Y component of the velocity.
    if {(($yoff < 0) && ($top <= 3)) ||
        (($yoff > 0) && ($bottom >= 147))} {
      set yoff [expr -.9 * $yoff]
    }
    .c move star $xoff $yoff
    update
}
```

Script Output



Tk stores the location of items in floating-point coordinates. Thus, the bouncing star does not completely stop until the *x* and *y* velocities fall below the precision of the floating-point library on your computer. The star will just move more and more slowly until you get bored and abort the program.

12.4.4 Finding Items, and Raising and Lowering Items

The canvas can be searched for items that meet certain criteria, such as their position in the display list, their location on the canvas, or their tags. The `canvas` widget subcommand that searches a canvas is the `find` subcommand. It will return a list of unique IDs for the items that match its search criteria. The list is always returned in the order in which the items appear in the display list, the first item displayed (the bottom if there are overlapping items) first, and so on.

The items in a `canvas` widget are stored in a display list. The items are rendered in the order in which they appear in the list. Thus, items that appear later in the list will cover items that appear earlier in the list. An item's position in the display list can be changed with the `raise` and `lower` subcommands.

Syntax: *canvasName* `raise` *tagOrId ?abovetagOrId?*

Move the identified item to a later position in the display list, making it display above other items.

`tagOrId`	A tag or unique ID that identifies the item.
`?abovetagOrId?`	The ID or tag of an item that this item should be above. By default, items are moved to the end position in the display list (top of the displayed items). With this option, an item can be moved to locations other than the top. In fact, you can `raise` an item to just above the lowest item, effectively lowering the item.

Syntax: *canvasName* `lower` *tagOrId ?belowtagOrId?*

Move the identified item to an earlier position in the display list, making it display below other items.

`tagOrId`	A tag or unique ID that identifies the item.
`?belowtagOrId?`	The ID or tag of an item that this item should be directly below. By default, items are moved to the first position in the display list (bottom of the displayed items). With this option, an item can be moved to locations other than the bottom. You can `lower` an item to just below the top item, effectively raising the item.

Syntax: *canvasName* `find` *searchSpec*

Find displayable items that match the search criteria and return that list.

searchSpec

The search criteria. May be one of:

`all`	Return all items in a canvas in the order in which they appear in the display list.
`withtag Tag`	Returns a list of all items with this tag.
`above tagOrId`	Returns the single item just above the one defined by `tagOrId`. If a tag refers to multiple items, the last (topmost) item is used as the reference.
`below tagOrId`	Returns the single item just below the one defined by `tagOrId`. If a tag refers to multiple items, the first (lowest) item is used as the reference.
`enclosed x1 y1 x2 y2`	Returns the items totally enclosed by the rectangle defined by `x1, y1` and `x2, y2`. The `x` and `y` coordinates define opposing corners of a rectangle.
`overlapping x1 y1 x2 y2`	Returns the items that are partially enclosed by the rectangle defined by `x1, y1` and `x2, y2`. The `x` and `y` coordinates define opposing corners of a rectangle.
`closest x y ?halo? ?start?`	Returns the item closest to the X/Y location described with the `x` and `y` parameters.

If more than one item overlies the X/Y location, the one that is later in the display list is returned. When using the `closest` search specifier, the `halo` parameter is a positive integer that is added to the actual bounding box of items to determine whether they cover the X/Y location.

The `start` parameter can be used to step through a list of items that overlie the X/Y position. If this is present, the item returned will be the highest item that is before the item defined by the `start` parameter. If the display list contained the item IDs in the order 1 2 3 4 9 8 7 6 5, and all items overlapped position 100,50, the command

```
$canvasName find -closest 100 50 0
```

would return item 1, the lowest object in the display table.

The command

```
$canvasName find -closest 100 50 0 8
```

would return item 9, the item immediately below 8.

The next example shows some overlapping items and how their position in the display list changes the appearance of the display. This example redefines the default font for the `text` items. This is explained in the next section.

Note that the `upvar` command is used with the `array` argument. In Tcl, arrays are always passed by reference, not by value. Thus, if you need to pass an array to a procedure, you will call the procedure with the name of the array and use `upvar` to reference the array values.

Example 5

Script Example

```
# proc showDisplayList {canv array}--
#  Prints the Display List for a canvas.
#  Converts item ID's to item names via an array.
# Arguments
#  canv   Canvas to display from
#  array  The name of an array with textual names for the
#         display items
# Results
#  Prints output on the stdout.
 proc showDisplayList {canv array} {
   upvar $array id
   set order [$canv find all]
   puts "display list (numeric): $order"
   puts -nonewline "display list (text):   "
   foreach i $order {
     puts -nonewline "$id($i)"
   }
   puts ""
}

canvas .c -height 150 -width 170
pack .c
```

```
# create a light gray rectangle with dark text.

set tclSquare \
   [.c create rectangle 10 20 110 140 \
        -fill gray70 -outline gray70 ]
set tclText [.c create text 60 30 -text "Tcl\nIs\nTops" \
    -fill black -anchor n -font [list arial 20 bold ] \
    -justify center ]

# create a dark gray rectangle with white text.
set tkSquare \
    [.c create rectangle 60 20 160 140 -fill gray30 \
        -outline gray30 ]
set tkText [.c create text 110 30 -text "Tk\nTops\nTcl" \
    -fill white -anchor n -font [list arial 20 bold ] \
    -justify center ]

# Initialize the array with the names of the display items
#   linked to the unique number returned from the
#   canvas create.
foreach nm [list tclSquare tclText tkSquare tkText] {
  set id([subst $$nm]) "$nm"
}

 # Update the display and canvas
 update;

# Show the display list
puts "\nAt beginning"
showDisplayList .c id

# Pause to admire the view
after 1000

# Raise the tclSquare and tclText items to the end \
#   (top) of the display list
.c raise $tclSquare
.c raise $tclText

# Update, display the new list, and pause again
update;
puts "\nAfter raising tclSquare and tclText"
showDisplayList .c id
after 1000

# Raise the tkText to above the tclSquare (but below tclText)
```

```
.c lower $tkText $tclText

# Update and confirm.
update;
puts "\nAfter 'lowering' tkText"
showDisplayList .c id

puts ""
puts "Find reports that $id([.c find above $tclSquare]) \
  is above tclSquare"
puts "Find reports that $id([.c find below $tclSquare]) \
  is below tclSquare"
puts "and items: [.c find enclosed 0 0 120 150] are \
  enclosed within 0,0 and 120,15"
```

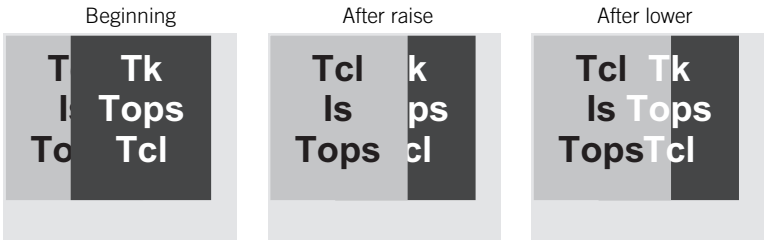
Script Output

```
At beginning
display list (numeric): 1 2 3 4
display list (text):   tclSquaretclTexttkSquaretkText

After raising tclSquare and tclText
display list (numeric): 3 4 1 2
display list (text):   tkSquaretkTexttclSquaretclText

After 'lowering' tkText
display list (numeric): 3 1 4 2
display list (text):   tkSquaretclSquaretkTexttclText

Find reports that tkText  is above tclSquare
Find reports that tkSquare  is below tclSquare
and items: 1 2 are  enclosed within 0,0 and 120,15
```



12.4.5 Fonts and Text Items

Fonts can be just as complex a problem as some 6,000 years of human endeavor with written language can make them. They can also be quite simple. The following discusses the platform-independent, simple method of dealing with fonts. Other methods are described in the on-line documentation under `font`. The pre-8.0 releases of Tk name fonts using the X Window system naming convention as follows.

```
-foundry-family-weight-slant-setwidth-addstyle-pixel-point-resx-resy-spacing-
        width-charset-encoding.
```

If you need to work with a version of Tk earlier than 8.0, you are probably on a UNIX system and can determine a proper name of an available font with the X11 command `xlsfonts` or `xfontsel`.

With the 8.0 release of Tcl and Tk, life became much simpler: a font is named with a list of attributes, such as `family ?size? ?style? ?style?` If the Tcl interpreter cannot find a font that will match the font you requested, it will find the closest match. It is guaranteed that a font will be found for any syntactically correct set of attributes. It's not guaranteed that you'll get the same font on different systems. For example, if you have a 19.5 point font defined for your system, but the end user does not, the display will be rendered with the closest available font, which might be 18 or 22 point.

The attributes that define a font are:

`family` The family name for a font. On all systems `times`, `helvetica`, and `courier` are defined. A list of the defined fonts can be obtained with the `font families` command.

`size` The size for this font. If this is a positive value, it is the size in points (1/72 of an inch). The scaling for the monitor is defined when Tk is initialized. If this value is a negative number, it will be converted to positive and will define the size of the font in pixels. Defining a font by point size is preferred and should cause applications to look the same on different systems.

`style` The style for this font. May be one or more of `normal` (or `roman`), `bold`, `italic`, `underline`, or `overstrike`.

Information about fonts can be obtained with the `font` command. The following discuss only some of the more commonly used subcommands.

Syntax: `font families ?-displayof` *`windowName?`*

Returns a list of valid font families for this system.

`-displayof windowName` If this option is present, it defines the window (`frame`, `canvas`, `topwindow`) to return this data for. By default, this is the primary graphics window ".".

The `font measure` command is useful when you are trying to arrange text on a display in an aesthetic manner.

Syntax: `font measure` *`font`* `?-displayof` *`windowName? text`*

The `font measure` command returns the number of horizontal pixels the string `text` will require if rendered in the defined font.

`font` The name of the font, as described previously.

`-displayof` *`windowName`* If this option is present, it defines the window (`frame`, `canvas`, `topwindow`) to return this data for. By default, this is the primary graphics window ".".

`text` The text to measure.

If Tk cannot obtain an exact match to the requested font on your system, it will find the best match. You can determine what font is actually being used with the `font actual` command.

Syntax: font actual *font* ?-displayof *windowName?*

Return the actual font that will be used when font is requested.

font	The name of the requested font.
-displayof *windowName*	If this option is present, it defines the window (frame, canvas, topwindow) to return this data for. By default, this is the primary graphics window ".".

With Tk 8.1, Tcl and Tk support the Unicode character fonts. These are encoded by preceding the four-digit Unicode value with \u. You can obtain information on the Unicode character sets at *http://Unicode.org*.

Unicode provides a standard method for handling letters other than the U.S. ASCII alphabet. The Unicode alphabet uses 16 bits to describe 65,536 letters. Each alphabet is assigned a range of numbers to represent its characters. With Unicode, you can represent Japanese Katakana, Korean Hangul, French, Spanish, Russian, and most other major alphabets.

The following example steps through the fonts available on a modern Linux system and displays some sample text in ASCII and Unicode. Note that if the Unicode fonts are not supported in a given font, the Unicode numbers are displayed instead. This example was run using Tcl/Tk version 8.6b2.

Example 6

Script Example

```
# Create a canvas and link it to a scrollbar
canvas .c -height 950 -width 800 -yscrollcommand {.ysb set}
scrollbar .ysb -orient vertical -command {.c yview}
grid .c .ysb -sticky ns
set ypos 15

# Get the list of available font families
set families [font families]

# Step through the available families
foreach family $families {

  .c create text 4 $ypos -font [list times 16 bold] \
      -text "$family" -anchor nw

  .c create text 240 $ypos -font [list $family 16] \
      -text "test line -- UNICODE: \u30Ab \u042f \u3072" \
      -anchor nw

  .c create text 4 [expr $ypos+25] -font [list roman 14] \
      -text "[font actual [list $family 16]]" -anchor nw
  .c create line 4 [expr $ypos+45] 800 [expr $ypos+45]
  incr ypos 50
}
```

```
update

# Configure the canvas's scroll area to be the
# bounding box of all available items.

.c configure -scrollregion [.c bbox all]
```

Script Output

12.4.6 Using a Canvas Larger than the View

The previous example more than filled the area of the canvas. The actual area that's being filled isn't known until run time when the script learns what fonts are available. The `-scrollregion` option lets you define how much of the canvas is to be controlled by a scrollbar.

When you create a canvas, you can provide `-width xsize` and `-height ysize` arguments to define the size of the canvas to display on the screen. If you do not also use a `-scrollregion` argument, the canvas will be `xsize` by `ysize` in size, and the entire canvas will be visible.

The -scrollregion command will let you declare that the actual canvas is larger than its visible portion. For example,

```
canvas .c -width 200 -height 150 -scrollregion {0 0 4500 2000}
```

creates a window 200 pixels wide and 150 pixels high into a canvas that is actually 4500 × 2000 pixels.

If you attach scrollbars to this canvas, you can scroll this 200 × 150 pixel window around the larger space. The -xscrollincrement and -yscrollincrement arguments will let you set how much of the canvas should scroll at a time. For instance, if you are displaying a lot of text in a canvas, you would want to set the -yscrollincrement to the height of a line. This will cause the canvas to display full lines of text, instead of displaying partial lines at the top and bottom of the canvas as you scroll through the text.

The next example shows a checkerboard pattern, with each square labeled to show its position in the grid. Because the -xscrollincrement is set to the size of a square, the full width of each square is displayed. Because the -yscrollincrement is half the size of a square, half the height of a square can be displayed.

Note that the -scrollregion can include negative coordinates as well as positive. You can draw any location in a canvas, but you will not be able to scroll to it unless you have set the -scrollregion to include the area you have drawn to.

If you have a complex image to create, and you do not know what the size will be until it is drawn, you can draw the image to a canvas and then determine the bounding box for the image with the *canvasName* bbox command.

```
set coords [$canvasName bbox all]
```

Example 7

Script Example

```
# Create the canvas with a small viewing area
# and a much larger scrolling area

canvas .c -width 200 -height 150 -scrollregion \
    {-100 -200 4500 2000} -yscrollcommand \
    {.scrollY set} -xscrollcommand {.scrollX set} \
    -xscrollincrement 50 -yscrollincrement 25

# Attach scrollbars
scrollbar .scrollY -orient vertical -command ".c yview"
scrollbar .scrollX -orient horizontal -command ".c xview"

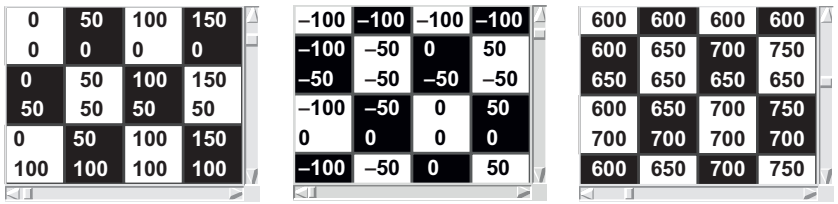
# grid the widgets
grid .c -row 0 -column 0
grid .scrollY -row 0 -column 1 -sticky ns
grid .scrollX -row 1 -column 0 -sticky ew

# Fill the canvas with alternating black & white
# boxes containing the box coordinates in a
```

```
# contrasting color

for {set y -200} {$y < 2000} {incr y 50} {
  set bottom [expr $y + 50]
  for {set x -100} {$x < 4500} {incr x 50} {
    set right [expr $x+50]
    if {(($y + $x) % 100) == 50} {
      set textColor white
      set fillColor black
    } else {
      set textColor black
      set fillColor white
  }
  .c create rectangle $x $y $right $bottom \
    -fill $fillColor -outline gray30
  .c create text [expr $x+25] [expr $y+25] \
    -text "$x\n$y" -font \
    [list helvetica 16 bold] -fill $textColor
  }
}
```

Script Output



12.5 THE `bind` AND `focus` COMMANDS

Section 12.1.3 mentioned that an event can be linked to a canvas item, and when that event occurs it can trigger an action. You can also bind actions to events on other widgets. This section discusses how this is done.

Most of the Tcl widgets have some event bindings defined by default. For example, when you click on a `button` widget, the default action for the `ButtonPress` event is to evaluate the script defined in the `-command` option.

12.5.1 The `bind` Command

The `bind` command links an event to a widget and a script. If the event occurs while the focus is on that widget, the script associated with that event will be evaluated.

Syntax: bind *widgetName eventType script*

Define an action to be executed when an event associated with this widget occurs.

widgetName The widget to have an action bound to it.

eventType The event to trigger this action. Events can be defined in one of three formats:

alphanumeric	A single printable (alphanumeric or punctuation) character defines a KeyPress event for that character.
<<virtualEvent>>	A virtual event defined by your script with the event command.
<modifier-type-detail>	This format precisely defines any event that can occur. The fields of an event descriptor are described in the following. Where two names are together, they are synonyms for each other. For example, either Button1 or B1 can be used to the left mouse button click.

script The script to evaluate when this event occurs.

The most versatile technique for defining events is to use the <modifier-type-detail> convention. With this convention, an event is defined as zero or more modifiers, followed by an event-type descriptor, followed by a detail field.

modifier There may be one or more occurrences of the modifier field, which describes conditions that must exist when an event defined by the type field occurs. For example, Alt and Shift may be pressed at the same time as a letter KeyPress. Not all computers support all available modifiers. For instance, few computers have more than three buttons on the mouse, and most general-purpose computers do not support a Mod key. The modifiers Double, Triple, and Quadruple define rapid sequences of two, three, or four mouse clicks. Valid values for modifier are

Button1	Button2	Button3	Button4	Button5
B1	B2	B3	B4	B5
Mod1	Mod2	Mod3	Mod4	Mod5
M1	M2	M3	M4	M5
Meta	Alt	Control	Shift	Lock
M				
Double	Triple	Quadruple		

`type` This field describes the type of event. Only one `type` entry is allowed in an event descriptor. Valid values for `type` are:

`Activate`	`Enter`	`Map`
`ButtonPress`, `Button`	`Expose`	`Motion`
`ButtonRelease`	`FocusIn`	`MouseWheel`
`Circulate`	`FocusOut`	`Property`
`Colormap`	`Gravity`	`Reparent`
`Configure`	`KeyPress`, `Key`	`Unmap`
`Deactivate`	`KeyRelease`	`Visibility`
`Destroy`	`Leave`	

`detail` The `detail` provides the final piece of data to identify an event precisely. The value the `detail` field can take depends on the content of the `type` field.

`type`	*`detail`*
`ButtonPress`	`detail` will be the number of a mouse button (1 through 5).
`ButtonRelease`	If the button number is specified, only events with that button will be bound to this action. If no `detail` is defined for a `ButtonPress` or `ButtonRelease` event, any mouse-button event will be bound to this action.
`KeyPress`	`detail` will specify a particular key, such as `a` or `3`.
`KeyRelease`	Non-alphanumeric keys are defined by names such as `Space` or `Caps_Lock`. An X11 manual will list all of these. On a UNIX/Linux system you can run `xkeycaps` (*www.jwz.org/xkeycaps/*) to obtain a list of the available key types. If no `detail` is defined for a `KeyPress` or `KeyRelease` event, any keyboard event will be bound to this action.

Event Definition Examples

Event Definition	Description
`<B1-Motion>`	This event is generated if the left mouse button is pressed and the mouse moves; that is, when you drag an item on a screen.
`a`	Generates an event when the `A` key is pressed.
`Shift_L`	This does not define an event. `Shift_L` is not an alphanumeric.
`<Shift_L>`	This event is generated when the left shift key is pressed. No event is generated when the key is released.

<KeyRelease-Shift_L> Generates an event when the left shift key is released.

<Control-Alt-Delete> Generates an event if all three are pressed simultaneously. Not a recommended event for MSDOS-based hardware platforms.

The bind command can pass information about the event that occurred to the script that is to be evaluated. This is done by including references to the events in the script argument to the bind command. The complete list of the information available about the event is defined in the bind on-line documentation. The information that can be passed to a script includes the following.

%x The X component of the cursor's location in the current window when the event occurs.

%y The Y component of the cursor's location in the current window when the event occurs.

%b The button that was pressed or released if the event is a ButtonPress or ButtonRelease.

%k The key that was pressed or released if the event is a KeyPress or KeyRelease.

%A The ASCII value of the key pressed or released if the event is a KeyPress or KeyRelease.

%T The type of event that occurred.

%W The path name of the window in which the event occurred.

```
bind .window <KeyPress> {keystroke %k}
```

Will invoke the procedure keystroke with the value of the key when a KeyPress event occurs and .window has focus.

```
bind .win <Motion> {moveitem %x %y}
```

Will invoke the procedure moveitem with the x and y location of the cursor when a Motion event occurs and .win has focus.

12.5.2 The canvas Widget bind Subcommand

Events can be bound to a drawable item in a canvas, just as they are bound to widgets. The command to do this uses the same values for eventType and script as the bind command uses.

Syntax: canvasName bind *tagOrId eventType script*

Bind an action to an event and canvas object.

tagOrId The tag assigned when an object is created, or the object identifier returned when an object is created.

eventType The event to trigger on, as described previously.

script A Tcl script to evaluate when this event occurs.

The following example is a network diagram showing the outside world, the firewall and four internal routers. When a mouse enters any of the router boxes a popup label will appear displaying the parameters of that router.

Example 8

Script Example

```
##########################################################################
# proc showStatus {x y item}--
#  Report the status of an item in the global dictionary.
#  In a real-world system, the dictionary would be updated
#    at intervals.
```

```
#
# Arguments
#   x     Cursor X location
#   y     Cursor Y location
#   item  Index into the dictionary
#
# Results
#   Temporary label is created.
#
proc showStatus {x y item} {
  global routerDict

  catch {destroy .statusLabel}

  set status "Name: $item\n"
  foreach {k v} [dict get $routerDict $item] {
    append status "$k:     $v\n"
  }

  label .statusLabel -text $status -relief solid
  place .statusLabel -x $x -y $y
}

##########################################################################
# proc drawItem {cvs itemDict item x y}--
#    Recursive draw to step through 'feeds' elements in dict
# Arguments
#   cvs         Canvas to draw on
#   itemDict    Dictionary of drawable items
#   item        The index of the item to draw
#   x           X location of center for item
#   y           Y location of center for item
#
# Results
#   Display is modified
#
proc drawItem {cvs itemDict item x y} {

  # First item in list is the outside world
  # Draw as oval.
  # All other items are internal routers.
  # Draw as rectangle.

  if {[lindex [dict keys $itemDict] 0] eq $item} {
    set type oval
  } else {
    set type rectangle
  }
```

```
  $cvs create $type [expr {$x - 40}] \
      [expr {$y - 20}] [expr {$x + 40}] \
      [expr {$y + 20}] -fill white -tag $item
  $cvs create text $x $y -text [string totitle $item] \
      -tag $item
  $cvs bind $item <Enter> "showStatus %x %y $item"

  # If this item doesn't feed anything, return.
  if {![dict exists $itemDict $item feeds]} {
    return
  }

  set feeds [dict get $itemDict $item feeds]
  set xPos  [expr {$x - ((([llength $feeds]-1) * 100) / 2)}]

  set yPrev [expr {$y + 20}]
  set yTop  [expr {$y + 50}]
  set yCenter [expr {$y + 70}]

  foreach feed $feeds {
    drawItem $cvs $itemDict $feed $xPos $yCenter
    $cvs create line $xPos $yTop $x $yPrev
    incr xPos 100
  }
}

# Define a dictionary with an entry for each element
# on the network.
#
# Each entry is a dict of parameters (speed, status, etc)

foreach item {internet firewall} \
        feeds {firewall {wireless 10-B-T Gig-Ethernet Fiber}} \
    status {up up} {
  dict lappend routerDict $item feeds $feeds status $status
}

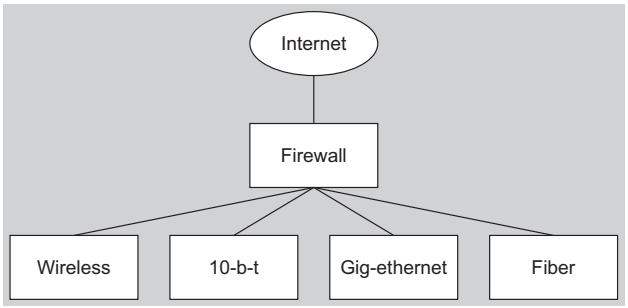
foreach sub {wireless 10-B-T Gig-Ethernet Fiber} \
        availability {public private private private} \
    speed {1-Mbps 10-Mbps 1000-Mbps 10000-Mbps} \
    status {down up up up} {

  dict lappend routerDict $sub availability $availability \
      status $status speed $speed
}
```

```
set cvs [canvas .c -width 500 -height 250]
grid $cvs

drawItem $cvs $routerDict internet 250 30
```

Script Output



12.5.3 Focus

When a window manager is directing events to a particular item (window, widget, or canvas item), that window is said to have the focus. Some events are always passed to an item that requests them, whether the item is defined as having focus or not. The cursor entering an item always generates an event, and a destroy event will always be transmitted to any applicable item. The keyboard events, however, are delivered only to the widget that currently has the focus.

The window manager controls the top-level focus. The default behavior for Macintosh and MS Windows systems is to click a window to place the focus in a window. On UNIX/X Window systems, the window managers can be set to require a click or to place the focus automatically on the window where the cursor resides.

Within the Tcl application, the focus is in the top-level window, unless a widget explicitly demands the focus. Some widgets have default actions that acquire the focus. An `entry` widget, for instance, takes the focus when you click it or tab to that widget. If you want to allow `KeyPress` events to be received in a widget that does not normally gain focus, you must add bindings to demand the focus.

The next example shows a label widget `.l` with three events bound to it: `Enter`, `Leave`, and `KeyPress`. When the cursor enters the `.l` widget, it acquires the focus, and all `KeyPress` events will be delivered to the script defined in the `bind .l <KeyPress>` command. When the cursor leaves label `.l`, no events are passed to this item.

The label `.l2` shares the same `-textvariable` as label `.l`. When `lvar` is modified in response to an event on `.l`, it is reflected in `.l2`. However, `.l2` never grabs the focus, as `.l` does. The `bind .l2 <KeyPress> {append lvar "L2";}` command has no effect, since `.l2` never has focus.

Example 9

Script Example

```
label .l1 -width 40 -textvariable lvar -background gray80
label .l2 -width 40 -textvariable lvar
```

```
grid .l1
grid .l2

bind .l1 <KeyPress> {append lvar %A;}

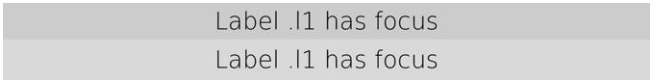
bind .l1 <Enter> {
    puts "Enter Event - grabbing focus"
    focus .l1
    }

bind .l1 <Leave> {
    puts "Leave Event - Returning focus"
    focus .
    }

bind .l2 <KeyPress> {append lvar "L2";}

puts "These events are valid on label .l1: [bind .l1]"
```

Script Output




```
These events are valid on label .l1: <Leave> <Enter> <Key>
# I moved the cursor into label .l1
Enter Event — grabbing focus
# I typed "Label .l1 has focus"
# I moved the cursor into label .l2
Leave Event — Returning focus
# I typed "This has no effect"
```

12.6 CREATING A WIDGET

Now that we know how to create drawable items in a canvas and bind actions to events in that canvas, we can create our own widgets. For example, suppose we want an Opera/Firefox-style button that does one thing when you click it and another if you hold down the mouse button for more than one second.

To create this we need to write a procedure that will create a canvas, draw something appropriate in that canvas, and bind scripts to the appropriate events. The following is a specification for a delayed-action button.

- Will be invoked as `delayButton name ?-option value?`
- Required options will be:
  `-bitmap bitmapName`
  The bitmap to display in this button.

`-command1 script`
A script to evaluate when button is clicked.
`-command2 script`
A script to evaluate when button is held down.

- Optional arguments will include:
  `-height number`
  The height of the button.
  `-width number`
  The width of the button.
  `-foreground color`
  The foreground color for text and bitmap.
  `-background color`
  The background color for the button and bitmap.
  `-highlight color`
  The background color when the cursor enters the button.
  `-text Text to display below bitmap`
  The optional text to display below the bitmap.
  `-font fontDescriptor`
  The font for displayed text.
- Clicking the widget with a left mouse button will invoke the `command1` script.
- Holding down the left mouse button will invoke the `command2` script.
- The button will default to 48 × 48 pixels.
- The button will default to a medium gray color.
- The highlight color will default to light gray.
- The button background will become the highlight color when a cursor enters the button.
- The button background will return to normal color when the cursor leaves the button.

This section shows a set of code that meets these specifications. The example shows the `delayButton` being used to display three simple buttons. There are a few things to notice in the next example. In the initialization, the configuration options are assigned default values before examining the argument list for modification. This lets the user override the defaults. Using the `info locals` to validate the options makes it easy to modify the code in the future. If a new configuration option becomes necessary (perhaps a `-textColor` instead of using the same color for the bitmap and text), you need only add a new default setting and change the code that uses the new value. The parsing code will not need to be modified.

In contrast, parsing for required options needs to have a list of required options to check against. When the required options are changed, this list must also be modified. After the button is created event bindings can be assigned to it. These bindings cause the procedures in the `delayButton` namespace to be evaluated when the left mouse button is pressed or released. Note the way the `after` command is used to schedule the commands in `scheduleCommand2`, and to evaluate a command in `invokeCommand1`.

When button 1 is pressed, the `scheduleCommand2` procedure is evaluated. This procedure schedules the *`command2`* script to be evaluated in one second, and saves the identifier returned by the `after`

command. It also appends a command that will delete the saved identifier when the command2 script is invoked.

When the button is released, the invokeCommand1 procedure will check to see if there is still an event identifier for this window in the afters array. If there is, the <ButtonRelease> event came before the after event was evaluated. In this case, the command1 script will be evaluated, and the command2 event will be canceled. If there is no event identifier for this window in the afters array, it means the command2 script was already evaluated, and the procedure will return without evaluating the command1 script.

Also pay attention to the namespace commands. Because the after scripts are evaluated in global scope, full namespace identifiers are used for script names, and uplevel #0 is used to force scripts to evaluate in the global scope.

Putting the registerCommands, scheduleCommand2, cancelCommand2, and invokeCommand1 commands into a namespace protects these commands from possible collision with other commands and does not pollute the global space. Creating these procedures inside the namespace also allows them to create and share the associative arrays afters, command1Array and command2Array without exposing these arrays to procedures executing outside this namespace.

The sample script for the delayButton does not include all the necessary procedures to use the menus or generate other data. It is only a skeleton to demonstrate the use of the delayButton.

Example 10

Script Example

```
 ##############################################################################
# proc delayButton {delayButtonName args}--
#  Create a delayButton icon that will evaluate a command when
#  clicked and display an information label if the cursor
#  rests on it
#
# Arguments
#  delayButtonName  The name of this delayButton
#  args     A list of -flag value pairs
#
# Results
#  Creates a new delayButton and returns the name to the
#  calling script
#  Throws an error if unrecognized arguments or required args
#  are missing.

proc delayButton {delayButtonName args} {
  ## Initialization

  #  define defaults
  set height 48
  set width 48
  set background gray80
```

```
 set highlight gray90
 set foreground black
 set text " "
 set command1 " "
 set command2 " "
 set bitmap " "
 set font {Helvetica 10}

## Parse for Unsupported Options
 #  Step through the arguments.
 #  Confirm that each -option is valid (has a default),
 #  and set a variable for each option that is defined.
 #   The variable for -foo is "foo".
 foreach {option val} $args {
   set optName [string range $option 1 end]
   if {![info exists $optName]} {
         error "Bad argument $option - \
         must be one of [info locals]" \
         "Invalid option"
   }
   set $optName $val
 }

##  Parse for Required Options
 #  These arguments are required.
 set reqOptions [list -command1 -command2 -bitmap]

 #  Check that the required arguments are present
 #  Throw an error if any are missing.
 foreach required $reqOptions {
   if {[lsearch -exact $args $required ] < 0} {
       error "delayButton requires a $required option" \
       "Missing required option"
   }
 }

##  Button Creation
 #  Create the canvas for this delayButton
 canvas $delayButtonName -height $height -width $width \
     -background $background

 # Place the bitmap in it.
 $delayButtonName create bitmap [expr $width/2] \
     [expr $height/2] -bitmap $bitmap -foreground $foreground
  #  If there is text, it goes on the bottom.

 if {![string match $text ""]} {
```

```
    $delayButtonName create text [expr $width/2] \
        [expr $height-1] \
        -text $text -fill $foreground \
        -font $font -anchor s
  }

  ##  Binding Definition
  #  Bind the button click to evaluate the $command
  #   script in global scope

  bind $delayButtonName <ButtonPress-1> \
      "::delayButton::scheduleCommand2 $delayButtonName"

  bind $delayButtonName <ButtonRelease-1> \
      "::delayButton::invokeCommand1 $delayButtonName"

  # Bind the background to change color when the
  #  cursor enters
  bind $delayButtonName <Enter> \
      "$delayButtonName configure -background $highlight"

  # Bind the background to change color when the cursor
  # leaves, and cancel any pending display of an info label.
  bind $delayButtonName <Leave> \
  "$delayButtonName configure -background $background;\
  ::delayButton::cancelCommand2 $delayButtonName;"

  ## Command Registration
  # Register the commands
  ::delayButton::registerCommands $delayButtonName \
      $command1 $command2
  # And return the name
  return $delayButtonName
}

# The registerCommands, scheduleCommand2, cancelCommand2, and
# invokeCommand1 commands are defined within this namespace to
# 1) avoid polluting the global space.
# 2) make the  "afters" array of identifiers returned by the
#   after command private to these procedures.
# 3) make the "afters" array persistent
# 4) protect the  "command1Array" and "command2Array" variables
#   from the rest of the program
# 5) make the  "command1Array" and "command2Array" variables
#   persistent

namespace eval delayButton {
```

```
variable afters
variable command1Array
variable command2Array
##########################################################################
# proc registerCommands {delayButtonName cmd1 cmd2}--
#  registers commands to be evaluated for this button
# Arguments
#  delayButtonName  The name of the delayButton
#  cmd1     Command to evaluate if clicked
#  cmd2     Command to evaluate if held
# Results
#  New entries are created in the command1Array
#  and command2Array variables.
proc registerCommands {delayButtonName cmd1 cmd2} {
  variable command1Array
  variable command2Array
  set command1Array($delayButtonName) $cmd1
  set command2Array($delayButtonName) $cmd2
}

##########################################################################
#  proc scheduleCommand2 {name }--
#  Cancel any existing "after" scripts for this button.
#
# Arguments
#  name  The name of this delayButton
#
# Results
#  Places an item in the "after" queue to be evaluated
#  1 second from now
proc scheduleCommand2 {delayButtonName } {
  variable afters
  variable command2Array
  cancelCommand2 $delayButtonName
  set cmd [format "%s; %s" $command2Array($delayButtonName) \
      "[namespace current]::cancelCommand2 $delayButtonName"]
  set afters($delayButtonName) [after 1000 $cmd]
}

##########################################################################
# proc cancelCommand2 {delayButtonName}--
#  Cancels the scheduled display of an Info label
# Arguments
#  delayButtonName The name of the delayButton
#
# Results
#  If this label was scheduled to be displayed,
```

```
  #  it is disabled.
  proc cancelCommand2 {delayButtonName} {
    variable afters
    if {[info exists afters($delayButtonName)]} {
       after cancel $afters($delayButtonName);
    }
    set afters($delayButtonName) ""
  }

  ##########################################################################
  #  proc invokeCommand1 {delayButtonName}--
  #  Deletes the label associated with this button, and
  #  resets the binding to the original actions.
  # Arguments
  #  delayButtonName  The name of the parent button
  #
  # Results
  #  No more label, and bindings as they were before the
  #  mouse paused
  #  on this button
  proc invokeCommand1 {delayButtonName} {
    variable afters
    variable command1Array

    # If there is nothing in the 'after' identifier
    # holder we've already done the 'command 2',
    # and this release is not part of a click.
    if {[llength $afters($delayButtonName)] == 0} {
      return
    }

    # Cancel the command 2 execution
    cancelCommand2 $delayButtonName
    # And do command 1
    uplevel #0 $command1Array($delayButtonName)
  }
}

# The following script shows how the  delayButton} can be used
##########################################################################
# proc makeMenu {args}--
#  generate and post a demonstration menu
# Arguments
#  args  A list of entries for the menu
#
# Results
#  Creates a menu of command items with no commands associated
```

```
#  with them.
proc makeMenu {args} {
  # Destroy any previous .m window
  catch {destroy .m}

  # Create menu, add items
  menu .m
  foreach item $args {
  .m add command -label $item
  }
  .m post [winfo pointerx .] [winfo pointery .]
}

# Create a button with text
delayButton .b1 -bitmap info -text "INFO" \
    -command1 {tk_messageBox -type ok -message "Information"} \
    -command2 {makeMenu "Tcl Info" "Tk Info" "TclOO Info" "Expect Info"}


# Create a button without text
delayButton .b2 -bitmap hourglass \
    -command1 {tk_messageBox -type ok \
        -message [clock format [clock seconds]]} \
    -command2 {makeMenu "Time in London" \
 "Time in New York" "Time in Chicago"}

 # Create a button with non-default colors.
delayButton .b3 -bitmap warning \
    -command1 {reportLatestAlert} \
    -command2 {reportAllAlerts} \
    -background #000000 -foreground #FFFFFF \
    -highlight #888888

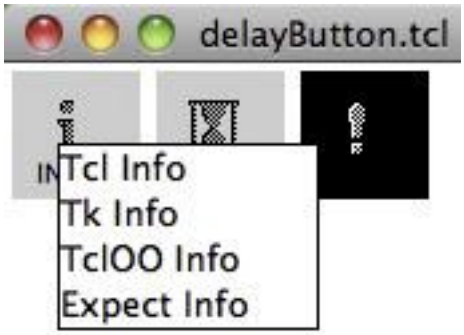
grid .b1 .b2 .b3
```

Script Output

Initial button state

After clicking Info

After holding left-button on Info



12.7 A HELP BALLOON: INTERACTING WITH THE WINDOW MANAGER

It also would be useful to be able to attach pop-up help balloons to widgets. Tk does not include a pop-up help widget as one of the basic widgets, but one can be created with just a few lines of code. The help balloon in this example is derived from an example on the Tcler's Wiki site (*http://wiki.tcl.tk*). The behavior of the help balloon is as follows.

- Schedule a help message to appear after a cursor has entered a widget.
- Destroy a help message when the cursor leaves a widget.

Scheduling the window to appear, and destroying it when it is no longer required, can be done with the bind command, described previously. These bindings are established in the balloon procedure. When the cursor enters a widget, an after event is scheduled to display the help message, and the balloon is destroyed when the cursor leaves the requested window.

The %W argument to the bind command is replaced by the name of the window that generated the event. This lets the script connect the help balloon to the window that created it. Displaying a balloon requires the following steps.

1. Confirm that the cursor is still inside the window associated with this help balloon.
2. Destroy any previous balloon associated with this window.
3. Create a new window with the appropriate text.
4. Map this window to the screen in the appropriate place.

This is done in the balloon::show procedure. Confirming that the cursor is still inside the window associated with this help balloon and placing the help balloon in the correct location of the screen requires interacting with both the window manager and the Tk interpreter. The window manager controls how top-level windows are placed, whereas the Tk interpreter controls how widgets are placed within the application.

The Tk wm and winfo commands provide support for interacting with window features such as window size, location, decorations, and type. The wm command interacts with the window manager, whereas the winfo command interacts with windows owned by the wish interpreter.

On a Macintosh or Microsoft Windows platform, the window manager is merged into the operating system. On an X Window platform, the window manager is a separate program ranging from the simple wm and twm managers to the configurable modern window managers, such as Gnome/Enlightenment and CDE/KDE. There is a core of features supported by all window managers, and there are some subcommands supported only on certain platforms. Again, check the on-line documentation for your platform.

The wm command is used to interact with top-level windows, the root screen and system-oriented features such as the focus model. The syntax of the wm command is similar to other Tk commands, with the command name (wm) followed by a subcommand and a set of arguments specific to this subcommand.

Syntax: wm *subcommand args*

Here are a few of the commonly used wm subcommands.

The wm title command lets your script define the text in the top decoration.

Syntax: wm title *windowName text*

Assigns *text* as the title displayed in the window manager decoration.

windowName	The window to set the title in.
text	The new title.

Example 11

```
wm title . "Custom Title"
grid [label .l -text "Title Demonstration"]
```

The `wm geometry` command lets you control where windows are placed, and how large they are. Note that all `wm` commands are "suggestions" to the window manager. If your program requests unsupported behavior (like a 1280 × 1024 window on a 640 × 480 screen), the window manager may reject the request.

Syntax: `wm geometry` *`windowName ?geometryString?`*

Queries or modifies the window's location or size.

`windowName`	The window to be queried or set.
`?geometryString?`	If this is defined, it instructs the window manager to change the size or location of the window. If not defined, the geometry of the window is returned. The format of the returned geometry string will be: *`width × height + xLocation + yLocation`* A geometry string defined by a user can include just the dimension information (i.e., `500 × 200`), or the location information (i.e., `+50+200`). The usual technique for defining location is with plus (+) markers, which denote pixels to the right and down from the top left corner. You can also describe the location with minus (−) markers, in which case the location is measured in pixels left or up from the bottom right corner.

Example 12

```
# This example moves the primary window slightly

# Retrieve the geometry
set g [wm geometry .]

# Parse it into width, height, x and y components
lassign [split $g {+-x}] width height xloc yloc

# Move the window down and right
incr xloc 5
incr yloc 5
set newGeometry [format "+%s+%s" $xloc $yloc]
wm geometry . $newGeometry
```

The overrideredirect subcommand will turn off the decorative borders that let you resize and move windows. By default, top-level widgets are created with full decorations, as provided by the window manager.

Syntax: wm overrideredirect *windowName boolean*

Sets the override-redirect flag in the requested window. If true, the window is not given a decorative frame and cannot be moved by the user. By default, the override-redirect flag is false.

windowName The name of the window for which the override-redirect flag is to be set.

boolean A Boolean value to assign to the override-redirect flag.

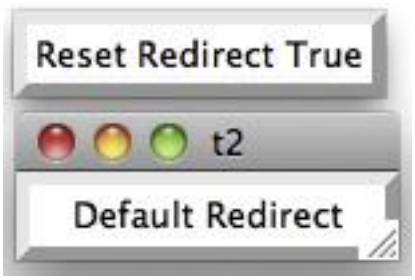
The wm overrideredirect command should be given before the window manager transfers focus of a window. Unlike most Tcl/Tk commands, you may not be able to test this subcommand by typing commands in an interactive session. The difference between override-redirect true and false looks as follows.

Example 13

```
# Create a toplevel window with no decorations
toplevel .t1 -border 5 -relief raised
label .t1.l -text "Reset Redirect True"
grid .t1.l
wm overrideredirect .t1 1

# Create a toplevel window with normal decorations
toplevel .t2 -border 5 -relief raised
label .t2.l -text "Default Redirect"
grid .t2.l

# Place the windows and raise them to the top of
# the display list.
wm geometry .t1 +300+300
wm geometry .t2 +300+400
raise .t1
raise .
```



The `protocol` command allows a script to examine or modify how certain windowing system events are handled.

Syntax: `wm protocol` *`window ?protocolName? ?script?`*

Return information about protocol handlers attached to a window, or assign a protocol handler to a window.

`window`	The name of a top-level Tk window (the main window or one created with the `toplevel` command).
`?protocolName`	The name of an atom corresponding to a window manager protocol. The only protocol in common use is `WM_DELETE_WINDOW`, the notification that a window has received a delete signal. If this field is absent, report all protocols that have handlers associated with them.
`?script?`	A script to evaluate when the *`protocolName`* message is received. If this field is absent, return the script associated with the *`protocolName`* atom.

The common use for the `wm protocol` command is to catch the `WM_DELETE_WINDOW` request, and give the user an option to save work before exiting (or abort the exit completely).

This is similar to using the `bind` command to catch the notification that a window has been destroyed: `bind . <Destroy> {cleanExit }`. The difference is that the `wm protocol WM_DELETE_WINDOW` command catches the request to delete a window, and can abort destroying the window, whereas the `bind . <Destroy>` command catches the notification that the window has already been destroyed.

Example 14

Script Example

```
# Confirm that a user wishes to exit the application.
proc confirmClose {} {
  set answer [tk\_messageBox -type "yesno" \
      -message "Do you wish to exit" ]

  if {[string match $answer "yes"]} {
    # Might add call to a 'save' procedure here
    exit
  } else {
  return
  }
}

wm protocol . WM_DELETE_WINDOW confirmClose

puts "ALL Protocols with handlers: [wm protocol .]"
puts "Handler for WM_DELETE_WINDOW: \
    [wm protocol . WM_DELETE_WINDOW]"
```

Script Output

```
ALL Protocols with handlers: WM_DELETE_WINDOW
Handler for WM_DELETE_WINDOW:  confirmClose
```

The wm command gives the Tcl programmer access to information controlled by the window manager. The winfo command gives the Tcl programmer access to information controlled by the Tk application. This includes information such as what windows are children of another window, what type of widget a window is, the size and location of a widget, the location of the cursor, and so on.

The pointerxy subcommand will return the coordinates of the cursor.

Syntax: winfo pointerxy *window*

Return the *x* and *y* location of the mouse cursor. These values are returned in screen coordinates, not application window coordinates.

window The mouse cursor must be on the same screen as this window. If the cursor is not on this screen, each coordinate will be −1.

Example 15

```
# Report the cursor location
label .l -textvar location
grid .l
while {1} {
  set location [winfo pointerxy .]
  update
  after 10
}
```

The containing subcommand will return the name of a window that encloses a pair of coordinates.

Syntax: winfo containing *rootX rootY*

Returns the name of the window that encloses the X and Y coordinates.

rootX An X screen coordinate. (0 is the left edge of the screen.)

rootY A Y screen coordinate. (0 is the top edge of the screen.)

Example 16

```
# This label displays 'outside' when the cursor is outside
#  the label, and 'inside' when the cursor enters the window.
label .l -textvar where
grid .l

while {1} {
  # Get the location of the cursor
  set location [winfo pointerxy .]
```

```
    set in [eval winfo containing $location]

    # Is the cursor inside .l?
    if {[string match $in .l]} {
      set where inside
    } else {
      set where outside
    }
    update
    after 10
  }
```

The `winfo height`, `winfo width`, `winfo rootx`, and `winfo rooty` return the same information as the `wm geometry` command. However, whereas the `wm geometry` command can only be used for top-level windows, the `winfo` commands can also be used for individual widgets.

Syntax: `winfo height` *winName*

Syntax: `winfo width` *winName*

Return height or width of a window.

winName The name of the window.

Example 17

```
puts "Toplevel window is [winfo width .] x [winfo height .]"
```

Syntax: `winfo rootx` *winName*

Syntax: `winfo rooty` *winName*

Return *x* or *y* location of a window in screen coordinates.

winName The name of the window.

Example 18

```
label .l
pack .l
puts "Label is at X: [winfo rootx .l] Y: [winfo rooty .l]"
```

The `wm` and `winfo` commands are used in the `balloon::show` procedure to confirm that a help balloon is still required, and map it to the correct location if it is. The first step, confirming that the cursor is still within the window, can be done with two `winfo` commands. The `pointerxy` subcommand

will return the coordinates of the cursor, and the containing subcommand will return the name of a window that encloses a pair of coordinates.

A help balloon window should not have the decorations added by the window manager, as we do not want the user to be able to move this window, iconify it, and so on. The decorations are added by the window manager, not managed by Tk, so removing the decorations is done with the wm command. The subcommand that handles this is overrideredirect.

This example uses the message widget. A label only displays text exactly as it is formatted, whereas a message widget displays multi-line text and will add newlines to maintain a height/width ratio. The message is lighter weight than the text widget, discussed in the next chapter.

Syntax: message *name ?options?*

Create a message widget.

name	A name for the message widget. Must be a proper window name.
?options?	Options for the message include:
-text	The text to display in this widget.
-textvar	The variable which will contain the text to display in this widget.
-aspect	An integer to define the aspect ratio: (Xsize/Ysize) * 100
-background	The background color for this widget.

The final step is to place the new window just under the widget that requested the help balloon. The window that requests the help will be a window managed by Tk, so we can use the winfo command to determine its height and X/Y locations. Placing a top-level window on the screen is a task for the window manager, so the wm geometry command gets used.

Example 19

Script Example

```
 ##########################################################################}
# proc balloon {w help}--
#  Register help text with a widget
# Arguments
# w      The name of a widget to have a help balloon associated it
# help   The text to associate with this window.
# Results
# Creates bindings for cursor Enter and Leave to display and
#  destroy a help balloon.

proc balloon {w help} {
    bind $w <Any-Enter> "after 1000 [list balloon::show %W [list $help]]"
    bind $w <Any-Leave> "destroy %W.balloon"
}

namespace eval balloon {
```

```
#############################################################
# proc show {w text }--
#   Display a help balloon if the cursor is within window w
#
# Arguments
# w     The name of the window for the help
# text  The text to display in the window
#
# Results
# Destroys any existing window, and creates a new
#   toplevel window containing a message widget with
#   the help text
proc show {w text} {
  # Get the name of the window containing the cursor.
  set currentWin [eval winfo containing  [winfo pointerxy .]]

  # If the current window is not the one that requested the
  #   help, return.
  if {![string match $currentWin $w]} {
    return
  }

  # The  new toplevel window will be a child of the
  #  window that requested help.
  set top $w.balloon

  # Destroy any previous help balloon
  catch {destroy $top}

  # Create a new toplevel window, and turn off decorations
  toplevel $top -borderwidth 1
  wm overrideredirect $top 1

  # If Macintosh, do a little magic.
  if {$::tcl_platform(platform) == "macintosh"} {
    # Daniel A. Steffen added an 'unsupported1' command
    # to make this work on macs as well, otherwise raising the
    # balloon window would immediately post a Leave event
    # leading to the destruction of the balloon... The
    # 'unsupported1' command makes the balloon window into a
    # floating window which does not put the underlying
    # window into the background and thus avoids the problem.
    # (For this to work, appearance manager needs to be present.)

    # In Tk 8.4, this command is renamed to:
    # ::tk::unsupported::MacWindowStyle
```

```
      unsupported1 style $top floating sideTitlebar
    }

    # Create and pack the message object with the help text
    set hint [message $top.txt -aspect 200 \
        -background lightyellow -font fixed -text $text]
    pack $hint

    # Get the location of the window requesting help,
    #  use that to calculate the location for the new window.
    set wmx [winfo rootx $w]
    set wmy [expr [winfo rooty $w]+[winfo height $w]]

    wm geometry $top \
      [winfo reqwidth $hint]x[winfo reqheight $hint]+$wmx+$wmy

    # Raise the window, to be certain it's not hidden below
    # other windows!.
    raise $top
  }
}

# Create a test button with a popup hint
button  .b -text Exit -command exit
balloon .b "Push me if you're done with this"
grid .b
```

Script Output

12.8 THE image OBJECT

Creating items such as lines and arcs on a canvas is one way of generating a picture, but for some applications a rasterized image is more useful. There are two ways of creating images in Tcl. The `create bitmap` canvas command will display a single color image on a canvas (as shown in Section 12.3.7). The `create image` canvas command will display an image object on the canvas.

The canvas bitmap item is easy to use but is not as versatile as the image object. You can use the image object to create simple bitmap images or full-color images. The full-color image objects can be shrunk, zoomed, subsampled, and moved from image object to image object. The image objects can be displayed in other widgets, including `button`, `label`, and `text` widgets.

The standard Tk distribution supports images defined in the X-Bitmap, GIF, Portable Pixmap, and Portable GrayMap formats. Some extensions exist that provide support for other formats, such as JPEG and PNG. (See Chapter 14 for Jan Nijtmans's Img Extension which adds more image support to Tk.) This section will explain how to create and use image objects.

12.8.1 The `image` Command

To create an `image` item on a canvas, you must first create an `image` object. The `image` command supports creating, deleting, and getting information about the `image` objects within the Tk interpreter. The `image create` command creates an `image` object from a file or internal data and returns a handle for accessing that object.

Syntax: `image create` *`type ?name? ?options?`*

Create an `image` object of the desired type, and return a handle for referencing this object.

`type` The type of `image` that will be created. Tk versions 8.0 and more recent support:

- `photo` A multicolor graphic.
- `bitmap` A two-color graphic.

`?name?` The name for this `image`. If this is left out, the system will assign a name of the form `imageX`, where `X` is a unique numeric value.

`?options?` These are options specific to the type of `image` being created. The appropriate options are discussed in regard to `bitmap` and `photo`.

An image can be deleted with the `image delete` command.

Syntax: `image delete` *`name ?name?`*

Delete the named image.

`name` The handle returned by the `image create` command.

The `image` command can be used to get information about a single `image`, or all `images` that exist in the interpreter.

Syntax: `image height` *`name`*

Return the height of the named image.

Syntax: `image width` *`name`*

Return the width of the named image.

Syntax: `image type` *`name`*

Return the type of the named `image`.

For each of these commands, the *`name`* parameter is the image name returned by the `image create` command. You can retrieve a list of existing image names with the `image names` command.

Syntax: `image names`

Return a list of the image object names.

As with the Tk widgets, when an image is created with the `image create` command, a command with the same name as the `image` is also created. This image object command can be used to manipulate the object via the *`imageName`* `cget` and *`imageName`* `configure` commands. These commands behave in the same way as other widget `cget` and `configure` subcommands. The configuration options that can be modified with the `configure` subcommand depend on the type of `image` created and are the same as the options available to that `image create type` command.

12.8.2 Bitmap Images

Bitmap images (which are different from a canvas `bitmap` item) are created with the command `image create bitmap`, as described previously. The options supported for this command include

`-background color`	The background color for this bitmap. If this is empty (the default), background pixels will be transparent.
`-foreground color`	The foreground color for this `image`. The default is black.
`-data data`	The data that defines this image. This must be in the same format as an X Windows system bitmap file.
`-file filename`	A file containing the X-Bitmap data.

The X Window system bitmap file is quite simple. It was designed to be included in C programs and looks like a piece of C code. The format is as follows.

```
# define name_width width
# define name_height height
static unsigned char name_bits[] = {
  data1, data2, ...
}
```

The good news is that this is simple and easy to construct. The `bitmap` program on a UNIX system will let you construct an X-Bitmap file, or you can use the `imagemagick` or `PBM` utilities to convert an MS Windows or Mac bitmap file to an X-Bitmap file.

Example 20

Script Example

```
 # Create and pack a canvas
pack [canvas .c -background white -height 60 -width 100]

# This is the hourglass bitmap from the predefined bitmaps in
# the Tk 8.0 distribution: tk8.0/bitmaps/hourglass.bmp
set hourglassdata {
# define hourglass_width 19
# define hourglass_height 21
static unsigned char hourglass_bits[] = {
   0xff,0xff,0x07,0x55,0x55,0x05,0xa2,0x2a,0x03,0x66,0x15,0x01,
   0xa2,0x2a,0x03,0x66,0x15,0x01,0xc2,0x0a,0x03,0x46,0x05,0x01,
```

```
    0x82,0x0a,0x03,0x06,0x05,0x01,0x02,0x03,0x03,0x86,0x05,0x01,
    0xc2,0x0a,0x03,0x66,0x15,0x01,0xa2,0x2a,0x03,0x66,0x15,0x01,
    0xa2,0x2a,0x03,0x66,0x15,0x01,0xa2,0x2a,0x03,0xff,0xff,0x07,
    0xab,0xaa,0x02};
}

# Create a bitmap image from the hourglass data
set bm [image create bitmap -data $hourglassdata]

# And create a displayable canvas image item from that image.
.c create image 50 30 -image $bm

# And label the image.n
.c create text 50 50 -text "bitmap image"
```

Script Output

12.8.3 Photo Images

The image data for a `photo image` may be PPM, PGM, or GIF. The data may be in binary form in a disk file or encoded as base64 data within the Tcl script. base64 encoding is a technique for converting binary data to printable ASCII with the least expansion. This format is used by all e-mail systems that support the MIME (Multipurpose Internet Mail Extensions) protocol (which most modern mail programs do). Strange as it sounds, the most portable way of converting a file from binary to base64 encoding is to mail it to yourself, save the mail, and extract the data you want from the saved mail file.

If you are running on a UNIX system, you may have the `mmencode` or `mimencode` program installed. These are part of the `metamail` MIME mail support.

The `tcllib` (*http://Tcllib.sf.net/*) collection of useful tools is included in the ActiveTcl distribution and includes the `base64` package. This package includes `encode` and `decode` functions within the `base64` namespace that will convert data to and from base64 format within a script.

```
package require base64
set b64Stream [base64::encode $binaryData]
```

The `image create photo` command supports several options. These include

`-data string`	The base64 encoded data that defines this image.
`-file filename`	A file that contains the image data in binary format.
`-format format`	The format for this image data. Tk will attempt to determine the type of image by default. The possible values for this option in version 8.4 are `gif`, `pgm`, and `ppm`. Tcl 8.6 adds support for PNG images.

As with the image create bitmap command, this command creates a new command with the same name as the new image, which can be used to manipulate the image via the *imageName* cget and *imageName* configure commands.

The photo type images support other subcommands, including the following.

Syntax: *imageName* copy *sourceImage ?options?*

Copy the image data from sourceImage to the image *imageName*. Options include:

-from x1 y1 x2 y2	The area to copy from in the source image. The coordinates define opposing corners of a rectangle.
-to x1 y1 x2 y2	The area to copy to in the destination image.
-shrink	Shrink the image if necessary to fit the available space.
-zoom	Expand the image if necessary to fit the available space.

Syntax: *imageName* get *x y*

Return the value of the pixel at coordinates x, y. The color is returned as three integers: the red, green, and blue intensities.

Syntax: *imageName* put *data* ?-to *x1 y1 ?x2 y2??*

Put new pixel data into an image object.

data	The data is a string of color values, either as color names or in #rgb format. By default, these pixels will be placed from the upper left corner, across each line, and then across the next line.
-to x1 y1 x2 y2	Declares the location for the data to be placed. If only x1, y1 are defined, this is a starting point, and the edges of the image will be used as end points. If both x1, y1 and x2, y2 are defined, they define the opposing corners of a rectangle.

The following example creates an image from base64 data, copies the data to another image, extracts some data (from the base of the T), and draws a box around the extracted data. In the extracted data, each pixel is grouped within curly braces and displayed as a red, blue, and green intensity value. The {0 0 0} pixels are black, and the {60 248 52} pixels are green.

Example 21

Script Example

```
 # Create a canvas
canvas .c -height 100 -width 300 -background white
pack .c

# Define an image - in base 64 format
set img {
```

```
ROlGODdhQAAwAKEAAP///wAAADz4NAD//ywAAAAAQAAwAAAC/oSPqcvtD6OctNqLs
16h+w+G4kiWAWOm6hqi7AuLbijU9k3adHxyYtOBBoQjokfAm4GQQeawZBw6X8riFH
q9sqo/rZQY9aq4NK3wPI1ufdilE/ORj9m5rNcoT5Hd/PIxb7IXZ9d1xOMh+Ne3CBh
IZ+UX1DR52KNgguOGBtToeFkJBWqZIJojmlgKipp6uDoyABsrGxCr+sgC25FLO6Db
W+kq8svr8bvbehtzXDyclOw7+9rsCw1DJos9TQ1yrK2XnB3Nrb3r/f35EY4dskxtf
k46rt5OLP/uGZ8+T8/fSz+Hjtk+duSM3RvBZV8/gsys3VI4TRyvYbXWBBQYjhXAJX
z61GmEh4BEto8bQ5JkFewkyAMqU6VsSWKDzJkOa9q8iTNnhQIAOw==
}

# Create an image from the img data
set i [image create photo -data $img]

# Now, create a displayable canvas image item from the image
.c create image 25 40 -image $i -anchor nw

# Create an image with no data,
set i2 [image create photo -height 50 -width 70]

$i2 put white -to 0 0 70 50
# And copy data from the original image into it.
$i2 copy $i -from 1 1 50 40 -to 10 10 60 50

# Now display the image
.c create image 140 40 -image $i2 -anchor nw

# Display some of the values in the new image
puts "Values from 24 24 to 29 31"
for {set y 24} {$y < 31} {incr y} {
    for {set x 24} {$x < 29} {incr x} {
    puts -nonewline \
    "[format "% 12s" [list [$i2 get $x $y]]]"
   }
   puts ""
}

# Draw a n outline around the area we sampled:
for {set x 24} {$x < 29} {incr x} {
    $i2 put {#FFF #FFF} -to $x 23
    $i2 put {#FFF #FFF} -to $x 30
}

for {set y 24} {$y < 31} {incr y} {
    $i2 put {#FFF} -to 22 $y
    $i2 put {#FFF} -to 23 $y
    $i2 put {#FFF} -to 29 $y
    $i2 put {#FFF} -to 30 $y
}
```

```
# And label the images.
.c create text 25 5 -text "photo image 1" \
    -font {helvetica 12} -anchor nw
.c create text 140 5 -text "copied and \nmodified" \
    -font {helvetica 12} -anchor nw
```

Script Output

```
Values from 24 24 to 29 31
     {0 0 0} {60 248 52} {60 248 52}     {0 0 0}     {0 0 0}
     {0 0 0} {60 248 52} {60 248 52}     {0 0 0}     {0 0 0}
     {0 0 0} {60 248 52} {60 248 52}     {0 0 0}     {0 0 0}
     {0 0 0} {60 248 52} {60 248 52}     {0 0 0}     {0 0 0}
 {60 248 52} {60 248 52} {60 248 52} {60 248 52}     {0 0 0}
     {0 0 0}     {0 0 0}     {0 0 0}     {0 0 0}     {0 0 0}
     {0 0 0}     {0 0 0}     {0 0 0}     {0 0 0}     {0 0 0}
```

Detail of copy/modified image

12.8.4 Revisiting the `delaybutton` Widget

The bitmaps used in the `delayButton` example are all predefined in the Tk interpreter. If you have a bitmap of your own you would like to use, you would have to define it in a data file using the X Window system bitmap format. For example, if the bitmap image is in a file named *bitmap.xbm*, the `delayButton` could be created with the following command.

```
delayButton .f.bFile -bitmap @bitmap.xbm ...
```

Tk has no method for defining canvas `bitmap` item data in a script. Tk does allow you to define an `image` of type `bitmap` from data included in the script. The `image` command is the preferred way of dealing with images on canvases.

Unfortunately, you cannot access the predefined bitmaps with the `image` command, though you can load the data from the distribution (under the *TkDistribution/bitmaps* directory), as was just done in.

Following is a modified version of the `delayButton` procedure that uses an `image` instead of a `bitmap` and a script that creates its own bitmaps to display in the buttons. The code that executes within the `delayButton` namespace does not change.

The `info` and `warning` bitmap data are taken from the files *bitmaps/questhead.bmp* and *bitmaps/warning.bmp*. The data for the `clockIcon` is a base64 encoded GIF image. It may not be obvious

in the book, but the clock has a red body, a blue border, and yellow hands. The points to note in this version of the script are as follows.

- The -bitmap options are replaced with -image options.
- The bitmap objects are replaced by image objects.
- The -foreground option is not supported in the canvasName create image command but is supported as a subcommand of the image object.

Example 22

Script Example

```
# Load the delayButton namespace code.

source delayButton.tcl

#################################################################
# proc delayButton {delayButtonName args}--
#  Create a delayButton icon that will evaluate a command when
#  clicked and display an information label if the cursor
#  rests on it
#
# Arguments
#  delayButtonName  The name of this delayButton
#  args     A list of -flag value pairs
#
# Results
#  Creates a new delayButton and returns the name to the
#  calling script
#  Throws an error if unrecognized arguments or required args
#  are missing.
proc delayButton {delayButtonName args} {
   # define defaults
   set height 48
   set width 48
   set background gray80
   set highlight gray90
   set foreground black
   set text ""
   set command1 ""
   set command2 ""
   set image ""
   set font {Helvetica 10}

   #   Step through the arguments.
   #   Confirm that each -option is valid (has a default),
   # and set a variable for each option that is defined.
   #   The variable for -foo is "foo".
```

```
foreach {option val} $args {
  set optName [string range $option 1 end]
     if {![info exists $optName]} {
       error "Bad argument $option - \
   must be one of [info locals]" \
   "Invalid option"
     }
     set $optName $val
 }

# These arguments are required.
set reqOptions [list -command1 -command2 -image]

# Check that the required arguments are present
# Throw an error if any are missing.
foreach required $reqOptions {
    if {[lsearch -exact $args $required ] < 0} {
      error "delayButton requires a $required option" \
      "Missing required option"
    }
}

# Create the canvas for this delayButton
canvas $delayButtonName -height $height -width $width \
    -background $background

# Place the image in it.
$delayButtonName create image [expr $width/2] \
    [expr $height/2] -image $image

# If there is text, it goes on the bottom.
if {![string match $text ""]} {
  $delayButtonName create text [expr $width/2] \
      [expr $height-1] \
      -text $text -fill $foreground \
      -font $font -anchor s
}

# Bind the button click to evaluate the $command
#  script in global scope
bind $delayButtonName <ButtonPress-1> \
    "::delayButton::scheduleCommand2 $delayButtonName"
bind $delayButtonName <ButtonRelease-1> \
    "::delayButton::invokeCommand1 $delayButtonName"

# Bind the background to change color when the
#  cursor enters
```

```
    bind $delayButtonName <Enter> \
        "$delayButtonName configure -background $highlight"

    # Bind the background to change color when the cursor
    #  leaves, and cancel any pending display of an info label.
    bind $delayButtonName <Leave> \
        "$delayButtonName configure -background $background;\
        ::delayButton::cancelCommand2 $delayButtonName;"

    # Register the commands
    ::delayButton::registerCommands $delayButtonName \
          $command1 $command2

    # And return the name
    return $delayButtonName
}

# The following script shows how the  delayButton} can be used

# makeMenu is the same as the previous procedure for creating
# an ad-hoc popup menu.

proc makeMenu {args} {
    catch {destroy .m}
    menu .m
    foreach item $args {
     .m add command -label $item
    }
    .m post [winfo pointerx .] [winfo pointery .]
}

# Create images for info, clock and warning
#  Use bitmap data for info and warning, and
#  a base64 GIF image for the clock

set info [image create bitmap -data {
# define info_width 8
# define info_height 21
static unsigned char info_bits[] = {
   0x3c, 0x2a, 0x16, 0x2a, 0x14, 0x00, 0x00, 0x3f, 0x15,
   0x2e, 0x14, 0x2c, 0x14, 0x2c, 0x14, 0x2c, 0x14, 0x2c,
   0xd7, 0xab, 0x55

}}]


image create photo clockIcon -data {
```

```
R0lGODlhIAAgALMAANnZ2QD/////AAAAADOZmZmZmf8AAMwAAJmZAMz
MzP//////////////////////yH5BAEAAAAALAAAAAAgACAAAAT/EA
AY5KTVXgDAvULAAYOcVBJQaxVi1BpARIYQQoFDTlrtCIEQEgIMZUAjJ
63WjEAICQGGAY2ctNoZCCEhBDigkZMeISidgRACgxzQyEmPEGNQagKB
hIQQBjRyOiPEGJSaQCAhIYQBjZzOCDEGpSYQSEgIYUAjJz1CjEGpCQQ
SEkIYOMhJjxBjUGoCgYSEEAYOctIjxBiUmkAgISGEAY2c9AgxBqUmEE
hICGFAIyc9QoxBqQkEEhKEGNDISYUsRQghqIFBiBGEGNDIOY+QpQghx
IDDGBOEGGGMAY2c8giBEBxySmNMmCGEAY2c5ggBh5xOGGMCISQEGAYO
ch4hxphzjU4TIJEhhAGNlEeIMeacUwZIZAhhGGPMEWLAAY2ctAZCSAh
hUCjEGNDISasJhJAQwoBGGjHGnHNOGQgkJIQwoJGTVjsDISSEAAcOct
JqZyCEhBBggUZOWuOMhJAQAgwFGjlptSYRQggMctJqLSRShjDnnFKIM
ecOERCARE5ahRi1SgAAJHJSOsaolQAQAQA7
}


set warning [image create bitmap -data {
# define warning_width 6
# define warning_height 19
static unsigned char warning_bits[] = {
   0x0c, 0x16, 0x2b, 0x15, 0x2b, 0x15, 0x2b, 0x16, 0x0a,
   0x16, 0x0a, 0x16, 0x0a, 0x00, 0x00, 0x1e, 0x0a, 0x16,
   0x0a}
}]

# Create the three buttons and grid them.
delayButton .b1 -image $info -text "INFO" \
    -command1 {tk_messageBox -type ok -message "Information"} \
    -command2 {makeMenu "Tcl Info" "Tk Info" "Expect Info"}

delayButton .b2 -image clockIcon \
    -command1 {tk_messageBox -type ok \
    -message [clock format [clock seconds]]} \
    -command2 {makeMenu "Time in London" \
    "Time in New York" "Time in Chicago"}

delayButton .b3 -image $warning \
    -command1 {reportLatestAlert} \
    -command2 {reportAllAlerts} \
    -background #AAA -foreground #FFF -highlight #888

grid .b1 .b2 .b3
```

Script Output

The behavior is the same with the `-bitmap` version and the `-image` version. The `-image` version of this widget needs more setup but is more versatile.

12.9 BOTTOM LINE

- The `canvas` widget creates a drawing surface.
- The `canvas` widget returns a unique identifier whenever a drawable item is created.
- Drawable items may be associated with one or more `tag` strings.
- The coordinates of a drawable item can be accessed or modified with the *`canvasName`* `coords` command.
  Syntax: *`canvasName`* `coords` *`tagOrId ?x1 y1?... ?xn yn?`*
- A drawable item can be moved on a `canvas` with the *`canvasName`* `move` command.
  Syntax: *`canvasName`* `move` *`tagOrId xoffset yoffset`*
- The space occupied by a drawable item is returned by the *`canvasName`* `bbox` command.
  Syntax: *`canvasName`* `bbox` *`tagOrId`*
- A list of items that match a search criterion is returned by the *`canvasName`* `find` command.
  Syntax: *`canvasName`* `find` *`searchSpec`*
- A drawable item's location in the display list can be modified with the *`canvasName`* `raise` command and *`canvasName`* `lower` command.
  Syntax: *`canvasName`* `raise` *`tagOrId? abovetagOrId?`*
  Syntax: *`canvasName`* `lower` *`tagOrId? belowtagOrId?`*
- A list of available fonts is returned with the `font families` command.
  Syntax: `font families ?-displayof` *`windowName`*
- The horizontal space necessary to display a string in a particular font is returned by the `font measure` command.
  Syntax: `font measure font ?-displayof` *`windowName? text`*
- The closest match to a requested font is returned by the `font actual` command.
  Syntax: `font actual font ?-displayof` *`windowName?`*
- Actions can be bound to events on a widget with the `bind` command.
  Syntax: `bind` *`widgetName eventType script`*
- Actions can be bound to events on a drawable item within a `canvas` widget, with the *`canvasName`* `bind` command.
  Syntax: *`canvasName`* `bind` *`tagOrId eventType script`*
- Two-color (`bitmap`) and multicolor (`photo`) images can be created with the `image create` command.
  Syntax: `image create type` *`?name? ?options?`*
- Images can be deleted with the `image delete` command.
  Syntax: `image delete` *`name ?name?`*
- Information about an image is returned by the `image height`, `image width`, and `image type` commands.
  Syntax: `image height` *`name`*
  Syntax: `image width` *`name`*
  Syntax: `image type` *`name`*

- A list of images currently defined in a script is returned by the `image names` command.
  Syntax: `image names`
- Photo image pixel data can be copied from one image to another with the *`imageName`* `copy` command.
  Syntax: *`imageName`* `copy` *`sourceImage? options?`*
- Photo image pixel data can be accessed or modified with the *`imageName`* `get` and *`imageName`* `put` commands.
  Syntax: *`imageName`* `get` *`x y`*
  Syntax: *`imageName`* `put` *`data`* `?-to` *`x1 y1 ?x2 y2??`*
- The `canvas create bitmap` command and the `image create bitmap` command are not the same but can be used to achieve equivalent results.

Canvas create bitmap	*Image create bitmap*
Can access internal bitmaps	Can use data defined within a script
Can reference a data file via `@filename`	Can reference a data file via `-file`
Can modify foreground via: `cvsName create bitmap \ -foreground $newcolor`	Can modify foreground via: `imageHandle configure \ -foreground $newcolor`

12.10 PROBLEMS

The following numbering convention is used in all Problem sections:

Number Range	Description of Problems
100–199	Short comprehension problems review material covered in the chapter. They can be answered in a few words or a 1–5-line script. These problems should each take under a minute to answer.
200–299	These quick exercises require some thought or information beyond that covered in the chapter. They may require reading a man page or making a web search. A short script of 1–50 lines should fulfill the exercises, which may take 10–20 minutes each to complete.
300–399	Long exercises may require reading other material or writing a few hundred lines of code. These exercises may take several hours to complete.

- *100* Can an object drawn on a Tk canvas have more than one tag?
- *101* What command would draw a blue line from 20,30 to 50,90 on a canvas named `.c`?
- *102* How many colors can be included in a canvas bitmap object?
- *103* What command would cause all canvas graphic objects on a canvas named `.c` with the tag `red` to turn red when a cursor passes over them?

- *104* What command will cause keyboard events to be sent to a label named `.l`?
- *105* Can a canvas be linked with a scrollbar?
- *106* What command will create an image object?
- *107* What command will display an image object?
- *108* Can you access the data in a single pixel of a Tk `image` object?
- *109* If an application is running on a 1024 × 768 pixel display, what command would resize the application to fill the display?
- *110* What command will return the height of a window?
- *200* Write a script that creates a canvas and draws a 20 × 30 pixel blue rectangle with a 5-pixel-wide green border, a red oval with a 2-pixel-wide yellow border, and a 3-pixel-wide red line that connects them.
- *201* Create a bouncing ball animation by creating a canvas and an oval and changing the oval location every 10 milliseconds. Allow the bouncing ball to only bounce to 90% of the height of the starting location, to simulate a bouncing ball settling down.
- *202* Modify the delaybutton example in Section 12.6 to require text instead of a bitmap, and size the button to match the text, instead of using a fixed-size button.
- *203* Modify the delaybutton example in Section 12.6 to require a `-helpText` option, and automatically attach the help balloon described in Section 12.7 when the delaybutton is created.
- *204* Lemon juice is a classic invisible ink. It dries invisibly until the paper is heated and the sugars in the lemon juice caramelize and turn brown. Write a Tcl script to simulate this by writing white text on a white background, and then slowly change the color using the `itemconfigure` command.
- *205* Modify the bouncing star example in Section 12.4.3 to include several non-moving circles. Highlight whichever circle is closest to the star as the star moves around.
- *206* Write a script that examines each pixel in an image object and counts the occurrences of pixels of different brightness values (referred to as "histogramming an image"). Display the results by printing the brightness value and count for any non-zero count.
- *300* Add bindings to the display in problem 201 so that you can "grab" the rectangle or oval by placing the cursor on it and holding down the left mouse button, and "drag" it by moving the cursor while the button is depressed.
- *301* Modify the script in problem 301 to update the line coordinates to always connect the two nearest edges of the rectangle and oval.
- *302* Create a simple "paint" package that will have radio buttons for select-drawing a rectangle or oval, and will allow a user to click on a location on the screen and drag a corner to define the size of the rectangle or oval.

- *303* The `winfo class` command will return the type of a window (`Button`, `Frame`, and so on). Use this command to write a recursive procedure that will place the names of all windows in an application into a tree data structure. Use a nested dict to hold the data as described in Chapter 6.
- *304* Write a Tk procedure that will display the elements in a tree data structure with each element on a single line, and indentation to denote how many levels deep a tree element is. You may need scrollbars to view an entire tree. The output should resemble the following.

```
.
  .buttonFrame
    .buttonFrame.file
    .buttonFrame.edit
    .buttonFrame.view
  .dataFrame
    .dataFrame.canvas
    .dataFrame.label
...
```

- *305* Modify the script created for problem 206 to display the results as a bar chart. Scale the bars so that the longest is 90% of the canvas size, and use a scrollbar to display all bars.

CHAPTER

The text Widget and htmllib

13

The message, label, and entry widgets are useful for applications with short prompts and small amounts of data to enter. That is, for applications that resemble filling out a form. However, some applications need to allow the user to display, enter, or edit large amounts of text. The text widget supports this type of application. The text widget supports the following.

- Emacs-style editing
- Arrow-key-style editing
- Scrollbars
- Multiple fonts in a single document
- Tagging a single character or multiple-character strings (in a similar manner to the way tags are applied to canvas objects)
- Marking positions in the text
- Inserting other Tk widgets or images into the text display
- Binding events to a single character or a multiple-character string

Stephen Uhler used the text widget to construct the html_library for rendering HTML documents. This is not part of the Tcl/Tk distribution but can be found on the companion website and is available via ftp at:

http://noucorp.com/tcl/utilities/htmllib/htmllib-rel_0_3_4.zip

This pure Tcl package will render html text and provides hooks for links and images. This chapter introduces the text widget and the html library.

13.1 OVERVIEW OF THE text WIDGET

An application may create multiple text windows, which can be displayed in either a top-level window, frame, canvas, or another text widget. The content of a text widget is addressed by line and character location. The text widget content is maintained as a set of lists. Each list consists of the text string to display, along with any tags, marks, image annotations, and window annotations associated with this text string.

The text widget displays text longer than the width of the widget in one of two ways. It can either truncate the line at the edge of the widget or wrap the text onto multiple lines. The widget can wrap the text at a word or character boundary.

When text wraps to multiple lines, the number of lines on the display will be different from the number of lines in the internal representation of the text. When referencing a location within the text

Tcl/Tk: A Developer's Guide. DOI: 10.1016/B978-0-12-384717-1.00013-0

widget, the line and character position refers to the internal representation of the text, not what is displayed.

When a scrollbar is connected to the text widget, the user can modify which lines are displayed on the screen. This does not affect the content of the text widget, and the line and character positions still reflect the internal representation of the text, not the displayed text. For example, if the user scrolls to the bottom of a document so that the top line is the fiftieth line of the document, the line number is still 50, not 1.

A text string can have a tag associated with it. Tags in a text widget are similar to tags in the canvas object. You can apply zero or more tags to an item of text, and apply a given tag to one or more items of text. Tags are used to mark sections of text for special treatment (different fonts, binding actions, and so on). Tags are discussed in more detail later in the chapter.

A text widget can also have marks associated with it. Marks are similar to tags, except that where tags are associated with a text string, a mark is associated with a location in the text. You can have only a single instance of any mark, but you can have multiple marks referring to a single location. Marks are used to denote locations in the document for some action (inserting characters, navigating with a mouse, and so on). Marks are discussed in more detail later in the chapter.

13.1.1 Text Location in the text Widget

The text widget does not allow you to place text at any location, the way a canvas widget does. A text widget "fills" from the top left, and you can address any position that has a character defined at that location. If a program requests a position outside the range of available lines and characters, the text widget will return the nearest location that has a character defined (usually the end). Tcl uses a list to define the index that describes a location in the text widget.

```
position ?modifier1 modifier2...modifierN?
```

The position field may be one of the following.

line.character	The line and character that define a location in the internal representation of the text. Lines are numbered starting from 1 (to conform to the numbering style of other UNIX applications), and character positions are numbered starting from 0 (to conform to the numbering style of C strings). The character field may be numeric or the word end. The end keyword indicates the position of the newline character that terminates a line.
@x.y	The x and y parameters are in pixels. This index maps to the character with a bounding box that includes this pixel.
markID	The character just after the mark named markID.
tagID.first	The first occurrence of tag tagID.
tagID.last	The last occurrence of tag tagID.
windowName	The name of a window that was placed in this text widget with the window create widget command.

imageName	The name of an image that was placed in this text widget with the image create widget command.
end	The last character in the text widget.

The modifier fields in an index may be one of:

+num chars

-num chars

+num lines

-num lines

The location moves forward or backward by the defined number of characters or lines. The plus symbol (+) moves the location forward (right or down), and the minus symbol (−) moves the location backward (left or up).

linestart

lineend

The index refers to the beginning or end of the line that includes the location index. NOTE: {1.0 lineend} is equivalent to {1.end} and {1.0 linestart} is equivalent to {1.0}.

wordstart

wordend

The index refers to the beginning or end of the word that includes the location index.

Index Examples

The following are examples of valid and invalid index descriptions.

0.0

Not a valid index because line numbers start at 1. This will be mapped to the beginning of the text widget, index 1.0.

{.b wordend}

If a letter follows window .b, this refers to the location just before the first space after the .b window.

If window .b is at the end of a word, this refers to the location just after the space that follows window .b.

1.0 lineend

This is not a valid index; it is not a list.

[list mytag.first lineend -10c wordend]

Sets the index to the end of the word that has a letter 10 character positions from the end of the line that includes the first reference to the tag mytag. Note that the index is evaluated left to right, as follows.

1. Find the first occurrence of the tag mytag.
2. Move to the end of the line.
3. Count back 10 characters.
4. Go to the end of the word.

13.1.2 Tag Overview

The tag used in a text widget is similar to the tag used with the canvas object. A tag is a string of characters that can be used to identify a single character or text string. A tag can reference several text strings, and a text string can be referenced by multiple tags.

A tag's start and end locations define the text string associated with it. Note that the tag is associated with a location, not the character at a location. If the character at the location is deleted, the tag moves to the location of the nearest character that was included in a tagged area. If all the characters associated with a tag are deleted, the tag no longer exists.

Manipulating portions of text (setting fonts or colors, binding actions, and so on) is done by identifying the text portion with a tag and then manipulating the tagged text via the *textWidget* tag configure command. For instance, the command

```
$textWidget tag configure loud -font [times 24 bold]
```

will cause the text tagged with the tag loud to be drawn with large, bold letters. All text tagged loud will be displayed in a large, bold font.

The sel tag is always defined in a text widget. It cannot be destroyed. When characters in the text widget have been selected (by dragging the cursor over them with the mouse button depressed), the sel tag will define the start and end indices of the selected characters. The sel tag is configured to display the selected text in reverse video.

13.1.3 Mark Overview

Marks are associated with the spaces between the characters, instead of being associated with a character location. If the characters around a mark are deleted, the mark will be moved to stay next to the remaining characters. Marks are manipulated with the *textWidget* mark commands.

A mark identifies a single location in the text, not a range of locations.

A mark can be declared to have gravity. The gravity of a mark controls how the mark will be moved when new characters are inserted at a mark. The gravity may be one of:

left — The mark is treated as though it is attached to the character on the left, and any new characters are inserted to the right of the mark. Thus, the location of the mark on a line will not change.

right — The mark is treated as though it is attached to the character on the right, and new characters are inserted to the left of the mark. Thus, the location of the mark will be one greater character location every time a character is inserted.

The following two marks are always defined.

insert — The location of the insertion cursor.

current — The location closest to the mouse location. Note that this is not updated if the mouse is moved with button 1 depressed (as in a drag operation).

13.1.4 Image Overview

Images created with the `image create` command (covered in Chapter 12) can be inserted into a `text` widget. The annotation that defines an image uses a single character location regardless of the height or width of the image. The *`textWidget`* `image create` command can insert multiple copies of an image into a `text` widget. Each time an image is inserted, a unique identifier will be returned that can be used to refer to this instance of the image.

13.1.5 Window Overview

The *`textWidget`* `window create` command will insert a Tk widget such as a `button`, `canvas`, or even another `text` widget into a `text` widget. Only a single copy of any widget can be inserted. If the same widget is used as the argument of two `textWidget window create` commands, the first occurrence of the widget will be deleted, and it will be inserted at the location defined in the second command.

13.2 CREATING A `text` WIDGET

`Text` widgets are created the same way as other widgets, with a command that returns the specified widget name. Like other widgets, the `text` widget supports defining the height and width on the command line. Note that the unit for the height and width is the size of a character in the default font, rather than pixels. You can determine the font being used by a `text` widget with the `configure` or `cget` widget command. (See Chapter 11.)

Syntax: `text` *`textName ?options ?`*

Create a `text` widget.

`textName` The name for this text widget.

`?options?` Some of the options supported by the `text` widget are:

`-state` *`state`*

Defines the initial state for this widget. May be one of:

`normal`	The `text` widget will permit text to be edited.
`disabled`	The `text` widget will not permit text to be edited. This option creates a display-only version of the widget.

`-tabs` *`tabList`*

Defines the tab stops for this `text` widget. Tabs are defined as a location (in any screen distance format, as described in Chapter 11) and an optional modifier to define how to justify the text within the column. The modifier may be one of:

`left`	Left justify the text.
`right`	Right justify the text.
`center`	Center the text.

`numeric` Line up a decimal point, or the least significant digit at the tab location.

`-wrap` *style*

Defines how lines of text will wrap. May be one of:

`none` Lines will not wrap. Characters that do not fit within the defined width are not displayed.

`word` Lines will wrap on the space between words. The final space on a line will be displayed as a `newline`, rather than leave a trailing space on the line before the wrap. Words are not wrapped on hyphens.

`char` Lines will wrap at the last location in the `text` widget, regardless of the character. This is the default mode.

`-spacing1` *distance*

Defines the extra blank space to leave above a line of text when it is displayed. If the line of text wraps to multiple lines when it is displayed, only the top line will have this space added.

`-spacing2` *distance*

Defines the extra blank space to leave above lines that have been wrapped when they are displayed.

`-spacing3` *distance*

Defines the extra blank space to leave below a line of text when it is displayed. If the line of text wraps to multiple lines when it is displayed, only the bottom line will have this space added.

Example 1

Script Example

```
#  Create and grid the text widget
set txt [text .t —height 12 —width 72 —background white \
    —tabs {2i right 2.4i left 4.6i numeric} —font {helvetica 12}]
grid $txt

#  Insert several lines at the end location. Each new line
#  becomes the new last line in the widget

$txt insert end "The next lines demonstrate tabbed text\n"
```

```
$txt insert end "noTab \t Right Justified \t\
Left Justified \t Numeric\n"

$txt insert end "\n"

$txt insert end "line1 \t column1 \t column2 \t 1.0\n"

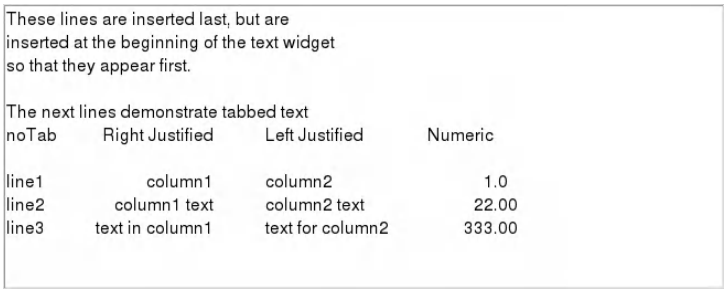
$txt insert end "line2 \t column1 text \t column2\
text \t 22.00\n"

$txt insert end "line3 \t text in column1 \t text\
for column2 \t 333.00\n"

#  Insert a few lines at specific line locations
#  to demonstrate some location example

$txt insert 1.0 "These lines are inserted last, but are\n"
$txt insert 2.0 "inserted at the\n"
$txt insert 2.end " beginning of the text widget\n"
$txt insert {3.5 linestart} "so that they appear first.\n"
```

Script Output



13.3 Text WIDGET SUBCOMMANDS

This section describes several of the more useful subcommands supported by the `text` widget. See the on-line documentation for your version of Tk for more details, and to see if other subcommands are implemented in the version you are using. The `dump` subcommand provides detailed information about the content of the `text` widget. Thus, it is very seldom used in actual programming, but can be very useful when debugging a complex display that's not doing quite what you expect. The `dump` subcommand is used in the following discussion of the `text` widget, as we examine how text strings, marks, and tags are handled by the `text` widget.

The dump subcommand returns a list of the text widget's content between two index points. Each entry in the list consists of sets of three fields: identifier, data, and index. The identifier may have one of several values that will define the data that follows. The data field will contain different values, depending on the value of the identifier. The index field will always be a value in line.char format. The values of identifier and the associated data may be one of the following.

text	The following data will be text to be displayed on the text widget. If there are multiple words in the text, the text will be grouped with curly braces.
tagon	Denotes the start of a tagged section of text. The data associated with this identifier will be a tag name. The index will be the location at which the tagged text starts.
tagoff	Denotes the end of a tagged section of text. The data associated with this identifier will be a tag name. The index will be the location at which the tagged text ends.
mark	Denotes the location of a mark. The data associated with this identifier will be a mark name. The index will be the location to the right of the mark.
image	Denotes the location of an image to be displayed in the text widget. The index will be the index of the character to the right of the image.
window	Denotes the location of a window to be displayed in the text widget. The index will be the index of the character to the right of the window.

13.3.1 Inserting and Deleting Text

The insert subcommand will insert one or more sets of text into the text widget. Each set of text can have zero or more tags associated with it.

Syntax: *textName* insert *index text ?tagList? ?moreText? ?moreTags? ?...?*

Insert text into text widget.

textName	The name of the text widget.
index	The index at which to insert this text, formatted as described in Section 13.1.1
text	A string of text to insert. If this text includes embedded newlines (\n), it will be inserted as multiple lines.
tagList	An optional list of tags to associate with the preceding text.

The following example creates a text widget, inserts three lines of text, and displays a formatted dump of the text widget content. Notice the foreach command with three arguments in a list to step through the dump output. The ability to use a list of arguments to a foreach command was added in Tcl revision 7.4. The foreach with a list of iterator variables is a useful construct for stepping through data formatted as sets of items, instead of as a list of lists.

Example 2

Script Example

```
text .t -height 5 -width 60 -background white \
  -font {helvetica 12}
grid .t

.t insert end "This text is tagged 'TAG1'. " TAG1
.t insert end "This is on the same line, 'TAG2'\n" TAG2
.t insert end \
    "The linefeed after 'TAG2' creates a new line.\n"

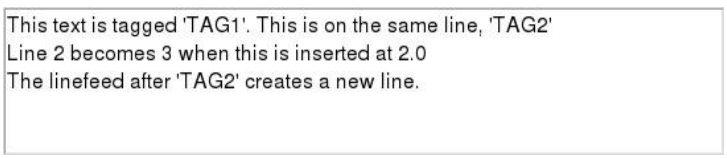
.t insert 2.0 \
    "Line 2 becomes 3 when this is inserted at 2.0\n" \
 INSERTED

set textDump [.t dump -all 1.0 end ]
puts "[format "%-6s %-45s %4s\n" ID DATA INDEX]"
foreach {id data index} $textDump {
  puts "[format {%-6s %-45s %4s} \
      $id [string trim $data] $index]"
}
```

Script Output

```
ID     DATA                                          INDEX

tagon  TAG1                                           1.0
text   This text is tagged 'TAG1'.                    1.0
tagoff TAG1                                          1.28
tagon  TAG2                                          1.28
text   This is on the same line, 'TAG2'              1.28
tagoff TAG2                                           2.0
tagon  INSERTED                                       2.0
text   Line 2 becomes 3 when this is inserted at 2.0  2.0
tagoff INSERTED                                       3.0
text   The linefeed after 'TAG2' creates a new line.  3.0
mark   insert                                         4.0
mark   current                                        4.0
text                                                  4.0
```



In the previous example, all lines are shorter than the width of the text widget. Since none of the lines wrap, the text widget display resembles the internal representation. The first line of the example inserts a line with the text *This text is tagged 'TAG1'*. at the end of the text widget. Since there was no text in the widget, the end was also the beginning. The dump shows that tag TAG1 starts at index 1.0 and ends at index 1.28. The text also starts at index 1.0.

Note that the second set of text is also inserted onto line 1. This is because the first set of inserted text did not have a newline character to mark it as terminating a line. The second set of text has a newline character at the end, which causes the next line to be inserted at index 2.0. However, the last insert command inserts text at location 2.0 and terminates that text with a newline, pushing the line that was at index 2.0 down to index 3.0.

The delete subcommand deletes the text between two locations.

Syntax: *textName* delete *startIndex ?endIndex?*

Remove text from a text widget.

startIndex	The location at which to start removing characters.
?endIndex?	If *endIndex* is greater than *startIndex*, delete the characters after *startIndex* to (but not including) the character after *endIndex*. If *endIndex* is less than *startIndex*, no characters are deleted. If *endIndex* is not present, only the character after *startIndex* is deleted.

```
# Clear all text in a widget
$textWidget delete 1.0 end
```

13.3.2 Searching Text

A script can search for text within a text widget using the *textWidget* search subcommand. This command will search forward or backward from an index point or search between two index points.

Syntax: *textName* search *?options? pattern startIndex ?endIndex?*

textName	The name of the text widget.	
search	Search the content of this text widget for text that matches a particular pattern.	
?options?	Options for search include:	
	Search Direction	These are mutually exclusive options.
	-forwards	The direction to search from the startIndex.
	-backwards	
	Search Style	These are mutually exclusive options. The default search style is to use glob rules.

	-exact	Search for an exact match.
	-regexp	Use regular expression rules to match the pattern to text.
	-nocase	Ignore the case when comparing pattern to the text.
	Other Options	These are not mutually exclusive options.
	-count *varName*	The -count option must be followed by a variable name. If the search is successful, the number of characters matched will be placed in varName.
	--	Signifies that this is the last option. The next argument will be interpreted as a pattern, even if it starts with a "-", and would otherwise be interpreted as an option. As with the switch command, it is good style to use the "--"option whenever you use the search subcommand. This will ensure that your code will not generate an unexpected syntax error if it is used to search for a string starting with a "-".
pattern	A pattern to search for. May be a glob or regexp pattern, or a text string.	
startIndex	The location in the text widget to start searching from.	
?endIndex?	An optional location in the text widget to stop searching.	

Example 3

Script Example

```
#  Create and pack a text widget
text .t –height 5 –width 68 –font {times 14}
grid .t

.t tag configure courier –font {courier 14 }

#  Insert some text
.t insert end "The default style for the "
.t insert end "search" courier
.t insert end " command is to use"
.t insert end " glob " courier
.t insert end "rules.\n"
```

```
.t insert end "The"
.t insert end " -regexp " courier
.t insert end \
    "flag will treat the pattern as a regular expression\n"

.t insert end "and, using the"
.t insert end " -exact " courier
.t insert end "flag will match an exact string."

#  "Exact" doesn't exist in this text, search will return ""
set pos \
    [.t search -exact -count matches -- "Exact" 1.0 end]
puts "Position of 'Exact' is '$pos' "

#  "-exact flag" does exist. This will match 11 characters
#  The two different fonts do not matter.
set pos \
    [.t search -exact -count matches -- "-exact flag" 1.0 end]

puts "Position of '-exact': $pos  matched $matches chars"

#  This regular expression search also searches for "-e"
#  followed by any characters except a space .
#  It will match to "-exact"
set pos \
    [.t search -regexp -count matches -- {-e[^ ]+} 1.0 end]
puts "Position of '-e\[^ \]+': $pos matched $matches chars"

#  This is an error - the lack of the "--" argument causes the
#   search argument "-e[^ ]*" to be treated as a flag.

set pos [.t search -regexp -count matches {-e[^ ]*} 1.0 end]
puts "This line is not evaluated
```

Script Output

```
Position of 'Exact' is ''
Position of '-exact': 3.15  matched 11 chars
Position of '-e[^ ]+': 3.15 matched 6 chars
Error in startup script: bad switch "-e[^ ]*": must be --,
-all, -backward, -count, -elide, -exact, -forward,
-nocase, -nolinestop, -overlap, -regexp, or -strictlimits
    while executing
".t search -regexp -count matchchars {-e[^ ]*} 1.0 end"
```

The default style for the search command is to use glob rules.
The -regexp flag will treat the pattern as a regular expression
and, using the -exact flag will match an exact string.

13.3.3 The mark Subcommands

The *textName* mark subcommands are used to manipulate marks in the text widget. Marks are used when you need to remember a particular location in the text. For instance, you can use marks to define an area of text to highlight, for a cut-and-paste operation, and so on. The *textName* mark set command defines a mark.

Syntax: *textName* mark set *markName index*

Set a mark in the text widget textName.

markName The name to assign to this mark.
index The index of the character to the right of the mark.

Example 4

```
# Put a mark at the beginning of the first three lines
# The marks are named start_1, start_2, etc
for {set line 1} {$line <= 3} {incr line}  {
  $textWidget mark set start_$line $line.0
}
```

The textName mark unset command will remove a mark.

Syntax: *textName mark unset markName ?... markNameN?*

Remove a mark from the text widget textName.

*markName** The name or names of the mark to remove.

Example 5

```
# Remove start_1, start_2 and start_3 marks
for {set line 1} {$line <= 3} {incr line} {
  $textWidget mark unset start_$line
}
```

You can obtain a list of all marks defined in a text widget with the *textName* mark names command.

Syntax: *textName* mark names

Return the names of all marks defined in the text widget textName.

Example 6

```
# Remove all start_* marks
foreach mark [$textWidget mark names] {
  if {[string first start_ $mark] == 0} {
    $textWidget mark unset \$mark\\
  }
}
```

You can search for marks before or after a given index position with the *textName* mark next and textName mark previous commands.

Syntax: *textName* mark next *index*

Syntax: *textName mark previous index*

Return the name of the first mark after (next) or before (previous) the *index* location.

index The index from which to start the search.

Example 7

Script Example

```
# Create a text widget for example
pack [set textWidget [text .t]]

# Put a mark at the beginning of the first three lines
# The marks are named start_1, start_2, etc
for {set line 1} {$line <= 3} {incr line}  {
  $textWidget mark set start_$line $line.0
}

# Step through the marks in a text widget
# Start at the beginning
set index 1.0
while {$index ne ""} {
  set index [$textWidget mark next $index]
  puts "Index is: $index"
}
```

Script Output

```
Index is: start_1
Index is: insert
Index is: current
Index is: start_3
Index is: start_2
Index is:
```

The next example shows how marks appear in a `text` widget.

Example 8

Script Example

```
grid  [text .t –height 5 –width 60 –font {helvetica 12}]

.t insert end "A mark will be set in this line,\n"
.t insert end "as shown by dump"
.t mark set demoMark 1.0

puts "The first mark is: [.t mark next 1.0]"
puts "The last mark is: [.t mark previous end]"

set textDump [.t dump –all 1.0 end ]

puts "[format "%–7s %–40s %6s\n" ID DATA INDEX]"

foreach {id data index} $textDump {
  puts "[format {%–7s %–40s %6s} $id [string trim $data] $index]"
}
```

Script Output

```
The first mark is: demoMark
The last mark is: current
ID      DATA                                      INDEX

mark    demoMark                                    1.0
text    A mark will be set in this line,            1.0
text    as shown by dump                            2.0
mark    insert                                     2.16
mark    current                                    2.16
text                                               2.16
```

13.3.4 Tags

Many of the interesting things that can be done with a `text` widget are done with tags. Tags define text that should be displayed in different fonts or colors, text with a highlighted background, text bound to an event, and so on.

Creating and Destroying Tags

Tags can be attached to text when the text is inserted, as shown in the example in section 13.2 or they can be added later with the *textName* `tag add` command. The *textName* `tag remove` command will remove a tag from a specific set of characters, and the `textName tag delete` command will remove all occurrences of a tag.

Syntax: *textName* tag add *tagName startIndex1 ?end1? ?start2 ... endN?*

`textName`	The `text` widget that will contain the tag.
`tag add`	Add a tag at the defined index points.
`startIndex ?end?`	The tag will be attached to the character at `startIndex` and will contain all characters up to but not including the character at `end`. If the `end` is less than the `startIndex`, or if the `startIndex` does not refer to any character in the text widget, no characters are tagged.

Example 9

```
# Tag the first letter of the page
$textWidget tag add "firstLetter" 1.0 1.1
```

Syntax: *textName* tag remove *tagName startIndex1 ?end1? ?start2 ... endN?*

`textName`	The `text` widget that contains the tag.
`tag delete`	Remove tag information for the named tags from characters in the ranges defined.
`tagName`	The names of tags to be removed.
`startIndex ?end?`	The range of characters from which to remove the tag information.

Example 10

```
# Remove the firstLetter tag on the first letter
$textWidget tag remove "firstLetter" 1.0 1.1
```

Syntax: *textName* tag delete *tagName ?...tagnameN*?

textName	The text widget that contains the tag.
tag delete	Delete all information about the tags named in this command.
*tagName**	The names of tags to be deleted.

Example 11

```
# Clear all tags in this widget
# NOTE: "tag names" is described in the next section
# Pre Tcl 8.5
# eval $textWidget tag delete [$textWidget tag names]
# Tcl 8.5 and newer
$textWidget tag delete *[$textWidget tag names]
```

Finding Tags

Your script can get a list of tags that have been defined at a particular location, a range of locations, or an entire text widget. You can also retrieve a list of locations that are associated with a tag. Multiple tags can reference the same location in a text file. For instance, the first character of a paragraph may be in a different font, tagged to show that it is the start of a keyword, and tagged as the first character after a page break. You can obtain a list of the tags at a location with the tag names command.

Syntax: *textName* tag names *index*

textName	The text widget that contains the tags.
tag names	Return a list of all tags defined at index.
index	The index point to check for tags.

Example 12

```
# Display the tags at the start of line 2:
puts [$textWidget tag names 2.0]
```

When you are processing the content of a text widget, you may want to step through the widget looking at tags in the order in which they appear. The nextrange and prevrange commands will step through a text widget, returning the index points that fall within the requested range of characters.

Note that the prevrange and nextrange commands expect the start and end locations in the opposite order. For the prevrange command, startIndex is greater than endIndex, whereas for nextrange the endIndex is greater than startIndex.

Syntax: *`textName`* tag nextrange *`tagName startIndex ?endIndex?`*

Syntax: *`textName`* tag prevrange *`tagName startIndex ?endIndex`*?

`textName`	The `text` widget that contains the tags.
`tag nextrange`	Return the first index defined after `startIndex` but before `endIndex`.
`tag prevrange`	Return the first index defined before `startIndex` but after `endIndex`.
`tagName`	The name of the tag to search for.
`startIndex endIndex`	The index points that defined the boundaries for the search.

Example 13

Script Example

```
# Create a text widget for example
set textWidget [text .t -font {helvetica 12} \
    -height 4 -width 20]
grid $textWidget
# Add tagged and untagged text text
for {set i 0} {$i < 3} {incr i} {
  $textWidget insert end "$i --" "myTag"
  $textWidget insert end "-$i\n"
}

# Initialize the "current" mark to the start of text
$textWidget mark set current 1.0

# Find the next range of the tag "myTag" after current
set tagRange [$textWidget tag nextrange "myTag" current]

# Loop  until the nextrange returns an empty string
while {![string match $tagRange ""]} {
  # Do something interesting with the tag data here
  puts "Range for myTag: $tagRange"

  # Set the current mark to the end of the tagged area
  $textWidget mark set  current [lindex $tagRange 1]

  # Find the next tagged range
  set tagRange [$textWidget tag nextrange "myTag" current]
}
```

Script Output


```
Range for myTag: 1.0 1.4
Range for myTag: 2.0 2.4
Range for myTag: 3.0 3.4
```

The `tag ranges` command will return a list of all index ranges a tag references.

Syntax: *textName* `tag ranges` *tagName*

textName	The `text` widget that contains the tags.
`tag ranges`	Return a list of index ranges that have been tagged with `tagName`.
tagName	The name of the tag to search for.

Example 14

Script Example

```
# Create a text widget for example
grid [set textWidget [text .t]]

# Add tagged and untagged text
for {set i 0}{$i < 3} {incr i} {
  $textWidget insert end "$i ––" "myTag"
  $textWidget insert end "–$i\n"
}

set tagRanges [$textWidget tag ranges "myTag"]
puts "The tagged ranges are: $tagRanges"
foreach {start end} $tagRanges {
   puts "‘[$textWidget get $start $end]’ is tagged"
}
```

Script Output

```
The tagged ranges are: 1.0 1.4 2.0 2.4 3.0 3.4
‘0 ––’ is tagged
```

```
'1 --' is tagged
'2 --' is tagged
```

The next example shows a more complex set of tags and text strings.

Example 15

Script Example

```
text .t -height 5 -width 65 -font {helvetica 12}
grid .t
.t insert end \
  "Text can have multiple tags at any given index.\n" T1

.t insert end \
  "You can get a list of tags at an index with 'tag names'.\n" T2

.t insert end \
  "You can search the text with prevrange and nextrange.\n" T3

.t insert end "You can get the ranges where a tag is \
  defined with 'ranges'" T4

# Set tags for
# firstchar: first character on a line
# firstline: first line on the text widget
# secondline: second line on the text widget
# firstpage: all characters on this text widget

.t tag add firstchar 1.0
.t tag add firstline 1.0 {1.0 lineend}
.t tag add firstpage 1.0 end
.t tag add secondline 2.0 {2.0 lineend}
.t tag add firstchar 2.0
.t tag add firstchar 3.0
.t tag add firstchar 4.0

# Show some results
puts "These tags are defined:\n  [.t tag names]\n"
puts "These tags are defined at index 1.0:"
puts "  [.t tag names 1.0]\n"
puts "Tag firstchar is defined for these ranges:"
puts "  [.t tag ranges firstchar]\n"
puts "Tag T2 is defined for the range:\n  [.t tag ranges T2]\n"
puts "The first occurrence of firstchar after the"
puts "  start of line 3 is:"
```

```
puts "  [.t tag nextrange firstchar 3.0]\n"
puts "The first occurrence of firstchar before the"
puts "  start of line 3 is:"
puts "  [.t tag prevrange firstchar 3.0]\n"
```

Script Output

```
These tags are defined:
  sel T1 T2 T3 T4 firstchar firstline firstpage secondline

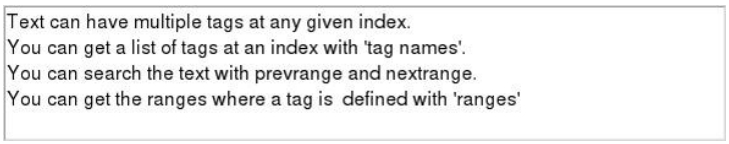
These tags are defined at index 1.0:
  T1 firstchar firstline firstpage

Tag firstchar is defined for these ranges:
  1.0 1.1 2.0 2.1 3.0 3.1 4.0 4.1

Tag T2 is defined for the range:
  2.0 3.0

The first occurrence of firstchar after the
  start of line 3 is:
  3.0 3.1

The first occurrence of firstchar before the
  start of line 3 is:
  2.0 2.1
```



Using Tags

Tags are the interface into the `text` widget that allows you to bind actions to events on sections of text, set colors, set fonts, set margins, set line spacing, and so on. This section discusses a few of the things that can be done with tags and provides an example of how to use tags. The complete list of options you can set with a tag is in the on-line documentation. The `tag bind` command will bind an action to an event in a tagged area of text.

Syntax: *textName* tag bind *tagName ?eventType? ?script?*

textName The name of the `text` widget.

`tag bind`	Bind an action to an event occurring on the tagged section of text, or return the script to be evaluated when an event occurs.
`tagName`	The name of the tag that defines the range of characters that will accept an event.
`?eventType?`	If the `eventType` field is set, this defines the event that will trigger this action. The event types are the same as those defined for `canvas` events in Section 11.5.
`script`	The script to evaluate when this event occurs.

Example 16

Script Example

```
# Create a text widget for example
set textWidget [text .t -height 10 -width 60\
    -font {helvetica 14}]
grid $textWidget

# insert some text
$textWidget insert end "This is text with a "
$textWidget insert end "secret" secretWord
$textWidget insert end " in it"

# Pop up a message box when someone clicks on the secret word
$textWidget tag bind secretWord <Button-1> \
  {tk_messageBox -type ok -message "You found the secret!"}
```

Script Output

After clicking the word secret.

The `tag configure` command will modify how tagged text is displayed. The `tag configure` command will allow you to set many of the options that control a display, including the following.

-foreground *color*	The foreground color for the text.
-background *color*	The background color for the text.
-font *fontID*	The font to use when displaying this text.
-justify *style*	How to justify text with this tag. May be left, right, or center.
-offset *pixels*	The vertical offset in pixels for this line from the base location for displaying text. A positive value raises the text above where it would otherwise be displayed, and a negative value lowers the text. This can be used to display subscripts and superscripts.
-lmargin1 *pixels* -lmargin2 *pixels* -rmargin *pixels*	The distance from the left and right edges to use as a margin. The rmargin value is a distance from the right edge of the text widget to treat as a right margin. The lmargin1 is a distance from the left edge to treat as a margin for display lines that have not wrapped, and lmargin2 is a distance to use as a margin for lines that have wrapped.
-underline *boolean*	Specifies whether or not to underline characters.

The next example displays a few lines of text in several fonts and binds the creation of a label with more information to a button click on the word tag.

Example 17

Script Example

```
#  Create and pack the text widge
text .t –height 6 –width 70 –font {helvetica 12}
grid .t

#  Insert some text with lots of tags
.t insert end "T" {firstLetter} "he " {normal}
.t insert end "tag" {code action}
.t insert end " command allows you to create displays\n" {normal}
.t insert end "with multiple fonts" {normal} "1" {superscript}
.t insert end "\n\n"
.t insert end "1" {superscript}
.t insert end "Tcl/Tk: A Developer's Guide, " {italic} \
    "Clif Flynt, " {bold}
.t insert end "Morgan Kaufmann, 2011"

#  Set the fonts for the various types of tagged text
.t tag configure italic –font {times 14 italic}
.t tag configure normal –font {times 14 roman}
.t tag configure bold –font {times 14 bold}
```

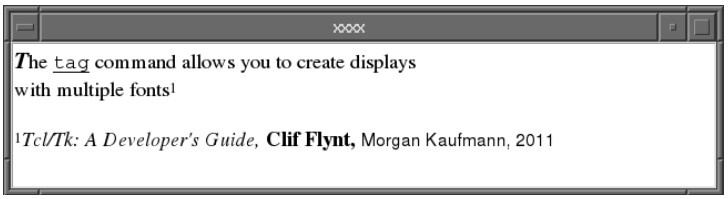
```
#  Set font and offset to make superscript text
.t tag configure superscript —font {times 10 roman} —offset 3

#  Set font for typewriter style font
.t tag configure code —font {courier 14 roman underline}

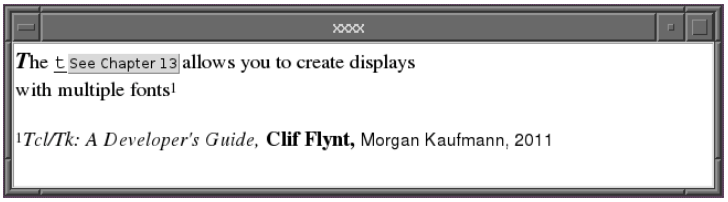
#  Define a font to make a fancy first letter for a word
.t tag configure firstLetter —font {times 16 italic bold}

#  Bind an action to the text tagged as "action"
.t tag bind {action} <Button—1> {
  label .l —text "See Chapter 13" —relief raised
  place .l —x 50 —y 10
}
```

Script Output



After Clicking "Tag":



13.3.5 Inserting Images and Widgets into a `text` Widget

Following the pattern used by the `canvas` widget, you can insert an image created with the `image create` command into a `text` widget with the *`textName`* `image create` command, and other Tk widgets can be inserted into a `text` widget with the *`textName`* `window create` command.

Syntax: *textName* image create *index ?options?*

textName	The name of the text widget this image will be placed in.
image create	Insert an image into a text widget.
index	The index at which to insert the image.
?options?	The options for the image create command include:
-image *handle*	The image object handle returned when the Tk image object was created. (See Section 10.8.)
-name *imageName*	A name to use to refer to this image. A default name will be returned if this option is not present.

Example 18

Script Example

```
# Create a bird graphic
set bird {
R0lGODlhIAAgAJEAANnZ2QAAAP///////yH5BAEAAAAALAAAAAAgACAAAAKWhI+p
y+OPo5yO2qtIHIJvFD8IPhrFDwAAxCPwjeID3N3dEfiYFB/zCD4mxcdMgo9H8TGT
4ONRfARAgo9pFP8IPqYaxccj+JhI8Y1gx93dHcUPgo+pSPGD4GMqUnyCj6lJ8Qk+
pibFJviYqhSb4GOqRlBC+Ji6EZQQPqZqBCWEj6kbQfAxdSOIPqYuBIXwMXW5/WGU
kz5SADs=
}
image create photo bird –data $bird

# Create and pack the text widget
text .t –height 2 –width 40 –background white –font {times 16 bold}
pack .t

.t image create 1.0 –image bird
.t insert 1.1 " Watch the birdie"
```

Script Output

In the same fashion, another Tk widget (even another text widget) can be placed in a text widget with the window create object command.

Syntax: *textName* window create *index ?options?*

textName	The name of the text widget that this window will be inserted into.
window create	Insert a Tk widget into the text widget.
index	The index at which to insert the Tk widget.
?options?	The options for the window create command include:
-window *widgetName*	The name given to this widget when it was created.
-create *script*	A script to evaluate to create the window if the -window option is not used.

The text widget inserts the new image or widget at the requested index. It treats images and windows as if they were a single character. If the image or widget is inserted into an existing line, the height of the line will be increased to the size of the image. If you want to create a display with columnar format, you can use two text widgets and put the text for the left column in one text widget, and the text for the right column in the other, and then use window create to place the two text windows side by side.

Example 19

Script Example

```
grid [text .t –width 60 –height 5 –background white]
label .l –text "label"
.t window create 1.0 –window .l
.t insert 1.1 " This is a label\n"
.t window create 2.0 –create {button .b1 –text "Button"}
.t insert 2.1 " This is a button\ndescribed on two lines"
```

Script Output

Notice in the previous example that *described on two lines* appears below the button, instead of having all text appear to the right. Images and windows appear on a line, just like text, and cannot span multiple lines. If you want to have an image that spans multiple lines, you must add a second text widget to hold the text, as shown in the next example.

The -borderwidth and -highlightthickness options are used with the second text widget to make it "invisible." By default, a text widget will have a border around it, which may not be the visual effect you wish.

Example 20

Script Example

```
set scroll {
R0lGODdhNQClAJEAAP///wAAAP///////ywAAAAANQClAAAC/4SPqcvtD6OctNqL
s968+z8RsQk+pi43xccg+Ji6jBQfkeBj6rJRfEwj+Ji6S/GD4FNsgo+pqxSf4B/F
JviYukmxAQAOCEBEBMHH1M2gJABA8QEAjuBj6iLFBrgzICARQfAxdS8oCUBEBAGJ
MIugED6m7gUlAYgIICQRAAAUEz6m7gUlAYgAICRBsCOC4GPqXlASiAgAQkKwI4Lg
Y+piUBKAiCD4FxGIiAghROIAXBB8iIhARIQQQiQOAFwAAaDYBB9TN4OSAABQfAyC
j6mrFBsA4O4IPqbuUmyCT7EJPqYuA8Um+Ji63EhxCD6mLjdSHIKPqcuNFJvgY+py
H8Um+P+YutxHsQk+pi73UWyCj6nLfRSf4GPqchvFJ/iYutxHsQk+pi73UWyCj6nL
fRSb4GPqch/FJ/iYutxG8Qk+pi63UXyCj6nLbRSf4GPqch/FJviYutxH8Qk+pi63
UXyCj6nLbRSf4GPqchvFJ/iYutxG8YPgY+pyA8UPgo+py2OUn+Bj6nIbxSf4mLrc
RvEJPqYut1H8IPiYutxA8YPgY+pyA8UPgo+pywOUPwg+pi43UHwj+Ji63EDxg+Bj
6nIDxQ+Cj6nLDRQ/CD6mLjdQ/CD4mLrcQPGN4GPqclN8I/iYutwU3wg+pi43UPwg
+Ji63EDxjeBj6nJTfCP4mLrcFN//CD6mLjfFN4KPqctN8Y3gY+pyU/wj+Ji6zBT/
CD6mLjPFP4KPqctN8Y3gY+pyU/wj+Ji6zBT/CD6mLjPFP4KPqctM8Y/gY+oyU/wj
+Ji6zBQfCT6mLi/FR4KPqctL8ZHgY+oyU/wj+Ji6zBT/CD6mLjPFP4KPqctM8Y/g
Y+pyU3wj+Ji63BT/CD6mLjPFP4KPqctM8Y/gY+pyU3wj+Ji63BTfCD6mLjfFN4KP
qctN8Y3gY+pyA8U3go+pyO3xjeBj6nJTfCP4mLrcFN8IPqYuN8U3go+pywOUPwg+
pi43UPwg+Ji63EDxjeBj6nJTfCP4mLrcQPGD4GPqcgPFD4KP/6nLDRQ/CD6mLjdQ
/CD4mLrcQPGD4GPqchvFJ/iYutxG8YPgY+pyA8UPgo+pywOUPwg+pi63UXyCj6nL
bRSf4GPqchvFJ/iYutxG8Qk+pi73UXyCj6nLbRSf4GPqchvFJ/iYutxG8Qk+pi73
UWyCj6nLfRSb4GPqch/FJviYutxH8Qk+pi63UXyCj6nLfRSb4GPqch/FJviYutxH
sQk+pi73UWyCj6nLjRSH4GPqciPFJviYutxHsQk+pi73UWyCj6nLfRSb4GPqciPF
IfiYumwUh+BDRAQiIoQQQiS+EeyICIKPqbsUHwEAMAg+pm5SbAC4AAIQEUHwMXUx
KAnBj8GIuKAkBB9TF4MSwreIuKAkBB9T94KSAABEEJwwiAgK4WPqXlACAIgIghNh
EUEhfEzdCOoAEBFBQCIMIighfEzdCOoAEBFASCIAICgJwcfUvaAEABTfAACQ4GPq
XlASAKD4BDsiguBj6l5QEgCACArhU2yCj6mLQUkIPgLFJviYukmxCf5RfIKPqasU
m+BT/CD4mLpM8TGD4GPqslF8RIKPqctJ8ZHgY+pyA8Um+Ji63P4wykmrvTjrzbv/
YCiOZEcUADs=
}
image create photo scroll -data $scroll
grid [text .t -width 65 -height 12 -background white]

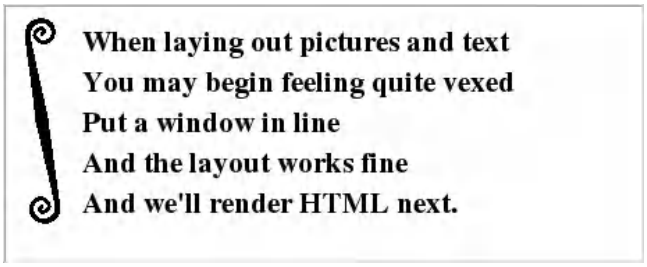
.t image create 1.0 -image scroll

text .t2 -width 35 -height 5 -background white \
    -font {times 16 bold} -borderwidth 0 \
    -highlightthickness 0

.t window create 1.1 -window .t2

.t2 insert 1.0 "When laying out pictures and text\n"
.t2 insert 2.0 "You may begin feeling quite vexed\n"
.t2 insert 3.0 "Put a window in line\n"
.t2 insert 4.0 "And the layout works fine\n"
.t2 insert 5.0 "And we'll render HTML next."
```

Script Output



13.4 HTML DISPLAY PACKAGE

HTML is currently the most portable protocol for distributing formatted text and the odds are good that HTML will continue to be important for many years. The `text` widget tags make it possible to use the `text` widget to display HTML text with little work.

Stephen Uhler wrote `html_library`, a library that will display HTML in a `text` widget. The `html_library` is a good example of what you can do with the `text` widget, so we will examine it. The HTML library is not a part of the standard Tk distribution. It is available on the companion website and on the web at the following address, among other locations.

- *http://noucorp.com/*

The `htmllib` parsing engine is used in the `tcllib` (*http://tcllib.sf.net/*) `htmlparse` package.

There is also an HTML display widget that is written in C. The C Language HTML widget is much faster at rendering pages, but the Tcl widget is easier to merge into applications for simple displays like HTML formatted documentation and embedded help pages.

13.4.1 Displaying HTML Text

Using the `html_library` package to display text is very simple, as follows.

Step	Example
1. Source the `html_library` code.	`source htmllib.tcl`
2. Create a `text` widget to contain the displayed HTML text.	`text .t`
3. Map the `text` widget to the display.	`pack .t`
4. Initialize the HTML library by calling `HMinit_win`.	`HMinit_win .t`
5. Display your HTML text by calling `HMparse_html`.	`HMparse_html HTML_TEXT "HMrender .t"`

The following example shows a text window being used to display some HTML text and a dump of the first couple of lines. Note that the tags are used to define the indenting and fonts.

Example 21

Script Example

```
# Load the htmllib scripts
source "htmllib.tcl"

# Create and display a text widget
grid [text .t —height 10 —width 50]

# Initialize the text widget
HMinit_win .t

# Define some HTML text
set txt {
<HTML><BODY>Test HTML
  <P><B>Bold Text</B>
  <P><I>Italics</I>
</BODY</HTML>
}

# and render it into the text widget
HMparse_html $txt "HMrender .t"

# Examine what's in the text widget
set textDump [.t dump —all 1.0 4.0]
puts "[format "%—7s %—30s %6s\n" ID DATA INDEX]"
foreach {id data index} $textDump {
  puts "[format {%—7s %—30s %6s} $id [string trim $data] $index]"
}
```

Script Output


```
ID      DATA                              INDEX

mark    current                             1.0
tagon                                       1.0
```

```
tagon   indent0                          1.0
tagon   font:times:14:medium:r           1.0
text    Test HTML                        1.0
tagoff  indent0                         1.11
tagoff                                  1.11
tagon   space                           1.11
text                                    1.11
text                                     2.0
tagoff  space                            3.0
```

You can see that the html library uses tags to define the fonts for the text widget to use to display the text.

13.4.2 Using html_library Callbacks: Loading Images and Hypertext Links

To display an image, the html_library needs an image widget handle. If the library tried to create the image, it would need code to handle all methods of obtaining image data, and the odds are good that the technique you need for your application would not be supported.

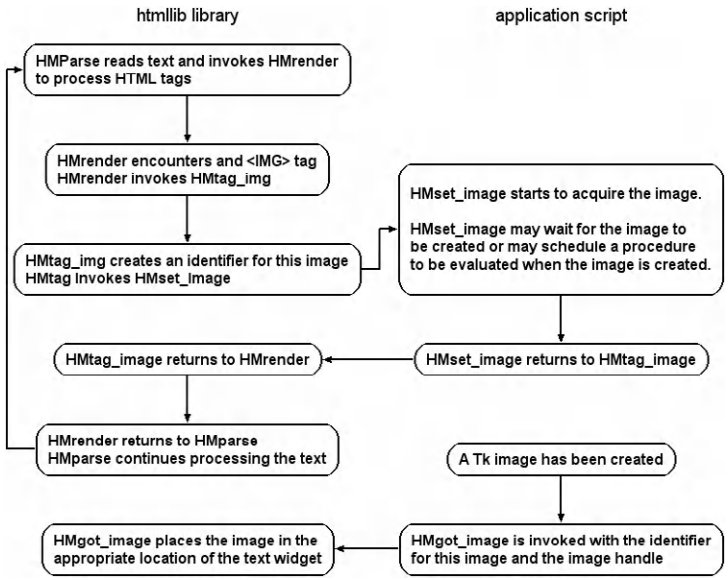
To get around this problem, the html_library calls a procedure that you supply (HMset_image) to get an image handle. This technique makes the library versatile. Your script can use whatever method is necessary to acquire the image data: load it from the Web, extract it from a database, code it into the script, and so on.

The requirements for handling hypertext links are similar. A browser will download a hypertext link from a remote site, a hypertext on-line help will load help files, and a hypertext GUI to a database engine might generate SQL queries. When a user clicks on a hypertext link, the html_library invokes the user-supplied procedure HMlink_callback to process the request.

Note that you must source html_library.tcl before you define your callback procedures. There are dummy versions of HMset_image and HMlink_callback in the html_library package that will override your functions if your script defines these procedures before it sources the html_library.tcl script.

One of the requirements of event-driven GUI programming is that procedures must return quickly. When the script is evaluating a procedure, it is not evaluating the event loop to see if the user has clicked a button (perhaps the Cancel button!). In particular, if the procedure acquiring image data is waiting for data to be read from a remote site, you cannot even use update to force the event loop to run. Therefore, the html_library package was designed to allow the script retrieving an image to return immediately and to use another procedure in the html_library to display the image when it is ready.

The following flowchart shows control flow when the html_library code encounters an <IMG> tag. Note that the creation of the image need not be connected to the flow that initiated the image creation (the HMset_image procedure). An outside event, such as a socket being closed, can invoke HMgot_image.



When `HMset_image` is called, it is passed a handle that identifies this image to the `html_library`. When the application script calls `HMgot_image`, it includes this handle and the image object handle of the new image object so that the `html_library` code knows which `<IMG>` tag is associated with this image object.

Syntax: `HMset_image` *`win handle src`*

Create an `image` widget from the appropriate data, and transfer that `image` widget back to the `html_library` by calling `HMgot_image`.

`win` The name of the `text` widget that will ultimately receive the image.

`handle` A handle provided by the `html_library` script that must be included with the `image` handle when `HMgot_image` is invoked.

`src` The textual description of the image source. This is the content of the `SRC="XXX"` field.

Syntax: `HMgot_image` *`handle image`*

Maps an `image` widget handle into a `text` widget in the location defined by data associated with *`handle`*.

`handle` The value that was passed to `HMset_image` as an argument.

`image` The image handle returned by an `image create` command.

The next example displays the Tcl feather logo. In this example, the `HMset_image` call is done as a hard-coded image creation. In an actual application, this would be code to parse the `src` parameter, and so on.

Example 22

Script Example

```
source htmllib.tcl
set logo {
R0lGODdhKwBAAKUAAP////r6+vX19e3t7fPz8+jo6Obm5uDg4NXV1aurq56e
nnV1dcjIyOLi4ufn/5TO9Ovr6+zs7OHh4Xt7e1JSUsXFxX9/f2FhYeTk5I+P
j3Jycn19fbW1tUVFRWtra5mZmY2Njd/f38LCwvHx8ZycnPf39wAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAACwAAAAAKwBAAAAG/kCAcEgsEgOC
pGAgMDqLyCRhQCOYDtgDIsFNKL7fxKH5NAq84LR6zU4UlNNqIZtk2+94tiCA
zvv/CgkAfICFeIIAfXYLYAuMhmoIQopsDGANlpBpB5N4AwyODg8NmpudbAsM
Awqpq6VpBkIIiwONjJ+PjayFBUIHiwygDA65uxBLhQO+wKwOoIOJexERCMV3
ygC/eQ5pCwkBEhMUFBWAZNrAxF/eAhLj4xbmy3cLzrkD7u8XgN9CBtuNMBh4
Ny5DGnsKuKVBBKAAQDADxL3TsCHNAGL1noUZ4hCPwgUH8o3bFwjisOxgJAlx
dSfTggISC2YoEODArTucVubRiIHg/rgJEyIMSKWQTU4ALPMk4OBzXIcCAzxg
sgYmls4/CT4OpXBBgM2ibHoJEcBPa1MQZxRoZIMNAFmsZn2GGIrSTtu3f8x+
iFlglh8ybv8wusCVg8QPBqjqGYL3kALCFkK8qODJTj8hhPJQIVxBMtddfhgO
qqxmAYEIWjsX3IK1CGk1A7R2OCBi5IhqflQOed2ogNYLvn8O4G1qSGZ6wYGb
nUAAkO7ReRZAYKp83IehrRn7WYCBqQWY4zhgDO2k8SIEki1EHFcBguI1os1b
Tr+eAgn3WANoD12BgnqJFdiU335+HDDBf+Og8wdgSdGTwAQXxOZfAwrmwaBg
CnxQ3Uzj/vO1RIOWPVYdBRMU8N4aBHy43RgfPFUbBR8I+EdbINo1wVP5hJDA
ibBxJNgBWjXgTomGtNXRHwZWMNAHuAEiFgD/FFKBkvZVxMsQFfoxUzyQWJUN
IACU8EUiuoSBx1FZ3gEAK2SGOYd8YKAJhiB9eNHHmgoI8QieX7zFZ5xYgjEI
nOKkseYCeuZZaJ95VqabX4EEAMZbYo65kaBp1NHGEJAKgqkahKKxpp1kwcmQ
InxWeqcXiKIhpp14tjnnboamWukXt5bgaq64wkfrK5BcRhywdpAxLLFrNHEc
sgtCx6w8cLZxrB/K4NUFFgZAxcQeUSghQBxUDFDAuAZckYUWDQgoSOAS+pXh
7rtOBAEAOw==\\
}
set HTMLtxt {
<HTML>
<HEAD><TITLE>Example of Embedded image</TITLE></HEAD>
  <BODY>
  <B>Tcl</B>
  <IMG src=logo>
  <B>Powered</B>
  </BODY>
</HTML>
}
##########################################
# proc HMset_image {win handle src {speed {0}}}-\
#  Acquire image data, create a Tcl image object,
#  and return the image handle.
#
# Arguments
# win     The text window in which the html is rendered.
# handle  A handle to return to the html library with
```

```
#         the image handle
# src     The description of the image from:
#    <IMG src=XX>
#
# Results
# This example creates a hard-coded image. and then invokes
# HMgot_image with the handle for that image.

proc HMset_image {win handle src {speed {0}}} {
  global logo
  puts "HMset_image was invoked with WIN: \
      $win HANDLE: $handle SRC: $src"
  # In a real application this would parse the src, and load the
  # appropriate image data.
  set img [image create photo -data $logo]
  HMgot_image $handle $img
  return ""
}
text .t -height 6 -width 50
pack .t
HMinit_win .t
HMparse_html $HTMLtxt "HMrender .t"
```

Script Output

```
HMset_image was invoked with WIN:  .t HANDLE: .t.9 SRC: logo
```

The `html_library` uses a technique similar to the image callback procedures to resolve hypertext links. When a user clicks on a hypertext link, the parsing engine invokes a procedure named `HMlink_callback` with the name of the `text` window and the content of the `href=value` field. The `html_library` provides a dummy `HMlink_callback` that does nothing. An application must provide its own `HMlink_callback` procedure to resolve hypertext links.

Syntax: `HMlink_callback` *`win href`*

A procedure that is called from the `html_library` package when a user clicks on an `<A href=value>` field.

`win` The `text` widget in which the new text can be rendered.

`href` The hypertext reference.

The next example shows an `HMlink_callback` that will display the HTML text contained in a Tcl variable. In a browser application, the `HMlink_callback` procedure would download the requested URL, and in a help application it would load the requested help file.

Example 23

Script Example

```
source htmllib.tcl

######################################################################
# proc HMlink_callback {win href}--
#  This procedure is invoked when a user selects a hypertext
#  link in the HTML display
#
#  In an actual browser, the href field would contain the URL of
# the HTML page to retrieve. In this example, it contains the
# name of a Tcl variable
#
# Argument
#  win    The window in which the new page will be displayed
#  href   The hypertext reference
#
proc HMlink_callback {win href} {
  global HTMLText2 HTMLText HTMLText3
  puts "HMlink_callback was invoked with win: $win href: $href"

  # Clear the window
  HMreset_win $win

  # Display new text
  # - href will be substituted to the name of a text string,
  #   and "set varName" returns the contents of a variable.
  #   "[set $href]" could also be written as "[subst $$href]"
  HMparse_html [set $href] "HMrender $win"
}

# Define three sets of simple HTML text to display
# This is the first text to display. It is an Unordered List of
# hypertext links to the other two sets of HTML text.
set HTMLText1 {
  <HTML>
    <HEAD><TITLE> Initial Text </TITLE></HEAD>
    <BODY>
    <UL>
      <LI>
```

```
        <A href=HTMLText2> Clicking this line will select text 2.</A>
      <LI>
        <A href=HTMLText3> Clicking this line will select text 3.</A>
    </UL>
  </BODY> </HTML>
}

# This text will be displayed if the user selects the
# top line in the list
set HTMLText2 {
  <HTML>
  <HEAD><TITLE> Initial Text </TITLE></HEAD>
  <BODY>
  <CENTER>This is text 2.</CENTER>
</BODY> </HTML>
}

# This text will be displayed if the user selects the
#  bottom line in the list
set HTMLText3 {
   <HTML>
   <HEAD><TITLE> Initial Text </TITLE></HEAD>
   <BODY>
   <CENTER>This is text 3.</CENTER>
   </BODY> </HTML>
}

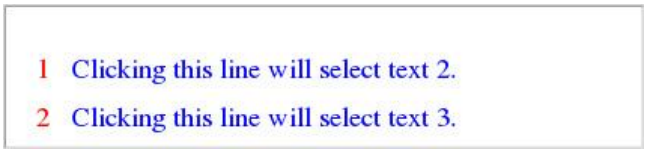
  # Create the text window for this display
  text .t -height 6 -width 60 -background white
  grid .t

# Initialize the html_library package and display the text.
HMinit_win .t
HMparse_html $HTMLText1 "HMrender .t"
```

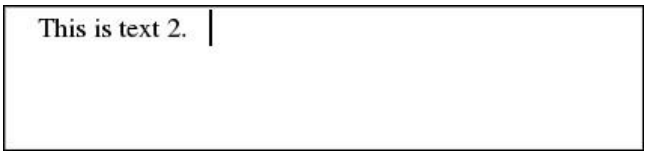
Script Output

```
HMlink_callback was invoked with win: .t href: HTMLText2
```

Before clicking the hypertext link



After clicking the top line



13.4.3 Interactive Help with the text Widget and htmllib

The next example shows how you can use the text widget bind command, an associative array, and the html library to create a text window in which a user can click on a word to get help. The tag bind command causes a mouse click event to invoke the ShowHelp procedure.

```
$ textWin tag bind help <Button-1>\
     [list textWithHelp::ShowHelp $textWin %x %y]
```

When someone clicks on the text widget, the text widget invokes the textWithHelp::ShowHelp procedure with the name of the text widget and the X and Y cursor location. Within the ShowHelp procedure, the format converts the X and Y position to a text widget index.

```
# Convert the X and Y cursor location into a text index.
set index [format "@%d,%d" $x $y]
```

That index is used to identify the start and end locations of the word that was clicked, and to get that word from the text widget with the following code.

```
# Get the word that surrounds that location from the text widget
set word [$txt get [list $index wordstart] [list $index wordend]]
```

By default, the text widget allows a user to edit the text with common emacs bindings or arrow keys. The last line:

```
$ txtWin configure -state disabled
```

disables the editing, making the window read-only. This is appropriate for a display you do not want a user to modify.

In the following example, only the word text has help associated with it, and all words in the text widget are tagged with the string help. For other applications, you might add a tag configure command to the textWithHelp to make words with help highlighted, and only tag the words that are in the index.

Example 24

Script Example

```
##############################################################
# proc textWithHelp {args}--
#  Creates a text widget with a binding for context
#    sensitive pop-up help
# Arguments
```

```
#   args   A list of arguments appropriate for the
#          text widget
#
# Results
#  Creates a new text widget with a binding on the tag 'help'
#  Returns the name of the new widget

proc textWithHelp {args} {
  set textWin [eval text $args]
  $textWin tag bind help <Button-1> \
       [list textWithHelp::ShowHelp $textWin %x %y]
  return $textWin
}

namespace eval textWithHelp {
  variable helpDocs

##############################################################
# proc addHelp {key text}--
#   Add help text to the help database.
# Arguments
#   key    The keyword to identify this help message
#   text   An HTML text message.
# Results
#   Updates the helpDocs array.

proc addHelp {key text} {
  variable helpDocs
  set helpDocs($key) $text
}

##############################################################
# proc ShowHelp {txt x y}--
#   Finds the word with the cursor and checks that this
#   word is an index into the helpDocs array.  If so, it
#   creates a popup window with a text widget, scrollbar,
#   and a 'Done' button.  The text widget will display the
#   HTML formatted text indexed by the keyword.
#
# Arguments
#   txt     The name of the parent window.
#   x       Cursor X location
#   y       Cursor Y location
# Results
#   Any previous popup help window associated with this text
#     widget is destroyed
#   If the cursor is on a word with help, a new toplevel
#     widget is created with the help text.
```

```
proc ShowHelp {txt x y} {
   variable helpDocs

   # Convert the X and Y cursor location into a text index.
   set index [format "@%d,%d" $x $y]

   # Get the word that surrounds that location from the
   #   text widget
   set word [$txt get [list $index wordstart] \
       [list $index wordend]]

   # If the word exists, do stuff, else just return.
   if {[info exists helpDocs($word)]} {

      # Destroy any existing window.
      catch {destroy $txt.help}

      # Create a new toplevel, and title it appropriately
      set help [toplevel $txt.help]
      wm title $help "Help for $word"

      # Create a new text widget, scrollbar and exit button
      text $help.t -width 60 -height 10 \
              -yscrollcommand "$help.sb set"
      scrollbar $help.sb -orient vertical \
              -command "$help.t yview"
      button $help.b -text "Done" -command "destroy $help"

      # Map the widgets to the toplevel window
      grid $help.t -row 1 -column 1
      grid $help.sb -row 1 -column 2 -sticky ns
      grid $help.b -row 2 -column 1

      # Initialize the text widget for HTML and render the
      #  text
       HMinit_win $help.t
       HMparse_html $helpDocs($word) "HMrender $help.t"
  }
 }
}

#
# A sample use of the previous code
#

# Load the html display library
```

```
source htmllib.tcl

# Add a help message to the help index.
textWithHelp::addHelp text \
 {<BODY>
  <CENTER><H4><CODE>text</CODE> widget</H4></CENTER>
  <CODE><B>Syntax: </B>text <I>textName ?options?</I></CODE>
  <P>
   The text widget can be used to display and edit text.
  </BODY>
 }

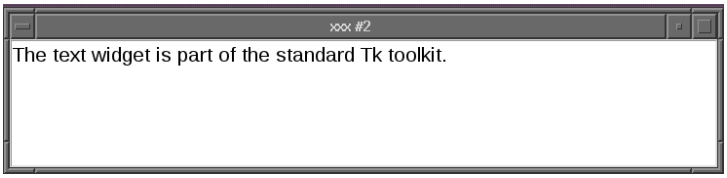
# Create and map the text window
set txtWin [textWithHelp .txt -background white -height 5 \
    -width 60 -font {arial 16} ]
grid $txtWin

# Insert text into the text widget.
$txtWin insert end \
    "The text widget is part of the standard Tk toolkit." help

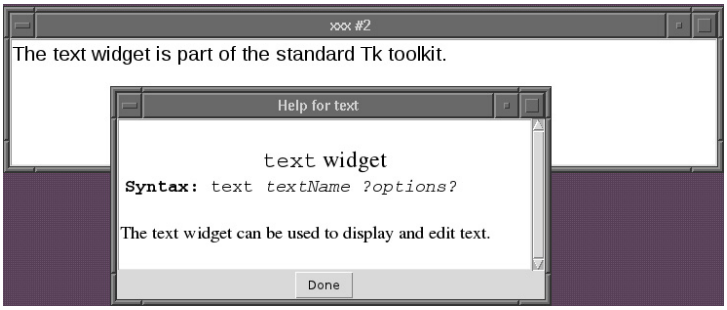
# Make the text window read only - disable editing.
$txtWin configure -state disabled
```

Script Output

Before clicking on the word text



After clicking on the word text



The previous example has hard-coded text to make an example that can be placed in the book. Your applications may use the http package to download pages from a remote site, load pages from a disk, generate pages from database records, or whatever technique your program specifications require.

The ease with which HTML support is added to the text widget may make you think that it would be easy to write a full browser with Tk. Before you do too much work on that project, take a look at Steve Ball's plume browser at *http://ftp.sunet.se/pub/lang/tcl/sorted/apps/plume-6.2/* and read Mike Doyle and Hattie Schroeder's book *Web Applications in Tcl/Tk*. Much of the work you will need to do to create a browser has already been done. Using a text widget to implement a browser display has the following problems.

- *Performance:* Displaying HTML in a text widget requires a great deal of parsing HTML and calculating layout parameters. This can be compute intensive, and becomes slow when done in an interpreted language like Tcl.
- *Layout limitations:* The text widget is optimized for pure text displays. As discussed in Section 13.3.5, you need to add extra windows to flow text around images, and so on.

D. Richard Hipp solved these problems with his TkHTML widget (*www.hwaci.com/sw/tkhtml/index.html*). This widget is the basis for several applications, including the full-featured BrowseX web browser (*http://browsex.com/*).

The htmllib.tcl package is useful as a lightweight, portable, and easily configured HTML display widget for applications like application help, runtime documentation, etc.

13.5 BOTTOM LINE

- The text widget will display formatted text.
- By default, the text widget allows the user to edit text.
- The text widget supports:
  - Multiple fonts
  - Multiple colors for foreground and background
  - Varying margins
  - Varying line spacing
  - Binding actions to characters or strings
  - Including images and other Tk widgets in text
- Locations in the text widget are identified as line and character positions. Only locations where text exists can be accessed.
- Lines that are longer than the display can wrap. If this occurs, the line and character index points reflect the internal representation of the data, not the display.
- A text widget is created with the text command.
  Syntax: `text` *`textName ?options?`*
- Text is inserted into a text widget with the insert subcommand.

Syntax: *textName* insert *index text ?tagList? ?moreText? ?moreTags? ?...?*

- An image is inserted into a text widget with the > *textName* image create subcommand.
  Syntax: *textName* image create *index ?options?*
- A Tk widget can be inserted into a text widget with the *textName* window create subcommand.
  Syntax: *textName* window create *index ?options?*
- Text is deleted from a text widget with the delete subcommand.
  Syntax: *textName* delete *startIndex ?endIndex?*
- You can search for patterns in a text widget with the search subcommand.
  Syntax: *textName* search *?options? pattern startIndex ?endIndex?*
- A mark is placed with the mark set subcommand.
  Syntax: *textName* mark set *markName index*
- A mark is removed with the mark unset subcommand.
  Syntax: *textName* mark unset *markName ?...? ?markNameN?*
- You can get a list of marks with the mark names subcommand.
  Syntax: *textName* mark names
- You can iterate through the marks in a text widget with the mark next and mark previous subcommands.
  Syntax: *textName* mark next *index*
  Syntax: *textName* mark previous *index*
- You can add tags either in the insert subcommand or with the tag add subcommand.
  Syntax: *textName* tag add *tagName startIndex1 ?end1? ?start2? ... ?endN?*
- You can remove a tag from a location with the tag remove subcommand.
  Syntax: *textName* tag remove tagName *startIndex1 ?end1? ?start2? ... ?endN?*
- You can remove all references to a tag with the tag delete subcommand.
  Syntax: *textName* tag delete *tagName ?tagname2? ... ?tagNameN?*
- You can get a list of tag names with the tag names subcommand.
  Syntax: *textName* tag names *index*
- You can iterate through tag locations with the tag nextrange and tag prevrange subcommands.
  Syntax: *textName* tag nextrange *tagName startIndex ?endIndex?*
  Syntax: *textName* tag prevrange *tagName startIndex ?endIndex?*
- You can bind an event to an action that occurs on a character or string with the tag bind subcommand.
  Syntax: *textName* tag bind *tagName ?eventType? ?script?*
- The htmllib.tcl library provides a set of procedures that will render HTML into a text widget. Images are inserted into the HTML document with the HMset_image and associated HMgot_image procedures.
  Syntax: HMset_image *win handle src*
  Syntax: HMgot_image *handle image*
- Hypertext links are inserted with a user-supplied HMlink_callback procedure.
  Syntax: HMlink_callback *win href*

13.6 PROBLEMS

The following numbering convention is used in all Problem sections:

Number Range	Description of Problems
100–199	Short comprehension problems review material covered in the chapter. They can be answered in a few words or a l–5 line script. These problems should each take under a minute to answer.
200–299	These quick exercises require some thought or information beyond that covered in the chapter. They may require reading a man page or making a web search. A short script of 1–50 lines should fulfill the exercises, which may take 10–20 minutes each to complete.
300–399	Long exercises may require reading other material or writing a few hundred lines of code. These exercises may take several hours to complete.

- *100* What is the top line in a text widget?
- *101* What index string would match the start of a word that includes the index `2.22`?
- *102* What units are used to define the height and width of a text widget?
- *103* What command will return a list of all marks in a text widget?
- *104* Can a mark exist in multiple locations in a `text` widget?
- *105* Can a tag exist in multiple locations in a `text` widget?
- *106* What command would display characters tagged `bold` using the font {`times 16 bold`}?
- *107* What Tcl script library can be loaded to render HTML data?
- *108* What command would cause the procedure `highlightText` to be invoked when a user clicks on a word in the `text` widget `.t`?
- *109* What marks are always present in a `text` widget?
- *110* What tags are always present in a `text` widget?
- *200* Create a `text` widget and insert the text words *Hello, World* into the widget.
- *201* Create a `text` widget and vertical scrollbar. Add 100 lines of text into the `text` widget and confirm that the scrollbar will display them all.
- *202* Write a procedure `proc insertText {txtWin text}` that will accept the name of a `text` widget and a string of text. The procedure should tag the first character of each sentence as "red", and insert the text into the `text` widget. Use `tag configure` to make the first letter of each sentence appear red.
- *203* Write a procedure that will report all tags that overlap some section of text.

- *204* Write a script using the `text` widget (not `canvas`) that shows an image in the center of the `text` widget, and text surrounding the image. There should be multiple characters displayed on each side of the image, and multiple lines above and below the image.
- *205* Write a procedure `proc highlightText {textName string}` that will search a `text` widget for all occurrences of `string` and highlight those occurrences.
- *300* Write a set of scripts with a label and `text` widget. The label should display the word currently under the mouse cursor as the mouse is moved about a `text` widget.
- *301* Write a procedure that will accept the name of a `text` widget as an argument and modify the text in that widget to show what areas are tagged, and what the tag is. This may require adding line feeds to the text to create blank lines to hold annotation.
- *302* Expand the example in section 13.4.3 by adding a procedure `insertText helpTextName text` that will check each word in the text string and underline all words for which help has been added.
- *303* Write an editor with a `text` widget, and a `File` menu that includes `Load` and `Save` options. The load and save commands can be implemented with a `toplevel` widget that contains an entry widget for the file name and a `Done` and `Cancel` button.
- *304* Write a multiple-choice test program that displays a set of questions, via a set of `radio button` widgets, for the user to select the answer.

CHAPTER

Tk Megawidgets

14

The widgets described in Chapter 11 are relatively small and simple objects created to serve a single purpose. They are designed to be combined into larger and more complex user interfaces in your application.

Many applications require more complex widgets. A program that allows a user to save or read data from a disk file will need a file selection widget that supports browsing. Many applications require a `listbox` or `text` widget that displays a scrollbar when the number of lines displayed exceeds the widget height.

Tk supports merging several widgets into a larger widget that can be reused in other applications. These compound widget interfaces are commonly called megawidgets.

The standard distribution of the Tk toolkit contains a few megawidgets, and many more megawidgets are supported in Tk extensions and packages, including `iwidgets` (<http://sourceforge.net/projects/incrtcl/>), and `Bwidgets` (<http://sourceforge.net/projects/tcllib/>).

Megawidget packages can be created using only Tk or with C language extensions to `wish`. This chapter will discuss the megawidgets included with the standard distribution and techniques for building script-level megawidget libraries using namespaces or TclOO.

14.1 STANDARD DIALOG WIDGETS

The standard distribution provides several megawidgets to simplify writing applications. These widgets extend the functionality of the Tk interpreter and save the script writer from having to reinvent some wheels.

The implementation of these widgets varies among the platforms Tcl supports. On the UNIX platform, they are all implemented as `.tcl` scripts in the *TK_LIBRARY* directory. The Windows and Macintosh platforms already support some of these widgets, in which case Tk uses the native implementation.

If you prefer to have your application use the Motif-style widgets instead of the native widgets (perhaps it is important to have the application appear the same across multiple platforms, rather than adhere to the platform look and feel), you can force the Motif look and feel by performing the following.

Tcl/Tk: A Developer's Guide. DOI: 10.1016/B978-0-12-384717-1.00014-2

1. Set the global variable `tk_strictMotif` to 1.
2. Delete the compound widget commands you wish to have appear as Motif.
3. Source `tk.tcl`.

A code snippet to change the color selector will resemble the following.

```
set tk_strictMotif 1
rename tk_chooseColor
source [file join $tk_library tk.tcl]
```

14.1.1 tk_optionMenu

This `menubutton` widget displays the text of the selected item on the button. The widget creation command returns the name of the menu associated with the button name provided by the script.

Syntax: `tk_optionMenu` *`buttonName varName val1 ?val2...valN?`*

Create a menu from which to select an item.

`buttonName`	A name for the button to be created. This name must not belong to an existing widget. Once the `tk_optionMenu` call has returned, this name can be used to display the widget.
`varName`	The text variable to be associated with the menu. The selected value will be saved in this variable.
`val*`	The values a user can select from on this menu. These values will become the elements in the menu, and the selected value will be displayed on the button face.

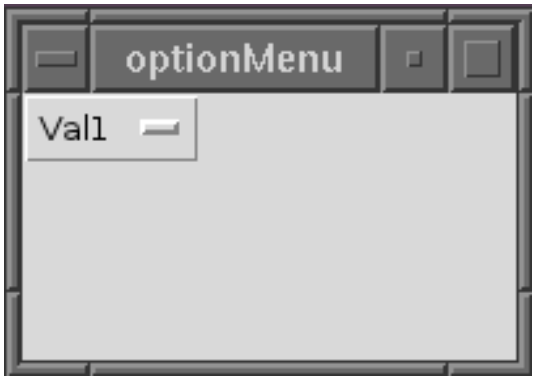
Example 1

Script Example

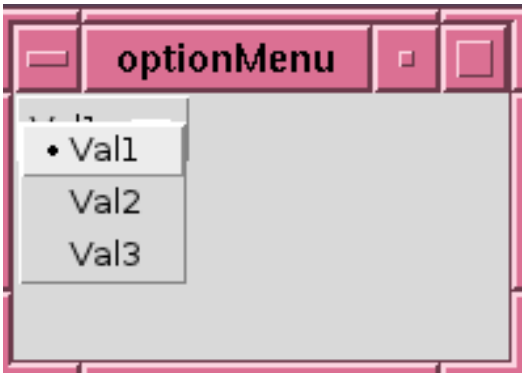
```
tk_optionMenu .button varName Val1 Val2 Val3
grid .button
```

Script Output

Before selection



After selection



14.1.2 tk_chooseColor

This color selector widget returns a properly formatted string that can be used for a color name. The string is the red-blue-green value displayed in the Selection field. This command creates a new window in the center of the screen and disables the other windows in the task that invoked it until interaction with this widget is complete.

Syntax: `tk_chooseColor`

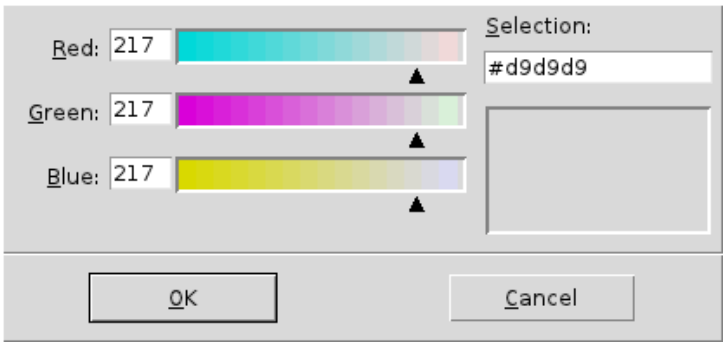
Example 2

Script Example

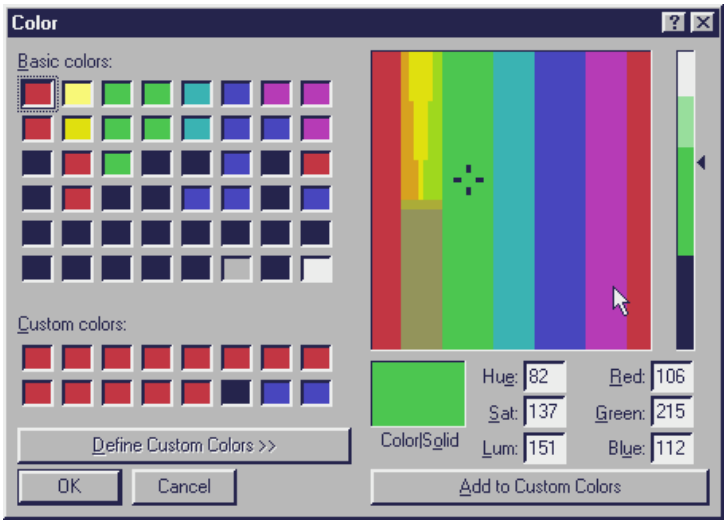
```
set newColor [tk_chooseColor]
```

Script Output

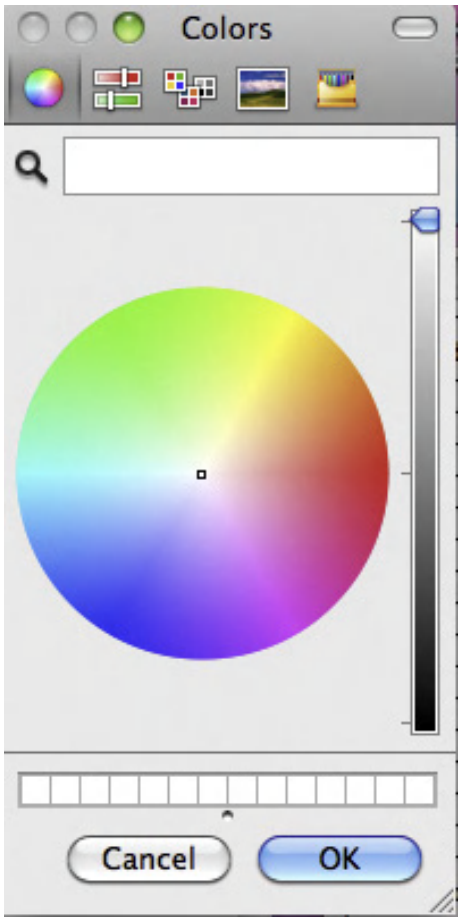
X Windows



MS-Windows



OS/X



14.1.3 tk_getOpenFile

This `file browser` widget returns the name of a file that already exists. If the user types in a nonexistent file name, an error message dialog box is displayed, and the window focus is returned to the `tk_getOpenFile` window. This command creates a new window on the screen and disables the other widgets in the task that invoked it until interaction with this widget is complete.

Syntax: `tk_getOpenFile` *?option value?*

Create a file-browsing widget to find an existing file.

?option value?

The options supported by the `tk_getOpenFile` megawidget include:

`-defaultextension` *ext* The *extension* string will be appended to a file name entered by users if they do not provide an extension. The default is an empty string. This option is ignored on the Macintosh.

`-filetypes` *patternList*	This list is used to create menu entries in the *Files of type:* menubutton. If file types are not supported by the platform evaluating the script, this option is ignored.
`-initialdir` *path*	The initial directory for the directory choice menubutton. Note that on the Macintosh the General Controls panel may be set to override application defaults, which will cause this option to be ignored.
`-initialfile` *fileName*	Specifies a default file name to appear in the selection window.
`-parent` *windowName*	Specifies the parent window for this widget. The widget will attempt to be placed over its parent but may be placed elsewhere by the window manager.
`-title` *titleString*	The title for this window. The window may be displayed without a title bar by some window managers.

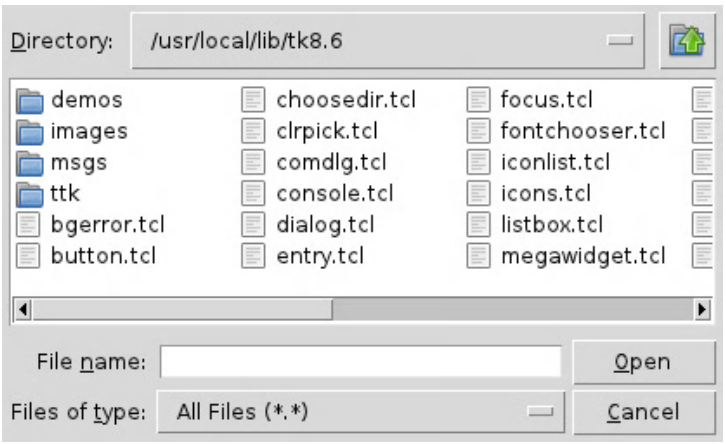
Example 3

Script Example

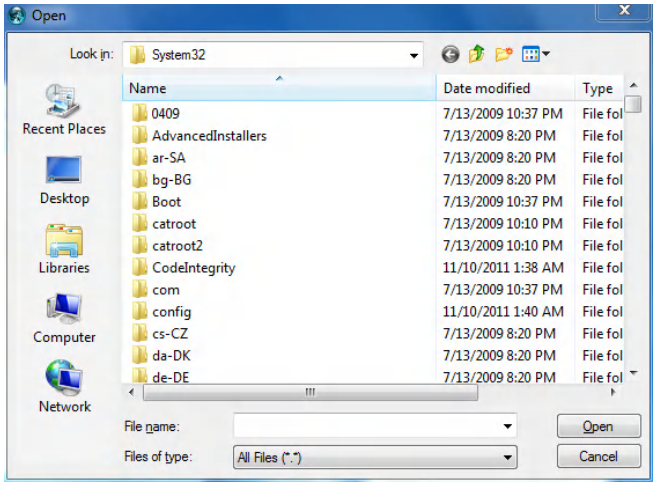
```
set typeList {
  { {Include Files} {.h} }
  { {Object Files}  {.o} }
  { {Source Files}  {.c} }
  { {All Files}     {.*} }
}
tk_getOpenFile -initialdir . -filetypes $typeList
```

Script Output

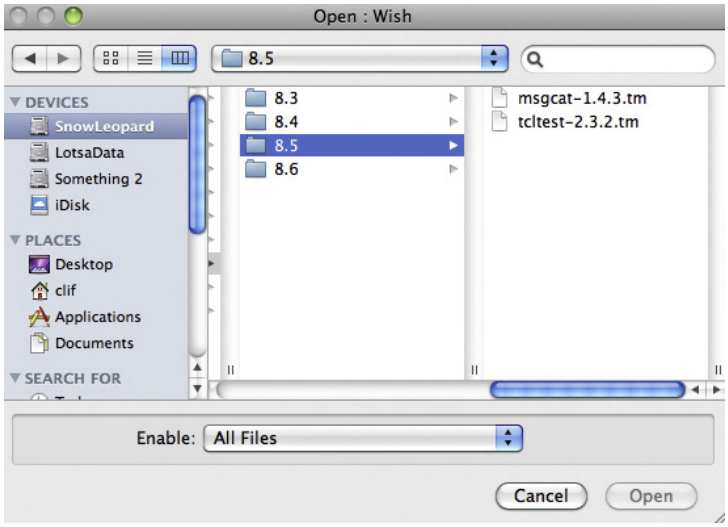
X Windows



MS-Windows



OS/X



14.1.4 tk_getSaveFile

This file-browsing widget returns the name of a selected file. If the selected file already exists, users are prompted to confirm that they are willing to overwrite the file. This widget creates a new window that is identical to the window created by `tk_getOpenFile`.

Syntax: `tk_getSaveFile` *?option value?*

Create a file-browsing widget to find an existing file or enter the name of a nonexistent file.

?option value? The tk_getSaveFile megawidget supports the same options as the tk_getOpenFile widget.

14.1.5 tk_messageBox

The tk_messageBox command creates a dialog window in one of several predefined styles. This command returns the name of the button that was clicked. This command creates a new window that takes focus until interaction with this widget is complete.

Syntax: tk_messageBox *?option value?*

Create a new window with a message and a set of buttons. When a button is clicked, the window is destroyed and the value of the button that was clicked is returned to the script that created this widget.

option value

Options for this widget include:

-message	The message to display in the box.
-title	The title to display in the window border.
-type	The type of message box to create. The type of box will determine what buttons are created. The options are:

abortretryignore	Displays three buttons: abort, retry, and ignore.
ok	Displays one button: ok.
okcancel	Displays two buttons: ok and cancel.
retrycancel	Displays two buttons: retry and cancel.
yesno	Displays two buttons: yes and no.
yesnocancel	Displays three buttons: yes, no, and cancel.

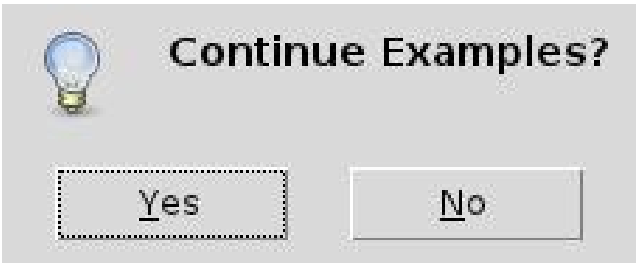
Example 4

Script Example

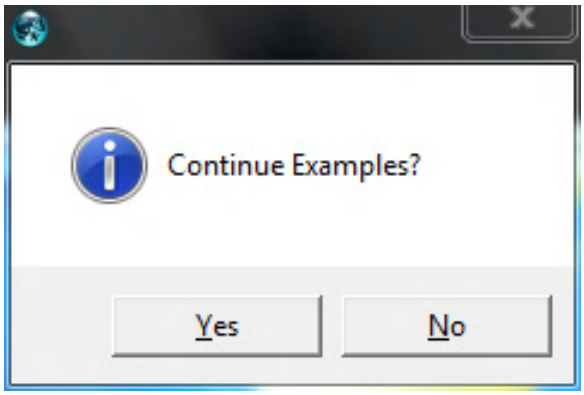
```
set clicked \
    [tk_messageBox -message "Continue Examples?" -type yesno]
```

Script Output

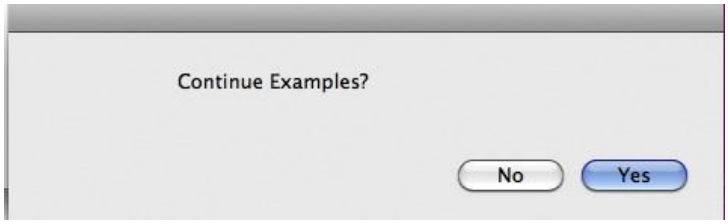
X Windows



MS-Windows



OS/X



14.1.6 tk_dialog

The `tk_dialog` megawidget creates a message window with a line of text, and one or more buttons labeled with text provided in the command line. When the user clicks a button, the button position is returned. The first button is number 0. If the user destroys the window, a –1 is returned. This command creates a new window in the center of the screen and disables the other windows in the task that invoked it until interaction with this widget is complete.

Syntax: `tk_dialog` *`win title text bitmap default string1 ?...stringN?`*

Create a dialog box and wait until the user clicks a button or destroys the box.

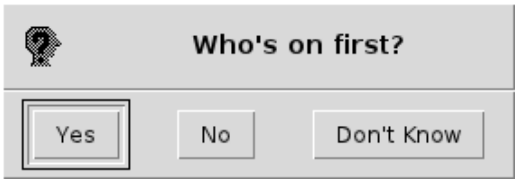
`win`	The name for this dialog box widget.
`title`	A string to place in the border of the new window. Whether or not the window is displayed with a border depends on the window manager you are using.
`text`	The text to display in the message box.
`bitmap`	If this is not an empty string, it must be a bitmap object or bitmap file name (not an image object) to display.
`default`	A numeric value that describes the default button choice for this window. The buttons are numbered from 0.
`string*`	The strings to place in the buttons.

Example 5

Script Example

```
set reply [tk_dialog .box "Example Dialog" \
    "Who's on first?"  \
    questhead 0 "Yes" "No" "Don't Know"]
```

X Windows



MS-Windows

OS/X

14.1.7 tk_popup

The tk_popup command creates a pop-up menu window at a given location on the display (not constrained within the wish window). The menu appears with a tear-off entry and takes focus. Note that this widget is not modal. The command after the tk_popup is evaluated immediately without waiting for the user to select an item from the menu. The following example shows the pop-up bound to a left button press to display a question based on the content of a text widget.

Syntax: `tk_popup` *`menu x y ?entry?`*

`tk_popup`	Create a pop-up menu somewhere on the display. This need not be within the Tcl/Tk application window.
`menu`	A previously defined `menu` object.
`x y`	The *x* and *y* coordinates for the menu in screen coordinates.
`?entry?`	A numeric value that defines a position in the menu. The defined position will be selected (active), and the menu will be placed with that entry at the x, y position requested.

Example 6

Script Example

```
proc question {} {
  global var;
  # Remove any existing .m menu.
  # Use catch in case there is no previous .m menu.
  catch {destroy .m}

  # Create a menu for this popup selector
  menu .m
  .m add command -font {helvetica 14} \
      -label {This dialog is from Abbott & Costello's \
      "Who's on First?" routine} \
      -command {tk_messageBox -type ok -message "Correct"}
  .m add command -font {helvetica 14} \
      -label "This dialog is from _War_and_Peace_" \
      -command {tk_messageBox -type ok -message "Wrong"}
  .m add command -font {helvetica 14} -label \
      "This dialog is from Monty Python's Parrot sketch" \
      -command {tk_messageBox -type ok -message "Wrong"}

# Get the size and location of this window with the "winfo"
#  command, which returns the geometry as
#  WIDTHxHEIGHT+XPOS+YPOS
# The scan command is similar to the C standard
#  library "sscanf" function.

  scan [winfo geometry .] "%dx%d+%d+%d" \
  width height xpos ypos

  # Put the popup near the bottom, right hand
  #  edge of the parent window.
  tk_popup .m [expr $xpos + ($width/2) + 50 ] \
      [expr $ypos + $height - 40] 1
}
```

```
# Create a text window, display it, and insert some text
#  from a famous dialog.

text .t -height 12 -width 75 -font {helvetica 14}
grid .t

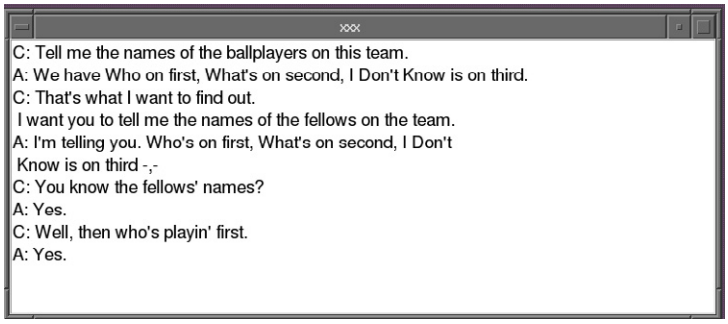
set dialog {
"C: Tell me the names of the ballplayers on this team.\n"
"A: We have Who on first, What's on second, "
"I Don't Know is on third.\n"
"C: That's what I want to find out.\n"
" I want you to tell me the names of the fellows"
" on the team.\n"
"A: I'm telling you. Who's on first, "
"What's on second, I Don't\n"
" Know is on third -\,-\n"
"C: You know the fellows' names?\n"
"A: Yes.\n"
"C: Well, then who's playin' first.\n"
"A: Yes.\n"
}

foreach line $dialog {
  .t insert end $line
}

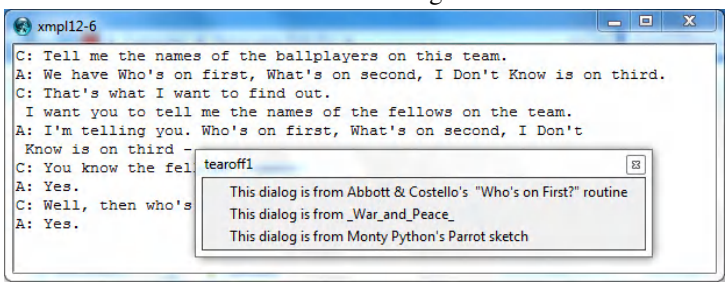
# Bind the right button to a question about this dialog.
bind .t <Button-1> question
```

Script Output

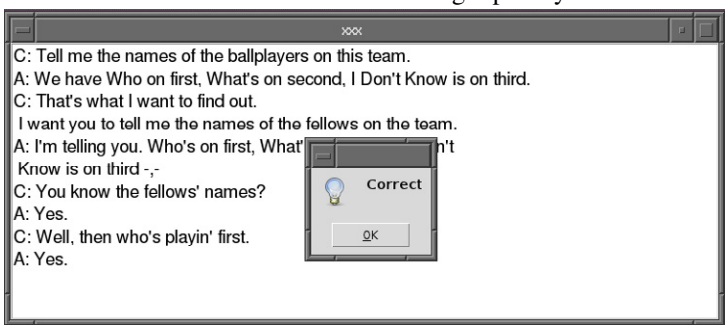
X Windows: Initial Text Widget.



X Windows: After Right Click.



X Windows: After Selecting top entry.



14.2 MEGAWIDGET BUILDING PHILOSOPHY

The megawidgets included in the standard distribution are useful, but the odds are good that you will end up needing to build your own widgets at some point. You can design a megawidget library in a number of ways. Each technique has features that may make it better or worse for your application. One of the primary trade-offs is versatility and ease of use versus speed of construction. If you expect to use the widget more than once, it is worthwhile to take the extra time to design the widget library for versatility. Consider the points discussed in the sections that follow when building your widget libraries.

14.2.1 Display in Application Window or Main Display?

The Tk distribution includes several megawidgets that open their own windows on the primary display rather than being loaded into the application window. Some widgets are useful as temporary top-level windows, whereas others are more useful if they can be packed into an application window.

A fallback position is to design a widget to be loaded in an existing window. You can create a wrapper to create a new `toplevel` window and `pack` the widget into that window when you need a pop-up.

14.2.2 Modal versus Modeless Operation

Most of the megawidgets in the standard Tk distribution operate in a modal style, freezing any other activities until the user has completed an interaction with this widget. This is appropriate behavior in some circumstances but not in others. Consider the applications in which your megawidget will be used to determine whether modal or modeless operation is required, or if script writers should be able to define the behavior, when they create the widget.

14.2.3 Widget Access Conventions

You can design a megawidget to follow its own conventions, or it can mimic the behavior of the standard Tk widgets. The widget creation command can return a handle that is used to identify this megawidget to procedures that manipulate it. Code using this technique would resemble the following.

```
set megaHandle [megaCreate -option value]
myMegaConfigureProc $megaHandle -newOption -newValue
```

The examples in this chapter demonstrate a better technique. They construct megawidgets that return the name of a procedure that accesses the megawidget. This technique mimics the behavior of the standard widgets and is implemented with the object-style technique introduced in Chapters 7 – 10. Code using this technique would resemble the following.

```
set mega [newMegaWidget -option value]
$mega configure -newOption newValue
```

14.2.4 Widget Frames

Most megawidgets have a parent frame that holds their subwidgets. This frame can be created by the megawidget creation command or provided by the calling procedure. If the frame is created by the widget creation command, calling script can provide the name (as names are given to Tk widgets) or the megawidget can generate a unique name.

If you provide the parent frame for the megawidget (instead of just providing a name), you can have the megawidget display itself in that frame automatically. This is not a generally good technique. It restricts the usefulness of the widget to applications that have similar display requirements. This technique is useful for widgets such as toolbar buttons that are always displayed side by side in a frame.

14.2.5 Configuration

The next person who uses your megawidget library is guaranteed not to like the configuration options you chose. For instance, the dialog box that informs users that the warp drives are about to explode and asks whether they would like do something about it should probably be in a bright color with big letters, rather than the usual dull gray with small letters. Configuration options can be:

- Set in a state variable and applied when the widget is created
- Set on the widget creation command line
- Set in a `configure` procedure associated with the widget
- Set with the `option add` command (discussed in Section 14.3.2) if the megawidget must use the `-class` option to define itself as a new class of widget

Supporting the first two and fourth options of the previous list is the most versatile design. This provides features that mimic the options supported by the normal Tk widgets.

14.2.6 Access to Subwidgets

A megawidget can be designed in an opaque manner, so that the component widgets cannot be accessed, or in a transparent manner so that the component widgets can be accessed and modified. If the goal of this megawidget is to provide a uniform interface across all applications that use it, the widget should be designed to restrict programmers from modifying the options of the subwidgets. The message box and file browsers provided by the OS platform are examples of this style of widget.

If the goal for this megawidget is to provide a tool for a wide variety of applications, the megawidget should be designed to allow the application script writer access to the subwidgets. This allows the application writer to modify options such as background color and font. This can be handled by the techniques described in the following.

Naming Convention

The megawidget creation command returns a base name for the megawidget. The subwidgets are accessed as `base.labelName`, `base.buttonOK`, and so on. This is the simplest technique to implement, although it may be difficult to use if you nest megawidgets within other megawidgets.

This technique's disadvantage is that it requires application-level code to know about the internals of the megawidget. This can make maintenance a headache.

Code using this technique might resemble:

```
set w [myMegaWidget .meg -option value]
$w.topLabel configure -font $bigFont
```

A Command to Return Subwidget Names

The megawidget package you design can include a `subwidget` command that will return the full name of a subwidget component of the megawidget. Code that uses this technique would resemble the following.

```
set button [$myMegaWidget subwidget buttonOK]
$button configure -text "A-OK"
```

Pass Commands to the Subwidget

The widget procedure can accept a list of subwidgets and command strings to be evaluated by the subwidget. Code that uses this technique would resemble the following.

```
set myBigWidget [megawidget .bigwidget -option value]
$myBigWidget widgetcmd mainTitle configure -text "Big Widget"
```

`Subwidget` Option

The widget procedures you create can accept (or require) a subwidget as an argument when they are invoked. Code that uses this technique would resemble the following.

```
$myMegawidget configure -subwidget buttonOK -text "A-OK"
```

14.2.7 Following Tk Conventions

The closer you follow the Tk conventions for names, options, and subcommands, the easier it is for script writers to use your megawidgets. If there are two equally good methods for creating the functionality you need, choose the one that mimics Tk.

14.3 FUNCTIONALITY THAT MAKES MEGAWIDGETS POSSIBLE

Tcl/Tk provides three sets of functionality that make it easy to build megawidgets with pure Tcl. All of these pieces have other, perhaps more important, uses, but they work together synergistically when used to construct megawidgets. The three features are the `rename` command, the `option` command, and the `-class` option.

14.3.1 The `rename` Command

The `rename` command was discussed briefly in Chapter 7.

We can also use the `rename` command to rename the command associated with a widget. For instance, in the next code snippet, a label named `.l` is created with the command `label .l -text "original"`. This creates both a widget named `.l` and a command named `.l`. The `rename` command renames the procedure `.l` to `mylabel`. The widget is still named `.l`, and that name must be used to `pack` the widget. However, the procedure is now named `mylabel`, and that is the name we must use to configure the widget.

```
label .l -text "original"
rename .l mylabel
pack .l
mylabel configure -text "new text"
```

When creating megawidgets, we will first create a frame to hold the widget components, and then rename that frame's procedure to something new, so that we can define a new command with the original name of the frame as the widget command. Stripping it down below the bare essentials, the following shows what happens.

Example 7

Script Example

```
# Create a frame to hold the subwidgets.
frame .megawidget
# ... Build and display the subwidgets
# Rename the frame procedure to a new name.
rename .megawidget megaWidgetFrame
# Create a new procedure named for the original frame.
proc .megawidget {...} {...}
```

14.3.2 The `option` Command

The `option` command lets the Tcl programmer interact with the Tk option database. The option database is a collection of patterns and values. When the `wish` interpreter creates a new window, it examines the options database to see if this window has any configuration options that match a pattern in the database. The value from the database is applied to the widget if an option is matched. You can set a value in the option database with the `option add` command.

Syntax: `option add` *`pattern value ?priority?`*

Add a definition to the options database.

`pattern` The pattern describing the widget option to set.

`value` A value to use when a widget that matches the pattern is created.

The form of a pattern is *`application.widget.optionName`*. The pattern `tkcon.mybutton.background` would match the background color of a button named `.mybutton` in an application named `tkcon`. The basic pattern supports a few variations to make it more versatile.

- Any field can be replaced by a wildcard (`*`). Thus, `*mybutton.background` would match the background of any button named `.mybutton`.
- Fields can be replaced by a generic name. The generic name will start with an uppercase letter. The generic name for a widget field is the name of the class of the object. For example,

  `*Button.background` will match the background of any button created in an application.

The Tcl interpreter will select options from the most specific pattern that matches a widget. Thus, using a widget name will override a generic widget option, as shown in the following example.

Example 8

Script Example

```
# All labels have white background
option add *Label*background white
# All widgets named alert have black background
#   and white foreground
option add *alert*background black
option add *alert*foreground white
pack [label .l -text "normal colors"]
pack [label .alert -text "ALERT Inverted"]
```

Script Output

The `option add` command can be used in any application when you want to change the default value for a set of widgets. For example, if you want to make large buttons you might use the `-font` option for each `button` command, or you could set the `*Button*Font` option to a larger font. One advantage of using the `option` command is that it is faster than using *`-option`* in a widget creation

command. In an application with many widgets, this can noticeably affect how quickly windows are built.

14.3.3 The `-class` Option

The Tcl interpreter can assign a `-class` option to a new `frame` or `toplevel` widget. The default class for a frame is `Frame`, and the default class for a top-level window is `Toplevel`. (This information is returned by the `winfo class` *`widgetName`* command.)

If we include the `-class` option with the `frame` or `toplevel` command, our script can override this to become a new class. We can set options for all members of this new class with the `option add` command.

Example 9

Script Example

```
# A simple Label/Entry megawidget
proc LabelEntry {frameName labelText varName} {
  frame $frameName -class Labelentry
  pack [label $frameName.l -text $labelText] -side left
  pack [entry $frameName.e -textvar $varName] -side left
  return $frameName
}

# Labelentry widgets all use large font
option add *Labelentry*font {Times 24 bold}
pack [LabelEntry .le "Enter your Name" name]
```

Script Output

14.4 BUILDING A MEGAWIDGET

The simple `LabelEntry` megawidget shown in the previous example makes it easier to develop a form-style application with many items to be filled out. Using this type of simple megawidget can make your applications easier to write and maintain. However, if you are developing a library of complex megawidgets for multiple projects, you will want them to be more flexible.

The first step in designing a library of megawidgets is deciding what sets of conventions you will follow. The actual conventions do not matter as much as consistently following the conventions. Consistency is one of the keys to creating a usable set of tools. This section describes building a modeless megawidget in a frame. This widget will behave like one of the basic Tk widgets. The conventions we will establish for this widget are as follows.

- The subwidget will create a new frame of class *Megawidget*.
- A configuration option can be defined for the entire megawidget and it will be propagated to all appropriate subwidgets using the `option` command.
- A subwidget is identified by a unique identifier within the megawidget. For example, `.parentwindow.megawidget.subwidget` will be identified as `subwidget`.
- A list of subwidget names will be returned by a `names` command, similar to `array names`.
- Subwidgets may also be accessed by full name. For example, `.parentwindow.megawidget.subwidget` may be identified as `.parentwindow.megawidget.subwidget`.
- A configuration option can be defined for a subwidget with a `widgetconfigure` command that duplicates the behavior of the `configure` command.
- A configuration option can be queried for a subwidget with a `widgetcget` command that duplicates the behavior of the `cget` command.
- A command can be passed to a subwidget with the `widgetcommand` command.
- Infrastructure data and procedures will be maintained in a namespace.

14.5 A SCROLLING `listbox` MEGAWIDGET

This section shows how a megawidget can be built using the conventions described in Section 14.4. The first example creates a simple scrolling `listbox` megawidget. The next example enhances this widget into a more complex megawidget that can take selected items from one scrolled listbox and place them in another. The final example puts a wrapper around that megawidget to make a modal file selector.

14.5.1 Scrolled listbox Description

The scrollable `listbox` megawidget we will construct will mimic the structure of the Tk widgets. The support procedures for this megawidget exist within a namespace. A single procedure (`scrolledLB`) exists in the global scope to create a `scrolledLB` megawidget.

The `scrolledLB` procedure will return the name of the new widget's parent frame and will create a procedure with that name in the global scope. This procedure will support several subcommands in the same manner as the Tk widgets. The `scrolledLB` configuration options can be set by:

- Using the `option add` command before constructing a `scrolledLB` widget.
- Using new options defined for this megawidget.
- Using the `widgetconfigure` widget subcommand after constructing a `scrolledLB` widget.
- Retrieving the full path of a subwidget with the `subwidget` command and using that widget's `configure` subcommand.

The `scrolledLB` procedure accepts any configuration values that are valid for a frame or that match specific options for this megawidget, and uses those values to configure the frame or the `scrollbar` and `listbox` subwidgets. In addition, the `scrolledLB` command accepts an option to define whether the scrollbar should be on the left or right side of the listbox.

Syntax: `scrolledLB` *`?frameName? ?-option value?`*

Create a `scrolledLB` megawidget. Returns the name of the parent frame for this widget for use with the `pack`, `grid`, or `place` layout managers and creates a procedure with the same name to be used to interact with the megawidget.

?frameName?	An optional name for this widget. If this argument is not present, the widget will be created as a unique child of the root frame, ".".
?-option value?	Sets of configuration values that can be applied to the scrollbar or listbox.

The widget procedure created by the `scrolledLB` command will accept the subcommands `configure`, `widgetconfigure`, `widgetcget`, `widgetcommand`, `names`, `insert`, `delete`, `selection`, and `subwidget`.

The `scrolledLB` widget `widgetconfigure` and `widgetcget` subcommands are similar to the standard Tk `configure` and `cget` commands except that they accept a subwidget argument. The configuration options will be applied only to those subwidgets.

Syntax: *scrolledLBName* `widgetconfigure` *subwidgetName ?-opt? ?val?*

Return the value of an option, or set an option if a value is supplied.

subwidgetName	The name of a subwidget or subwidgets to be configured or queried. If the widget is being queried and multiple subwidget names are present, the return value will be a list of configurations in the order in which the subwidgets appear in the list. NOTE: This field is different from standard widget usage.
-opt	An option name. If multiple options are being set, there may be multiple *-opt val* pairs. If the configuration is being queried, only one *-opt* argument may be present.
val	The value to assign to a configuration option.

The `insert` and `delete` subcommands pass their arguments to the `listbox` subwidget with no further parsing. Thus, they exactly duplicate the behavior of the `listbox` subcommands.

Syntax: *scrolledLBName* `insert` *index string*

Inserts an item into the `listbox` at the requested index position.

index	The location in the list before which this entry will be inserted. Any index value appropriate for a `listbox` widget may be used.
string	The text to insert into that location of the listbox.

Syntax: *scrolledLBName* `delete` *first ?last?*

Delete one or more items from the listbox.

first	The index of the first item to delete. The listbox items are numbered from 0.
?last?	The index of the last item in a range to delete. If this argument is not present, only the item identified by the *first* is deleted.

The `selection` subcommand returns the content of the selected entries of the listbox. Note that the `listbox curselection` subcommand returns a list of the indices of selected items, whereas this `selection` returns a list of the content.

Syntax: *scrolledLBName* selection
Returns the text of the selected item or items from the listbox.

The subwidget subcommand returns the full path name of a subwidget of the scrolledLB.

Syntax: *scrolledLBName* subwidget *name*
Returns the full name of a subwidget of this scrolledLB.
name The name of the subwidget to return. May be one of:
list The full name of the listbox widget.
scrollbar The full name of the scrollbar widget.
title The full name of the label widget.

14.5.2 Using the scrolledLB

The next example shows how to use the scrolled listbox. The widget creation command resembles that of a standard Tk widget.

Example 10
Script Example

```
# set the auto_path and declare that we require the
# scrolledLB package.
# The widget pkgIndex.tcl file is in the current directory
# Load the scrolledLB support

lappend auto_path.
package require scrolledLB

# The background of the listbox is white
# The other widgets are normal gray.
option add *Scrolledlb*Listbox*background white

# Create and pack the scrolled listbox
set lb [scrolledLB .sbox -listboxHeight 5 -listboxWidth 20 \
    -titleText "Pick your Banjo"]
grid $lb

# Fill the listbox
foreach brand { {Bacon & Day} Epiphone Gibson \
    Ludwig Paramount Orpheum Vega Weymann} {
  $lb insert end $brand
}

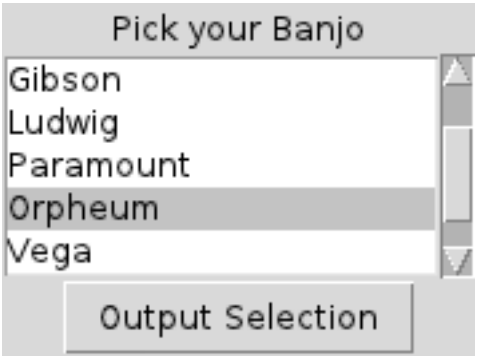
# Add a button to display the selection
set cmd [format {puts "Picking an [%s selection]"} $lb]
```

```
button .b -text "Output Selection" -command $cmd

grid .b
```

Script Output

Picking an Orpheum



14.5.3 Implementing the Scrollable listbox

The scrolledLB widget is implemented using a common pattern for widgets and data structures that mimic objects. There is a single global scope procedure to create the scrolledLB megawidget, and a set of support procedures for the scrolledLB in a private namespace. Creating a megawidget also creates a new procedure in the global scope with the same name as the parent frame. The new procedure invokes support procedures in the widget namespace scope. The scrolledLB megawidget is implemented with state variable and three procedures.

The state variable, scrolledLBState, exists within the scrolledLB namespace. This variable holds the information that needs to be maintained between evaluations of the scripts in this package. This information includes a number used to create unique names for the scrolledLB widgets when no name is provided in the creation command line, and a lookup table to map simple subwidget names such as list to the full widget path, such as .fileFrame.sbox.list. The scrolledLB procedure that creates a scrolledLB widget exists in the global scope.

The scrolledLB procedure calls the MakescrolledLB procedure in the widget namespace. The MakescrolledLB procedure creates the new megawidget and returns the widget name to the scrolledLB procedure. The scrolledLB procedure uses that name to create a new procedure in the global scope that will interact with the new megawidget. The new procedure interacts with the widget by invoking the *widgetName*Proc procedure that exists in the widget namespace.

The MakescrolledLB procedure creates a frame to hold the subwidgets that constitute this megawidget. That frame should have the same name as the megawidget. This lets us access the subwidgets as children of the parent megawidget.

However, there is a name conflict when you use the same name for both the megawidget and the frame that contains the megawidget components. The trick is that we want the megawidget to look

like a Tk object, but the megawidgets in these examples are frames holding some subwidgets and a procedure to access the subwidgets.

For instance, if you construct a megawidget named `.foo` and create the subwidgets within a frame named `.foo`, you cannot use `.foo` as the procedure name for the megawidget commands. The command name `.foo` is already in use by the frame `.foo`, and Tcl command and procedure names may not conflict. We solve the problem by renaming the procedure associated with the holder frame to a new name, and then writing a new procedure with the original name to interact with the megawidget.

This creates a naming convention that makes the frame and subwidgets appear as children to the megawidget. This is the way the megawidget and subwidgets are related to each other logically, although it is not the way the megawidget is implemented. The `MakescrolledLB` procedure creates a frame to hold the subwidgets and then renames the widget procedure associated with that frame before returning the frame name to the `scrolledLB` procedure. The `scrolledLB` procedure is evaluated in the global scope, and when it creates a new procedure with the name of the holder frame that procedure is created in the global scope.

For example, if your script invokes `scrolledLB` to create a `scrolledLB` named `.sl1`, the `MakescrolledLB` command will create a frame named `.sl1`. After the scrollbar and listbox have been packed into the frame `.sl1`, it will rename the frame widget command from `.sl1` to `.sl1.fr`. Since there is no longer a procedure named `.sl1`, the `scrolledLB` procedure can create a new procedure named `.sl1` that will invoke `scrolledLBProc` to process the megawidget commands.

We can do this because procedure names and window names are resolved from separate tables. When you create a frame `.foo`, the Tk interpreter creates a window named `.foo` and a procedure named `.foo` to access that window. You can rename the procedure `.foo` without affecting the window `.foo`.

Being able to disconnect the frame `.sl1` from the procedure `.sl1` allows us to duplicate the Tk convention of declaring a name for the megawidget and then using that name as the widget procedure. If we did not rename the frame widget procedure there would be no way to interact with the frame after we create a new procedure with that name.

After `MakescrolledLB` has created a frame to hold the subwidgets, it returns the name of the parent frame to `scrolledLB`, which creates a procedure in the global scope with the same name as the frame. This new procedure is the global portion of the second pair of procedures. Script writers will use this procedure when they need to interact with the megawidget. This procedure simply invokes `scrolledLBProc`, a procedure in the `scrolledLB` namespace to actually implement the widget commands.

Global Scope	`scrolledLB` **Namespace**
`scrolledLBName`	`scrolledLBProc`
Accepts the `scrolledLB` widget subcommands. Invokes `scrolledLBProc` to process the command with the name of the parent frame.	Performs the widget interactions requested by the widget subcommands.

These procedures are all that is needed to create a scrolled `listbox` megawidget. The actual implementation is broken up into few more procedures within the `scrolledLB` namespace just to make maintenance simpler.

14.5.4 The scrolledLB Code

The actual implementation of this code is broken up into a couple more procedures. Draw creates and maps the subwidgets into the parent frame, and DoWidgetCommand implements the widgetcommand subcommand.

Example 11

Script Example

```
# Name: Scrolled LB. tcl

package provide scrolledLB 1.0

##########################################################################
# proc scrolledLB {args}
#  Create a scrolledLB megawidget
#  External entry point for creating a megawidget.
# Arguments
#  ?parentFrame?   A frame to use for the parent,
#                  If this is not provided, the mega widget
#                  is a child of the root frame (".").
#
#  args            A set of key/value pairs to describe the listbox
#
# Results
#  Creates a procedure with the megawidget name for processing
#  widget commands
#  Returns the name of the megawidget

proc scrolledLB {args} {
  set newWidget [eval scrolledLB::MakescrolledLB $args]

  # Define a new command which passes control to the
  #  scrolledLB::scrolledLBProc command and return the results.

  set newCmd [format {return [namespace eval %s %s %s $args ]}\
      scrolledLB scrolledLBProc $newWidget]

  proc $newWidget {args} $newCmd

  return $newWidget
}

namespace eval scrolledLB {
  variable scrolledLBState

  # Assign a couple defaults
  set scrolledLBState(unique) 0
  set scrolledLBState(debug) 0
```

```
##########################################################################
# proc MakescrolledLB {args}
#  Create a scrolledLB megawidget
# Arguments
#  ?parentFrame?  A frame to use for the parent.
#     If this is not provided the megawidget
#     is a child of the root frame (".").
#  ?-scrollSide left/right?
#     The side of the listbox for the scrollbar
# Results
#  Returns the name of the parent frame for use as a
#  megawidget
proc MakescrolledLB {args} {
  variable scrolledLBState

  # Set the default name
  set holder .scrolledLB_$scrolledLBState(unique)
  incr scrolledLBState(unique)

  # If the first argument is a window path, use that as
  # the base name for this widget.
  if {[string first "." [lindex $args 0]] == 0} {
    set holder [lindex $args 0]
    set args [lreplace $args 0 0]
  }

  # Set Command line option defaults here
  #  height and width are freebies
  set scrolledLBState($holder.height) 5
  set scrolledLBState($holder.width) 20
  set scrolledLBState($holder.scrollSide) 1
  set scrolledLBState($holder.listboxHeight) 10
  set scrolledLBState($holder.listboxWidth) 20
  set scrolledLBState($holder.listSide) 0
  set scrolledLBState($holder.titleText) title

  foreach {key val } $args {
    set keyName [string range $key 1 end]
    if {![info exists \
        scrolledLBState($holder.$keyName)]} {
      regsub -all "$holder." \
          [array names scrolledLBState $holder.*] \
          "" okList
      error "Bad option" \
          "Invalid option '$key'.\n\
          Must be one of $okList"
```

```
    }
    set scrolledLBState($holder.$keyName) $val
  }

  # Create master Frame
  frame $holder -height $scrolledLBState($holder.height) \
      -width $scrolledLBState($holder.width) \
      -class ScrolledLB

  # Apply invocation options to the master frame, as appropriate
  foreach {opt val} $args {
    catch {$holder configure $opt $val}
  }

  Draw $holder

  # We can't have two widgets with the same name.
  #  Rename the base frame procedure for this
  #  widget to $holder.fr so that we can use
  #  $holder as the widget procedure name
  #  for the megawidget.
  uplevel #0 rename $holder $holder.fr

  # When this window is destroyed,
  #  destroy the associated command.
  bind $holder <Destroy> "+ rename $holder {}"
  return $holder
}

######################################################################
# proc Draw {parent}--
#  creates the subwidgets and maps them into the parent
# Arguments
#  parent   The parent frame for these widgets
#
# Results
#  New windows are created and mapped.
proc Draw {parent} {
  variable scrolledLBState

  set tmp [scrollbar $parent.yscroll  -orient vertical \
      -command "$parent.list yview" ]

  set scrolledLBState($parent.subWidgetName.yscroll) $tmp
  grid $tmp  -row 1 -sticky ns \
      -column $scrolledLBState($parent.scrollSide)
```

```
  set tmp [listbox $parent.list  \
      -yscrollcommand "$parent.yscroll set" \
      -height $scrolledLBState($parent.listboxHeight)\
      -width $scrolledLBState($parent.listboxWidth) ]

  set scrolledLBState($parent.subWidgetName.list) $tmp
  grid $tmp  -row 1 -column $scrolledLBState($parent.listSide)

  set tmp [label $parent.title  \
      -text $scrolledLBState($parent.titleText) ]

  set scrolledLBState($parent.subWidgetName.title) $tmp
  grid $tmp  -row 0 -column 0 -columnspan 2
}

#########################################################################
# proc DoWidgetCommand {widgetName widgets cmd}--
#  Perform operations on subwidgets
# Arguments
#  widgetName: The name of the holder frame
#  widgets:    A list of the public names for subwidgets
#  cmd:        A command to evaluate on each of these widgets
#
# Results
#  Does stuff about the subwidgets

proc DoWidgetCommand {widgetName widgets cmd} {
  variable scrolledLBState
  foreach widget $widgets {
    set index $widgetName.subWidgetName.$widget
    eval $scrolledLBState($index) $cmd
  }
}

proc selection {widgetName} {
  set lst [$widgetName.list curselection]
  set itemlst ""
  foreach l $lst {
     lappend itemlst [$widgetName.list get $l]
  }
  return $itemlst
}

#########################################################################
# proc scrolledLBProc {widgetName subCommand args}
#  The master procedure for handling this megawidget's
#  subcommands
```

```
# Arguments
#  widgetName:  The name of the scrolledLB
#    widget
#  subCommand     The subCommand for this cmd
#  args:     The rest of the command line
#    arguments
# Default subCommands are:
# configure - configure the parent frame
# widgetconfigure - configure or query a subwidget.
#    mega widgetconfigure itemID -key value
#    mega widgetconfigure itemID -key
# widgetcget - get the configuration of a widget
#    mega widgetcget itemID -key
# widgetcommand - perform a command on a widget
#    mega widgetcommand itemID commandString
# names - return the name or names that match
#    a pattern
#    mega names  # Get names of all widgets
#    mega names *a* # Get names of widgets
#    with an 'a' in them.
# subwidget - return the full pathname of a
#    requested subwidget
# Results
#  Evaluates a subcommand and returns a result if required.
proc scrolledLBProc {widgetName subCommand args} {
  variable scrolledLBState
  set cmdArgs $args

  switch -- $subCommand {
    configure {
       return [eval $widgetName.fr configure $cmdArgs]
    }

    widgetconfigure {
       set sbwid [lindex $cmdArgs 0]
       set cmd [lrange $cmdArgs 1 end]
       set index $widgetName.subWidgetName.$sbwid
       catch {eval \
           $scrolledLBState($index) configure $cmd}  rtn
       return $rtn
    }
    delete  -
    insert  {
       return [eval $widgetName.list $subCommand $cmdArgs]
    }
    selection {return [selection $widgetName]}
    widgetcget {
```

```
            if {[llength $cmdArgs] != 2} {
              error "$widgetName cget subWidgetName -option"
            }
            set sbwid [lindex $cmdArgs 0]
            set index $widgetName.subWidgetName.$sbwid
            set cmd [lrange $cmdArgs 1 end]
            catch {eval \
                $scrolledLBState($index)\
            cget $cmd}  rtn
            return $rtn
            }
         widgetcommand {
            return [eval DoWidgetCommand $widgetName $cmdArgs]
         }
         names {
            if {[string match $cmdArgs ""]} {
              set pattern $widgetName.subWidgetName.*
            } else {
              set pattern $widgetName.subWidgetName.$cmdArgs
            }
            foreach n [array names scrolledLBState $pattern] {
              foreach {d w s name} [split $n .] {}
                lappend names $name
              }
            return  $names
         }
         subwidget {
            set name [lindex $cmdArgs 0]
            set index $widgetName.subWidgetName.$name
            if {[info exists scrolledLBState($index)]} {
              return $scrolledLBState($index)
            }
         }
         default {
            error "bad command" "Invalid Command: \
                $subCommand \ n \
                must be configure, widgetconfigure, \
                widgetcget, names,\
                delete, insert, selection,  \
                or subwidget"
         }
      }
   }
}
```

14.6 NAMESPACES AND TK WIDGETS

When the command associated with a Tk widget is evaluated, it is evaluated in the global namespace even if the widget is created inside a namespace. For example, the following code will not work.

```
namespace eval badCode {
  variable clicked 0
  button .b -text "Click me" -command "set clicked 1"
}
```

When the button is clicked, it will create and assign a 1 to the variable `clicked` in the global scope, not assign a 1 to the variable `clicked` in the namespace `badCode`. There are two `namespace` subcommands that provide the hooks for using Tk widgets and namespaces. The `namespace current` command will return the current namespace, and `namespace code` *`script`* will make a script that will be evaluated in the current namespace.

Syntax: `namespace current`

Returns the absolute name of the current namespace.

The previous example can be modified to use the `namespace current` command to set the correct variable. After performing substitutions, the script registered with the button is `set ::goodCode::clicked 1`.

Example 12

```
namespace eval goodCode {
  variable clicked 0
  button .b -text "Click me" \
      -command "set [namespace current]::clicked 1"
}
```

The `namespace code` command will wrap a script with some namespace commands to force the script to be evaluated in the current namespace, instead of the namespace it's invoked from.

Syntax: `namespace code` *`script`*

Returns a script that will be evaluated in the current namespace.

`script` The script to be evaluated.

The first example can also be modified to work using the `namespace code` command. In the following example, the
`namespace code {set clicked 1}`
script is converted to
`namespace inscope ::goodCode {set clicked 1}`,
which is the script registered with the button. This is equivalent to :
`namespace eval ::goodCode {set clicked 1}`.

The `inscope` subcommand causes a script to be evaluated within an existing scope. It is usually used by the namespace procedures, not applications programmers.

```
namespace eval goodCode {
  variable clicked 0
  button .b -text "Click me" \
      -command "[namespace code {set clicked 1}]"
}
```

You will usually use `namespace current` when you need to provide a variable name to a Tk widget (for example, the `-textvariable` option), and `namespace code` with scripts associated with Tk widgets. If the script associated with a widget simply invokes a procedure within a namespace, you can use the `namespace current` command to add the namespace information to the procedure name. However, if you include Tcl commands such as `for`, `if`, `foreach`, and so on in your script, you must use `namespace code`.

Recommended procedure is to use `namespace code` for scripts associated with widgets. Since scripts associated with a widget tend to grow over time, it makes code maintenance easier.

14.6.1 Creating a Multiple Language Megawidget

Many applications need to be distributed with support for multiple languages. Tcl has features that help make this easy. Tcl uses Unicode to store text strings, which makes it easy to represent non-Latin alphabets (see Section 12.4.5 for more discussion). You can use the associative array as a translation table.

The next example is a megawidget to create forms of label/entry widget pairs in which the label can be displayed in one of several languages. The label text is converted using an associative array as a lookup table. The `Draw` procedure uses both the `namespace code` command to invoke the `worldForm::translate` procedure when the `translate` button is clicked, and the `namespace current` command to link the variable `worldFormState(language)` to the `tk_optionMenu` widget.

Example 13

Script Example

```
proc Draw {parent} {
  variable worldFormState
  button $parent.translate  \
      -text $worldFormState($parent.buttonText) \
      -command [namespace code [list translate $parent]]

  # ...
  tk_optionMenu $parent.options  \
      [namespace current]::worldFormState(language)  \
      French English German  Spanish

  # ...
}
```

The following example shows the complete code for this megawidget, and a short test example.

Example 14

Complete code

```
# name: worldForm.tcl

package provide worldForm 1.0

##########################################################################
# proc worldForm {args} -
#   Create a worldForm megawidget
#   External entry point for creating a megawidget.
# Arguments
#    ?parentFrame?  A frame to use for the parent,
#                   If this is not provided, the megawidget
#                   is a child of the root frame (".").
#
# Results
#  Creates a procedure with the megawidget name for processing
#  widget commands
#  Returns the name of the megawidget

proc worldForm {args} {
  set newWidget [eval worldForm::MakeworldForm $args]

  set newCmd [format {return [namespace eval %s %s %s $args ]}  \
      worldForm WorldFormProc $newWidget]

  proc $newWidget {args} $newCmd

  return $newWidget
}

namespace eval worldForm {
  variable worldFormState

  # Assign a couple defaults
  set worldFormState(unique) 0

  ##########################################################################
  #  proc unique {}--
  #  Return a unique number
  # Arguments
  #  None
  #
```

```
# Results
#  Modifies stateVar(unique)
#
proc unique {} {
  variable worldFormState
  return [incr worldFormState(unique)]
}

##########################################################################
# proc MakeworldForm {args} -
#  Create a worldForm megawidget
# Arguments
#  args    A list of arguments to define the widget.
#          If the first argument starts with a ".", it is
#          the parent frame for the widget.
#          If this is not provided, the megawidget
#          is a child of the root frame (".").
#
#          Other args are key/value pairs
# Results
#  Returns the name of the parent frame for use as a
#  megawidget

proc MakeworldForm {args} {
variable worldFormState

  # Set the default name
  set holder .worldForm_$worldFormState(unique)
  incr worldFormState(unique)

  # If the first argument is a window path, use that as
  # the base name for this widget.
  if {[string first "." [lindex $args 0]] == 0} {
    set holder [lindex $args 0]
    set args [lreplace $args 0 0]
  }

  # Set Command line option defaults here,
  #  height and width are freebies
  set worldFormState($holder.height) 5
  set worldFormState($holder.width) 20
  set worldFormState($holder.buttonText) "translate"
  set worldFormState($holder.row) 0
  set worldFormState($holder.row) 0
  set worldFormState($holder.row) 0
  set worldFormState($holder.language) english
```

```
  foreach {key val } $args {
    set keyName [string range $key 1 end]

    # Check that this is a valid key.
    if {![info exists  worldFormState($holder.$keyName)]} {

       # Not valid key, generate an error.
       # Extract valid keys from worldFormState array indices.

       regsub -all "$holder." \
       [array names worldFormState $holder.*] \
           "" okList
       error "Bad option"  "Invalid option '$key'.\n \
       Must be one of $okList"
    }
    set worldFormState($holder.$keyName) $val
  }

  # Create master Frame
  frame $holder -height $worldFormState($holder.height) \
      -width $worldFormState($holder.width)  -class WorldForm

  # Apply invocation options to the master frame, as
  # appropriate
  foreach {opt val} $args {
    catch {$holder configure $opt $val}
  }

  Draw $holder

  # We can't have two widgets with the same name.
  #    Rename the base frame procedure for this
  #    widget to $holder.fr so that we can use
  #    $holder as the widget procedure name
  #    for the megawidget.
  uplevel #0 rename $holder $holder.fr

  # When this window is destroyed,
  #  destroy the associated command.
  bind $holder <Destroy>  \
      "+[namespace code [list rename $holder {}]]"
  return $holder
}

##########################################################################
# proc Draw {parent}--
#  creates the subwidgets and maps them into the parent
```

```
# Arguments
#  parent   The parent frame for these widgets
#
# Results
#  New windows are created and mapped.

proc Draw {parent} {
  variable worldFormState
  set tmp [button $parent.translate \
      -text $worldFormState($parent.buttonText) \
      -command [namespace code [list translate $parent]] ]

  set worldFormState($parent.subWidgetName.translate) $tmp

  grid $parent.translate -row $worldFormState($parent.row)\
      -column 0

  set tmp [tk_optionMenu $parent.options\
      [namespace current]::worldFormState(language)\
      French English German  Spanish ]

  set worldFormState($parent.subWidgetName.options) $tmp

  grid $parent.options -row $worldFormState($parent.row)\
      -column 1
}

######################################################################
# proc DoWidgetCommand {widgetName widgets cmd}--
#       Perform operations on subwidgets
# Arguments
#  widgetName:  The name of the holder frame
#  widgets:     A list of the public names for subwidgets
#  cmd:     A command to evaluate on each of these widgets
# Results
#  Does stuff about the subwidgets

proc DoWidgetCommand {widgetName widgets cmd} {
  variable worldFormState
  foreach widget $widgets {
    set index $widgetName.subWidgetName.$widget
    eval $worldFormState($index) $cmd
  }
}

######################################################################
# proc translate {parent}--
```

```
#  Change the labels from one language to another
# Arguments
#  parent:  The parent widget to examine for label widgets
#
# Results
#  The labels are -configured with new text

proc translate {parent} {
  variable worldFormState

  set language [string tolower $worldFormState(language)]
  foreach w [winfo children $parent] {
    if {[string match [winfo class $w] Label]} {
       set old [$w cget -text]
       $w configure -text $worldFormState($language.$old)
    }
  }
}

########################################################################
# proc updateXlateTable {lst}--
#  Add index/value pairs to translation table
# Arguments
#  lst:  List of language and word pairs
#
# Results
#  The state array is updated
proc updateXlateTable {lst} {
  variable worldFormState

  # sample: english  Name french Nom spanish Nombre
  # array set State {
  #  english.Nom Name english.Nombre Name }
  # array set State {
  #  spanish.Nom Nombre spanish.Name Nombre}
  # For later xlate to english:
  #  "set word $State(english.$word)"

  foreach {lang word} $lst {
    set lookup ""
    foreach {l w} $lst {
       if {![string match $l $lang]} {
         lappend lookup $lang.$w $word
       }
    }
    array set worldFormState $lookup
  }
```

```
}

##########################################################################
#     proc insert {parent args}--
#  Insert a new Label/Entry pair
# Arguments
#  -width  Width of entry widget
#  -alternates List of language alternates:
#       {lang1 word1 lang2 word2}
#  -text     Text for label
#  -textvar  Textvariable for entry widget
# Results
#  Adds new label/entry pair and increments the row pointer.
proc insert {parent args} {
  variable worldFormState
  set width 10
  incr worldFormState($parent.row)
  set alternates " "

  foreach {key val} $args {
    set varName [string range $key 1 end]
    set $varName $val
  }

  set w [entry $parent.e[unique] -textvariable $textvariable \
     -width $width]

  grid $w -row $worldFormState($parent.row) -column 1
  set w [label $parent.l[unique] -text $text]
  grid $w -row $worldFormState($parent.row) -column 0
  updateXlateTable $alternates
}

##########################################################################
# proc WorldFormProc {widgetName subCommand args}
#  The master procedure for handling this megawidget's
#  subcommands
# Arguments
#  widgetName    The name of the worldForm widget
#  subCommand    The subCommand for this cmd
#  args:         The rest of the command line
#                arguments
# Default subCommands are:
# configure - configure the parent frame
# widgetconfigure - configure or query a subwidget.
#                   mega widgetconfigure itemID -key value
#                   mega widgetconfigure itemID -key
```

```
# widgetcget - get the configuration of a widget
#                   mega widgetcget itemID -key
# widgetcommand - perform a command on a widget
#                 mega widgetcommand itemID commandString
# names - return the name or names that match
#         a pattern
#    mega names       # Get names of all widgets
#    mega names *a*   # Get names of widgets
#                     # with an 'a' in them.
# subwidget - return the full pathname of a
#             requested subwidget
# Results
#  Evaluates a subcommand and returns a result if required.

proc WorldFormProc {widgetName subCommand args} {
  variable worldFormState
  set cmdArgs $args
  switch -- $subCommand {
    configure {
       return [eval $widgetName.fr configure $cmdArgs]
    }

    widgetconfigure {
       set sbwid [lindex $cmdArgs 0]

       set cmd [lrange $cmdArgs 1 end]
       set index $widgetName.subWidgetName.$sbwid

       catch {eval $worldFormState($index) configure $cmd} rtn
       return $rtn
    }

    insert  {
       return [eval insert $widgetName $cmdArgs]
    }

    widgetcget {
       if {[llength $cmdArgs] != 2} {
         error "$widgetName cget subWidgetName -option"
       }
       set sbwid [lindex $cmdArgs 0]
       set index $widgetName.subWidgetName.$sbwid
       set cmd [lrange $cmdArgs 1 end]
       catch {eval $worldFormState($index) cget $cmd} rtn
       return $rtn
    }
```

```
        widgetcommand {
           return [eval DoWidgetCommand $widgetName $cmdArgs]
        }

        names {
           if {[string match $cmdArgs ""]} {
             set pattern $widgetName.subWidgetName.*
           } else {
             set pattern $widgetName.subWidgetName.$cmdArgs
        }
        foreach n [array names worldFormState $pattern] {
           foreach {d w s name} [split $n .] {}
           lappend names $name
        }
        return $names
        }

        subwidget {
           set name [lindex $cmdArgs 0]
           set index $widgetName.subWidgetName.$name
           if {[info exists worldFormState($index)]} {
             return $worldFormState($index)
           }
        }
        default {
           error "bad command" "Invalid Command: $subCommand \n\
           must be configure, widgetconfigure,  widgetcget, names,\
           insert,   or subwidget"
        }
      }
    }
}
```

Script Example

```
source Ch14-World.tcl

grid [worldForm .x]

.x insert -text Name -textvariable name \
    -alternates {english Name french Nom \
    spanish Nombre german Name}

.x insert -text Address -textvariable address -width 20  \
    -alternates {english Address french Adresse \
    spanish Direcci n  german Adresse}
```

Script Output

Initial

After click translate

After select Spanish

After click translate

14.7 INCORPORATING A MEGAWIDGET INTO A LARGER MEGAWIDGET

The Tk design philosophy is that large complex things can be built out of smaller, less complex things. A megawidget is larger than a primitive widget, but it should still be possible to merge one or more megawidgets into a larger megawidget.

In this section, we will create a larger megawidget that contains two of the `scrolledLB` widgets, a `label`, a `button`, and an `entry` widget. This megawidget will allow a user to define a filter to extract a subset of the entries from the first listbox for display in the second listbox. When the widget is queried, it will return the selected items in the filtered listbox, or in the selected items in the unfiltered listbox if nothing was selected in the filtered box.

The justification for this widget is that sometimes there are too many items in a listbox to find those the user is interested in, but a simple filter could cut down the size to something manageable. This is similar to the effect that various file browsers get with the "filter" option, but this widget allows the user to see both the unfiltered list and the filtered list at the same time.

Like the `scrolledLB`, this megawidget is implemented as a procedure for creating the widget in the global scope with support procedures maintained in a namespace scope. As described in the previous section, there are some points to watch when using namespaces with Tk widgets. The button command string uses the `namespace code` command to cause the script to be evaluated in the current namespace.

If we knew that the `filteredLB` widget would always be defined at a top-level, we could just define the command string as `::filteredLB::DoFilter`, but just as the `scrolledLB` megawidget is contained within the `filteredLB` widget, the `filteredLB` widget may be contained in a larger widget, as shown in the file browser megawidget example in the next section.

Example 15

Complete Code

```
# Name: Filtered.tcl
lappend auto_path .
package provide filteredLB 1.0
package require scrolledLB

##########################################################################
# proc filteredLB {args}
#  Create a filteredLB megawidget
#  External entry point for creating a megawidget.
# Arguments
#  args     A list of arguments to define the widget.
#           If the first argument starts with a "."
#               it will be used as the name of the main frame.
#       If this is not provided, the megawidget
#           is a child of the root frame (".").
# Results
#  Creates a procedure with the megawidget name for
#    processing widget commands
#  Returns the name of the megawidget

proc filteredLB {args} {
  set newWidget [eval filteredLB::MakefilteredLB $args]

  set newCmd [format {return [namespace eval %s %s %s $args ]}\
      filteredLB FilteredLBProc $newWidget]

  proc $newWidget {args} $newCmd

  return $newWidget
}
namespace eval filteredLB {
  variable filteredLBState

  # Assign a couple defaults
  set filteredLBState(unique) 0

  ##########################################################################
  # proc MakefilteredLB {args}
  #  Create a filteredLB megawidget
  # Arguments
  #  args   A list of arguments to define the widget.
  #         If the first argument starts with a "."
  #               it will be used as the name of the main frame.
  #         If this is not provided, the megawidget
```

```
#         is a child of the root frame (".").
#
#             Subsequent args are -option/value pairs.
#     ?-scrollposition left/right?
#         The side of the listbox for the scrollbar
# Results
#  Returns the name of the parent frame for use as a
#  megawidget
proc MakefilteredLB {args} {
  variable filteredLBState

  # Set the default name
  set holder .filteredLB_$filteredLBState(unique)
  incr filteredLBState(unique)

  # If the first argument is a window path, use that as
  # the base name for this widget.
  if {[string first "." [lindex $args 0]] == 0} {
    set holder [lindex $args 0]
    set args [lreplace $args 0 0]
  }

  # Set Command line option defaults here,
  #  height and width are freebies
  set filteredLBState($holder.height) 5
  set filteredLBState($holder.width) 20
  set filteredLBState($holder.listbox1Height) 10
  set filteredLBState($holder.title1Text) title1
  set filteredLBState($holder.listbox1Width) 30
  set filteredLBState($holder.listbox2Height) 10
  set filteredLBState($holder.title2Text) title2
  set filteredLBState($holder.listbox2Width) 30
  set filteredLBState($holder.titleFont) {helvetica 20 bold}
  set filteredLBState($holder.titleText) title
  set filteredLBState($holder.patternVariable) filterVar
  set filteredLBState($holder.patternWidth) 10

  foreach {key val } $args {
    set keyName [string range $key 1 end]
    if {![info exists \
          filteredLBState($holder.$keyName)]} {
      regsub -all "$holder." \
          [array names filteredLBState $holder.*] \
          " " okList
      error "Bad option" \
          "Invalid option '$key'.\n\
          Must be one of $okList"
```

```
    }
  set filteredLBState($holder.$keyName) $val
}

# Create master Frame
frame $holder -height $filteredLBState($holder.height) \
    -width $filteredLBState($holder.width) \
    -class FilteredLB

# Apply invocation options to the master frame, as appropriate
foreach {opt val} $args {
  catch {$holder configure $opt $val}
}
Draw $holder

# We can't have two widgets with the same name.
#  Rename the base frame procedure for this
#  widget to $holder.fr so that we can use
#  $holder as the widget procedure name
#  for the megawidget.
uplevel #0 rename $holder $holder.fr

# When this window is destroyed,
#  destroy the associated command.
bind $holder <Destroy> "+ rename $holder {}"

return $holder
}

######################################################################
#  proc Draw {parent}--
# creates the subwidgets and maps them into the parent
# Arguments
#  parent   The parent frame for these widgets
#
# Results
#  New windows are created and mapped.
proc Draw {parent} {
  variable filteredLBState
  image create photo arrow -data {
  R0lGODlhEAAQAPABAAAAAP///yH5BAAAAAAALAAAAAAQABAAAAK
  RTJgwYcKECRMmTJgwYcKECRMmTJgwYcKECBMmTJgwYcKECRMmTJ
  gwYcKEABEmTJgwYcKEAAEmDAgQIECAAAEiDAgQIECAAAECDAgQI
  ECAAAEiTJgwYcKEAAEmTJgwYcKEABEmTJgwYcKEABMmTJgwYcKE
  CBMmTJgwYcKECRMmTJgwYcKECRMmTJgwYcKECRMmBQA7
  }
```

```
  set w [scrolledLB $parent.list1 \
      -listboxHeight $filteredLBState($parent.listbox1Height) \
      -titleText $filteredLBState($parent.title1Text) \
      -listboxWidth $filteredLBState($parent.listbox1Width)]

  set filteredLBState($parent.subWidgetName.list1) $w
  grid $w  -row 2 -column 0 -rowspan 2

  set w [scrolledLB $parent.list2   \
      -listboxHeight $filteredLBState($parent.listbox2Height) \
      -titleText $filteredLBState($parent.title2Text) \
      -listboxWidth $filteredLBState($parent.listbox2Width)]
  set filteredLBState($parent.subWidgetName.list2) $w
  grid $w  -row 2 -column 2 -rowspan 2

  set w [label $parent.title   \
      -font $filteredLBState($parent.titleFont) \
      -text $filteredLBState($parent.titleText)]
  set filteredLBState($parent.subWidgetName.title) $w
  grid $w  -row 0 -column 0 -columnspan 3

  set w [entry $parent.pattern   \
      -textvar $filteredLBState($parent.patternVariable) \
      -width $filteredLBState($parent.patternWidth)]
  set filteredLBState($parent.subWidgetName.pattern) $w
  grid $w  -row 1 -column 0 -columnspan 3

  set w [button $parent.go   \
      -image arrow \
      -command "[namespace code [list DoFilter $parent]]"]
  set filteredLBState($parent.subWidgetName.go) $w
  grid $w  -row 2 -column 1 -sticky n
}

######################################################################
#  proc DoWidgetCommand {widgetName widgets cmd}--
# Perform operations on subwidgets
# Arguments
#  widgetName: The name of the holder frame
#  widgets:    A list of the public names for subwidgets
#  cmd:        A command to evaluate on each of these widgets
# Results
#  Does stuff about the subwidgets
proc DoWidgetCommand {widgetName widgets cmd} {
  variable filteredLBState
  foreach widget $widgets {
    set index $widgetName.subWidgetName.$widget
```

```
    eval $filteredLBState($index) $cmd
  }
}

#########################################################################
#  proc DoFilter {holder}--
# Tests items in full listbox, and puts ones that match
#  a pattern into the second listbox.
#
# Arguments
#  holder  The name of the parent frame.
#
# Results
#  The second listbox (list2) may receive new entries.
proc DoFilter {holder} {
  variable filteredLBState

  set pattern [$filteredLBState($holder.subWidgetName.pattern) get]
  set fullListbox [[$holder subwidget list1] subwidget list]
  set filterListbox [[$holder subwidget list2] subwidget list]

  foreach item [$fullListbox get 0 end] {
    if {[string match $pattern $item]} {
       $filterListbox insert end $item
    }
  }
}

#########################################################################
#  proc Selection {holder}--
# Returns the selected items from the second listbox
#
# Arguments
#  holder   The name of the parent frame.
#
# Results
#  No changes to widget
proc Selection {holder} {
  set filterListbox [[$holder subwidget list2] subwidget list]
  set lst [$filterListbox curselection]
  set itemlst ""
  foreach l $lst {
    lappend itemlst [$filterListbox get $l]
  }
  if {[string match $itemlst ""]} {
    set fullListbox [[$holder subwidget list1] subwidget list]
    set lst [$fullListbox curselection]
```

```
    set itemlst ""
    foreach l $lst {
      lappend itemlst [$fullListbox get $l]
    }
  }
  return $itemlst
}

######################################################################
#  proc FilteredLBProc {widgetName subCommand args}
# The master procedure for handling this megawidget's
#  subcommands
# Arguments
#  widgetName:   The name of the filteredLB
#    widget
#  subCommand     The subCommand for this cmd
#  args:     The rest of the command line
#    arguments
# Default subCommands are:
# configure - configure the parent frame
# widgetconfigure - configure or query a subwidget.
#    mega widgetconfigure itemID -key value
#    mega widgetconfigure itemID -key
# widgetcget - get the configuration of a widget
#    mega widgetcget itemID -key
# widgetcommand - perform a command on a widget
#    mega widgetcommand itemID commandString
# names - return the name or names that match
#    a pattern
#    mega names # Get names of all widgets
#    mega names *a* # Get names of widgets
#    with an 'a' in them.
# subwidget - return the full pathname of a
#    requested subwidget
# Results
#  Evaluates a subcommand and returns a result if required.

proc FilteredLBProc {widgetName subCommand args} {
  variable filteredLBState

  set cmdArgs $args

  switch -- $subCommand {
    configure {
      return [eval $widgetName.fr configure $cmdArgs]
    }
```

```
widgetconfigure {
  set sbwid [lindex $cmdArgs 0]
  set cmd [lrange $cmdArgs 1 end]
  set index $widgetName.subWidgetName.$sbwid
  catch {eval \
      $filteredLBState($index) configure $cmd}  rtn
  return $rtn
}

delete  -
insert  {
  return [eval $widgetName.list1 $subCommand $cmdArgs]
}

selection {return [Selection $widgetName]}

widgetcget {
  if {[llength $cmdArgs] != 2} {
     error "$widgetName cget subWidgetName -option"
  }
  set sbwid [lindex $cmdArgs 0]
  set index $widgetName.subWidgetName.$sbwid
  set cmd [lrange $cmdArgs 1 end]
  catch {eval \
     $filteredLBState($index)\
     cget $cmd}  rtn
  return $rtn
}

widgetcommand {
  return [eval DoWidgetCommand $widgetName $cmdArgs]
}

names {
  if {[string match $cmdArgs ""]} {
    set pattern $widgetName.subWidgetName.*
  } else {
    set pattern $widgetName.subWidgetName.$cmdArgs
  }

  foreach n [array names filteredLBState $pattern] {
    foreach {d w s name} [split $n .] {}
    lappend names $name
  }
  return  $names
}
```

```
        subwidget {
          set name [lindex $cmdArgs 0]
          set index $widgetName.subWidgetName.$name
          if {[info exists filteredLBState($index)]} {
            return $filteredLBState($index)
          }
        }

        default {
        error "bad command" "Invalid Command: \
           $subCommand \n \
           must be configure, widgetconfigure, \
           widgetcget, names,\
           delete, insert, selection,  \
           or subwidget"
        }
      }
    }
  }
```

Using the `filteredLB` for the file browser is simple. You simply create the megawidget, map it, and fill it with the appropriate values, as shown in the following example.

Note the `option add` commands. These set widgets associated with the `filteredLB` widget to a medium gray, and widgets associated with the `scrolledLB` widget to a lighter gray. The `option` command will apply colors to the most specific option. The parts of the `filteredLB` widget that are implemented with the `scrolledLB` widget override the darker gray to become lighter, whereas the `title`, `entry`, and `button` widgets that are not part of a `scrolledLB` widget are assigned the medium gray color.

Example 16

Script Example

```
lappend auto_path.
package require filteredLB

option add *FilteredLB*Background #ddd
option add *ScrolledLB*Background #eee

set fb [filteredLB .dir -patternVariable tst  \
    -title1Text "Original Data" -title2Text "Filtered Data"  \
    -listbox1Height 5 -listbox2Height 5 \
    -titleText "Filtered directory contents"  \
    -listbox1Width 15 -listbox2Width 15]
```

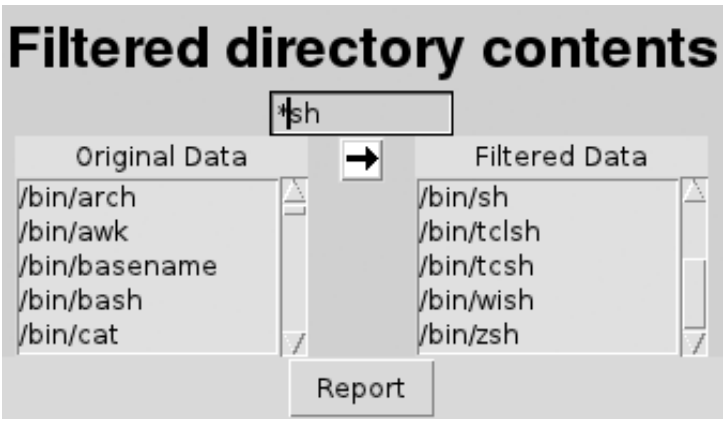
```
grid $fb

set path /bin

foreach f [lsort [glob $path/*]] {
    $fb insert end $f
}

button .report -text "Report" \
    -command {tk_messageBox -type ok -message [.dir selection]}
grid .report
```

Script Output



14.8 MAKING A MODAL MEGAWIDGET: THE `grab` AND `tkwait` COMMANDS

In this section we will take the `filteredList` megawidget we just developed and make a modal file selection widget. This widget is far from the last word in file selection widgets, but it makes a good example of how a modal widget can be constructed.

This widget is implemented as a single procedure that creates and displays the megawidget and returns the results of the user interaction when that interaction is complete. The condition for completion is that the user has either clicked the Quit button, clicked the OK button, or destroyed the widget using a window manager command.

This widget introduces the `grab` and `tkwait` commands that have not been used in previous examples. The `grab` command allows your application to examine and control how mouse and keyboard events are reported to the windows on your display. The `tkwait` command will cause a script to pause until an external event (such as a window being destroyed) occurs.

14.8.1 The grab Command

By default, whenever a cursor enters or leaves a widget, an Enter or Leave event is generated. When the cursor enters a widget, that widget is said to have focus, and all keystroke and button events will be reported to that widget. By using the grab command, you can declare that all keyboard and mouse events will go to a particular widget regardless of the location of the cursor. This requires a user to perform a set of interactions before your program releases the grab. Once the grab is released, the focus will follow the cursor and mouse and keyboard events will be distributed normally.

The grab command has several subcommands that let you determine whether other grabs are already in place (in which case your program should save that state and restore it when it is done) and set a new state. You can learn what window is currently grabbing events with the grab current command.

Syntax: grab current *?window?*

Returns a list of the windows that are currently grabbing events.

window If *window* is defined and a child of *window* has grabbed events, that child will be returned. If no child of the window has grabbed events, an empty string is returned. If *window* is not defined, all windows in the application that have grabbed events will be reported.

A window can either grab all events for that application or all events for the entire system. The default mode is to grab only events for the current application. If you need to grab events for the entire system, this can be done with the -global flag when a grab is requested. If a grab is in effect, you can determine whether it is for a local (default) or global scope with the grab status command.

Use care when using grab global. When your application grabs all events the rest of the windowing system is frozen until your application releases the grab global.

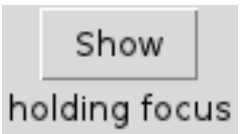
The following example shows the focus being grabbed by a label, instead of the button. Since the label is grabbing focus, the button never responds to <Enter> or button events.

Example 17

Script Example

```
grid [button .b -text "Show" \
    -command {puts [grab current]}]
grid [label .l -text "holding focus"]
grab .l
puts "Focus is held by: [grab current]"
```

Script Output




```
Focus is held by: .l
```

Syntax: `grab status` *`window`*

Returns the status of a `grab` on the defined window. The return will be one of:

`local`	The window is in local (default) mode. Only events destined for this application will be routed to this window.
`global`	The window is in global mode. All events in the system will be routed to this window.
`empty string`	There is no `grab` in effect for this window.

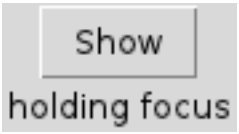
`window` The window for which status will be returned.

Example 18

Script Example

```
pack [button .b -text "Show" -command {puts [grab current]}]
pack [label .l -text "holding focus"]
grab .l
puts "Status for .l is: [grab status .l]"
```

Script Output




```
Status for .l is: local
```

You can declare that all events will go to a window with the `grab` or `grab set` command. These commands behave identically.

Syntax: `grab ?set?` *`?-global? window`*

Set a window to grab keyboard and mouse events.

`-global`	Grab all events for the system. By default, only events generated while this application has the focus will be directed to this window.
`window`	The window to receive the events.

14.8.2 The `tkwait` Command

Once all events in the system are directed to the modal widget, the widget has to know when the interaction is complete. We can force the Tcl interpreter to wait until the user has performed an interaction with the `tkwait` command. This command will wait until a particular event has occurred. The event may be a variable being modified (perhaps by a button press), a change in the visibility of a window, or a window being destroyed.

Syntax: `tkwait` *`EventType name`*

Wait for an event to occur.

`EventType`	A name describing the type of event that may occur. The content of *`name`* depends on which type of event has occurred.

Valid events are:

`variable` *`varName`*	Causes `tkwait` to wait until the variable *`varName`* changes value.
`visibility` *`windowName`*	Causes `tkwait` to wait until the window *`windowName`* either becomes visible or is hidden.
`window` *`windowName`*	Causes `tkwait` to wait until the window named in *`windowName`* is destroyed.

Example 19

Script Example

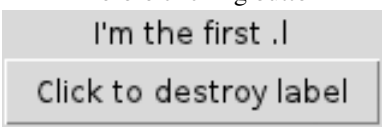
```
#  Create and display two widgets}
grid [label .l -text "I'm the first .l"]
grid [button .b -text "Click to destroy label" -command "destroy .l"]

#  The code pauses here until the window named .l is destroyed
#  This will happen when the button is clicked.
tkwait window .l

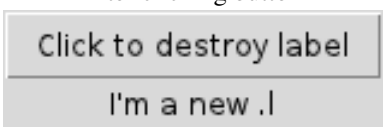
#  After the button is clicked, and .l is destroyed,
#   a new .l is created and packed.
grid [label .l -text "I'm a new .l"]
```

Script Output

Before clicking button



After clicking button



14.8.3 The Modal Widget Code

The next example shows how the `filtered listbox` megawidget can be used to create a modal file selection megawidget.

Example 20

Complete Code

```
## Name: getFileList.tcl
lappend auto_path "."
package require filteredLB
package provide fileSel 1.0

#########################################################################
#  proc getFileList {directory}-/-
#   Returns a list of selected files from the requested
#   directory.
#  Arguments
#   directory     The directory to return a list of files from
#
#  Results
#   Creates a new window and grabs focus until user
#     interacts with it.
#   Returns a list of the selected files.
#   Returns an empty list if:
#        no files are selected,
#       'QUIT' button is clicked
#       window is destroyed.

proc getFileList {directory} {
  global popupStateVar

  #  Find an unused, unique name for a toplevel window.
  #  winfo children returns a list of child windows.
  set unique 0
  set childlist [winfo children .]
  while {[lsearch childlist .dirList_$unique]!= -1} {
    incr unique
  }

  #  Create the new toplevel window that will hold this
  # widget
  set win [toplevel .dirList_$unique -class Fileselector]

  #  Create the widget as a child of the $win window
  #  And configure the listboxes for multiple selections
  set fl [filteredLB $win.fl1 -titleText "Select Files" \
      -title1Text "Files in Directory" \
      -title2Text "Filtered list" ]

  [$fl subwidget list1] \
      widgetconfigure list -selectmode multiple
```

```
[$fl subwidget list2] \
    widgetconfigure list -selectmode multiple

#  And fill the full list
foreach file [lsort [glob -nocomplain $directory/*]] {
  $fl insert end [file tail $file]
}

#  Pack the filteredList megawidget at the top of the
#  new window
pack $fl -side top

#  Create the buttons in a frame of their own, and pack them
set bframe [frame $win.buttons]

set quit [button $bframe.quit -text "QUIT" -command\
    {set popupStateVar(ret) ""}]

set ok [button $bframe.ok -text "OK" -command \
   "set popupStateVar(ret) \[$fl selection\]"]

pack $quit -side left
pack $ok -side left
pack $bframe -side bottom

#  bind the destroy event to setting the trigger variable,
#  so we can exit properly if the window is destroyed by
#  the window manager.
bind $win <Destroy> {set popupStateVar(ret) ""}

#  The following code is adapted from dialog.tcl in the
#  tk library. It sets the focus of the window manager
#  on the new window, and waits for the value of
#  popupStateVar(ret) to change.
#
#  When the user clicks a button, or destroys the window,
#  it will set the value of popupStateVar(ret)!,
#  and processing will continue
#  Set a grab and claim the focus.
set oldFocus [focus]
set oldGrab [grab current $win]
if {$oldGrab != ""} {
  set grabStatus [grab status $oldGrab]
}
grab $win
focus $win
```

```
    #  Wait for the user to respond, then restore the focus
    #  and return the index of the selected button. Restore
    #  the focus before deleting the window, since otherwise
    #  the window manager may take the focus away so we can't
    #  redirect it.  Finally, restore any grab that was in
    #  effect.
    tkwait variable popupStateVar(ret)
    catch {focus $oldFocus}

    #  It's possible that the window has already been
    #  destroyed, hence this "catch". Delete the
    #  <Destroy> handler so that popupStateVar(ret)
    #  doesn't get reset by it.
    catch {
     bind $win <Destroy> {}
     destroy $win
    }

    if {$oldGrab != ""} {
      if {$grabStatus == "global"} {
        grab -global $oldGrab
      } else {
        grab $oldGrab
      }
    }
    return $popupStateVar(ret)
}
```

Script Example

```
#  The widgets pkgIndex.tcl file is in the current directory
#  in this example.
lappend auto_path "."

#  Include the widgets package to get the
# fileSel megawidget.
package require fileSel

#  Create a label and entry widget to get a base directory path
set dirFrame [frame .f1]
label $dirFrame.l1 -text "base directory: "
entry $dirFrame.e1 -textvariable directory

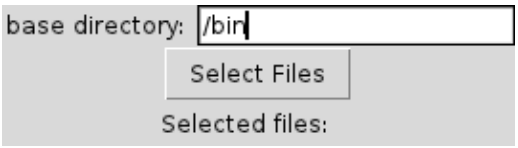
#  Create buttons to invoke the getFileList widget.
set buttonFrame [frame .f2]
button $buttonFrame.b1 -text "Select Files" \
    -command {set filelist [getFileList $directory]}
```

```
#  The selected files will be displayed in a label widget
#  with the -textvariable defined as the filelist returned
#  from the getFileList widget.
set selFrame [frame .sel]
set selectLabel [label $selFrame.label -text \
    "Selected files:"]
set selectVal [message $selFrame.selected \
    -textvariable filelist -width 100]

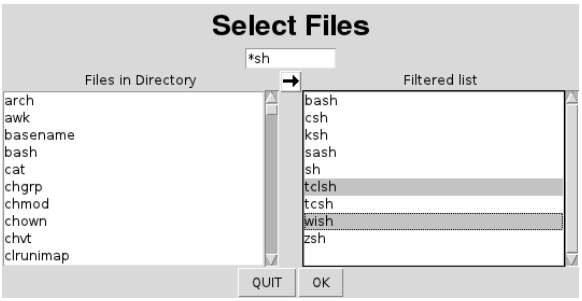
#  and pack everything
pack $dirFrame.l1 -side left
pack $dirFrame.e1 -side right
pack $dirFrame -side top
pack $buttonFrame -side top
pack $buttonFrame.b1 -side left
pack $selFrame -side top
pack $selectVal -side right
pack $selectLabel -side left
```

Script Output

Initial Window



File selection window, after selections



Initial window, after OK button clicked

14.9 AUTOMATING MEGAWIDGET CONSTRUCTION

You may have noticed that the three megawidgets presented in this chapter are very similar. In fact, the bulk of the code in these widgets is identical. Building custom megawidgets can be automated in a few ways, by working from a skeleton and filling in the pieces, by further automating building from a skeleton with a configuration file and a Tcl script, or by assembling compound widgets on the fly with another set of procedures.

14.9.1 Building Namespace Megawidgets Summary

The architecture of these megawidgets is consistent. There is a procedure in the global scope to create a megawidget, which calls procedures in the appropriate namespace to populate the megawidget and return the name of the procedure to interact with the megawidget. A second global scope procedure is created to interact with the megawidget. This procedure has the same name as the parent frame, and will evaluate scripts within the megawidget namespace to implement the widget commands.

The unique part of each megawidget is the procedure to draw the subwidgets, and the procedures that implement unique subwidget commands.

There are many ways to construct a compound widget using classic Tk widgets and namespaces. A compound widget can be built from a skeleton, the widgets can be created from a configuration file, or generated by a set of scripts.

Jeffrey Hobbs, *jeff@hobbs.org*, created a technique for automating megawidget creation with his `widget` package. His package will allow you to easily create a megawidget that uses a `subwidget` command or a naming convention to access the subwidgets. The subwidgets can be configured by accessing them as individual widgets, using either the name returned with the `subwidget` command or the subwidget naming convention to access the subwidgets. Configuration options related to the entire megawidget can be processed via the megawidget widget procedure, a procedure with the same name as that given to the megawidget.

14.10 BUILDING MEGAWIDGETS WITH TCLOO

Building megawidgets with namespaces is the traditional approach to compound and complex widgets. However, building megawidgets involves concepts like inheriting behavior and aggregating multiple objects into a single entity. These requirements are what object-oriented design styles are best at.

TclOO supports building megawidgets. Many of the design decisions are the same, whether you are building a megawidget in a namespace or as an TclOO class. Decisions like whether the widget should be modal or not, whether it should create a toplevel widget or be embedded in the applications window, whether it should mimic Tk behavior or follow its own patterns, etc. All of these patterns can be implemented with TclOO as easily or easier than with namespaces. TclOO even has features that make it easier to mimic the behavior of the Tk widgets.

As with building compound widgets using namespaces, there are a few different patterns that can be used to build TclOO megawidgets.

14.10.1 Using a Wrapper Proc for TclOO-Based Widgets

The simplest pattern is similar to the pattern used for namespace-based compound widgets—a global procedure as a wrapper to create a megawidget, perform the renaming and return a new widget command with the appropriate name.

The next example shows a scrolled listbox widget built using a factory procedure `scrolledLB`. The `scrolledLB` proc creates a compound widget using TclOO instead of in a namespace. The differences to look for are:

1. The `scrolledLB` command creates a new object, rather than calling a command in a namespace.
2. With TclOO the `State` variable is specific to each object, rather than being shared by all widgets using this namespace. Because each object has a private copy of State there is no need to include the widget name as part of the State array indices. This makes the references simpler.
3. Instead of a single `scrolledLBProc` with a switch statement to determine which widget command to evaluate, each subcommand is a method in the `scrolledLB` class.
4. When the widget is destroyed, the binding destroys the object, instead of renaming the proc to an empty string.
5. Subwidget options are identified as *subwidgetName.optionName*. For example, the height of the listbox is `list.height`.

Example 21

Script Example

```
package provide scrolledLB 2.0

##########################################################################
# proc scrolledLB {args}
#  Create a scrolledLB megawidget
#  External entry point for creating a megawidget.
# Arguments
#  ?parentFrame?   A frame to use for the parent,
#                  If this is not provided, the megawidget
#                  is a child of the root frame (".").
#
#  args            A set of key/value pairs to describe the listbox
#
# Results
#  Creates a procedure with the megawidget name for processing
#  widget commands
#  Returns the name of the megawidget

proc scrolledLB {winName args} {

  set newWidget [scrolledLB_OO new $winName {*}$args]

  # Define a new command which passes control to the
```

```
  #  scrolledLB::scrolledLBProc command and return the results.

  rename $newWidget  "$winName"

  return $winName
}

oo::class create scrolledLB_OO {
  variable State

  ##########################################################################
  # method constructor
  #  Create a scrolledLB megawidget
  # Arguments
  #  frameName  A frame to use for the parent.
  #    Subsequent arguments are -option/value pairs
  #  ?-scrollSide left/right?
  #     The column in the scrolledLB for the scrollbar
  # Results
  #  Returns the name of the parent frame for use as a
  #  megawidget

  constructor {frameName args} {

    # Set Command line option defaults here
    #  height and width are freebies
    array set State {
      height 5
      width 20
      list.background white
      list.height 10
      list.width 20
      scroll.side 1
      list.side 0
      title.text title
    }

    foreach {key val } $args {
      set keyName [string range $key 1 end]
      if {![info exists State($keyName)]} {
        error "Bad option" \
            "Invalid option '$key'.\n\
            Must be one of [array names State]"
      }
      set State($keyName) $val
    }
```

```
  # Create master Frame
  frame $frameName -height $State(height) \
      -width $State(width) \
      -class ScrolledLB

  # Apply invocation options to the master frame, as appropriate
  foreach {opt val} $args {
    catch {$frameName configure $opt $val}
  }

  my Draw $frameName

  # Apply subwidget options
  foreach {k v} $args {
    lassign [split $k .] sub opt
    set sub [string trim $sub -]
    my widgetconfigure $sub -$opt $v
  }

  # We can't have two widgets with the same name.
  #  Rename the base frame procedure for this
  #  widget to $frameName.fr so that we can use
  #  $frameName as the widget procedure name
  #  for the megawidget.

  uplevel #0 rename $frameName $frameName.fr
  set State(subWidgetName.frame) $frameName.fr

  # When this window is destroyed,
  #  destroy the associated command.
  bind $frameName <Destroy> "+ [self] destroy"
  return $frameName
}

######################################################################
# method Draw {parent}--
#  creates the subwidgets and maps them into the parent
# Arguments
#  parent   The parent frame for these widgets
#
# Results
#  New windows are created and mapped.
method Draw {parent} {
  variable State

  set tmp [scrollbar $parent.yscroll  -orient vertical \
      -command "$parent.list yview" ]
```

```
  set State(subWidgetName.yscroll) $tmp
  grid $tmp  -row 1 -sticky ns \
      -column $State(scroll.side)

  set tmp [listbox $parent.list  \
      -yscrollcommand "$parent.yscroll set" \
      -height $State(list.height)\
      -width $State(list.width) ]

  set State(subWidgetName.list) $tmp
  grid $tmp  -row 1 -column $State(list.side)

  set tmp [label $parent.title  \
      -text $State(title.text) ]

  set State(subWidgetName.title) $tmp
  grid $tmp  -row 0 -column 0 -columnspan 2
}

##########################################################################
# method DoWidgetCommand {widgetName widgets cmd}--
#  Perform operations on subwidgets
# Arguments
#  widgetName: The name of the holder frame
#  widgets:    A list of the public names for subwidgets
#  cmd:        A command to evaluate on each of these widgets
#
# Results

#  Does stuff about the subwidgets

method DoWidgetCommand {widgetName widgets cmd} {
  variable State
  foreach widget $widgets {
    set index subWidgetName.$widget
    eval $State($index) $cmd
  }
}

method selection {} {
  set lst [$State(subWidgetName.list) curselection]
  set itemlst ""
  foreach l $lst {
     lappend itemlst [$State(subWidgetName.list) get $l]
  }
  return $itemlst
}
```

```
method configure {args} {
       return [eval $widgetName.fr configure ${*}$args]
}

method widgetconfigure {args} {
       set sbwid [lindex $args 0]
       set cmd [lrange $args 1 end]
       set index subWidgetName.$sbwid
       catch {eval \
           $State($index) configure $cmd}  rtn
       return $rtn
    }


method delete {args} {
       return [eval $State(subWidgetName.list) delete {*}$args]
    }

method insert  {args} {
       return [eval $State(subWidgetName.list) insert {*}$args]
    }

method widgetcget {args} {
       if {[llength $args] != 2} {
         error "$widgetName cget subWidgetName -option"
       }
       set sbwid [lindex $args 0]
       set index subWidgetName.$sbwid
       set cmd [lrange $args 1 end]

       catch {$State($index) cget $cmd}  rtn
       return $rtn
       }

method widgetcommand {args} {
       return [DoWidgetCommand $widgetName {*}$args]
    }

method names {args} {
       if {[string match $args ""]} {
         set pattern subWidgetName.*
       } else {
         set pattern subWidgetName.$args
       }
       foreach n [array names State $pattern] {
         foreach {w s name} [split $n .] {}
           lappend names $name
```

```
            }
          return  $names
        }

    method subwidget {args} {
          set name [lindex $args 0]
          set index subWidgetName.$name
          if {[info exists State($index)]} {
            return $State($index)
          }
        }
}

# Example of use

option add *ScrolledLB*background #aaa

set slb [scrolledLB .slb -list.background white -list.height 4 -scroll.side 2]

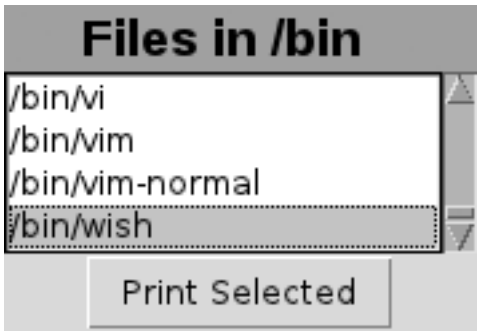
$slb widgetconfigure title -font {arial 14 bold} -text "Files in /bin"
puts "SLB: $slb"

foreach file [lsort [glob /bin/*]] {
  $slb insert end $file
}

grid .slb

button .b -text "Print Selected" -command "puts \[$slb selection\]"
grid .b
```

Script Output




```
/bin/wish
```

14.10.2 Using a Class Name as a Widget Type

One issue with this solution is that the procedure creates an object for a class that does not have the same name as the factory method—the `scrolledLB` procedure creates a `scrolledLB_OO` objct.

The normal way to create a TclOO object is

`className` `create` *`objectName ?args?`*

or

`className` `new` *`?args?`*

while the Tk widget create commands follow the pattern *`widgetType widgetName ?args?`*

As you might expect, there is a technique to allow a TclOO className to be used like a Tk widgetType. The technique is overriding the `unknown` static method.

Each TclOO class has a default method named `unknown`. The default method generates an error informing the user of acceptable commands:

Example 22

Script Example

```
oo::class create test {
}

test noSuchCommand
```

Script Output

```
unknown method "noSuchCommand": must be create, destroy or new
```

You can override the `unknown` class as described in Section 10.6 by adding your own static `unknown` method.

Section 10.6 explained that in order to add a class method, you need to define a `classmethod` procedure. A `classmethod` procedure looks like this:

Example 23

Script Example

```
proc ::oo::define::classmethod {name {args ""} {body ""}} {
   # Create the method on the class if the caller gave
   # arguments and body
   if {[llength [info level 0]] == 4} {
        uplevel 1 [list self method $name $args $body]
   }
   # Get the name of the class being defined
   set cls [lindex [info level -1] 1]
   # Make connection to private class "my" command by
   # forwarding
```

```
    uplevel forward $name [info object namespace $cls]::my $name
}
```

The new `unknown` method will examine the arguments and determine whether the first argument starts with a period. If so, then the illegal method is an attempt to create a new object. The `unknown` method can create the new object and return the name or else pass control to the default `unknown` method with the `next` command.

An `unknown` method will resemble this:

```
classmethod unknown {w args} {
  if {[string match .* $w]} {
    [self] new $w {*}$args
    return $w
  }
  next $w {*}$args
}
```

As with other compound widget patterns, we need to rename the window procedure to a new name that will not conflict with the name of the new object. This is done in the `constructor` by renaming the `widgetName` to a new string and then using the `self` command within the constructor to rename itself to `$widgetName`.

The critical part of the constructor looks like this:

```
constructor {widgetName args} {
  set State(parent) [frame $widgetName -class labelEntry]

  rename ::$widgetName ::$widgetName.frame
  rename [self] ::$widgetName
```

The next example shows a label/entry compound widget using this pattern. This widget has only a constructor and `configure` method. It uses a convention of identifying configuration options for the subwidgets as *`subwidgetName.option`*. To set the foreground of the label to black, the code would be `$megawidget configure -label.foreground black`

Example 24

Script Example

```
proc ::oo::define::classmethod {name {args ""} {body ""}} {
   # Create the method on the class if the caller gave
   # arguments and body
   if {[llength [info level 0]] == 4} {
        uplevel 1 [list self method $name $args $body]
   }
```

```
    # Get the name of the class being defined
    set cls [lindex [info level -1] 1]
    # Make connection to private class "my" command by
    # forwarding
    uplevel forward $name [info object namespace $cls]::my $name
}

oo::class create labelEntry {
  variable State
  superclass oo::class

  classmethod unknown {w args} {
    if {[string match .* $w]} {
      [self] new $w {*}$args
      return $w
    }
    next $w {*}$args
  }


  constructor {widgetName args} {

    set State(parent) [frame $widgetName -class labelEntry]

    rename ::$widgetName ::$widgetName.frame
    rename [self] ::$widgetName

    # A label named label and an entry widget named entry.
    label $State(parent).label
    entry $State(parent).entry

    # Grid the widgets in the frame
    grid $State(parent).label $State(parent).entry

    my configure {*}$args
  }

  method configure {args} {
    # Step through the args and apply them as required.
    # "-entry.width" becomes "parent.entry configure -width"

    foreach {k v} $args {
      lassign [split $k .] type opt
      set type [string trim $type -]
      catch {$State(parent).$type configure -$opt $v} err
    }
  }
```

```
}

# A sample for using this compound widget.

# Create two labelEntry widgets.  One for Author and one for Title
labelEntry .title -label.text "Title: " \
    -entry.textvariable State(title) -entry.width 23
labelEntry .author -label.text "Author: " \
    -entry.textvariable State(author)

# Grid the widgets,  author above title
grid .author
grid .title

# Assign values to be displayed in the entry widgets.
set State(title) "Tcl/Tk: A Developer's Guide"
set State(author) "Clif Flynt"

# Reconfigure the author label to be invert video, to demonstrate
#  the configure method.

.author configure -label.background black  -label.foreground white
```

Script Output

14.10.3 Callbacks into TclOO Widgets

When a Tk widget is created it creates the window and a command by the same name in the global scope. When a callback is invoked (as with a button click), this also happens in the global scope. In order to link a callback to an object's method, we need to use the `self` command that was discussed in Section 10.5.

The `self` command returns information about the call tree related to the current object. The most useful return value for a GUI programmer is that name of the current object. This name can be used to register a callback for a button, an `after` event, `fileevent`, etc.

The `my` command also returns information about the current object. The `my varname` returns a fully qualified name that can be used as a `textvariable` in a Tk widget.

The next example demonstrates how to build a TclOO class that creates a label that uses the object variable as a `-textvariable`, and a button that invokes an object method when clicked.

Example 25

Script Example

```
oo::class create OO_gui {

  variable objectVar

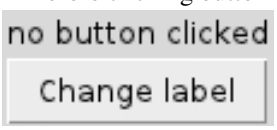
  constructor {} {
    set objectVar "no button clicked"
    label .l -textvariable "[my varname objectVar]"
    button .b -text "Change label" -command "[self] changeLabel"
    grid .l
    grid .b
  }

  method changeLabel {} {
    set objectVar "The button was clicked"
  }
}

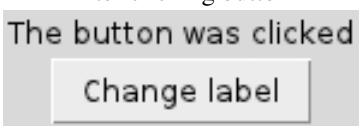
OO_gui new
```

Script Output

Before clicking button



After clicking button



14.10.4 Combining TclOO Compound Widgets

Multiple TclOO-based megawidgets can be combined into larger compound widgets by simply creating the megawidgets the same way you would include standard Tk widgets.

The next example demonstrates creating an `entryForm` widget using the previous `labelEntry` megawidget. Note that the `-textvariable` is defined using the `my varname` command and is passed to the `labelEntry` object as the value for the `-entry.textvariable` option.

The submit button uses the `self` command to invoke an object method.

Example 26

Script Example

```
proc ::oo::define::classmethod {name {args ""} {body ""}} {
   # Create the method on the class if the caller gave
   # arguments and body
```

```
    if {[llength [info level 0]] == 4} {
         uplevel 1 [list self method $name $args $body]
    }
    # Get the name of the class being defined
    set cls [lindex [info level -1] 1]
    # Make connection to private class "my" command by
    # forwarding
    uplevel forward $name [info object namespace $cls]::my $name
}

oo::class create labelEntry {
  variable State
  superclass oo::class

  classmethod unknown {w args} {
    if {[string match .* $w]} {
      [self] new $w {*}$args
      return $w
    }
    next $w {*}$args
  }


  constructor {widgetName args} {

    set State(parent) [frame $widgetName -class labelEntry]

    rename ::$widgetName ::$widgetName.frame
    rename [self] ::$widgetName

    # A label named label and an entry widget named entry.
    label $State(parent).label
    entry $State(parent).entry

    # Grid the widgets in the frame
    grid $State(parent).label $State(parent).entry

    my configure {*}$args
  }

  method configure {args} {
    # Step through the args and apply them as required.
    # "-entry.width" becomes "parent.entry configure -width"

    foreach {k v} $args {
      lassign [split $k .] type opt
      set type [string trim $type -]
```

```
      catch {$State(parent).$type configure -$opt $v} err
    }
  }
}

oo::class create entryForm {

  variable Values

  constructor { {parentFrame {}} } {
    set w [labelEntry $parentFrame.name -label.text "Name: " \
      -entry.textvariable [my varname Values(name)] -entry.width 15]
    grid $w

    set w [labelEntry $parentFrame.email -label.text "Email: " \
      -entry.textvariable [my varname Values(email)] -entry.width 15]
    grid $w

    set w [labelEntry $parentFrame.passphrase -label.text "Passphrase: " \
      -entry.textvariable [my varname Values(passphrase)] \
      -entry.show * -entry.width 15]
    grid $w

    button $parentFrame.b -text "Submit" -command "[self] submit"
    grid $parentFrame.b
  }

  method submit {} {
    set msg "Name: $Values(name)\nEmail: $Values(email)\nSecret: $Values(passphrase)"
    tk_messageBox -type ok -message $msg
  }
}

entryForm new
```

Script Output

Before clicking Submit button

After clicking Submit button

14.10.5 Adding New Functionality to a TclOO Widget

Needing to change the functionality of a widget is almost as common as needing to put multiple widgets into a single compound widget. The TclOO-inheritance model supports adding new functionality to a TclOO-based widget and overriding behaviors you want to change.

The `superclass` directive tells TclOO to inherit characteristics from another class and merge them into the current class. This lets you expand functionality by taking a previously designed class and adding new methods to it.

Control can be passed from one class to another using the `next` command. This allows you to use functionality in the existing class's methods while adding new functionality, or to completely replace one set of functionality with another. The `next` command can appear at any point in a method and will pass control to the method with the same name in a superclass. Look at the discussion of class hierarchies in Chapter 10 for more details on where the control may be placed.

The next example adds a validation support to the `labelEntry` compound widget. The new class is named `validatedLabelEntry`. It uses the `superclass` command to inherit the `labelEntry` functionality of creating a label and entry widget, providing support for the `configure` command and constructor.

The `validatedLabelEntry` class requires a `-validate` option when it is created. The value for this option can be any value supported by the `string is` command. The `string is` command is used to validate the contents of the entry widget after each keystroke.

There is no support for a `-validate` flag in the `labelEntry` class, so that option is tested and removed before the `labelEntry` constructor is invoked with `next`. Control returns to the `validatedLabelEntry` object after the label and entry widgets have been created. At this point, the `validatedLabelEntry` constructor can add a new binding to the *`parentWidget`*`.entry` widget to perform the validation.

The validation is performed in a method of the `validatedLabelEntry` class. The `bind` command in the constructor uses the `self` command to register the method as a script to invoke and adds the name of the entry widget to the script.

When a key is released, the `validatedLabelEntry` class's validate method is invoked. If the string in the entry widget is valid, it is configured for black letters on a white background. If it's not valid, the entry widget is configured for inverse video.

Example 27

Script Example

```
proc ::oo::define::classmethod {name {args ""} {body ""}} {
   # Create the method on the class if the caller gave
   # arguments and body
   if {[llength [info level 0]] == 4} {
        uplevel 1 [list self method $name $args $body]
   }
   # Get the name of the class being defined
   set cls [lindex [info level -1] 1]
   # Make connection to private class "my" command by
   # forwarding
```

```
    uplevel forward $name [info object namespace $cls]::my $name
}

oo::class create labelEntry {
  variable State
  superclass oo::class

  classmethod unknown {w args} {
    if {[string match .* $w]} {
      [self] new $w {*}$args
      return $w
    }
    next $w {*}$args
  }


  constructor {widgetName args} {

    set State(parent) [frame $widgetName -class labelEntry]

    rename ::$widgetName ::$widgetName.frame
    rename [self] ::$widgetName

    # A label named label and an entry widget named entry.
    label $State(parent).label
    entry $State(parent).entry

    # Grid the widgets in the frame
    grid $State(parent).label $State(parent).entry

    my configure {*}$args
  }

  method configure {args} {
    # Step through the args and apply them as required.
    # "-entry.width" becomes "parent.entry configure -width"

    foreach {k v} $args {
      lassign [split $k .] type opt
      set type [string trim $type -]
      if {[catch {$State(parent).$type configure -$opt $v} err]} {
        puts "FAILED: $State(parent).$type configure -$opt $v"
        puts $err
      }
    }
  }
}
```

```
oo::class create validatedLabelEntry  {

  variable State
  superclass labelEntry

  classmethod unknown {w args} {
    if {[string match .* $w]} {
      [self] new $w {*}$args
      return $w
    }
    next $w {*}$args
  }

  constructor {winName args} {
    if {[catch {dict get $args -validate} State(dataType)]} {
      error "validatedLabelEntry must include -validate\n\
      integer, alpha, graph, etc"
    }

    set args [dict remove $args -validate]
    next $winName {*}$args
    bind $winName.entry <KeyRelease> \
        "[self] validate $winName.entry"
  }

  method validate {entryWidget} {
     # Get the current contents of the entry widget
     set tmp [$entryWidget get]

     # If the string is valid, entry widget is black on white
     # If the string is not valid, entry uses reverse video
     if {[string is $State(dataType) $tmp]} {
       my configure -entry.background white
       my configure -entry.foreground black
     } else {
       my configure -entry.background black
       my configure -entry.foreground white
     }
  }
}

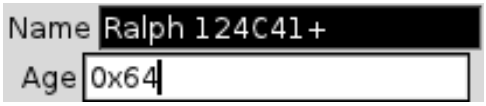
# An example of use with one alphabetic entry and
# one numeric entry.

validatedLabelEntry  .name -label.text Name -validate alpha
validatedLabelEntry  .age -label.text Age -validate integer
```

```
grid .name
grid .age
```

Script Output

The name `Ralph 124C41+` contains non-alphabetic characters, so it is in reverse video. The string `0x64` is a valid integer, so it is in normal video.



14.11 BOTTOM LINE

- This chapter has described the megawidgets included in the Tk distribution and shown how megawidgets can be constructed using Tk without any C language extensions using either namespaces or TclOO.
- The megawidgets included in the Tk distribution are:
  Syntax: `tk_optionMenu` *`buttonName varName val1 ?val2...valN?`*
  Syntax: `tk_chooseColor`
  Syntax: `tk_getOpenFile` *`?option value?`*
  Syntax: `tk_getSaveFile` *`?option value?`*
  Syntax: `tk_messageBox` *`?option value?`*
  Syntax: `tk_dialog` *`win title text bitmap default \`*
  `string1 ?... stringN?`
  Syntax: `tk_popup` *`menu x y ?entry?`*
- You can determine what windows (if any) have grabbed events with the `grab current` command.
  Syntax: `grab current` *`?window?`*
- You can determine whether a window that is grabbing events is grabbing them for the local task or all tasks with the `grab status` command.
  Syntax: `grab status` *`window`*
- A Tk window can be declared to be the recipient of all keyboard and mouse events with the `grab` command.
  Syntax: `grab set` *`?-global? window`*
- The `rename` command will change the name of a command.
  Syntax: `rename` *`oldName ?newName?`*
- You can define a default value for various options with the `option add` command.
  Syntax: `option add` *`pattern value ?priority?`*
- The `namespace current` command will return the full name of the current namespace.
  Syntax: `namespace current`
- The `namespace code` command will wrap a script so that it can be evaluated in the current space.
  Syntax: `namespace code` *`script`*
- A Tk script can be forced to wait for an event to occur with the `tkwait` command.

Syntax: `tkwait` *`EventType name`*

- A megawidget can be made modal with the `grab`, `focus`, and `tkwait` commands.
- You can construct your own megawidgets by providing a wrapper procedure that arranges widgets within a frame.
- The name of a frame can be disassociated from the name of the procedure used to configure the frame with the `rename` command.
- Megawidgets can be built using namespaces to conceal the internal structure of the megawidget.
- Megawidgets can be built using TclOO to conceal the internal structure of the megawidget. It is easier to retain state information with a TclOO class than a namespace because each widget gets a private state with a class but shares state with all widgets that use a namespace.
- Megawidgets can be nested within other megawidgets to support more complex interactions. This can be done with both namespace-based and TclOO-based megawidgets.
- TclOO-based megawidgets can inherit and extend functionality from other TclOO-based megawidgets.
- A TclOO-based megawidget needs to use the `self` command to register a method as a callback for a primitive widget, `bind`, `trace` or other command that registers a callback.
- A TclOO class uses the `my varname` to link an object variable to a widget (as with the `-textvariable` option).
- The `next` command transfers control to a superclass or mixin method with the same name as the current method. The `next` command can be used at any position in a method.

14.12 PROBLEMS

The following numbering convention is used in all Problem sections:

Number Range	Description of Problems
100–199	Short comprehension problems review material covered in the chapter. They can be answered in a few words or a 1–5-line script. These problems should each take under a minute to answer.
200–299	These quick exercises require some thought or information beyond that covered in the chapter. They may require reading a man page, or making a web search. A short script of 1–50 lines should fulfill the exercises, which may take 10–20 minutes each to complete.
300–399	Long exercises may require reading other material, or writing a few hundred lines of code. These exercises may take several hours to complete.

- *100* Which standard megawidget will allow a user to select an existing file from which to load data?
- *101* Which standard megawidget will allow a user to select a new file to which data can be written?
- *102* Which standard megawidget provides a generic dialog facility?

- *103* Can an application be written to use a Motif-style file selector on a Windows platform, instead of the native Windows-style file selector?
- *104* Which standard megawidget will give a user the option of selecting "Yes" or "No" as the answer to a question?
- *105* What two commands will follow this code fragment to:
  a. Configure the frame with a red background?
  b. Grid the frame in row 1, column 2?
  ```
  frame .f
  rename .f myFrame
  ```
- *106* What command would set the font for all buttons to {Times 24 bold}?
- *107* What command will force a `wish` script to pause until the window .l is destroyed?
- *108* If the button .b is clicked after the following code fragment has executed, will the application exit? If not, why?
  ```
  pack [entry .e -textvar tst]
  pack [button .b -text "EXIT" -command "exit"]
  update
  grab set .e
  ```
- *109* What option to grab will cause all events to be trapped by an application?
- *110* What command will transfer control to a superclass method?
- *111* What command will define an object variable as a label's `-textvariable`?
- *112* Can two megawidget objects of the same class be used in a compound widget?
- *200* Write a procedure that will accept a frame name as an argument, and will create a frame with that name and grid a text widget and scrollbar into the frame. The procedure should return the name of the text widget.
- *201* Using the TclOO pattern described in Section 14.10.1, construct a megawidget with a label, and a canvas below it.
- *202* Modify the megawidget constructed in the previous exercise to accept the following options.
  `-image imageName`
  An image to display in the canvas.
  `-title text`
  Text to display in the `label` widget.
- *203* Write a procedure that will create a new top-level window and wait for a user to click one of two buttons. The return from the procedure will be the text of the button that was clicked.
- *204* Modal widgets are commonly created in a new top-level window. Can a modal widget be created within a frame? Why or why not?
- *300* Use the pattern described in Section 14.10.2 to create a megawidget with a prompt, an entry widget, and a listbox. Allow a user to type a value into the `entry` widget, or select a value from

the listbox. When a listbox entry is selected, it should appear in the entry box. The widget should support the following widget procedures.

`insert value`
Insert a value into the listbox.
`get`
Get the content of the entry widget.

- *301* Expand the `labelEntry` example from Section 14.10.2 to include a `cget` method which will return the value of a configuration option for an internal widget. Use the *`-widgetName.optionName`* naming convention that was used for the `configure` command—i.e., the foreground attribute of the entry widget is identified as `-entry.foreground`.
- *302* Modify the `scrolledLB` widget from Section 14.5.4 from a namespace-based compound widget to be a TclOO class.
- *303* Modify the `filteredLB` megawidget from Section 14.7 to work as a TclOO-based widget. Use the TclOO-based `scrolledLB` widget reworked in Exercise 302.
- *304* The `filteredLB` compound widget can be created as an aggregate of two `scrolledLB` objects and a button. Use the `filteredLB` class from Exercise 303 as a superclass and create a file selector similar to the one described in Section 14.8.3.

CHAPTER

Extending Tcl 15

One of the greatest strengths of the Tcl package is the ease with which it can be extended. By adding your own extensions to the interpreter you can:

- Use compiled code to perform algorithms that are too computation-intensive for an interpreted language.
- Add support for devices or data formats that are not currently supported by Tcl.
- Create a rapid prototyping interpreter for an existing C language library.
- Add Tk graphics to an existing application or set of libraries.
- Create a script-driven test suite for an application or library.

Extensions can be linked with the Tcl library to create a new `tclsh` or `wish` interpreter, or they can be loaded into a running `tclsh` . Note that support for loading binary objects into a running `tclsh` requires a version of the Tcl library that is more recent than 7.5 and an operating system that supports dynamic runtime libraries. Microsoft Windows, Macintosh OS, Linux, and many flavors of UNIX support this capability.

Whether you are building a new `tclsh` or a loadable binary package, and whether you are linking to an existing library or are writing your own functions, the steps to follow are the same. This chapter describes the components of an extension, steps through creating a simple Tcl extension, discusses how to move data between the Tcl interpreter and your C language functions, and discusses how to handle complex data such as structures.

In Tcl version 8.0 a major modification was introduced to the Tcl internals. Prior to this (Tcl 7.x and earlier), all data was stored internally as a string. When a script needed to perform a math operation, the data was converted from string to native format (integer, long, float, double, and so on), the operation was performed, and then the native format data was converted back to a string to be returned to the script. This process caused Tcl scripts to run rather slowly.

With Tcl revision 8.0 and later, the data is stored internally in a `Tcl_Obj` structure. The `Tcl_Obj` structure contains a string representation of the data and a native representation of the data. The two representations are kept in sync in a lazy manner; the native and string representations are calculated when needed and retained as long as they are valid.

This change improved the speed of the Tcl interpreter, while adding minimal complexity for extension writers. Most of the Tcl library functions that interface with data now come in two forms: one to deal with old-style string data and one to deal with new-style `Tcl_Obj` data.

If you are writing a new extension, you can use all of the new `Tcl_Obj` -oriented commands (which have the word `Obj` in their names). If you need to link with an earlier version of Tcl (for instance, if you need to use an extension that exists only for Tcl 7.5), you will need to use the older version of the commands.

Tcl/Tk: A Developer's Guide. DOI: 10.1016/B978-0-12-384717-1.00015-4

This chapter covers both sets of commands. If you do not need to be able to link with an older version of Tcl, it is recommended that you use the `Tcl_Obj` objects for your data and the `Tcl_Obj` API to interface with the interpreter.

This chapter provides an overview of how to construct an extension, first from the functional viewpoint of what functionality is required in an extension and how the parts work together and then from the structural viewpoint of how the code modules should be assembled. This is followed by an example of a simple extension, and discussions of more advanced topics, such as using lists and arrays, and creating extensions with subcommands.

15.1 FUNCTIONAL VIEW OF A TCL EXTENSION

The interface between the Tcl library and an extension is implemented with several functions. On Windows systems these are documented under the Tcl Library Procedures entry in the Tcl Help menu. On UNIX systems, they are documented in files named *Tcl_*.3* in the *doc* directory. On the Macintosh these functions are documented under the *HTML DOCS* folder. The required commands are described in this chapter, but read the on-line documentation for more details.

The interface to the Tcl library includes several data types that are specific to Tcl. Those we will be dealing with are as follows.

`Tcl_Interp *`	A pointer to a Tcl interpreter state structure.
`Tcl_ObjCmdProc *`	A pointer to a function that will accept Tcl objects as arguments.
`Tcl_CmdProc *`	A pointer to a function that will accept strings as arguments.
`ClientData`	A one-word value that is passed to functions but never used by the Tcl interpreter. This allows functions to use data that is specific to the application, not the interpreter. For instance, this argument could be used to pass a pointer to a database record, a window object, or a C++ `this` pointer.

15.1.1 Overview

An extension must implement the following sets of functionality.

- Initialize any persistent data structures.
- Register new commands with the Tcl interpreter.
- Accept data from the Tcl interpreter.
- Process the new commands.
- Return results to the calling scripts.
- Return status to the calling scripts.

15.1.2 Initialize Any Persistent Data Structures

Your extension must include an initialization function that will be called by the Tcl interpreter when the extension is loaded into the interpreter. The code that performs the initialization will depend on your application.

15.1.3 Register New Commands with the Interpreter

You register a new command with the Tcl interpreter by calling a function that will insert the name of the new command into the Tcl command hash table and associate that name with a pointer to a function that will implement the command. There are two functions that can be invoked to register a new command with the Tcl interpreter. One will register the new command as a function that can use Tcl objects, and the other will register the new command as a function that requires string arguments. Extensions that are to be linked with versions of Tcl more recent than 8.0 should create Tcl object-based functions rather than string-based functions.

Syntax: `int Tcl_CreateObjCommand (interp, cmdName, func, clientData, deleteFunc)`

Syntax: `int Tcl_CreateCommand (interp, cmdName, func, clientData, deleteFunc)`

`Tcl_CreateObjCommand Tcl_CreateCommand`

These functions register a new command with the Tcl interpreter. The following actions are performed within the Tcl interpreter.

- Register *cmdName* with the Tcl interpreter.
- Define `clientData` data for the command.
- Define the function to call when the command is evaluated.
- Define the command to call when the command is destroyed.

`Tcl_CreateCommand` is the pre-8.0 version of this command. It will still work with 8.0 and newer interpreters but will create an extension that does not run as fast as an extension that uses `Tcl_CreateObjCommand`.

`Tcl_Interp *interp`

This is a pointer to the Tcl interpreter. It is required by all commands that need to interact with the interpreter state. Your extensions will probably just pass this pointer to Tcl library functions that require a Tcl interpreter pointer. Your code should not attempt to manipulate any components of the interpreter structure directly.

`char *cmdName`

The name of the new command, as a `NULL`-terminated string.

`Tcl_CmdProc *func`

`Tcl_ObjCmdProc *func`

The function to call when *cmdName* is encountered in a script. A pointer to a `Tcl_CmdProc *` is used with pre-8.0 interpreters, and a pointer to a `Tcl_ObjCmdProc` is used with recent interpreters.

`ClientData clientData`

A value that will be passed to *func* when the interpreter encounters *cmdName* and calls *func*. The `ClientData` type is a word, which on most machines is the size of a pointer. You can allocate memory and use this pointer to pass an arbitrarily large data structure to a function.

`Tcl_CmdDeleteProc *deleteFunc`

A pointer to a function to call when the command is deleted. If the command has some persistent data object associated with it, this function should free that memory. If you have no special processing, set this pointer to `NULL`, and the Tcl interpreter will not register a command deletion procedure.

These commands will return `TCL_OK` to indicate success or `TCL_ERROR` to indicate a failure. These are defined in the file `tcl.h`, which should be `#included` in your source.

```
Tcl_CreateObjCommand(interp, "demo", Demo_Cmd,
    (ClientData) NULL, NULL);
```

Modern Tcl extensions create the new commands in a namespace. To create new commands within a namespace, simply add the namespace identifier to the name of the command being added to the interpreter, as in the following example.

```
Tcl_CreateObjCommand(interp, "demoNamespace::demo", Demo_Cmd,
    (ClientData) NULL, NULL);
```

15.1.4 Accept Data from Tcl Interpreter

When a command is evaluated, the interpreter needs to pass data from the script to the function. The Tcl interpreter handles this with a technique similar to that used by the C language for receiving command line parameters. The function that implements the C command receives an integer argument count and an array of pointers to the Tcl script arguments. For example, if you register a command `foo` with the interpreter as

```
Tcl_CreateObjCommand(interp, "foo", fooCmd, NULL, NULL);
```

and later write a script command

```
foo one 1 two 2
```

`fooCmd` will be passed an argument count of 4 and an argument list that contains `one`, `1`, `two`, and `2`.

When the Tcl interpreter evaluates the Tcl command associated with a function in your extension, it will invoke that function with a defined set of arguments. If you are using Tcl 8.0 or newer and register your command using `Tcl_CreateObjCommand`, the Tcl command arguments will be passed to your function as an array of objects. If you are using an older version of Tcl or register your command using the `Tcl_CreateCommand` function, the arguments will be passed as an array of strings. The function prototype for the C function that implements the Tcl command will take one of these two forms:

Syntax: `int func(clientData, interp, objc, objv)`

Syntax: `int func(clientData, interp, argc, argv)`

`func`	The function to be called when `cmdName` is evaluated.
`ClientData clientData`	This value was defined when the command was registered with the interpreter.
`Tcl_Interp *interp`	A pointer to the Tcl interpreter. It will be passed to Tcl library commands that need to interact with the interpreter state.
`int objc`	The count of objects being passed as arguments.

`Tcl_Obj *objv[]`	An array of Tcl objects that were the arguments to the Tcl command in the script.
`int argc`	In pre-8.0 versions of Tcl, this is the count of argument strings being passed to this function.
`char *argv[]`	In pre-8.0 versions of Tcl, this is an array of strings. Each array element is an argument passed to the Tcl command. Versions of Tcl prior to 8.0 used strings instead of objects for arguments. The objects are much more efficient and should be used unless you have compelling reasons for maintaining compatibility with older code.

If you are using Tcl version 8.0 or newer and register the command with the `Tcl_CreateObjCommand` function, your function must accept an array of `Tcl_Obj` pointers. If you are using a version of Tcl older than 8.0, your function must accept arguments as an array of `char` pointers. In either case, the third argument will be an integer that describes the number of arguments in the array. Your function may return `TCL_OK`, `TCL_ERROR`, `TCL_RETURN`, `TCL_BREAK`, or `TCL_CONTINUE`. The most frequently used codes are `TCL_OK` and `TCL_ERROR`.

`TCL_BREAK`, `TCL_RETURN`, and `TCL_CONTINUE` are used for building program flow control commands such as looping or branching commands. Since Tcl already includes a complete set of looping and branching commands, if you think you need to implement a new flow control command you may want to look at your application very closely to see if you have missed an existing command you could use.

Converting Tcl Data to C Data

Once your function has been called, you will need to convert the data to the format your function (or the library calls it invokes) can use. In versions of Tcl earlier than 8.0, all data was stored as strings. Whenever data was needed in another format, it needed to be converted at that point, and the native format of the data was discarded when that use was finished.

With Tcl 8.0 and newer, the data is stored internally in a structure that maintains both a string and a native representation of the data. In this case, your function can use Tcl's support for extracting the values from the `Tcl_Obj` structure.

Data Representation

In order to convert the data, you need to know a little about how the Tcl interpreter stores data internally. With versions of Tcl that pass the arguments as strings, you can convert the data from the Tcl script to native (int, float, and so on) format with the standard library calls `sscanf`, `atof`, `atol`, `strtod`, `strtol`, `strtoul`, and so on.

As of Tcl 8.0, the Tcl interpreter stores data in `Tcl_Obj` structures. The `Tcl_Obj` structure stores its data in two forms: a string representation and a native representation. The data in a `Tcl_Obj` object is accessed via a set of function calls that will be discussed later in this section.

It is not necessary to know the details of the `Tcl_Obj` structure implementation, because all interactions with a Tcl object will be done via Tcl library functions, but you will understand the function calls better if you understand the design of the `Tcl_Obj` structure. The two primary design features of the Tcl object are the data's dual-ported nature (the string and native representations) and the interpreter's use of references to objects rather than copies of objects.

A `Tcl_Obj` structure maintains a string representation of the data as well as the native representation. The conversion between these two representations of the data (the dual nature of the object) is handled in a lazy manner. The data is converted between formats only when necessary. When one representation of the data is modified, the other representation is deleted to indicate that it must be regenerated when needed again. For example, in the commands:

```
set x "12"
```

The Tcl interpreter creates an object and defines the string representation as 12.

```
puts "The value of X is: $x"
```

The `puts` command does not require a conversion to native mode, so the string representation of `$x` is displayed and no conversion to integer is made.

```
incr x 2
```

The string representation of the data in the `$x` object is now converted to native (integer) representation, and the value 2 is added to it. The old string representation is cleared to show that the native representation is valid.

```
set y [expr $x+2]
```

When `$x` is accessed by the `expr` command, the object's native representation is used. After the addition is performed, a new object is created to hold the result. There is no need to convert the integer representation of `x` into a string, so it will be left blank. This new object is assigned to the variable `y`, with a native representation (integer) but no string representation defined.

```
puts "The value of Y is: $y"
```

When the `puts` is evaluated, the Tcl script needs a string representation of the object, the integer is then converted to a string, and the string representation is saved for future use.

If the value of `$x` were only displayed and never used for a calculation (as a report generator would treat the value), it would never be converted to integer format. Similarly, if the value of `$y` were never displayed (simply used in other calculations), the conversion to a string would never be made. Since most applications deal with data in native or string representation in batches, this lazy conversion increases the speed of the Tcl interpreter.

Converting data from one format to another is referred to as *shimmering*. These conversions can slow a program. Avoiding such behavior will improve program speed slightly, but are usually not the limiting factor in program speed.

The reference counts associated with Tcl objects allow the Tcl interpreter to maintain pointers to objects, instead of making copies of any objects referenced in more than one location. This lets the interpreter maintain a smaller number of objects than would otherwise be necessary. When an object is created, it is assigned a reference count of 0.

When a Tcl object is referenced by other objects (for example, the object is associated with a variable), the reference counter is incremented. The reference counter will be incremented by one when a variable is declared as the `-textvariable` for a `label` or passed as an argument to a procedure. When an object that references another Tcl object is destroyed (with the `unset` command or by destroying an associated widget), the reference count for the associated object is decremented by one. If the reference count becomes 0 or less, the object is deleted and its memory is returned to the heap.

Obtaining the Data

Since arguments are passed to your function in an array of pointers, the function can access data as `argument[0]`, `argument[1]`, and so on. For example, if you register your command using the `Tcl_CreateCommand` function, you could print out the arguments to a function with code such as the following.

```
int myFunc(ClientData data, Tcl_Interp *interp,
           int argc, char **argv) {
  int i;
  for (i=0; i<argc; i++) {
    printf("argument %d is %s\n", i, argv[i]);
  }
}
```

Recent versions of Tcl pass the data as an array of pointers to Tcl objects. In that case you need to use the Tcl conversion functions to extract either the string or native representation of the data.

Syntax: `int Tcl_GetIntFromObj(`*`interp, objPtr, intPtr`*`)`

`Tcl_GetIntFromObj`	Retrieve an integer from the object.
`Tcl_Interp *`*`interp`*	A pointer to the Tcl interpreter.
`Tcl_Obj *`*`objPtr`*	A pointer to the object that contains an integer.
`int *`*`intPtr`*	A pointer to the integer that will receive this data.

Syntax: `int Tcl_GetDoubleFromObj(`*`interp, objPtr, dblPtr`*`)`

`Tcl_GetDoubleFromObj`	Retrieve a double from the object.
`Tcl_Interp *`*`interp`*	A pointer to the Tcl interpreter.
`Tcl_Obj *`*`objPtr`*	A pointer to the object that contains a double.
`Double *`*`dblPtr`*	A pointer to the double that will receive this data.

The `Tcl_GetIntFromObj` and `Tcl_GetDoubleFromObj` functions return a `TCL_OK` if they successfully extract a numeric value from the object, or return `TCL_ERROR` if they cannot generate a numeric value for this object. (For instance, if the object contains an alphabetic string, there is no integer or floating-point equivalent.)

Tcl data is always available as a string. When you need to get the string representation of an object's data, you use the `Tcl_GetStringFromObj` command.

Syntax: `char *Tcl_GetStringFromObj (`*`objPtr, lengthPtr`*`)`

`Tcl_GetStringFromObj`	Retrieve a byte string from the object.
`Tcl_Obj *`*`objPtr`*	A pointer to the object that contains a string.
`int *`*`lengthPtr`*	A pointer to an `int` that will receive the number of characters in this data. If this value is `NULL`, the length will not be returned.

Note that the `Tcl_GetStringFromObj` function does not follow the format of the previous `Get` functions. It returns the requested data as a `char` pointer rather than returning a status. Since the data

is always available in a string format, this command can fail only if the object pointer is invalid, in which case the program has other problems and will probably crash.

The `Tcl_GetStringFromObj` command can place the number of valid bytes in the `char` pointer in an integer pointer (`lengthPtr`). The data in a string may be binary data (accessed via the Tcl `binary` command, for instance), in which case the length pointer is necessary to track the number of bytes in the string.

15.1.5 Returning Results

The function implementing a Tcl command can return results to a script in the following ways.

- Return a single value as the result.
- Return a list of values as the result.
- Modify the content of a script variable named as an argument.
- Modify the content of a known script variable.

If you desire to have your function return a value to the script as the result of evaluating the command (which is how most functions return their results), your code must call a function that sets the return to be a particular value. In old versions of Tcl, this can be done by passing a string. The modern Tcl interpreters require your code to create a new `Tcl_Obj`. The object created within the function will have a reference count of 0. If the return value from the function is assigned to a variable, the object's reference count will be incremented; otherwise, the object will be deleted when the command has finished being evaluated. You can create a new object with one of the following commands.

Syntax: `Tcl_Obj *Tcl_NewIntObj (intValue)`

Syntax: `Tcl_Obj *Tcl_NewDoubleObj (dblValue)`

Syntax: `Tcl_Obj *Tcl_NewStringObj (bytes, length)`

`Tcl_NewIntObj`	Create a new Tcl object with an integer value.
`Tcl_NewDoubleObj`	Create a new Tcl object with a double value.
`Tcl_NewStringObj`	Create a new Tcl object from a byte string.
`intValue`	The integer to assign to the new object.
`dblValue`	The double to assign to the new object.
`bytes`	An array of bytes to copy to the new object.
`length`	The number of bytes to copy to new object. If this is a negative value, all bytes up to the first `NULL` byte will be copied.

In Tcl earlier than 8.0, since all variables were maintained as strings, a function could create an ASCII string to return with the `sprintf` command or could modify data in an existing string with the standard string library commands. After a return value has been created, it can be assigned with one of the following commands.

Syntax: `void Tcl_SetObjResult (interp, objPtr)`

Syntax: `void Tcl_SetResult(`*`interp, string, freeProc`*`)`

`Tcl_SetObjResult`	Makes the Tcl interpreter point to *`objPtr`* as the result of this function. If this function has already had a result object defined, that object will be replaced by the object pointed to by `objPtr`. This function should be used with Tcl 8.0 and more recent versions.
`Tcl_SetResult`	Copies the string into the result string for this function, replacing any previous string that was there. This function is for use with pre-8.0 versions of Tcl.
`Tcl_Interp *`*`interp`*	A pointer to the Tcl interpreter.
`Tcl_Obj *`*`objPtr`*	A pointer to the object that will become the result.
`char *`*`string`*	A string to copy to the object.
`freeProc`	The name of a procedure to call to free the memory associated with the string when this object is destroyed. Must be one of:
	`TCL_STATIC`
	The string was defined in static memory and will remain constant.
	`TCL_DYNAMIC`
	The memory for the string was allocated with a call to `Tcl_Alloc`. It will be returned to the Tcl memory pool.
	`TCL_VOLATILE`
	The string was allocated from a nonpersistent area of memory (probably declared on the call stack) and may change when the function exits. In this case, the Tcl interpreter will allocate a safe space for the string and copy the memory content.

If the new command needs to return several pieces of information, you may prefer to pass the command the name of one or more Tcl variables to place the results in. This is similar to passing pointers to a C function. In this case, the code will modify the content of an existing Tcl variable.

The most generic function for modifying a Tcl variable is `Tcl_SetVar`. This function will accept the name of a variable and will either modify an existing variable or create a new variable by that name. The value to be assigned is passed to this function as a string, which can be easily obtained from a Tcl object, or generated as needed.

Syntax: `char *Tcl_SetVar(`*`interp, varName, newValue, flags`*`)`

`Tcl_SetVar`	Assign a value to a variable. Create a new Tcl variable if necessary.
`interp`	The pointer to the Tcl interpreter.
`varName`	A NULL-terminated string containing the name of the variable.

newValue	A NULL-terminated string containing the value to be assigned to this variable.
flags	One or more or'd-together flags to fine tune the behavior of the assignment. By default (0), the value is assigned as a string to a variable in the current scope when the command is invoked. The flags may be one or more of:
TCL_GLOBAL_ONLY	When Tcl tries to resolve the name to a Tcl variable, it will look in the global scope, instead of the local scope.
TCL_NAMESPACE_ONLY	Tcl looks for a variable defined only within the current namespace.
TCL_LEAVE_ERR_MSG	If this is set and an error occurs, the error message is left in the interpreter's result string. The error message can be retrieved with the `Tcl_GetObjResult` or `Tcl_GetStringResult` function.
TCL_APPEND_VALUE	Setting this bit causes the new value to be appended to the original data in the variable.
TCL_LIST_ELEMENT	If this flag is set, the new value is converted to a valid Tcl list element before being assigned (or appended) to the variable.

If your code has access to a Tcl object, it can assign a native format value to the variable with one of the following commands.

Syntax: `void Tcl_SetIntObj (objPtr, intValue)`

Syntax: `void Tcl_SetDoubleObj (objPtr, dblValue)`

Tcl_SetIntObj	Set the value of the integer representation of an object. If the object was not already an integer type, it will be converted to one if possible.
Tcl_SetDoubleObj	Set the value of the integer representation of an object. If the object was not already a double type, it will be converted to one if possible.
Tcl_Obj *objPtr	A pointer to the object that will contain the new value.
int intValue	The integer to assign to this object.
double dblValue	The double to assign to this object.

An object's string representation can be modified in several ways, either completely replacing one byte string with a new byte string, appending a single string, appending a string from another object, or appending a list of strings.

Syntax: `void Tcl_SetStringObj (objPtr, bytes, length)`

Syntax: `void Tcl_AppendToObj (objPtr, bytes, length)`

Syntax: `void Tcl_AppendObjToObj (objPtr, appendObjPtr)`

Syntax: `void Tcl_AppendStringsToObj (objPtr, string, ...,NULL)`

`Tcl_SetStringObj`	Redefine the string value of an object.
`Tcl_AppendToObj`	Append a string to the string representation of the data in an object.
`Tcl_AppendObjToObj`	Append the string representation of the value of one object to the string currently in an object.
`Tcl_AppendStringsToObj`	Append one or more strings to the string currently in an object.
`Tcl_Obj *objPtr`	A pointer to the object that contains the string.
`char *bytes`	An array of bytes to copy to the object.
`int length`	The number of bytes to copy to the new object. If this is a negative value, all the bytes up to the first `NULL` byte will be copied.
`Tcl_Obj *appendObjPtr`	A pointer to an object that contains a string to be appended.
`char *string`	A `NULL` -terminated string of characters. You may not use a binary string with the `Tcl_AppendStringsToObj` command.

15.1.6 Returning Status to the Script

When a function returns execution control to the Tcl interpreter, it must return its status. The status should be either `TCL_OK` or `TCL_ERROR`. If the function returns `TCL_OK`, the object defined by `Tcl_SetObjResult` will be returned to the script, and script will continue execution. If the function returns `TCL_ERROR`, an error will be generated, and unless your script is trapping errors with the `catch` command the execution will stop and error messages will be returned.

By default, the error messages returned will be the Tcl call stack leading to the Tcl command that caused the error. You may want to add other information to this, such as why a file write failed, what database seek did not return a value, or what socket can no longer be contacted. You can add more information to that message by invoking the function `Tcl_AddErrorInfo`, `Tcl_SetErrorCode`, `Tcl_AddObjErrorInfo`, or `Tcl_SetObjErrorCode`.

Syntax: `void Tcl_AddErrorInfo (interp, message)`

Syntax: `void Tcl_SetErrorCode (interp, element1, element2 ...NULL)`

Syntax: `void Tcl_AddObjErrorInfo (interp, message, length)`

Syntax: `void Tcl_SetObjErrorCode (interp, objPtr)`

`Tcl_AddObjErrorInfo`	Append additional text to the information object. This information object can be accessed within the Tcl script as the global variable `errorInfo`.

`Tcl_AddErrorInfo`	Append additional text to the information to be returned to the script. This function should be used with versions of Tcl before 8.0.
`Tcl_SetObjErrorCode`	Set the `errorCode` global variable to the value contained in the Tcl object. If the object contains a list, the list values are concatenated to form the return. By default `errorCode` will be `NONE`.
`Tcl_SetErrorCode`	Set the `errorCode` global variable to the value of the concatenated strings.
`Tcl_Interp *interp`	A pointer to the Tcl interpreter.
`char *message`	The message to append to the `errorInfo` global variable. This string will have a `newline` character appended to it.
`Tcl_Obj *objPtr`	A pointer to the object that will contain the error code.
`char *element`	A `NULL`-terminated ASCII string representation of a portion of the error code.

If a system error occurs, your function can invoke `Tcl_PosixError` to set the `errorCode` variable from the C language global `errno`. The behavior of this command varies slightly for different platforms. You should check the on-line documentation for your platform and Tcl revision before using it.

15.1.7 Dealing with Persistent Data

There are circumstances in which an extension needs to maintain a copy of some persistent data separate from the script that is being evaluated, while allowing the script to describe the piece of data with which it needs to interact. For instance, a file pointer must be maintained until the file is closed, and a Tcl script may have several files open simultaneously, accessing one file and then another.

If your extension's data requirements are simple, it may be sufficient to allocate an array of items and assign them to scripts as necessary. For instance, if you write an extension that interfaces with a particular piece of hardware and there will never be more than one of these devices on a system, you may declare the control structure as a static global in your C code.

For more complex situations, the Tcl library includes functions that let you use a Tcl hash table to store key and value pairs. Your extension can allocate memory for a data structure, define a key to identify that structure, and then place a pointer to the structure in the hash table, to be accessed with the key. The key may be an alphanumeric string that can be returned to the Tcl script. When a Tcl script needs to access the data, it passes the key back to the extension code, which then retrieves the data from the hash table.

The Tcl hash table consists of a `Tcl_HashTable` structure that is allocated by your extension code and `Tcl_HashEntry` structures that are created as necessary to hold key/value pairs. You must initialize the hash table before using it with the functions that access hash table entries. Once the table is initialized, your code can add, access, or delete items from the hash table.

Syntax: `void Tcl_InitHashTable (tablePtr, keyType)`

`Tcl_InitHashTable`	Initializes the hash table.
`Tcl_HashTable *tablePtr`	A pointer to the `Tcl_HashTable` structure. The space for this structure must be allocated before calling `Tcl_InitHashTable`.
`int keyType`	A Tcl hash table can use one of three different types of keys to access the data saved in the hash table. The acceptable values for `keyType` are:
`TCL_STRING_KEYS`	The hash table will use a `NULL`-terminated string as the key. This is the most commonly used type of hash key. The string representation of a `Tcl_Obj` object can be used as the key, which makes it simple to pass a value from a script to the Tcl hash table functions.
`TCL_ONE_WORD_KEYS`	The hash table will use a single-word value as the key. Note that if the word is a pointer, the value of the pointer is used as the key, not the data that the pointer references.
`positiveInteger`	If a positive integer is used as the `keyType`, the key will be an array of the number of integers described. This allows complex binary data constructs to be used as keys. Note that the constructs used as keys must be the same size.

Once a hash table has been initialized, the `Tcl_CreateHashEntry`, `Tcl_FindHash Entry`, and `Tcl_DeleteHashEntry` commands can be used to create, query, or remove entries from the hash table.

Syntax: `Tcl_HashEntry *Tcl_CreateHashEntry(tablePtr, key, newPtr)`

Syntax: `Tcl_HashEntry *Tcl_FindHashEntry(tablePtr, key)`

Syntax: `void Tcl_DeleteHashEntry(entryPtr)`

`Tcl_CreateHashEntry`	Allocates and initializes a new `Tcl_HashEntry` object for the requested `key`. If there was a previous entry with this `key`, `newPtr` is set to `NULL`.
`Tcl_FindHashEntry`	This function returns a pointer to the `Tcl_HashEntry` object that is associated with the `key` value. If that `key` does not exist in this hash table, this function returns `NULL`.
`Tcl_DeleteHashEntry`	This removes a hash table entry from the table. After this function has been called, `Tcl_FindHashEntry` will return a `NULL` if used with this `key`. This does not destroy data associated with the hash entry. Your functions that interact with the hash table must do that.

`Tcl_HashTable *tablePtr`	A pointer to a Tcl hash table. This table must be initialized by `Tcl_InitHashTable` before being used by these commands.
`char *key`	The *key* that defines this entry. This value must be one of the types described in the `Tcl_InitHashTable` call.
`int *newPtr`	This value will be 1 if a new entry was created, and 0 if a *key* with this value already existed.
`Tcl_HashEntry *entryPtr`	A pointer to a hash table entry.

A `Tcl_HashEntry` contains the key value that identifies it and a `ClientData` data object. The `ClientData` type is a word-sized object. On most modern machines, this is the same size as a pointer, which allows you to allocate an arbitrary data space and place the pointer to that space in a `Tcl_HashEntry`. You can manipulate a `Tcl_HashEntry` object with the `Tcl_SetHashValue` and `Tcl_GetHashValue` commands.

Syntax: `ClientData TclGetHashValue (entryPtr)`

Syntax: `void TclSetHashValue (entryPtr, value)`

`TclGetHashValue`	Retrieve the data from a `Tcl_HashEntry`.
`TclSetHashValue`	Set the value of the data in a `Tcl_HashEntry`.
`Tcl_HashEntry *entryPtr`	A pointer to a hash table entry.
`ClientData value`	The value to be placed in the `clientData` field of the `Tcl_HashEntry`.

The following code will create a hash table, add an entry, and retrieve the data.

Example 1

Code Example

```
void hashSnippet () {

    // The hash table pointer
  Tcl_HashTable *hashTable;

    // firstEntry will point to a hash entry we create
  Tcl_HashEntry *firstEntry;
    // secondEntry will point to a hash entry extracted from
    //  the table
  Tcl_HashEntry *secondEntry;

  int isNew;
  char *insertData, *retrievedData;
  char *key;

  key = "myKey";
  insertData = "This is data in the hash table";
```

```
  // Allocate the space and initialize the hash table

  hashTable = (Tcl_HashTable *)
      malloc(sizeof(Tcl_HashTable));
  Tcl_InitHashTable(hashTable, TCL_STRING_KEYS);

  // Get a hash Entry for the key, and confirm it is a
  // new key

  firstEntry = Tcl_CreateHashEntry(hashTable, key, &isNew);

  if (isNew == 0) {
    printf("Bad key - this key already exists!");
    return;
  }

  // Define the value for this entry.

  Tcl_SetHashValue(firstEntry, insertData);

  /* ------------------------------------------------------
   * At this point, the data has been placed in the hash
   * table. In an actual application, these sections of
   * code would be in separate functions.
   * ------------------------------------------------------*/

  // Retrieve the hash entry with the key.

  secondEntry = Tcl_FindHashEntry(hashTable, key);
    if (secondEntry == (Tcl_HashEntry *) NULL) {
    printf("Failed to find %s\n", key);
    return;
  }

  // Extract the data from the hash entry

  retrievedData = (char *)Tcl_GetHashValue(secondEntry);

  // Display the data, just to prove a point

  printf("Retrieved this string from hashTable: \n%s\n",
    retrievedData);
}
```

Script Output

```
Retrieved this string from hashTable:
This is data in the hash table
```

15.2 BUILDING AN EXTENSION

The bulk of your extension may be a library of code that has already been developed and tested, a library you need to test and validate, or a set of code you will write for this specific extension. In each of these cases, you will use the functions described previously to connect the application code to the interpreter. The next step is the mechanics of constructing the extension.

15.2.1 Structural Overview of an Extension

An extension consists of one or more source code files, one or more include files, and a `Makefile` or VC++, Borland, or CodeWarrior Project files. The source code files must contain a function with initialization code, and that function must conform to the naming conventions described in Section 15.2.2.

The source code files will include at least one function that adds new commands to the interpreter and at least one function to implement the new commands. A common structure is to create two files: one with the initialization function that adds the new commands to the interpreter and a second file with the functions that implement the new commands.

All code that uses functions from the Tcl library will need to include `tcl.h`. The `tcl.h` file has all the function prototypes, `#define`, data definitions, and so on required to interact with the Tcl library functions.

15.2.2 Naming Conventions

There are a few naming conventions involved with writing a Tcl extension. Some of these are required in order to interact with the Tcl interpreter, and some are recommended in order to conform to the appearance of other Tcl extensions. If you write your extension to conform to the recommended standards, it will be easier for your extension to be used and maintained by others.

These conventions are described in the *Tcl/Tk Engineering Manual*, and the *Tcl Extension Architecture* (TEA) guide, found on the companion website. They can also be acquired from *http://www.tcl.tk/doc/*. Most of the conventions mentioned in these documents are discussed in this chapter, but you should check the source documents for more details. It may save you from having to rewrite your code later.

In the following tables, you should replace the string *`ExtName`* with the name of your extension. Note the capitalization, which is part of the naming convention.

Function Names

The extension initialization function is expected to have a specific name in order for it to be automatically found by the Tcl extension loading commands.

Function Name	Description
`ExtName``_Init`	This function is required. It initializes an extension by creating any persistent data objects and registering new commands with the Tcl interpreter. This entry point is used to initialize the extension when a DLL (Dynamic Link Library) or shared library is loaded. The capitalization is important. For example,

	for an extension named `ext` the `Init` function would be `Ext_Init`.
ExtName`_AppInit`	This function is called to initialize a stand-alone `tclsh` interpreter with the extension compiled into it. This is not needed when you compile a loadable extension.
extNameCommand`_Cmd`	These entry points are optional, but this is the naming convention used for the C code that will be invoked when the command *command* is evaluated by a Tcl script. For example, if the extension named `ext` implements the Tcl command `foo`, you would put the code implementing the `foo` command in the function `extFoo_Cmd`.

File Names

There are certain conventions followed in Tcl to make it easier for other maintainers to work with your code. Your extension will still work if you do not follow these guidelines, but consistency is a good thing.

File Names	Description
extName`Int.h`	This file is required. It will contain the `#include` statements, `#define` statements, data structure definitions, and function prototypes that are used by the code in this extension. This is for the package's internal use. This file will be included by all extension code files.
extName`.h`	This file is optional. If your extension includes a library with a C interface, the external API definitions should be in this file.
extName`.c`	This file is recommended. It will contain the C language functions that implement the extension. If your extension is small, it may also include the *ExtName*`_Init` function.
extName`_Init.c`	This file is optional. If your extension is medium sized, you may use a file named like this for the *ExtName*`_Init` function, and put the functions that implement the extension into the *extName*`.c` file, or further subdivide the extension as shown in the following.
extName`Cmd.c`	This file is optional. If the code to interface between Tcl and the C code is large, you may want to separate the code that creates the new commands in the Tcl interpreter from the *extName*`.c` file and place that code in a separate file.
extName`CmdAL.c` *extName*`CmdMZ.c`	These files are optional. If you have a truly large extension, it can make the code easier to follow if you split the functions that implement the commands into smaller files. One convention used for this is to put the commands that start with the letter A through some other letter in one file, and those that start with a character after the breakpoint letter in another file.

extNameCommand.c	These files are optional. If your extension has commands that accept a number of subcommands, or if the command is implemented with several functions, it can make the code easier to follow if you split the functions that implement a command into a separate file. Thus, the functions that implement command `foo` would be in `extFoo.c`, whereas those that implement command `bar` would be in `extBar.c`.
extName.dll *extName*.lib *extName*.so *extName*.shlb *extName*.sl *extName*CFM68K.shl	A file in one of these formats will be created for your extension when you have completed compiling your extension. The Tcl interpreter will use the `extName` part of the extension to find the default *ExtName*_`Init` function.

If you cannot follow this naming convention for the extension loadable library file, you can force the Tcl interpreter to find the initialization function by declaring the extension name in the `load` command as follows.

```
load wrongname.dll myextension
```

Not following the naming convention for the extension library will make it impossible to use `pkg_mkIndex` to construct a `tclIndex` file, but you can build the index line with a text editor if necessary.

Directory Tree

The *Tcl Extension Architecture* (TEA) document defines a directory tree for Tcl extensions. Again, you do not need to follow these guidelines to make a working extension, but it will be easier to maintain and port your package if you follow them. The templates and skeletons included with the TEA materials on the companion website (also available at *www.tcl.tk*) can make writing an extension easier. Your extension

allows simple relative addressing to be used by the `makefiles` to find the appropriate files.

```
%>ls sources
myExtension tcl8.6 tk8.6
```

The following files should be placed at the top level of your directory.

`README`	A short discussion of what the package does and what the user may expect to find in this directory.
`license`	The distribution license for this package. Tcl is distributed with the Berkeley license, which is very open. You may elect to use the same licensing for your extension, the more restrictive GNU Copyleft, or your corporation's license agreement.
`changes`	A list of changes that have happened in the package. This should tell a user what to expect when they upgrade to a new version.

There will be several subdirectories under the main directory. These may include the following.

`generic`	This directory will contain C source code files that are not platform dependent. The *`extName`*`Int.h` file and *`extName`*`_Init.c` files, and the files that implement the extension functionality, should be here.
`unix`	Any UNIX-specific files should be in this directory. If there are platform-specific functions (perhaps using system libraries), a copy of the functions for the UNIX systems should be here. A UNIX-compatible `Makefile` or `configure` file should be included in this directory.
`win`	Any MS Windows specific files should be in this directory. An `nmake`-compatible `Makefile` or the VC++ or Borland project files should be included.
`mac`	Any Macintosh-specific files should be in this directory. A Mac-specific `Makefile` or CodeWarrior project files should also be included here.
`compat`	If your extension requires certain library features that may not exist, or are broken on some platforms, put source code files that implement the library calls in this directory.
`doc`	The documentation files should be put here. The standard for documenting Tcl packages is to use the UNIX `nroff` macros used with the main Tcl documentation. These can be converted to machine-native formats as necessary.
`tests`	It is good policy to have a set of regression tests to confirm that the package works as expected.
`library`	If your package has Tcl scripts associated with it, they should go here.

15.3 AN EXAMPLE

This section constructs a simple extension that will demonstrate the mechanisms discussed in this chapter. On the companion website you will find this `demo` code, and a `dummyTclExtension` kit with a Tcl script that will create a skeleton extension similar to the `demo` extension. This example follows the coding standards in the *Tcl/Tk Engineering Manual*, which is included on the companion website. Where the manual does not define a standard, it is noted that this convention is the author's, not the Tcl standard. Otherwise, the naming conventions and so on are those defined by the Tcl community.

This `demo` extension does not perform any calculations. It just shows how an extension can be constructed and how to acquire and return data using the hash functions. The `demo` package implements one Tcl command and five subcommands. The subcommands are `debug`, `create`, `get`, `destroy`, and `set`.

Syntax: `demo debug` *`level`*

`demo debug`	Sets a static global variable in the C code that turns on or off internal debug output.
`level`	The level to assign to this variable. A value of zero will disable the debugging output, and a non-zero value will enable that output.

Syntax: `demo create` *`key raw_message`*

`demo create`	Create a hash table entry.
`key`	The key for this hash table entry.
`raw_message`	The string to assign to this hash.

Syntax: `demo get` *`key`*

`demo get`	Retrieves the value of an entry from the hash table and returns the string saved with that key.
`key`	The key for the entry to return.

Syntax: `demo destroy` *`key`*

`demo destroy`	Deletes an entry in the hash table.
`key`	The key that identifies which entry to delete.

Syntax: `demo set` *`arrayName index Value`*

`demo set`	An example of how to set the values in an associative array. It sets the requested index to the requested value and sets the index `alpha` of the array to the value `NewValue`. The index `alpha` and value `NewValue` are hard-coded in the procedure `Demo_SetCmd`. This subcommand executes the C code equivalent of
	`set` *`arrayName`*`(`*`index`*`)` *`Value`*
	For example, the command
	`demo set myArray myIndex "my Value"`
	is equivalent to
	`set myArray(myIndex) "my Value"`
	Both commands will return the string `"my Value"`.

`arrayName`	The name of a Tcl array. It need not exist before calling this subcommand.
`index`	An index into the array. This index will have `Value` assigned to it.
`Value`	The value to assign to *`arrayName`*`(`*`index`*`)`.

The `demo` extension is arranged in the style of a large package with many commands, to show how the functionality can be split across files and functions. It consists of the following files.

`demoInt.h`	This include file has the definitions for the structures used by this extension and the function prototypes of the functions defined in `demoInit.c`, `demoCmd.c`, and `demoDemo.c`.
`demoInit.c`	This file contains the `Demo_Init` function.
`DemoCmd.c`	This file contains the `Demo_Cmd` function, which is invoked when a `demo` Tcl command is evaluated in a script. This file also includes the descriptions of the subcommands associated with the `demo` command.
`demoDemo.c`	This file contains the functions that implement the `demo` subcommands.

15.3.1 demoInt.h

The demoInt.h include file will be included by all the source files in the demo extension. It includes the version number information for this extension, the include files that will be used by other functions, some magic for Microsoft VC++, definition of data structures used by the demo extension, and the function prototypes.

Example 2
demoInt.h

```
 /*
  * demoInt.h --
  *
  * Declarations used by the demotcl extension
  *
  */
#ifndef _DEMOINT
#define _DEMOINT
 /*
1
  * Declare the #includes that will be used by this extension
  */
#include <tcl.h>
#include <string.h>
 /*
2
  * Define the version number.
  * Note: VERSION is defined as a string, not integer.
  */
#define DEMO_VERSION "1.0"
 /*
3
  * VC++ has an alternate entry point called
  * DllMain, so we need to rename our entry point.
  */
#if defined(__WIN32__)
# define WIN32_LEAN_AND_MEAN
# include <windows.h>
# undef WIN32_LEAN_AND_MEAN
# if defined(_MSC_VER)
# define EXPORT(a,b) __declspec(dllexport) a b
# define DllEntryPoint DllMain
# else
# if defined(__BORLANDC__)
#   define EXPORT(a,b) a _export b
# else
#   define EXPORT(a,b) a b
# endif
```

```
# endif
#else
# define EXPORT(a,b) a b
#endif
 /*
4
  * The CmdReturn structure is used by the subroutines to
  * pass back a success/failure code and a Tcl_Obj result.
  *
  * This is not an official Tcl standard return type. I
  * find this works well with commands that accept subcommands.
  */
typedef struct cmd_return {
  int status;
  Tcl_Obj *object;
} CmdReturn;
 /*
5
  * Function Prototypes for the commands that actually do
  * the processing.
  * Two macros are used in these prototypes:
  *
  * EXTERN EXPORT is for functions that must interact with
  *   the Microsoft or Borland C++ DLL loader.
  * ANSI_ARGS_ is defined in tcl.h
  * ANSI_ARGS_ returns an empty string for non-ANSI C
  *   compilers, and returns its arguments for ANSI C
  *   compilers.
  */
EXTERN EXPORT(int,Demo_Init) _ANSI_ARGS_ ((Tcl_Interp *));
EXTERN EXPORT(int,Demo_Cmd) _ANSI_ARGS_ ((ClientData,
  Tcl_Interp *, int, Tcl_Obj **));
CmdReturn *demo_GetCmd _ANSI_ARGS_ ((ClientData,
  Tcl_Interp *, int, Tcl_Obj **));
CmdReturn *demo_SetCmd _ANSI_ARGS_ ((ClientData,
  Tcl_Interp *, int, Tcl_Obj **));
CmdReturn *demo_CreateCmd _ANSI_ARGS_ ((ClientData,
  Tcl_Interp *, int, Tcl_Obj **));
CmdReturn *demo_DestroyCmd _ANSI_ARGS_ ((ClientData,
   Tcl_Interp *, int, Tcl_Obj **));
void demo_InitHashTable _ANSI_ARGS_ (());
/*
* For debugging printf's.
*/
#define debugprt if (demoDebugPrint>0) printf
/* End _DEMOINT */
#endif
```

Note the following regarding the previous example.

1. The demoInt.h file has the #include definitions that will be used by the source code files in the demo package. If the extension requires several include files, this convention makes it easier to maintain the list of include files.
2. The DEMO_VERSION string will be used in the Demo_Init function to define the script global variable demo_version. All packages should include a *packageName*_version variable definition. Defining this variable allows scripts to check the version of the package they have loaded.
3. There are some conventions that Microsoft VC++ demands for code that will be dynamically loaded. If your extension will need to compile only on UNIX or Macintosh, you may delete this section and simplify the function prototypes.
4. The CmdReturn structure is not a part of the Tcl standard. I use it to allow functions that implement subcommands to return both a status and a Tcl_Obj to the function that implements the primary command. The status field will be assigned the value TCL_OK or TCL_ERROR. The object field will be a pointer to a Tcl Object, or NULL if the function has no return. If you find another convention more suited to your needs, feel free to use that instead.
5. These are the function prototypes. All of the functions in the source code files should be declared here.

15.3.2 demoInit.c

The demoInit.c file is one of the required files in an extension. At a minimum this file must define the Demo_Init function, as shown in the following example.

Example 3

demoInit.c

```
#include "demoInt.h"
/* CVS Revision Tag */
#define DEMOINIT_REVISION "$Revision: 1.5 $"
/*
 *   *
 * Demo_Init
 *
 * Called from demo_AppInit() if this is a standalone
 * shell, or when the package is loaded if compiled
 * into a binary package.
 *
 * Results:
 * A standard Tcl result.
 *
 * Side effects:
 * Creates a hash table.
 * Adds new commands
 * Creates new Tcl global variables for demo_version and
 *    demoInit_revision.
 *
```

```
 *   */
int Demo_Init(Tcl_Interp *interp) {
 /* interp Current interpreter. */
 /*
1
  * If this application will need to save any
  * data in a hash table initialize the hash table.
  */
Demo_InitHashTable ();
 /*
2
  * Call Tcl_CreateCommand for commands defined by this
  * extension
  */
Tcl_CreateObjCommand(interp, "demo", Demo_Cmd,
    (ClientData) NULL, NULL);
 /*
3
  * Define the package for pkg_mkIndex to index
  */
Tcl_PkgProvide(interp, "demo", DEMO_VERSION);
 /*
4
  * Define the version for this package
  */
Tcl_SetVar((Tcl_Interp *) interp, "demo_version",
    DEMO_VERSION, TCL_GLOBAL_ONLY);
 /*
5
   * Not a requirement. I like to make the source code
  * revision available as a Tcl variable.
  * It's easier to track bugs when you can track all the
  * revisions of all the files in a release.
  */
Tcl_SetVar((Tcl_Interp *) interp, "demoInit_revision",
    DEMOINIT_REVISION, TCL_GLOBAL_ONLY);
 /*
6
  */
  return TCL_OK;
}
```

Note the following regarding the previous example.

1. Tcl allows any package to create its own hash table database to store and retrieve arbitrary data. If a package needs to share persistent information with scripts, you will probably need to save that data

in a hash table and return a key to the script to identify which data is being referenced. The hash table is discussed in more detail with the functions that actually interact with the table.

2. The Tcl_CreateObjCommand function creates a Tcl command demo and declares that the function Demo_Cmd is to be called when this command is evaluated.
3. The Tcl_PkgProvide command declares that this package is named demo, and the revision defined by DEMO_VERSION. DEMO_VERSION is #defined in demoInt.h.
4. This Tcl_SetVar command defines a global Tcl variable demo_version as the version number of this package. This definition allows a script to check the version number of the demo package that was loaded.
5. This is not part of the Tcl standard. My preference is to include the source control revision string in each module to make it easy to determine just what versions of all the code were linked into a package. This is particularly important when in a crunch phase of a project and making several releases a day to testers who are trying to convey what behavior was seen on what version of the code.
6. Finally, return TCL_OK. None of these function calls should fail.

15.3.3 demoCmd.c

The demoCmd.c file is where most of your extension code will exist. For small extensions, this file will contain all of the code that implements your package functionality. The Demo_Cmd function introduces a couple of new Tcl library functions.

The Tcl_GetIndexFromObj function searches a list of words for a target word. This function will return the index of the word that either exactly matches the target word or is a unique abbreviation of the target word. This function is used in this example to extract a subcommand from a list of valid subcommands.

Syntax: int Tcl_GetIndexFromObj (*interp, objPtr, tblPtr, msg, flags, indexPtr*)

Tcl_GetIndexFromObj sets the variable pointed to by *indexPtr* to the offset into *tablePtr* of the entry that matches the string representation of the data in *objPtr*. An item in *tablePtr* is defined as a match if it is either an exact match to a string in the table or a unique abbreviation for a string in the table. This function returns TCL_OK if it finds a match or TCL_ERROR if no match is found. If Tcl_GetIndexFromObj fails, it will set an error message in the interpreter's result object.

interp A pointer to the Tcl interpreter.

tablePtr A pointer to a NULL-terminated list of strings to be searched for a match.

msg A string that will be included in an error message to explain what was being looked up if *Tcl_GetIndexFromObj* fails. The error message will resemble the following.

bad *msg* "*string*": must be *tableStrings*

msg The string defined in the msg argument.

string The string representation of the data in objPtr.

tableStrings The strings defined in *tablePtr*.

flags The *flags* argument allows the calling code to define what matches are acceptable. If this flag is TCL_EXACT, only exact matches will be returned, rather than allowing abbreviations.

indexPtr A pointer to the integer that will receive the index of the matching field.

The demoCmd.c file contains the C functions that are called when demo commands are evaluated in the Tcl script. In a larger package with several commands, this file would contain several entry points.

Example 4

demoCmd.c

```
#include "demoInt.h"

/*
1
* Define the subcommands
*
* These strings define the subcommands that the demo command
* supports.
*
* To add a new subcommand, add the new subcommand string,
* #define, and entry in cmdDefinition.
*
* Note: Order is important.
*
* These are the subcommands that will be recognized
*
*/

static char *subcommands[] = {
    "create", "set", "get", "debug", "destroy", NULL};

/*
2
* These #defines define the positions of the subcommands.
* You can use enum if you are certain your compiler will
* provide the same numbers as this.
*/

#define M_create 0
#define M_set  1
#define M_get  2
#define M_debug 3
#define M_destroy 4

/*
```

3

```
* The cmdDefinition structure describes the minimum and
* maximum number of expected arguments for the subcommand
* (including cmd and subcommand names), and a usage message
* to return if the argument count is outside the expected
* range.
*/

typedef struct cmd_Def {
  char *usage;
  int minArgCnt;
  int maxArgCnt;
  } cmdDefinition;

static cmd_Def definitions[5] = {
  {"create key raw_message", 4 , 4},
  {"set arrayName index Value", 5, 5},
  {"get key ", 3, 3},
  {"debug level", 3, 3},
  {"destroy key", 3,3}
};

/*
```

4

```
 * If demoDebugPrint != 0, then debugprt will print debugging
 * info. This value is set with the subcommand debug.
 */

int demoDebugPrint = 0;

/*
```

5

```
 *--------------------------------------------------------------
 *
 * Demo_Cmd --
 *
 *   Demo_Cmd is invoked to process the "demo" Tcl command.
 *   It will parse a subcommand, and perform the requested
 *    action.
 *
 * Results:
 *   A standard Tcl result.
 *
 * Side effects:
 *
 *--------------------------------------------------------------
 */
```

```
int Demo_Cmd(ClientData demo,
       Tcl_Interp *interp,
       int objc,
       Tcl_Obj *objv[]) {
  /* ClientData demo;        /* Not used. */
  /* Tcl_Interp *interp;     /* Current interpreter. */
  /* int objc;               /* Number of arguments. */
  /* Tcl_Obj *CONST objv[];  /* Argument objects. */
 int cmdnum;
 int result;
 Tcl_Obj *returnValue;
 CmdReturn *returnStruct;
 ClientData info;

 /*
  * Initialize the return value
  */

 returnValue = NULL;
 returnStruct = NULL;

 /*
6
  * Check that we have at least a subcommand,
  * else return an Error and the usage message
  */

 if (objc < 2) {
   Tcl_WrongNumArgs(interp, 1, objv,
       "subcommand ?options?");
   return TCL_ERROR;
 }

 /*
7
  * Find this demo subcommand in the list of subcommands.
  * Tcl_GetIndexFromObj returns the offset of the recognized
  * string, which is used to index into the command
  * definitions table.
  * /

 result = Tcl_GetIndexFromObj(interp, objv[1], subcommands,
                "subcommand", TCL_EXACT, &cmdnum);

 /*
```

```
8
     * If the result is not TCL_OK, then the error message is
     * already in the Tcl Interpreter, this code can
     * immediately return.
     */

    if (result != TCL_OK) {
      return TCL_ERROR;
    }

    /*

9
     * Check that the argument count matches what's expected
     * for this Subcommand.
     */

    if ((objc < definitions[cmdnum].minArgCnt) ||
      (objc > definitions[cmdnum].maxArgCnt) ) {
      Tcl_WrongNumArgs(interp, 1, objv,
              definitions[cmdnum].usage);
      return TCL_ERROR;
    }

    result = TCL_OK;

    /*

10
     * The subcommand is recognized, and has a valid number of
     * arguments Process the command.
     */

    switch (cmdnum) {
      case M_debug: {
            char *tmp;
            tmp = Tcl_GetStringFromObj(objv[2], NULL);
            if (TCL_OK !=
               Tcl_GetInt(interp, tmp, &demoDebugPrint)) {
                return (TCL_ERROR);
            }
            break;
      }
      case M_destroy: {
            returnStruct =
              Demo_DestroyCmd((ClientData) &info,
                  interp, objc, objv);
            break;
```

```
    }
    case M_create: {
         returnStruct =
           Demo_CreateCmd((ClientData) &info,
               interp, objc, objv);
         break;
    }
    case M_get:      {
         returnStruct =
           Demo_GetCmd((ClientData) &info,
               interp, objc, objv);
         break;
    }
    case M_set:      {
         returnStruct =
           Demo_SetCmd((ClientData) &info,
               interp, objc, objv);
         break;
    }
    default:       {
         char error[80];
         sprintf(error,
           "Bad sub-command %s. Has no switch entry",
           Tcl_GetStringFromObj(objv[1], NULL));
         returnValue = Tcl_NewStringObj(error, -1);
         result = TCL_ERROR;
    }
  }

  /*
11
   * Extract an object to return from returnStruct.
   * returnStruct will be NULL if the processing is done
   * in this function and no other function is called.
   */

  if (returnStruct != NULL) {
    returnValue = returnStruct->object;
    result = returnStruct->status;
    free (returnStruct);
  }

  /*
12
   * Set the return value and return the status
   */
```

```
    if (returnValue != NULL) {
      Tcl_SetObjResult(interp, returnValue);
    }
    return result;
}
```

Note the following regarding the previous script example.

1. The `subcommands` array of strings will be passed to the `Tcl_GetIndexFromObj` function that will identify a valid subcommand from this list.
2. These `#defines` create a set of textual identifiers for the positions of the subcommands in the list. They will be used in a `switch` command in the main code to select which function to call to implement the subcommands.
3. The `cmdDef` structure is not part of the official Tcl coding standards. It is strongly recommended that functions check their arguments and return a usage message if the arguments do not at least match the expected count. This can be done in each function that implements a subcommand, or in the function that implements the main command. I prefer to use this table to define the maximum and minimum number of arguments expected and the error message to return if the number of arguments received is not within that range.
4. There is a `#define` macro used to define `debugprt` in `demoInt.h`. This will reference the `demoDebugPrint` global variable. This is not part of the official Tcl coding standards. I find it convenient to use `printf` for some levels of debugging.
5. This is the standard header for a C function in a Tcl extension, as recommended in the *Tcl/Tk Engineering Manual*.
6. If there are not at least two arguments, this command was not called with the required subcommand argument.
7. The `Tcl_GetIndexFromObj` call will set the `cmdnum` variable to the index of the subcommand in the list, if there is a match.
8. If the `Tcl_GetIndexFromObj` call returned `TCL_ERROR`, it will have set an error return as a side effect. If the return value is not `TCL_OK`, this function can perform any required cleanup and return a `TCL_ERROR` status. The interface with Tcl interpreter is already taken care of.
9. This section of code is not part of the official Tcl coding standard. The tests can be done here or in the individual functions that will be called from the `switch` statement. The call to `Tcl_WrongNumArgs` sets the return value for this function to a standard error message and leaves the interface with the interpreter ready for this function return.
10. When the execution has reached this point, all syntactical checks have been completed. The subcommand processing can be done in this function, as is done for the `demo debug` command, or another function can be called, as is done for the other subcommands.

    The functions that implement the commands are named using the naming convention `Demo_`*`subcommand`*`Cmd`.

    Note that this `switch` statement does not require a default case statement. This code will be evaluated only if the `Tcl_GetIndexFromObj` function returned a valid index. This code should not be called with an unexpected value in normal operation.

However, most code will require maintenance at some point in its life cycle. If a new command were added, the tables updated, and the `switch` statement not changed, a silent error could be introduced to the extension. Including the default case protects against that failure mode.

11. The `CmdReturn` structure is not part of the official Tcl coding style. I find it useful to return both status and an object from a function, and this works. You may prefer to transfer the status and object with C variables passed as pointers in the function argument list.
12. The `returnValue` is tracked separately from the `returnStruct` so that subcommands processed within this function as well as external functions can set the integer result code and `returnStruct` object to return values to the calling script.

15.3.4 demoDemo.c

The `demoDemo.c` file has the functions that implement the subcommands of the `demo` command. The naming convention is that the first `demo` indicates this is part of the `demo` package, and the second `demo` indicates that this file implements the `demo` command. If there were a `foo` command in this `demo` package, it would be implemented in the file `demoFoo.c`.

Most of these functions will be called from `demoCmd.c`. The exception is the `Demo_InitHashTable` function, which is called from `Demo_Init`. This function is included in this file to allow the hash table pointer (`demo_hashtbl`) to be a `static` global and keep all references to it within this file.

Demo_InitHashTable

This function simply initializes the Tcl hash table and defines the keys to be `NULL` -terminated strings.

Example 5

Demo_InitHashTable

```
#include "demoInt.h"

static Tcl_HashTable *demo_hashtblPtr;

extern int demoDebugPrint;

/*-------------------------------------------------------------
* void Demo_InitHashTable ()--
*  Initialize a hash table.
*  If your application does not need a hash table, this may be
*   deleted.
*
* Arguments
*  NONE
*
* Results
*  Initializes the hash table to accept STRING keys
```

```
*
* Side Effects:
*  None
------------------------------------------------------------*/

void Demo_InitHashTable () {
  demo_hashtblPtr = (Tcl_HashTable *)
        malloc(sizeof(Tcl_HashTable));
  Tcl_InitHashTable(demo_hashtblPtr, TCL_STRING_KEYS);
}
```

Demo_CreateCmd

This function implements the `create` subcommand. `Demo_CreateCmd` is the first function we have discussed that checks arguments for more than syntactic correctness and needs to return status messages other than syntax messages provided by the Tcl interpreter.

This function uses both the `Tcl_AddObjErrorInfo` and the `Tcl_AddErrorInfo` function to demonstrate how each can be used. The error messages are appended to the return value for `Demo_CreateCmd` in the order in which the error functions are called.

The `Tcl_SetErrorCode` function concatenates the string arguments into the Tcl script global variable `errorCode`. It adds spaces as required to maintain the arguments as separate words in the `errorCode` variable.

Example 6

Demo_CreateCmd

```
/*------------------------------------------------------------
* CmdReturn *Demo_CreateCmd ()--
*  Demonstrates creating a hash entry.
*  Creates a hash entry for Key, with value String
* Arguments
*  objv[0]:  "demo"
*  objv[1]:  "create"
*  objv[2]:  hash Key
*  objv[3]:  String
*
* Results
*  Creates a new hash entry. Sets error if entry already
*  exists.
*
* Side Effects:
*  None
------------------------------------------------------------*/

CmdReturn *Demo_CreateCmd (ClientData info,
            Tcl_Interp *interp,
```

```
            int objc,
            Tcl_Obj *objv[]) {
    Tcl_HashEntry *hashEntryPtr;
    CmdReturn *returnStructPtr;
    char *returnCharPtr;
    char *tmpString;
    Tcl_Obj *returnObjPtr;
    char *hashEntryContentsPtr;
    char *hashKeyPtr;
    int isNew;
    int length;

    /*
1    * Print that the function was called for debugging
     */
    debugprt("Demo_CreateCmd called with %d args\n", objc);

    /*
2    * Allocate the space and initialize the return structure.
     */

    returnStructPtr = (CmdReturn *) malloc(sizeof (CmdReturn));
    returnStructPtr->status = TCL_OK;
    returnCharPtr = NULL;
    returnObjPtr = NULL;

    /*
3    * Extract the string representation of the object
     * argument, and use that as a key into the hash table.
     *
     * If this entry already exists, complain.
     */

    hashKeyPtr  = Tcl_GetStringFromObj(objv[2], NULL);
    hashEntryPtr = Tcl_CreateHashEntry(demo_hashtblPtr,
        hashKeyPtr, &isNew);

    if (!isNew) {
      char errString[80];

      sprintf(errString,
        "Hashed object named \"%s\" already exists.\n",
        hashKeyPtr);

      /*
```

```
4
        * Both of these strings will be added to the Tcl script
        * global variable errorInfo
        */

       Tcl_AddErrorInfo(interp, "error in Demo_CreateCmd");

       Tcl_AddObjErrorInfo(interp, errString,
           strlen(errString));

       /*
5
        * This SetErrorCode command will set the Tcl script
        * variable errorCode to "APPLICATION" "Name in use"
        */

       Tcl_SetErrorCode(interp, "APPLICATION", \
           "Name in use", (char *) NULL);

       /*
        * This defines the return string for this subcommand
        */

       Tcl_AppendResult(interp, "Hashed object named \"",
         hashKeyPtr, "\" already exists.", (char *) NULL);

       returnStructPtr->status = TCL_ERROR;

       goto done;
     }

     /*
6
      * If we are here, then the key is unused.
      * Get the string representation from the object,
      * and make a copy that can be placed into the hash table.
      */

     tmpString = Tcl_GetStringFromObj(objv[3], &length);
     hashEntryContentsPtr = (char *) malloc(length+1);
     strcpy(hashEntryContentsPtr, tmpString);
     debugprt("setting: %s\n", hashEntryContentsPtr);
     Tcl_SetHashValue(hashEntryPtr, hashEntryContentsPtr);

     /*
7
      * Set the return values, cleanup and return
      */
```

```
done:
  if ((returnObjPtr == NULL) && (returnCharPtr != NULL)) {
    returnObjPtr = Tcl_NewStringObj(returnCharPtr, -1);
  }
  returnStructPtr->object = returnObjPtr;
  if (returnCharPtr != NULL) {free(returnCharPtr);}
  return returnStructPtr;
  }
```

Note the following in regard to the previous example.

1. This is not a part of the Tcl standard. I find that in many circumstances I need to generate execution traces to track down the types of bugs that show up once every three weeks of continuous operation. Real-time debuggers are not always appropriate for this type of problem, whereas log files may help pinpoint the problem. I like to be able to enable an output message whenever a function is entered.
2. The `returnStructPtr` is initialized in each of the functions that process the subcommands. It will be freed in the `Demo_Cmd` function after this function returns.
3. If the `hashKeyPtr` already exists, `isNew` will be set to zero, and a pointer to the previous `hashEntryPtr` will be returned. If your code sets the value of this entry with a `Tcl_SetHashValue` call, the old data will be lost.
4. Note that the `sprintf` call in this example is error prone. If the key is more than 37 characters, it will overflow the 80 characters allocated for `errString` . A robust application would count the characters, allocate space for the string, invoke `Tcl_AddError` to copy the error message to the error return, and then free the string.
5. Each time `Tcl_AddError` or `Tcl_AddObjError` is called, it will append the argument, followed by a `newline` character to the end of the global script variable `errorInfo`.
   The `Tcl_SetErrorCode` function treats each string as a separate list element. The first field should be a label that will define the format of the following data.
6. Note that the string data in the third argument to the `create` subcommand is copied to another area of memory before being inserted into the hash table. The argument object is a temporary object that will be destroyed, along with its data, when the `demo create` command is fully evaluated.
   The `Tcl_HashEntry` structure accepts a pointer as the data to store. It retains the pointer, rather than copying the data to a new place. Thus, if the string pointer returned from the `Tcl_GetStringFromObj(objv[3],...` call were used as the data in the `Tcl_SetHashValue` call, the string would be destroyed when the command completed, and the data in those memory locations could become corrupted.
7. All of the functions that implement the `demo` subcommands use a flow model of testing values, setting failure values if necessary, and using a `goto` to exit the function. This can also be done using a structured programming flow, but becomes complex and difficult to follow when there are multiple tests.

Demo_GetCmd

The `Demo_GetCmd` will return the string that was inserted into the hash table with a `demo create` command. This function follows the same flow as the `Demo_CreateCmd` function

Example 7
Demo_GetCmd

```
/*-------------------------------------------------------------
* CmdReturn *Demo_GetCmd ()--
*  Demonstrates retrieving a value from a hash table.
*  Returns the value of the requested item in the hash table
*
* Arguments
*  objv[0]:  "demo"
*  objv[1]:  "get"
*  objv[2]:  hash Key
*
* Results
*  No changes to hash table. Returns saved value, or sets error.
*
* Side Effects:
*  None
-------------------------------------------------------------*/

CmdReturn *Demo_GetCmd (ClientData info,
             Tcl_Interp *interp,
             int objc,
             Tcl_Obj *objv[]) {
  Tcl_HashEntry *hashEntryPtr;
  CmdReturn *returnStructPtr;
  char *returnCharPtr;
  Tcl_Obj *returnObjPtr;
  char *hashKeyPtr;
  debugprt("Demo_GetCmd called with %d args\n", objc);

  /*
   * Allocate the space and initialize the return structure.
   */

  returnStructPtr = (CmdReturn *) malloc(sizeof (CmdReturn));

  returnStructPtr->status = TCL_OK;
  returnStructPtr->object = NULL;
  returnCharPtr = NULL;
  returnObjPtr  = NULL;

  /*
   * Get the key from the argument
   * and attempt to extract that entry from the hashtable.
   * If the returned entry pointer is NULL, this key is not in
   * the table.
   */
```

```
    hashKeyPtr = Tcl_GetStringFromObj(objv[2], NULL);
    hashEntryPtr =
         Tcl_FindHashEntry(demo_hashtblPtr, hashKeyPtr);

    if (hashEntryPtr == (Tcl_HashEntry *) NULL) {
      char errString[80];
      Tcl_Obj *objv[2];
      Tcl_Obj *errCodePtr;

      /*
1
       * Define an error code as a list.  Set errorCode.
       */

       objv[0] = Tcl_NewStringObj("APPLICATION", -1);
       objv[1] = Tcl_NewStringObj("No such name", -1);

       errCodePtr = Tcl_NewListObj(2, objv);

      /*
       * This string will be placed in the global variable
       * errorInfo
       */

      sprintf(errString,
         "Hash object \"%s\" does not exist.", hashKeyPtr);
      Tcl_AddErrorInfo(interp, errString);

      /*
       * This string will be returned as the result of the
       *  command.
       */

      Tcl_AppendResult(interp,
          "can not find hashed object named \"",
          hashKeyPtr, "\"", (char *) NULL);
      returnStructPtr->status = TCL_ERROR;
      goto done;
    }

    /*
2
    * If we got here, then the search was successful and we can
    * extract the data value from the hash entry and return it.
    */

    returnCharPtr = (char *)Tcl_GetHashValue(hashEntryPtr);
```

```
  debugprt("returnString: %s\n", returnCharPtr);

done:
  if ((returnObjPtr == NULL) && (returnCharPtr != NULL)) {
    returnObjPtr = Tcl_NewStringObj(returnCharPtr, -1);
  }

  returnStructPtr->object = returnObjPtr;
  if (returnCharPtr != NULL) {free(returnCharPtr);}
  return returnStructPtr;
}
```

Note the following in regard to the previous example.

1. `Demo_GetCmd` uses the `Tcl_SetObjErrorCode` function to set the global script variable `errorCode`. This function assigns the `errCodePtr` to the error code. It does not make a copy of the object.
2. The character pointer `returnCharPtr` will be set to point to the string that was stored in the hash table. This is not a copy of the data. The data is copied when `returnObjPtr` is created with the `Tcl_NewStringObj(returnCharPtr, -1)` function.

Demo_DestroyCmd

The `Demo_DestroyCmd` function will remove an item from the hash table and release the memory associated with it. This example uses yet another technique for setting the error returns. The default behavior is to set the global script variable `errorInfo` with the same string as the function return and to set the `errorCode` value to `NONE`. This function simply allows that to happen.

Example 8
Demo_DestroyCmd

```
/*-------------------------------------------------------------
* CmdReturn *Demo_DestroyCmd ()--
*  Demonstrate destroying an entry in the hash table.
* Arguments
*  objv[0]:  "demo"
*  objv[1]:  "destroy"
*  objv[2]:  hash Key
*
* Results
*  Deletes the hash table entry and frees the memory
*  associated with the hash object being stored.
*
* Side Effects:
*  None
-------------------------------------------------------------*/
```

```
CmdReturn *Demo_DestroyCmd (ClientData info,
             Tcl_Interp *interp,
             int objc,
             Tcl_Obj *objv[]) {
  Tcl_HashEntry *hashEntryPtr;
  CmdReturn *returnStructPtr;
  char *returnCharPtr;
  Tcl_Obj *returnObjPtr;
  char *hashEntryContentsPtr;
  char *hashKeyPtr;
  debugprt("Demo_DestroyCmd called with %d args\n", objc);

  /*
   * Allocate the space and initialize the return structure.
   */
  returnStructPtr = (CmdReturn *) malloc(sizeof (CmdReturn));

  returnStructPtr->status = TCL_OK;
  returnStructPtr->object = NULL;
  returnCharPtr = NULL;
  returnObjPtr  = NULL;

  /*
  * Extract the string representation from the argument, and
  * use it as a key into the hash table.
  */

  hashKeyPtr = Tcl_GetStringFromObj(objv[2], NULL);
  hashEntryPtr = Tcl_FindHashEntry(demo_hashtblPtr,
          hashKeyPtr);

  /*

1
   * If the hashEntryPtr returns NULL, then this key is not in
   * the table. Return an error.
   */

  if (hashEntryPtr == (Tcl_HashEntry *) NULL) {
    /*
     * Tcl_AppendResult sets the return for this command.
     * The script global variable errorInfo will also be
     * set to this string.
     * The script global variable errorCode will be set to
     * "NONE"
     */

    Tcl_AppendResult(interp,
```

```
            "cannot find hashed object named \"",
            hashKeyPtr, "\"", (char *) NULL);
        returnStructPtr->status = TCL_ERROR;
        goto done;
      }

      /*
2
       * Retrieve the pointer to the data saved in the hash table
       * and free it. Then delete the hash table entry.
       */

      hashEntryContentsPtr =
         (char *)Tcl_GetHashValue(hashEntryPtr);
      free(hashEntryContentsPtr);

      Tcl_DeleteHashEntry(hashEntryPtr);

    done:
      if ((returnObjPtr == NULL) && (returnCharPtr != NULL)) {
        returnObjPtr = Tcl_NewStringObj(returnCharPtr, -1);
      }

      returnStructPtr->object = returnObjPtr;

      if (returnCharPtr != NULL) {free(returnCharPtr);}

      return returnStructPtr;
    }
```

Note the following in regard to the previous example.

1. This function allows the interpreter to set the `errorInfo` and `errorCode` values. The string `cannot find hashed object...` will be returned as the result of the command and will also be placed in the `errorInfo` variable.
2. The data saved in the hash table is the pointer to the string that was created in the `Demo_CreateCmd` function. Once the `Tcl_HashEntry` pointer is destroyed, that pointer will be an orphan unless there is other code that references it. In this example, there is no other code using the pointer, so the memory must be released.

Demo_SetCmd

The `Demo_SetCmd` demonstrates creating or modifying an array variable in the calling script. By default, the variable will be set in the scope of the command that calls the `demo set` command. If

`demo set` is called from within a procedure, the variable will exist only while that procedure is being evaluated. The Tcl interpreter will take care of releasing the variable when the procedure completes.

`Demo_SetCmd` uses the `Tcl_ObjSetVar2` function to set the value of the array variable. The `Tcl_ObjSetVar2` and `Tcl_ObjGetVar2` functions allow C code to get or set the value of a Tcl script variable. `Tcl_ObjSetVar2` and `Tcl_ObjGetVar2` are the functions called by the Tcl interpreter to access variables in a script. Any behavior the Tcl interpreter supports for scripts is also supported for C code that is using these functions. You can modify the behavior of these commands with the *`flags`* argument. By default, the behavior is as follows.

- When accessing an existing variable, the `Tcl_ObjGetVar2` command first tries to find a variable in the local procedure scope. If that fails, it looks for a variable defined with the `variable` command in the current namespace. Finally, it looks in the global scope.
- The `Tcl_ObjSetVar2` function will overwrite the existing value of a variable by default. The default is not to append the new data to an existing variable.
- When referencing an array, the array name and index are referenced in separate objects. By default, you cannot reference an array item with an object that has a string representation of `name(index)`.

Syntax: `char *Tcl_SetVar2(`*`interp, name1, name2, newstring, flags`*`)`

Syntax: `char *Tcl_GetVar2(`*`interp, name2, name1, flags`*`)`

`Tcl_SetVar2`	Creates or modifies a script variable. The value of value of the referenced variable will be set to *`newstring`*. This function returns a `char *` pointer to the new value. This function is for use with versions of Tcl older than 8.0.
`Tcl_GetVar2`	Returns the string value for the variable or array reference identified by the string values in *`name1`* and *`name2`*. This function is for use with versions of Tcl older than 8.0.
`Tcl_Interp *`*`interp`*	A pointer to the Tcl interpreter.
`char *`*`name1`*	A string that references either a simple Tcl string variable or the name of an associative array.
`char *`*`name2`*	If this is not `NULL`; it is the index of an array variable. If this is not `NULL`, *`name1`* must contain the name of an array variable.
`char *`*`newstring`*	A string that contains the new value to be assigned to the variable described by *`name1`* and *`name2`*.
`int` *`flags`*	The *`flags`* parameter can be used to tune the behavior of these commands. The value of `flags` is a bitmap composed of the logical OR of the following fields: `TCL_GLOBAL_ONLY` Setting this flag causes the variable name to be referenced only in the global scope, not a namespace or local procedure scope. If both this and the `TCL_NAMESPACE_ONLY` flags are set, this flag is ignored.

`TCL_NAMESPACE_ONLY`

Setting this flag causes the variable to be referenced only in the current namespace scope, not in the current local procedure scope. This flag overrides `TCL_GLOBAL_ONLY`.

`TCL_LEAVE_ERR_MSG`

This flag causes an error message to be placed in the interpreter's result object if the command fails. If this is not set, no error message will be left.

`TCL_APPEND_VALUE`

If this flag is set, the new value will be appended to the existing value, instead of overwriting it.

`TCL_LIST_ELEMENT`

If this flag is set, the new data will be converted into a valid list element before being appended to the existing data.

`TCL_PARSE_PART1`

If this flag is set and the *id1Ptr* object contains a string defining an array element (*arrayName(index)*), this will be used to reference the array index, rather than using the value in *id2Ptr* as the index.

Syntax: `Tcl_Obj *Tcl_ObjSetVar2(`*interp, id1Ptr, id2Ptr, newValPtr, flags*`)`

Syntax: `Tcl_Obj *Tcl_ObjGetVar2(`*interp, id1Ptr, id2Ptr, flags*`)`

`Tcl_ObjSetVar2` Creates or modifies a Tcl variable. The value of the referenced variable will be set to the value of the *newValPtr* object. The `Tcl_Obj` pointer will be a pointer to the new object. This may not be a pointer to *newValPtr* if events triggered by accessing the `newValPtr` modify the content of *newValPtr*. This can occur if you are using the `trace` command to observe a variable. See the discussion on using `trace` in Section 16.1 and read about the `trace` command in your on-line help to see how this can happen.

`Tcl_ObjGetVar2` Returns a Tcl object containing a value for the variable identified by the string values in *id1Ptr* and *id2Ptr*.

`Tcl_Interp *`*interp* A pointer to the Tcl interpreter.

`Tcl_Obj *`*id1Ptr* An object that contains the name of a Tcl variable. This may be either a simple variable or an array name.

`Tcl_Obj *`*id2Ptr* If this is not `NULL`, it contains the index of an array variable. If this is not `NULL`, *id1Ptr* must contain the name of an array variable.

`Tcl_Obj *`*newValPtr* A pointer to an object with the new value to be assigned to the variable described by *id1Ptr* and *id2Ptr*.

`int flags`	The `flags` parameter can be used to tune the behavior of these commands. The value of `flags` is a bitmap composed of the logical OR of the fields described for `Tcl_SetVar2` and `Tcl_GetVar2`.

Example 9
Demo_SetCmd

```
/*-------------------------------------------------------------
* CmdReturn *Demo_SetCmd ()--
*  Demonstrates setting an array to a value
* Arguments
*  objv[0]:  "demo"
*  objv[1]:  "set"
*  objv[2]:  arrayName
*  objv[3]:  Index
*  objv[4]:  Value
*
* Results
*  Sets arrayName(Index) to the Value
*  Also sets arrayName(alpha) to the string "NewValue".
*  Returns the string "NewValue"
*
* Side Effects:
*  None
-------------------------------------------------------------*/

CmdReturn *Demo_SetCmd (ClientData info,
             Tcl_Interp *interp,
             int objc,
             Tcl_Obj *objv[]) {
  Tcl_Obj *returnObjPtr;
  Tcl_Obj *indexObjPtr;
  Tcl_Obj *arrayObjPtr;
  Tcl_Obj *valueObjPtr;
  CmdReturn *returnStructPtr;
  debugprt("Demo_SetCmd called with %d args\n", objc);
  /*
   * Allocate the space and initialize the return structure.
   */
  returnStructPtr = (CmdReturn *) malloc(sizeof (CmdReturn));
  returnStructPtr->status = TCL_OK;
  returnObjPtr  = NULL;

  /*
1
   * Use Tcl_ObjSetVar2 to set the array element "Index" to
```

```
     * "Value"
     */
    arrayObjPtr = objv[2];
    Tcl_ObjSetVar2(interp, arrayObjPtr, objv[3], objv[4], 0);

    /*
2
     * Create two new objects to set a value for index "alpha"
     * in the array.
     */

    indexObjPtr = Tcl_NewStringObj("alpha", -1);
    valueObjPtr = Tcl_NewStringObj("NewValue", -1);

    returnObjPtr = Tcl_ObjSetVar2(interp, arrayObjPtr,
        indexObjPtr, valueObjPtr, 0);
    returnStructPtr->status = TCL_OK;
    returnStructPtr->object = returnObjPtr;

    /*
3
     * Delete the temporary objects by reducing their RefCount
     *  The object manager will free them, and associated
     * memory when the reference count becomes 0.
     */

    Tcl_DecrRefCount(indexObjPtr);

    /*
4
     * Don't delete valueObjPtr - it's the returnObjPtr
     * object. The task will core dump if you clear it,
     *  and then use it.
     *  Tcl_DecrRefCount(valueObjPtr);
     */

    return(returnStructPtr);
    }
```

Note the following in regard to the previous example.

1. This code implements the equivalent of `set` *`arrayName(Index) Value`*.
2. This section of code implements the equivalent of `set` *`arrayName`*`(alpha) NewValue`. The value assigned to `returnObjPtr` is a pointer to `valueObjPtr`, because there is no extra processing attached to the `valueObjPtr` variable.

3. Note that you should use the `Tcl_DecrRefCount` function to free Tcl objects. Do not use the `free` function.
4. The `valueObjPtr` is also created in this function. A pointer to this object will be returned by the `Tcl_ObjSetVar2` call, but the reference count for this object is not incremented.

   Since this object is being returned as the `returnObjPtr`, the reference count should not be decremented. If a pointer to this object is passed to `Tcl_DecrRefCount`, the object will be destroyed, and `returnObjPtr` will point to a nonexistent object. When the Tcl interpreter tries to process the nonexistent object, it will not do anything pleasant.

15.4 COMPLEX DATA

If you need to handle complex data (such as a structure) in your extension, there are some options.

- You can define the structure in your Tcl script as a list of values and pass that list to extension code to parse into a structure.
- You can use an associative array in the Tcl script, where each index into the associative array is a field in the structure.
- You can create a function that will accept the values for a structure in list or array format, parse them into a structure, place that structure in a hash table, and return a key to the Tcl script for future use.

The following example shows how a Tcl associative array can be used to pass a structure into a C code extension. The code expects to be called with the name of an array that has the appropriate indices set. The indices of the array must have the same names as the fields of the structure.

Example 10

Using an Associative Array to Define a Structure

```
...
  Tcl_Obj   *indexPtr;
  Tcl_Obj   *structElementPtr;

  struct demo {
    int   firstInt;
    char  *secondString;
    double thirdFloat;
  } demoStruct;

  /* The names of the fields to be used as array indices */

  char *fields[] = {"firstInt", "secondString",
             "thirdFloat", NULL };

  /*
```

```
 * Create an object that can be used with Tcl_ObjGetVar2
 * to find the object identified with the array and
 * index names
 */

indexPtr = Tcl_NewStringObj(fields[0], -1);

/*
 * Loop through the elements in the structure/ indices of
 * the array.
 */

for (i=0; i<3; i++) {

  /*
   * Set the index object to reference the field
   * being processed.
   */

   Tcl_SetStringObj (indexPtr, fields[i], -1);

  /*
   * Get the object identified as arrayName(index)
   * If that value is NULL, there is no
   * arrayName(index): complain.
   */

  structElementPtr =
      Tcl_ObjGetVar2(interp, objv[2], indexPtr, 0);
  if (structElementPtr == NULL) {
    Tcl_AppendResult(interp,
       "Array Index \"", fields[i],
       "\" is not defined ", (char *) NULL);
    returnStructPtr->status = TCL_ERROR;
    goto done;
  }

  /*
   * This is a strange way to determine which structure
   * element is being processed, but it works in the loop.
   *
   * If an illegal value is in the object, the error
   * return will be set automatically by the interpreter.
   */

  switch (i) {
  case 0: {
        int t;
```

```
          if (TCL_OK !=
            Tcl_GetIntFromObj(interp,
                structElementPtr, &t)) {
               returnStructPtr->status = TCL_ERROR;
               goto done;
          } else {
            demoStruct.firstInt = t;
          }
        break;
        }
    case 1: {
          char *t;
          if (NULL == (t =
            Tcl_GetStringFromObj(structElementPtr, NULL))) {
              returnStructPtr->status = TCL_ERROR;
              goto done;
          } else {
            demoStruct.secondString=
               (char *)malloc(strlen(t)+1);
            strcpy(demoStruct.secondString, t);
          }
        break;
        }
    case 2: {
          double t;
          if (TCL_OK !=
            Tcl_GetDoubleFromObj(interp,
                 structElementPtr, &t)) {
              returnStructPtr->;status = TCL_ERROR;
              goto done;
          } else {
            demoStruct.thirdFloat = t;
          }
        break;
        }
    }
  }

printf("demoStruct assigned values: \n %d\n %s\n %f\n",
  demoStruct.firstInt, demoStruct.secondString,
  demoStruct.thirdFloat);
...
```

The following example demonstrates populating a structure using the `arraystr` subcommand.

Example 11

Script Example

```
set struct(firstInt) "12"
set struct(secondString) "Testing"
set struct(thirdFloat) "34.56"
demo arraystr struct
```

Script Output

```
demoStruct assigned values:
12
Testing
34.560000
```

The next example demonstrates a pair of functions that use a list to define the elements of a structure and save the structure in a hash table. These functions implement two new subcommands in the `demo` extension: `demo liststr` *`key list`* and *`key list`* and `demo getstr` *`key`*.

These functions introduce the Tcl list object. A Tcl list object is an object that contains a set of pointers to other objects. There are several commands in the Tcl library API to manipulate the list object. This example introduces only a few that are necessary for this application.

Syntax: `int Tcl_ListObjLength (`*`interp, listPtr, lengthPtr`*`)`

Places the length of the list (the number of list elements, not not the string length) in the *`lengthPtr`* argument. function returns either `TCL_OK` or `TCL_ERROR`.

`Tcl_Interp *`*`interp`*	A pointer to the Tcl interpreter.
`Tcl_Obj *`*`listPtr`*	A pointer to the list object.
`int *`*`lengthPtr`*	A pointer to the integer that will receive the length of this list.

Syntax: `int Tcl_ListObjIndex (`*`interp, listPtr, index, elementPtr`*`)`

Places a pointer to the object identified by the *`index`* argument in the *`elementPtr`* variable. This function returns either `TCL_OK` or `TCLERROR`.

`Tcl_Interp *`*`interp`*	A pointer to the Tcl interpreter.
`Tcl_Obj *`*`listPtr`*	A pointer to the list object.
`int` *`index`*	The index of the list element to return.
`Tcl_Obj **`*`elementPtr`*	The address of a pointer to a `Tcl_Obj` that will be set to point to the `Tcl_Obj` that is referenced by *`index`*.

Syntax: `Tcl_Obj* Tcl_NewListObj (`*`count, objv`*`)`

Returns a pointer to a new `Tcl_Obj` that is a list object object pointing to each of the objects in the *`objv`* array of objects.

`int` *`count`*	The number of objects defined in the *`objv`* array of objects.
`Tcl_Obj *`*`objv[]`*	An array of pointers to Tcl objects that will be included in this list.

Example 12

List to Structure Example

```
/*-------------------------------------------------------------
* CmdReturn *Demo_ListstrCmd (ClientData info--
*  Accepts a key and a list of data items that will be used to
*  fill a pre-defined structure.
*  The newly filled structure pointer is saved in a hash
*  table, referenced by 'key_Value'.
*
* Arguments:
*  objv[0] "demo"
*  objv[1] "liststr"
*  objv[2] key_Value
*  objv[3] structure_List
*
* Results:
*  Places a structure pointer in the hash table.
*
* Side Effects:
*  None
-------------------------------------------------------------*

CmdReturn *Demo_ListstrCmd (ClientData info,
             Tcl_Interp *interp,
             int objc,
             Tcl_Obj *objv[]) {
  Tcl_HashEntry *hashEntryPtr;
  CmdReturn *returnStructPtr;
  char *returnCharPtr;
  Tcl_Obj *returnObjPtr;
  Tcl_Obj *listElementPtr;
  int listlen;
  int isNew;
  int length;
  char *tmpString;
  int i;
  char *hashEntryContentsPtr;
  char *hashKeyPtr;
  struct demo {
    int   firstInt;
    char  *secondString;
    double thirdFloat;
  } *demoStruct;
  debugprt("Demo_ListstrCmd called with %d args\n", objc);

  /*
```

```
 * Allocate the space and initialize the return structure.
 */
returnStructPtr = (CmdReturn *) malloc(sizeof (CmdReturn));

returnStructPtr->status = TCL_OK;
returnStructPtr->object = NULL;
returnCharPtr = NULL;
returnObjPtr  = NULL;

/*
 * Allocate space for the structure
 */

demoStruct = (struct demo *) malloc(sizeof (struct demo));

/*
 * Get the length of the list, then step through it.
 */
Tcl_ListObjLength(interp, objv[3], &listlen);

for(i=0; i< listlen; i++) {
  /*
   * Extract a list element from the list pointer
   */
  Tcl_ListObjIndex(interp, objv[3], i, &listElementPtr);

  debugprt("Position: %d : Value: %s\n",
    i, Tcl_GetStringFromObj(listElementPtr, NULL));

  /*
   * A strange way to determine which structure element
   * is being processed, but it works in the loop.
   * If an illegal value is in the object, the error
   * return will be set automatically by the interpreter.
   */

  switch (i) {
  case 0: {
        int t;
        if (TCL_OK !=
          Tcl_GetIntFromObj(interp,
               listElementPtr, &t)) {
             returnStructPtr->status = TCL_ERROR;
            goto done;
        } else {
          demoStruct->firstInt = t;
        }
```

```
        break;
        }

    case 1: {
          char *t;
          if (NULL == (t =
            Tcl_GetStringFromObj(listElementPtr,
                 NULL))) {
              returnStructPtr->status = TCL_ERROR;
              goto done;
          } else {
            demoStruct->secondString=
              (char *)malloc(strlen(t)+1);
            strcpy(demoStruct->secondString, t);
          }
        break;
        }

    case 2: {
          double t;
          if (TCL_OK !=
            Tcl_GetDoubleFromObj(interp,
                listElementPtr, &t)) {
              returnStructPtr->status = TCL_ERROR;
              goto done;
          } else {
            demoStruct->thirdFloat = t;
          }
        break;
        }
    }
}

/*
 * Extract the string representation of the object
 * argument, and use that as a key into the hash table.
 *
 * If this entry already exists, complain.
 */
hashKeyPtr = Tcl_GetStringFromObj(objv[2], NULL);
hashEntryPtr = Tcl_CreateHashEntry(demo_hashtblPtr,
    hashKeyPtr, &isNew);
if (!isNew) {
  char errString[80];
  sprintf(errString,
    "Hashed object named \"%s\" already exists.\n",
    hashKeyPtr);
```

```
    /*
     * Both of these strings will be added to the Tcl
     * script global variable errorInfo
     */

    Tcl_AddObjErrorInfo(interp, errString,
        strlen(errString));
    Tcl_AddErrorInfo(interp, "error in Demo_CreateCmd");

    /*
     * This SetErrorCode command will set the Tcl script
     * variable errorCode to "APPLICATION {Name exists}"
     */

    Tcl_SetErrorCode(interp, "APPLICATION", "Name exists",
        (char *) NULL);

    /*
     * This defines the return string for this subcommand
     */

    Tcl_AppendResult(interp, "Hashed object named \"",
        hashKeyPtr, "\" already exists.", (char *) NULL);

    returnStructPtr->status = TCL_ERROR;
    goto done;
  }

  /*
   * If we are here, then the key is unused.
   * Get the string representation from the object,
   * and make a copy that can be placed into the hash table.
   */

  tmpString = Tcl_GetStringFromObj(objv[3], &length);

  hashEntryContentsPtr = (char *) malloc(length+1);
  strcpy(hashEntryContentsPtr, tmpString);
  debugprt("setting: %s\n", hashEntryContentsPtr);
  Tcl_SetHashValue(hashEntryPtr, demoStruct);

done:
  if ((returnObjPtr == NULL) && (returnCharPtr != NULL)) {
    returnObjPtr = Tcl_NewStringObj(returnCharPtr, -1);
  }
```

```
  returnStructPtr->object = returnObjPtr;
  if (returnCharPtr != NULL) {free(returnCharPtr);}
  return returnStructPtr;
}

/*-------------------------------------------------------------
* CmdReturn *Demo_GetstrCmd ()--
*  Demonstrates retrieving a structure pointer from a hash
*  table, and stuffing that into a list.
*
* Arguments
*  objv[0]:  "demo"
*  objv[1]:  "getstr"
*  objv[2]:  hash_Key
*
* Results
*  No changes to hash table. Returns saved value, or sets error.
*
* Side Effects:
*  None
-------------------------------------------------------------*/

CmdReturn *Demo_GetstrCmd (ClientData info,
             Tcl_Interp *interp,
             int objc,
             Tcl_Obj *objv[]) {
  Tcl_HashEntry *hashEntryPtr;
  CmdReturn *returnStructPtr;
  char *returnCharPtr;
  Tcl_Obj *returnObjPtr;
  Tcl_Obj *listPtrPtr[3];
  char *hashKeyPtr;
  struct demo {
    int   firstInt;
    char  *secondString;
    double thirdFloat;
  } *demoStruct;
  debugprt("Demo_GetCmd called with %d args\n", objc);
  /*
   * Allocate the space and initialize the return structure.
   */

  returnStructPtr = (CmdReturn *) malloc(sizeof (CmdReturn));

  returnStructPtr->status = TCL_OK;
  returnStructPtr->object = NULL;
  returnCharPtr = NULL;
```

```
returnObjPtr = NULL;

/*
 * Get the key from the argument
 * And attempt to extract that entry from the hashtable.
 * If the returned entry pointer is NULL, this key is not
 * in the table.
 */

hashKeyPtr = Tcl_GetStringFromObj(objv[2], NULL);
hashEntryPtr = Tcl_FindHashEntry(demo_hashtblPtr,
    hashKeyPtr);

if (hashEntryPtr == (Tcl_HashEntry *) NULL) {
  char errString[80];
  Tcl_Obj *errCodePtr;

  /*
   * This string will be returned as the result of the
   * command.
   */

  Tcl_AppendResult(interp,
      "can not find hashed object named \"",
      hashKeyPtr, "\"", (char *) NULL);
  returnStructPtr->status = TCL_ERROR;
  goto done;
}
/*
 * If we got here, then the search was successful and we
 * can extract the data value from the hash entry and
 * return it.
 */
demoStruct = (struct demo *)Tcl_GetHashValue(hashEntryPtr);

/*
 * Create three objects with the values from the structure,
 * and then merge them into a list object.
 *
 * Return the list object.
 */

listPtrPtr[0] = Tcl_NewIntObj(demoStruct->firstInt);
listPtrPtr[1] =
   Tcl_NewStringObj(demoStruct->secondString, -1);
listPtrPtr[2] =
   Tcl_NewDoubleObj((double) demoStruct->thirdFloat);
```

```
    returnObjPtr = Tcl_NewListObj(3, listPtrPtr);
    debugprt("returnString: %s\n", returnCharPtr);

  done:
    if ((returnObjPtr == NULL) && (returnCharPtr != NULL)) {
      returnObjPtr = Tcl_NewStringObj(returnCharPtr, -1);
    }
    returnStructPtr->object = returnObjPtr;
    if (returnCharPtr != NULL) {free(returnCharPtr);}
    return returnStructPtr;
  }
```

The following script example demonstrates using the `liststr` and `getstr` subcommands.

Example 13

Script Example

```
%demo liststr key1 [list 12 "this is a test" 34.56]
% demo getstr key1
```

Script Output

```
12 {this is a test} 34.56
```

15.5 EMBEDDING THE TCL INTERPRETER

The most common use of Tcl is as an interpreter with extensions to add functionality. The most common *c*ommercial use of the Tcl interpreter is to embed it within a C, C# or C++ application. In this architecture, the mainline code is compiled and the Tcl interpreter is invoked to run specific scripts—to read a configuration file, display a GUI, to allow a user to customize behavior, etc.

This technique is very powerful, since it provides you with the speed and power of a compiled language with the versatility and ease of customization of an interpreted language.

Examples of this style of programming include most of the high-end CAD and EDA packages, Cisco's IOS, the TuxRacer game, the sendmail tclmilter and the Apache Rivet package. These applications define configuration options and simple operations in a Tcl script file and then perform the CPU-intensive operations within a compiled section.

Embedding the Tcl interpreter within a C application requires a few items:

1. The Tcl include files (`tcl.h, tk.h`).
2. The Tcl library for the target system (`libTcl.a, libTcl.so, libTcl.dll.`)
3. Some support code in the application to link to the Tcl interpreter.

The include files and libraries are included with the standard Tcl distributions, or they will be created if you compile Tcl and Tk from source (described in the next section).

Embedding Tcl or Tk into a C or C++ application uses these functions:

`Tcl_FindExecutable`	Fills internal Tcl variable that is used by `info nameofexecutable`. This is not absolutely required, but is good to include.
`Tcl_CreateInterp`	Creates a new interpreter. After this command has been completed the new interpreter can be used for some applications. The compiled commands will be available, but Tcl commands that are implemented in scripts (for example, `parray`) will not be loaded until `Tcl_Init` has been called.
`Tcl_Init`	Loads the Tcl initialization scripts to fully initialize the interpreter and create all standard commands.
`Tk_Init`	Loads the Tk initialization scripts. This is only valid if you have linked the Tk interpreter into your application.
`Tcl_Eval`	Evaluates a Tcl script and returns `TCL_OK` or `TCL_FAIL`.
`Tcl_Exit`	Clean up and exit.

The `Tcl_FindExecutable` and `Tcl_CreateInterp` function calls can be done in any order. Both of these should be done before calling the `Tcl_Init` or `Tk_Init` functions. The `Tcl_Init` function should be invoked before `Tk_Init`. The `Tcl_Eval` function should not be called until the previous functions have returned.

The first two function calls when initializing an embedded Tcl interpreter should be `Tcl_FindExecutable` and `Tcl_CreateInterp`. These two calls can be done in either order, but both should be done before calling `Tcl_Init`.

Syntax: `void Tcl_FindExecutable` *`(argv0)`*

Determines the complete path to the executable. If the *`argv0`* argument includes a rooted path that is used, else the current working directory and *`argv[0]`* value are used.

`argv0` C and C++ applications pass the command line arguments as an array of strings. The first element of this array will be the name of the executable.

Syntax: `Tcl_Interp *Tcl_CreateInterp ()`

Creates a new interpreter and defines the basic commands.

The `Tcl_CreateInterp` command returns a pointer to the new Tcl interpreter. This is actually a pointer to the Tcl interpreter state structure. Multiple interpreters share the same code base, but each have their own state structure.

This interpreter pointer is passed to most Tcl library functions.

The `Tcl_Init` and `Tk_Init` functions both require the tt Tcl_Interp pointer.

Syntax: `int Tcl_Init` *`(interp)`*

Initializes the Tcl Interpreter. This involves loading and evaluating the `init.tcl` script and other scripts in the library.

`interp` A pointer to the `Tcl_Interp` structure returned by a call to `Tcl_CreateInterp`.

The Tcl_Init and Tk_Init functions return an integer return that conforms to normal C conventions—a "0" (TCL_OK) indicates success and a non-zero return (TCL_FAIL) indicates an error. Information about the failure can be obtained by using Tcl_GetVar to retrieve the contents of the errorInfo global variable.

After an interpreter has been created and initialized your application can send scripts to be evaluated. These scripts can be simple Tcl commands, or long and complex scripts. There are several flavors of the Tcl_Eval command depending on whether the script is already in a Tcl object, a file, or included in the code as a string. The most basic function is Tcl_Eval.

Syntax: Tcl_Eval *(interp, string)*

Evaluate a string of Tcl commands within an interpreter.

interp A pointer to the Tcl_Interp structure returned by a call to Tcl_CreateInterp.

string A printable string consisting of one or more Tcl commands.

The last step is to invoke Tcl_Exit to exit your application. This is preferred over the normal exit function because it will do a bit more cleanup of the Tcl interpreter and make sure no data is left in limbo.

Syntax: Tcl_Exit *(status)*

Clean up the Tcl interpreter state and exit.

status An integer status to return to the OS. The standard is to return 0 for a success and non-zero for some exception exit.

The next example is a minimal main.c which will create and initialize a Tcl interpreter and then load and evaluate the commands in a file.

Example 14

main.c

```
#include <stdlib.h>
#include <tcl.h>
#include <tk.h>

main(int argc, char *argv[]) {

  // Tcl 'glue' variables
  Tcl_Interp *interp;     /* Interpreter for application. */
  int rtn;

  // Create the interp and initialize it.
  Tcl_FindExecutable(argv[0]);
  interp = Tcl_CreateInterp();

  if (Tcl_Init(interp) == TCL_ERROR) {
      return TCL_ERROR;
```

```
    }

    if (Tk_Init(interp) == TCL_ERROR) {
        return TCL_ERROR;
    }

    // Run the Tcl script file - hardcoded for myScript.tcl

    rtn = Tcl_Eval(interp, "source myScript.tcl");

    if (rtn != TCL_OK) {
        printf("Failed Tcl_Eval: %d \n%s\n", rtn,
           Tcl_GetVar(interp, "errorInfo", TCL_GLOBAL_ONLY));
        exit(-1);
    }
    Tcl_Exit(0);
}
```

In the previous example, the script performs all the operations. It might load another extension, create a GUI, create a web server or anything else.

If you're embedding Tcl within a set of compiled code, you probably have a set of compiled functions that you'd like the Tcl scripts to have access to. These can be built into the executable, rather than loaded from a compiled library, as is done with extensions.

The functions described in the previous sections about building a Tcl extension are also used to create new commands to be used with an embedded interpreter.

The next example shows a `main.c` with a new command being created. The new command is `factorcount`, which finds the number of factors a value has by dividing the value by all smaller integers to see which will divide it evenly. This function has no use except that it chews up fewer CPU cycles when compiled than it does when interpreted.

Example 15

main.c with new command

```
#include <stdlib.h>
#include <math.h>
#include <tcl.h>
#include <tk.h>

#include "./factorcountInt.h"

main(int argc, char *argv[]) {

  // Tcl 'glue' variables
  Tcl_Interp *interp;     /* Interpreter for application. */
```

```
    int rtn;

    // Create the interp and initialize it.
    interp = Tcl_CreateInterp();
    Tcl_FindExecutable(argv[0]);

    if (Tcl_Init(interp) == TCL_ERROR) {
        return TCL_ERROR;
    }

    if (Tk_Init(interp) == TCL_ERROR) {
        return TCL_ERROR;
    }

    // Add the factorcount command
    Tcl_CreateObjCommand(interp, "factorcount",
        Factorcount_Cmd, (ClientData) NULL, NULL);

    // Run the Tcl script file - hardcoded for myScript.tcl

    rtn = Tcl_Eval(interp, "source myScriptFactor.tcl");

    if (rtn != TCL_OK) {
        printf("Failed Tcl_Eval: %d \n%s\n", rtn,
           Tcl_GetVar(interp, "errorInfo", TCL_GLOBAL_ONLY));
        exit(-1);
    }
}
```

The `Factorcount_Cmd` function is mostly code to check that the Tcl command was called correctly. In a real application there would probably be more logic than checking that the script writer used the right parameters.

Example 16

factor.c

```
#include "./factorcountInt.h"

int Factorcount_Cmd(ClientData factorcount,
             Tcl_Interp *interp,
             int objc,
             Tcl_Obj * CONST *objv) {

    /* ClientData factorcount;           /* Not used. */
    /* Tcl_Interp *interp;         /* Current interpreter. */
```

```
/* int objc;                    /* Number of arguments. */
/* Tcl_Obj *CONST objv[];       /* Argument objects. */

// Tcl variables
int result;
Tcl_Obj *returnValue;

// function variables
int product;  // Product to find factors for - from user
int count;    // Count of factors
int i;        // loop variable
int f2;       // Temporary factor variable

// Assume everything will be ok
result = TCL_OK;

/*
 * Initialize the return value
 */

returnValue = NULL;

/*
 *  Check that we have a command and number
 */

if (objc < 2) {
    Tcl_WrongNumArgs(interp, 1, objv,
            "factorCount number");
    return TCL_ERROR;
}

  if (TCL_OK !=
      Tcl_GetIntFromObj(interp, objv[1], &product)) {
          result = TCL_ERROR;
          goto done;
  }

  count = 0;
  for (i=1; i < product; i++) {
    f2 = product/i;
    if (f2*i == product) {
      count++;
    }
  }

  returnValue = Tcl_NewIntObj(count);
```

```
done:

  /*
   * Set the return value and return the status
   */

  if (returnValue != NULL) {
      Tcl_SetObjResult(interp, returnValue);
  }

  return result;
}
```

The `factorcountInt.h` file is mostly the previously described boilerplate with these lines to define the new command:

factorcountInt.h

```
/***
extern "C" {
***/

EXTERN EXPORT(int,Factorcount_Cmd) _ANSI_ARGS_
        ((ClientData, Tcl\_Interp *, int, Tcl\_Obj * CONST *));

/***
  }
***/
```

The example script to go with this script will create a small GUI and invoke the `factorcount` command to evaluate the user input.

Example 17

Script Example

```
set done 0

label .l1 -text "Enter Number:"
entry .e1 -textvar product

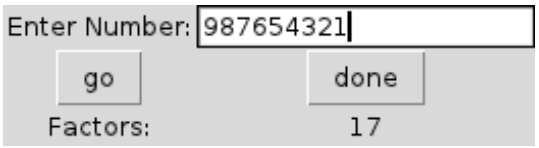
button .b1 -text "go" -command {set answer [factorcount $product]}
button .b2 -text "done" -command {set done 1}

label .l2 -text "Factors:"
label .l3 -textvar answer
```

```
grid .l1 .e1
grid .b1 .b2
grid .l2 .l3

vwait done
```

Script Output



The script uses `vwait` to force the script to run until the user exits. An application that intends to be GUI driven might follow `Tcl_Eval` with a call to `Tcl_MainLoop` to enter the event loop and stay there until the interpreter exits.

Syntax: `Tk_MainLoop ()`

Calls the event loop processing code repeatedly. Returns when the last window is destroyed.

15.6 BUILDING TCL AND TK FROM SOURCES

If your application requires more than just simple interfaces to the Tcl and Tk libraries, odds are good that you'll want to look at the source or build your libraries from the source and include debugging information. Tcl and Tk are relatively easy to build from source. The source distribution includes support for building Tcl/Tk under Linux, Unix, Mac OS/X and Windows. The build files include support for GCC compiler as well as MS Visual C, Borland C and XCode.

The first step in building your own Tcl and Tk libraries and interpreters is to acquire the sources. The Tcl source tree is maintained `core.tcl.tk`, with mirrors at `www.sourceforge.net`.

The best way to get the most current bleeding edge or most recently maintained source code is to use the `fossil` repositories at `http://core.tcl.tk`.

The baseline Tcl and Tk sources are maintained using the `fossil` distributed code management application written by D. Richard Hipp. The executable (or source) for this can be obtained from `http://www.fossil-scm.org`.

Once you have `fossil` installed on your system you can install the Tcl and Tk sources by opening a command or xterm window and typing commands similar to these.

Example 18

Script Example

```
fossil clone http://core.tcl.tk/tcl tcl.fos
fossil clone http://core.tcl.tk/tk tk.fos
```

```
mkdir tcl
cd tcl
fossil open ../tcl.fos
cd ..
mkdir tk
fossil open ../tk.fos
```

Once the sources are unpacked, the `tcl` and `tk` folders will include folders named `macosx`, `unix` and `win`. Within each of these folders is a `README` file with information about how to build Tcl or Tk for that platform. This information changes occasionally as platforms evolve, so it's best to check the README before trying to build Tcl or Tk.

Depending on the tools you have installed in your development environment (minimally, a C compiler, a make utility and a library utility), the steps to compile will resemble the following:

- `Windows`
  The source code distribution includes a Developer Studio workspace and project file.
  Most commonly, Tcl is compiled from the command line using the `nmake` application that is provided with Visual C++ 6.0.
  If you use `nmake` the command to compile Tcl is:
  ```
  cd sourceFolder/win
  nmake -f makefile.vc
  ```
- `Linux/Unix`
  The Tcl and Tk distributions include a `configure` script to determine the exact configuration of your system and allow you to fine-tune the options.
  The steps to compile Tcl and Tk are:
  ```
  cd sourceFolder/unix
  ./configure
  make
  ```
- `Mac OSX`
  If you have installed the GNU make and GCC, Tcl and Tk can be built using the provided makefile.
  ```
  cd sourceFolder/macosx
  make
  ```

Once a new interpreter has been created, you can install it with the command `make install`.

15.7 BOTTOM LINE

- A Tcl extension can be built for several purposes, including the following.
  a. Adding graphics to an existing library
  b. Adding new features to a Tcl interpreter

c. Creating a rapid prototyping interpreter from a library
d. Creating a script-driven test package for a library

- A Tcl extension must include the following.
  a. An *ExtName*_Init function
  b. Calls to Tcl_CreateCommand to create new Tcl commands
  c. Code to implement each new Tcl command
- A Tcl extension should also include the following.
  a. An include file named *extName*Int.h
  b. A makefile or IDE project files
  c. Separate directories for generic code, platform-specific files, documentation, and tests

  A Tcl extension may include other files as necessary to make it a maintainable, modular set of code.
- New Tcl commands are defined with the Tcl_CreateObjCommand or Tcl_CreateCommand command.

  Syntax: int Tcl_CreateObjCommand (*interp, cmdName, func, clientData, deleteFunc*)

  Syntax: int Tcl_CreateCommand (*interp, cmdName, func, clientData, deleteFunc*)
- The C function associated with the new Tcl command created by Tcl_Create...Command should have a function prototype that resembles the following.

```
int funcName (clientData, interp, objc, objv)
```

- You can get a native or a string representation from a Tcl object with the Tcl_GetIntFromObj, Tcl_GetDoubleFromObj, or Tcl_GetStringFromObj command.

  Syntax: int Tcl_GetIntFromObj (*interp, objPtr, intPtr*)

  Syntax: int Tcl_GetDoubleFromObj (*interp, objPtr, doublePtr*)

  Syntax: char *Tcl_GetStringFromObj(*objPtr, lengthPtr*)
- You can create a new object with one of the following functions.

  Syntax: Tcl_Obj *Tcl_NewIntObj(*intValue*)

  Syntax: Tcl_Obj *Tcl_NewDoubleObj(*doubleValue*)

  Syntax: Tcl_Obj *Tcl_NewStringObj (*bytes, length*)
- You can modify the content of an object with one of the following functions.

  Syntax: void Tcl_SetIntObj (*objPtr, intValue*)

  Syntax: void Tcl_SetDoubleObj (*objPtr, doubleValue*)

  Syntax: void Tcl_SetStringObj (*objPtr, bytes, length*)

  Syntax: void Tcl_AppendToObj (*objPtr, bytes, length*)

  Syntax: void Tcl_AppendObjToObj (*objPtr, appendObjPtr*)

  Syntax: void Tcl_AppendStringsToObj (*objPtr, string, ..., NULL*)
- The results of a function are returned to the calling script with the functions Tcl_SetObjResult and Tcl_SetResult.

  Syntax: void Tcl_SetObjResult (*interp, objPtr*)

  Syntax: void Tcl_SetResult (*interp, string, freeProc*)

- The status of a function is returned to the calling script with the functions `Tcl_AddObjErrorInfo`, `Tcl_AddErrorInfo`, `Tcl_SetObjErrorCode`, and `Tcl_SetErrorCode`, which set the `errorCode` and `errorInfo` global Tcl variables.
  Syntax: `void Tcl_AddObjErrorInfo (interp, message, length)`
  Syntax: `void Tcl_AddErrorInfo (interp, message)`
  Syntax: `void Tcl_SetObjErrorCode (interp, objPtr)`
  Syntax: `void Tcl_SetErrorCode (interp, element1, element2...NULL)`
- Tcl provides an interface to fast hash table database functions to allow tasks to save and retrieve arbitrary data based on keys. A hash table must be initialized with the `Tcl_InitHashTable` function.
  Syntax: `void Tcl_InitHashTable (tablePtr, keyType)`
- Hash table entries may be manipulated with the following functions.
  Syntax: `Tcl_HashEntry *Tcl_CreateHashEntry (tablePtr, key, newPtr)`
  Syntax: `Tcl_HashEntry *Tcl_FindHashEntry (tablePtr, key)`
  Syntax: `void Tcl_DeleteHashEntry (entryPtr)`
- Data may be retrieved or deposited in a `Tcl_HashEntry` with the following functions.
  Syntax: `ClientData TclGetHashValue (entryPtr)`
  Syntax: `void TclSetHashValue (entryPtr, value)`
- A match to a string can be found in an array of strings with the `Tcl_GetIndexFromObj` function.
  Syntax: `int Tcl_GetIndexFromObj (interp, objPtr, tblPtr, msg, flags, indexPtr)`
- The content of Tcl variables can be accessed with the following functions.
  Syntax: `Tcl_Obj *Tcl_ObjGetVar2 (interp, id1Ptr, id2Ptr, flags)`
  Syntax: `char *Tcl_GetVar2 (interp, name2, name1, flags)`
- The content of Tcl script variables can be set with the following functions.
  Syntax: `Tcl_Obj *Tcl_ObjSetVar2 (interp, id1Ptr, id2Ptr, newValPtr, flags)`
  Syntax: `char *Tcl_SetVar2 (interp, name1, name2, newstring, flags)`
- A Tcl list is implemented with a Tcl object that contains an array of pointers to the elements of the list. You can create a Tcl list within a C function with the `Tcl_NewListObj` function.
  Syntax: `Tcl_Obj* Tcl_NewListObj (count, objv)`
- You can get the length of a list with the `Tcl_ListObjLength` function.
  Syntax: `int Tcl_ListObjLength (interp, listPtr, lengthPtr)`
- You can retrieve the list element from a particular index in a list with the `Tcl_ListObjIndex` function.
  Syntax: `int Tcl_ListObjIndex (interp, listPtr, index, elementPtr)`

  Syntax: A Tcl or Tk interpreter can be embedded within a C application using the `Tcl_FindExecutable`, `Tcl_CreateInterp`, `Tcl_Init`, `Tk_Init` and `Tcl_Eval` functions.

  Syntax: Tcl and Tk can be built from source on Unix, Linux, Mac OSX and Windows systems.

  Syntax: The Tcl source code is available from `http://core.tcl.tk` and `http://www.sourceforge.net`

15.8 PROBLEMS

The following numbering convention is used in all Problem sections:

Number Range	Description of Problems
100–199	Short comprehension problems review material covered in the chapter. They can be answered in a few words or a 1–5 line script. These problems should each take under a minute to answer.
200–299	These quick exercises require some thought or information beyond that covered in the chapter. They may require reading a man page or making a web search. A short script of 1–50 lines should fulfill the exercises, which may take 10–20 minutes each to complete.
300–399	Long exercises may require reading other material, or writing a few hundred lines of code. These exercises may take several hours to complete.

- *100* If you have a standalone application with no source code, or library documentation, is this a likely candidate for a Tcl extension? Why or why not?
- *101* What Tcl library call adds a new command to a Tcl interpreter?
- *102* What Tcl command will return the integer value of the data in a Tcl object? What will be returned if the object does not contain integer data?
- *103* What data format is always valid for Tcl data?
- *104* Can the `Tcl_SetVar` function be used to assign a value to a variable that does not exist ?
- *105* Can you assign a string of binary (not ASCII) data to a Tcl object?
- *106* What value should a successful command return?
- *107* What command will add a string to the `errorInfo` global variable?
- *108* Given an extension named `checksum`, what should the initialization function be named?
- *109* What would be the path and name of the include file for internal use by the extension foo? (Following the TEA path and name conventions.)
- *110* Which library command will append a string onto the data in a Tcl object?
- *111* Which library command will create a new Tcl object with an integer value?
- *112* Which library command will append a new list element to a Tcl list variable?
- *113* Can a Tcl interpreter be embedded in a compiled application?.

- *200* If you have an application built from an internally developed function library, describe how you could use a Tcl extension to perform automated integration and unit testing for the package.
- *201* Find the sample extension code on the companion website and compile that on your preferred platform.
- *202* Write a Tcl extension that will calculate and return a simple checksum (the sum of all bytes) of the content of a Tcl variable.
- *203* Write a Tcl extension that will return a string formatted with the `sprintf` command. This new command will resemble the Tcl `format` command.

- *204* Extend the embedded Tcl example in Section 15.5 with a new command to return the area of a circle of a given radius. The area of a circle is `3.14 * radius * radius`.
- *205* Modify the embedded Tcl example in Section 15.5 to have an internal string with a script to evaluate instead of sourcing an external Tcl file.
- *300* Write a Tcl extension that will save data in a structure, as shown in Example 10. Add a subcommand to retrieve the values from the structure and populate a given array.
- *301* Port the extension written for problem 202 to at least one other platform: MS Windows, UNIX, or Macintosh.
- *302* Write a set of test scripts for the extension written in problem 203 to exercise the `sprintf` function and prove that it behaves as documented.
- *303* Port the extension and tests written in problems 203 and 302 to another platform. Confirm that `sprintf` behaves the same on both platforms.

CHAPTER

16 Applications with Multiple Environments

Many applications need to control more than one application environment. For example:

- A TCP/IP server needs to support multiple clients.
- A test platform may be testing several sets of hardware at once.
- Games may be tracking multiple users.
- Tcl script may be controlling several processor blades in a multi-processor environment.

Tcl provides several ways to manage the multiple environments. These techniques range from pure Tcl solutions to compiled code.

Here are the commonly used techniques. Each will be discussed in more detail later in this chapter.

- Event loop

  You can write many multiple environment applications using the Tcl event queue to evaluate procedures when they are required and the `upvar` command to map a persistent state variable into a procedure's local scope. This technique is used by the `tclhttpd` application and the `http` package to support multiple concurrent connections.

  This style works best when all of your threads use the same set of procedures, but require separate sets of state data.
- Slave interpreters

  If your application requires private procedures or modifies procedures during execution, you can create a new interpreters for each execution thread. Each new thread has it's own procedures, packages and global variables. An interpreter can be created either as a full-function interpreter, or a *safe* interpreter that cannot touch a filesystem or interact with the outside world.

  Interpreters are created as a hierarchy, with a single parent having multiple child interpreters, each of which may have more child interpreters.

  Using slave interps is suitable for Agent style applications, machine control and testing applications and for network application that interact with potentially malicious clients.
- Thread package

  The default Tcl interpreter is thread safe, but does not include thread support. The thread extension can be downloaded from sourceforge or (if you are using an ActiveTcl distribution) installed using `teacup`.

  The `thread` package allows a Tcl application to start multiple operating system threads. These threads can communicate and coordinate with each other using mutexes and messages.

  Each thread is a peer, there is no hierarchy among threads.

  This technique works well if you need the operating system to establish the threads and perhaps take advantage of multiple processors.

Tcl/Tk: A Developer's Guide. DOI: 10.1016/B978-0-12-384717-1.00016-6

- C library thread support

  The Tcl library includes several commands for creating new threads and interpreters and controlling the interactions between the threads.

  If you are working with an application that creates threads for each connection (i.e., `sendmail` creates a new thread for each mail message) then you will need to use the Tcl library thread functions to initialize the threads.

  This technique is necessary when interfacing with existing object code libraries that use threading.

16.1 EVENT LOOP

The Tcl interpreter includes an event loop to distribute events to procedures that can process them. These events can include data becoming available on a socket, a timer expiring, a variable being modified, mouse events and more.

In order to use the event loop your code must:

1. Register scripts to be processed when an event occurs.
2. Wait for events.

You can register a script to be processed using one of these commands:

- `fileevent`
  To register a script to be evaluated when data is available to be read from a file or socket.
- `trace`
  To register a script to be evaluated when a variable or procedure is accessed.
- `after`
  To register a script to be evaluated when a timer event occurs.
- `bind`
  To register a script to be evaluated when a Windows event (button press, mouse move, etc.) occurs.

How you cause your application to wait for events depends on whether your application is run using `wish` or `tclsh`.

If you are using `wish` to evaluate your program, you don't need to do anything to cause your application to wait for events. A `wish` application automatically waits for events after it has processed all the commands in a script file.

The `tclsh` interpreter defaults to evaluating all the commands in a script and then exiting. The `vwait` command (discussed in Chapter 4) will cause a Tcl script to pause and wait until a variable changes value. This command is commonly used to create modal user interfaces and to force a network server to run until something explicitly causes it to exit.

16.1.1 Using `fileevent`

The `fileevent` command registers a script to be evaluated whenever a channel requires service. The most common use of this is to register a script to be evaluated when a channel has data to be processed. This allows an application to perform a `read` or `gets` command only when data is available instead of performing a blocking read or requiring a round-robin loop.

If your application is controlling a slow device, like a modem, the `fileevent` command can be used to evaluate a script when the device buffer is ready to accept fresh data.

A single application can register scripts to handle file events associated with multiple channels. This technique is used in TCP/IP daemons (like a web or chat server) to support multiple simultaenous connections.

The `socket` and `gets` commands are introduced in Chapter 4.

Syntax: `fileevent` *`channel direction ?script?`*

Register a script to be evaluated when a channel can be serviced.

`channel`	The channel to be read from or written to.
`direction`	The direction of data flow. Acceptable values are `readable` or `writeable`
`?script?`	An optional script to be evaluated when the channel can be processed. If this argument is not provided, the previously registered script is returned.

This example shows a procedure that reads a line of data from a socket and then prints it to stdout. When the line of data is the word `exit`, the variable `done` is modified and the `vwait` command stops waiting, allowing the task to exit.

Example 1

Using Fileevent

```
# proc readData {channel} --
#    Read data from a channel and print it to stdout
# Arguments
#    channel: An open socket or other channel that \
#    has data available for reading.
#
#
proc readData {channel} {
  global done
  set len [gets $channel line]
  if {$len < 0} {
    puts "FAIL "
  }
  if {$line eq "exit"} {
    set done 1
  }
  puts "Just read $len bytes in $line"
}

# Open a socket connection to port 53210.
# Port 53210 is attached to a TCP/IP server that generates data
set channel [socket 127.0.0.1 53210]

# Register a procedure to be evaluated when there is data
```

```
# to be read
fileevent $channel readable "readData $channel"

set done 0
vwait done
```

If your application is processing multiple connections, it probably has state information associated with each connection that it needs to keep track of.

The `upvar` command is a useful way to link state data to a set of procedures. The `upvar` command links a variable in a higher-level scope to a variable in the local scope. This technique is used in the `http` package.

The `upvar` command is discussed in detail in Chapter 7.

Syntax: `upvar` *?level? variableName newName*

?level?	Optional level in which the variable exists.
	Formats for *level* are:
	-number Number represents the number of calling scopes to count up. `-1` is the scope that the current procedure was called from.
	#number Number represents the number of calling scopes to count down from the global scope. The global scope is *#0*.
	By default, the value of `level` is `-1`: the scope of the process that called the current procedure.
variableName	A variable that already exists.
newName	A local variable.

The following example shows a simple TCP/IP server that reads lines of data and changes the status and data indices of a state variable. The state variables exist in the global scope, and each has a unique name which is mapped to the local scope variable `State`.

When the server accepts a socket connection a new channel is created and the `acceptClient` procedure is evaluated with the name of the new channel. This procedure initializes the state variable and registers the `readData` procedure and the name of the channel as an argument as a script to be invoked when data is available for reading.

Each time data becomes available to be read, the `readData` procedure is evaluated. This procedure uses the name of the channel to map the global state variable into the local variable named `State`, and then reads and processes the available data.

Example 2

Using Fileevent with Multiple Clients

```
# proc acceptClient {channel ip port}--
#    Accept a connection from a remote client
# Arguments
#   channel     The channel assigned to this connection
```

```
#   ip          The IP address of the client
#   port        The port assigned to this connection
# Results
#   A new channel is opened and and fileevent assigned
#
proc acceptClient {channel ip port} {
  upvar #0 State_$channel State
  set State(ip) $ip
  set State(status) "Waiting"
  fileevent $channel readable "readData $channel"
}

# proc readData {channel}
#  Read data when it is available and change status as required.
#  Initially starts with status== "Waiting".
#  When data is read, status changes to "Reading"
#  When the word 'DONE' is read, collected input is output
# Arguments
#  channel  The channel to read from
# Results
#  Global variable State_$channel is modified.
#  Initially starts with status == "Waiting" and data == {}
#  When data is read, status changes to "Reading", and the
#   characters that were read are lappended to data.
#  When the word 'DONE' is read, collected input is output
#   and the status is changed back to "Waiting" and the
#   data is set to {}
#
proc readData {channel} {
  upvar #0 State_$channel State
  set len [gets $channel line]

  switch $State(status) {
    Waiting {
      set State(status) "Reading"
      set State(data) [list $line]
    }
    Reading {
      if {$line eq "DONE"} {
        puts "Read these lines from $State(ip): $State(data)"
        set State(status) "Waiting"
        set State(data) ""
      } else {
        lappend State(data) $line
      }
    }
  }
}
```

```
socket –server acceptClient 53210
set done 0
vwait done
```

When this code runs and a connection is made (perhaps using a `telnet` session to test the server) a new variable `State_sock`*X* will be created.

Script Output

```
% info globals S*
State_sock5
% parray State_sock5
State_sock5(data) = first line second line
State_sock5(ip) = 127.0.0.1
State_sock5(status) = Reading
```

16.1.2 Using `trace`

The `trace` command allows you to register a script to be evaluated when a variable or a command is accessed. The script can be registered to be evaluated when a variable is read, written or destroyed or when a command is entered, left, renamed, or deleted.

Syntax: `trace add` *`type name ops script`*

Add a script to be evaluated when an operation occurs on item *`name`*

`type` The classification of the *`name`*. May be one of:

- `command` Register a script to be evaluated if a command is renamed or deleted.
- `execution` Register a script to be evaluated when a procedure is entered or exited.
- `variable` Register a script to be evaluated when a variable is read, modified or deleted.

`name` The name of the item having a trace script added to it.

`ops` The operation that can happen to *`name`*. When this operation occurs *`script`* will be evaluated.

The `trace` command allows a variable to be used to control execution flow. In a process control environment, you might put a trace on a `failureCount` variable to cause an analysis (and perhaps correction) procedure to be invoked whenever there is a new failure.

The code below shows the skeleton for a small adventure type game. It has three locations defined, and two commands: `go` and `teleport`. The `go` command will allow a player to move in a defined direction, while the `teleport` command allows a player to move directly to any location.

The `trace` command is used to cause the `description` procedure to be invoked whenever the `player1(location)` variable is modified. Using `trace` instead of invoking the `description` procedure directly from the `go` and `teleport` procs reduces the coupling between the procedures that manipulate the data (`go` and `teleport`) and the user interface `description` code.

The upvar command is used to map the player1 variable to the myState variable. This is a hook to allow this skeleton to be used for a multi-player adventure game with different variables for player1, player2, etc.

Note that this example demonstrates how to use trace, not necessarily the best way to write an adventure game. In particular, using eval to process user input is very dangerous. A user might type in a command like "exec rm -rf /*", for instance. The interp command is discussed later in this chapter.

Example 3

Using Trace to Trigger a Procedure

```
# Index is a location
# Value is a list of direction/new location pairs

array set places {
  house {up roof down basement}
  roof {down house}
  basement {up house}
}

# Description to print for locations
array set descript {
  house {You are inside a single room house}
  roof {You are on the roof.  Look at those clouds!}
  basement {You are in the basement.  It's spooky here.}
}

# Print the description and possible moves.
proc describe {name index operation} {
  global places descript
  upvar $name myState

  puts "\n$descript($myState(location))"

  foreach {direction destination} $places($myState(location)) {
    puts "You can go $direction (to the $destination)"
  }
}

# Teleport to a destination
proc teleport {destination player} {
  upvar $player myState
  global places
  if {[info exists places($destination)]} {
    set myState(location) $destination
  }
}
```

```
# Go in a direction
proc go {direction player} {
  global places
  upvar $player myState

  array set moves $places($myState(location))

  if {![info exists moves($direction)]} {
    puts "You can't go $direction from $myState(location)"
  } else {
    set myState(location) $moves($direction)
  }
}

# Place a trace on the player variable to invoke 'describe'
trace add variable player1(location) write describe

# Initialize the player to the house
# The describe procedure will be invoked by trace
set player1(location) house

# Loop forever reading and evaluating commands.

while {1} {
  puts —nonewline "Now what? " ; flush stdout
  set cmd [gets stdin]
  eval $cmd player1
}
```

16.1.3 Using `after`

The `after` command can cause a program to pause (perhaps to synchronize with some hardware) or can schedule a script to be evaluated after a delay.

This is the syntax which will cause a script to pause for a given number of milliseconds:

Syntax: `after` *`ms`*

Pause for *`$ms`* milliseconds

`ms` Time in milliseconds

This syntax will schedule a script to be evaluated after *`$ms`* milliseconds have elapsed. The mainline script will continue running after this command is evaluated.

The script that is scheduled to be evaluated in the future will be run from the event loop, so it is important that the main script eventually pause to enter the event loop. If the script is being evaluated by `tclsh` this is commonly done with the `vwait` command. If the script is being evaluated by `wish` the event loop is entered whenever there is no procedure being evaluated.

Syntax: after *ms script*

Schedule *$script* to be evaluated after *$ms* milliseconds have elapsed and continue processing the current script.

ms Time in milliseconds

script A script to evaluate.

A procedure can be invoked at intervals using the after command. The next example shows a heartBeat procedure that will send a short message to a remote site every 20 seconds.

Example 4

A Repeating Procedure

```
proc heartBeat {socket} {
  puts $socket "active"
  flush $socket
  after 20000 heartBeat
}
```

The after command places the heartBeat procedure in the list of events to be processed. The heartBeat command is not recursive—it exits after each evaluation and a new procedure is created when the timer expires.

16.1.4 Using bind

The bind command is used by the Tk extension to bind an action to an event that occurs on a window. This is the mechanism behind the behavior of buttons, scrollbars and other interactive widgets.

You can add new actions to existing widgets with the bind command. To assign a binding to a window, use this syntax:

Syntax: bind *tag event script*

tag An identifier for the widget. May be a class of widget (Button), an individual widget (.myButton) or the word all to bind to all widgets in the application.

event An X-Windows event which will activate the binding.

script A script to evaluate when the event occurs. May include % identifiers to allow the window system to pass data to the script.

The example below adds a binding to a button to display a help message when you right-click on it.

Example 5

Adding New Functionality to a Button

```
# Create an exit button
button .exit –text "Exit" –command exit

# Add a special binding to this button.
bind .exit <ButtonRelease–3> {tk_messageBox –type ok \
```

```
    -message "Clicking this button will exit, losing all your work"}

grid .exit
```

16.1.5 Tcl Interpreters

Most Tcl applications use a single Tcl interpreter. The Tcl library supports creating multiple interpreters within the same executable. Each interpreter has its own set of procedures, global variables, channels, etc. although they share the event loop and view of the file system.

A new interpreter can be a full-featured interpreter, just like the primary interpreter, or it can be a *safe* interpreter. A safe interpreter is the same as a normal interpreter except that all the commands that can interact with the operating system have been removed. Safe interpreters cannot open files or sockets, perform `exec` commands, etc.

Multiple interpreters can be used:

- to provide a separate environment for multiple copies of procedures that each have their own state. For example, an email server can use a separate interp for each mail message being read. This lets each interp maintain a separate state for what may be a complex set of interactions with a remote client.
- to provide a safe environment for running scripts or commands received from untrusted sources. This use is common for Agent style programs or client/server applications that interact with unknown/untrusted entities.

New Tcl interpreters are created by an existing interpreter and have a master–slave relationship with the creator. A master interp can create multiple slaves, and each slave can be a master to multiple slave interps, creating a tree of interps rooted at the original interpreter.

Each interpreter is named as a rooted list. For slaves of the primary interpreter, just the name of the interpreter is sufficient. Slaves of slaves are named from the primary interpreter down. If a slave interpreter named `aa` is created, and then a slave of `aa` named `bb` is created, the new slave would be named {`aa bb`}.

Interpreters are created and manipulated using the `interp` command. This command has a lot of features and a lot of power. This section will introduce a few of the capabilities. Reading the man page will provide more details.

The `interp create` command will create a new slave interpreter.

Syntax: `interp create` *`?-safe? ?--? ?name?`*

Create a new interpreter. Optionally may be *a safe* interpreter and may have a specific name defined.

`-safe`	Make this a safe interpreter — one that cannot interact with the file system, networks or operating system.
`--`	Marks the end of options.
`name`	An optional name for this interp. Default names resemble `interp0`, `interp1`, etc.

When a new interpreter is created, a new command with the same name is created in the parent interpreter. A master can interact with a slave using that name, or the `interp eval` command.

Syntax: `interp eval` *`name script`*

Evaluate script in the named interp

`name` The name of a slave interpreter.

`script` The script to evaluate in that interpreter.

This code snippet creates three interpreters. The interpreter `aa` is a slave of the primary interpreter. The `bb` and `cc` interpreters are slaves of `aa`. This snippet shows a few of the ways that interpreters can be created and how code can be evaluated in the slave interpreters using the `interp eval` or the interpreter commands made by `interp create`.

Example 6

Evaluating Code in a Slave Interpreter

```
# Create a slave of the primary interp
interp create aa

# Create bb as a slave of aa
aa eval {interp create bb}

# Create cc as a slave of aa
interp create {aa cc}

# This sequence returns 3
interp eval aa {set ax 1}
aa eval {expr {$ax + 2}}

# This sequence returns 4
interp eval {aa bb} {set bx 2}
aa eval {bb eval {expr $bx+2}}

# This is an error
# cc is a command in the aa interp
# it is not visible from the primary interpreter
{aa cc} eval {set cx 3}
```

To revisit the simple `adventure` game described in Section 16.1.2, we can use a safe interpreter to accept data from a user and evaluate the commands. The safe interpreter is a sandbox that can interact with the user, but not with anything else on the system.

The `interp transfer` command will move an open channel from one interpreter to another. This can be used to accept a connection in the main Tcl interpreter, create a new *safe* interpreter, and then transfer the channel to the new interpreter. The new interpreter can interact with that channel, but not with any other channel including `stdin` and `stdout`.

Syntax: `interp transfer` *`primaryInterp channel destInterp`*

Transfer channel from the `primaryInterp` interpreter to a `destInterp`. The destination may be a safe interpreter.

`primaryInterp` The interpreter in which the channel currently exists.

channel	The channel to be moved to a new interpreter.
destInterp	The interpreter to receive the channel and commands to interact with the channel.

This example shows using the `interp eval` and `interp transfer` commands to implement the game in the previous example in a safe manner as a client/server game.

Example 7

Using a Safe Slave with a Socket

```
# The defineGame string contains the basic information
set defineGame {
  # Index is a location
  # Value is a list of direction/new location pairs
  array set places {
    house {up roof down basement}
    roof {down house}
    basement {up house}
  }

  # Other variables and procs ellided to
  # make the new code more visible.

  # Go in a direction
  proc go {direction player channel} {
    global places
    upvar #0 $player myState

    array set moves $places($myState(location))

    if {![info exists moves($direction)]} {
      puts $channel "You can't go $direction from $myState(location)"
    } else {
      set myState(location) $moves($direction)
    }
  }
}


# proc accptSock {channel ip port}--
#    Accept a socket connection
# Arguments
#   channel The channel assigned to this new connection
#   ip      The IP address of the client socket
#   port    The port assigned to this connection
#
```

```
# Results
#   Creates a new safe interp and initializes it to
#   play a simple adventure game.

proc accptSock {channel ip port} {
  global sock
  global defineGame

  set sock(ch) $channel

  # Create a new save interpreter
  set ni [interp create —safe]

  # $id will hold the name of the player state
  # variable.  It will be used to hold state within
  # the slave interpreter.
  set id player_$ni

  # Transfer the channel to the slave.
  interp transfer {} $channel $ni

  # Send the variable and procedure definitions to
  # the new interpreter
  $ni eval $defineGame

  # Define a readData procedure in the new interp
  $ni eval [list proc readData $channel \
      "eval \[gets $channel\] $id $channel"]

  # Assign a fileevent to cause readData to be evaluated
  # when there is data to be read.
  $ni eval [list fileevent $channel readable \
      "readData $channel"]

  # put a trace on the player state variable's
  # location index
  $ni eval [list trace add variable ${id}(location) \
      write "describe $channel"]

  # Initialize the player to the house
  # The describe procedure will be invoked by trace
  # within the slave interpreter
  $ni eval [list set ${id}(location) house]
}

socket —server accptSock 12345
```

16.2 THREADS

Each running application is a process. It has an identity within the operating system and is allocated memory, file descriptors and is provided with timeslices for operation. If you need to run multiple copies of an application, you can create multiple processes.

The downside of multiple processes is that creating a process is fairly expensive. The executable binary must be read from the disk, new entries must be created in the process table, etc. If the processes need to interact with each other, they need to do this through the operating system using pipes, sockets or other IPC support, which is also expensive.

For applications like a mail or web server that need a new environment for each client the cost of creating a new process is too high. A mail or web server would not be useful if it needed to read the mail server or web server code from disk each time a new email or HTTP request arrived.

A thread provides an independent state for an application within the application. A thread has its own stack and variables (much like a process), but shares the system interface with other threads in a process. It does not need to read the executable code from the disk and does not need a new entry in the operating system process table. Communications between threads can be handled within an application without needing operating system calls.

Most modern operating systems (Windows, Linux, MacOS) and languages (Java, C++, C, etc.) support creating threads within an application. The library calls to create threads vary between operating systems. As with other platform specific features, Tcl provides a platform neutral approach to creating threads at both the script and compiled code layer.

The core Tcl interpreter is thread-safe. A multi-threaded application that embeds the Tcl interpreter can create and evaluate multiple threads. Using threads at the "C" layer is discussed later in this chapter. This section will introduce the thread basics. The man page provides more options and details.

You can create true multithreaded applications with pure Tcl if the application loads the Tcl `Thread` package. The `Thread` package is available with the Tcl source code files at `http://sourceforge.net/projects/tcl/files/Thread Extension/`

The Tcl model for threading is to have one or more interpreters per thread. Each thread *must* have its own interpreter. The `Thread` package automatically creates a new interpreter for each thread that is created.

Since each thread uses a separate interpreter, each thread has a completely separate environment, with its own procedures, global variables, etc. Each new thread is initialized with the default set of global variables and basic commands, but the new thread is not cloned from a previous thread. Each new thread must explicitly load any special packages, procedure definitions, and so forth, just as a non-threaded Tcl application needs to load packages, etc.

The basic actions for using a thread are:

`thread::create`	Create a new thread.
`thread::send`	Evaluate commands within a thread.
`thread::eval`	Evaluate commands within a thread.
`thread::wait`	Causes a thread to enter the event loop.
`thread::release`	Free a thread.

Syntax: `thread::create` *`?-joinable? ?-preserved? ?script?`*

Create a new thread. Optionally include a command for the new thread to evaluate immediately. Returns a unique identifier string for this thread.

-joinable	Creates a *joinable* thread. A *joinable* thread can be joined to another thread which will block until the *joinable* thread exits. A *joinable* thread must be joined before it exits to avoid memory leaks.
-preserved	Preserve this thread even after it has evaluated a script. By default, if *?script* is present, the new thread evaluates the script and exits.
script	A script to be evaluated immediately. This is non-blocking. The `thread::create` command will return immediately and continue evaluating commands in the parent thread. When the new thread has completed evaluating the commands it will exit unless the `thread::wait` command is evaluated in the new thread.

Just creating a thread is not usually of much use. Your application will need to communicate with the thread, either to send it data to process or receive results.

The two commands for communicating between threads are `thread::send` and `thread::eval`.

Syntax: `thread::send` *?-async? ?-head? threadID script ?variableName?*

Send a string to be evaluated to a thread. By default, wait for the evaluation to be complete before returning.

-async	The `thread::send` command returns immediately instead of waiting for the script to complete processing. This option is useful for scripts that create more threads to communicate directly to the parent thread. A parent thread can use `vwait` to wait for the target thread to complete processing while allowing a GUI in the main thread to stay active.
-head	Force this script to the head of the event queue.
threadID	A thread identifier, as returned by `thread::create`.
script	A script to be evaluated in the target thread.
variableName	The name of an optional variable to receive the results of the script.

The next example shows how to create a thread and send it scripts to process. The first script sent to this thread creates a procedure named `showInfo` which will pause for 2 seconds (to simulate a task that takes time to complete) and will then print out an identifier string and the elapsed time since the application started.

The loop sends eight scripts to the thread, alternating between the normal send and adding the `-head` option. The `thread::send` command sends messages to the thread, which are placed on the thread interpreter's event queue and evaluated in the order that they are removed from the queue. Using the `-head` flag causes a message to be placed first on the queue, delaying the messages that were already on the queue.

By default, the `thread::send` command will block until it has finished being evaluated. If the `thread::send` commands in this example did not use the `-async` option, they would block, and there would never be more than one script in the event queue, and the scripts would be evaluated in the order they are sent, instead of stacking them on the event queue.

Example 8

Sending Messages to a Thread

```
package require Thread

set t [thread::create]

set procDef {proc showInfo {string} {
  global initial
  after 2000
  puts "$string Elapsed Time: [expr [clock seconds] - $initial]"
  }
}

thread::send $t $procDef

thread::send $t "set initial [clock seconds]"

for {set i 0} {$i < 4} {incr i} {
  thread::send -async $t "showInfo $i"
  thread::send -head -async $t "showInfo {$i HEAD}"
}
puts "Completed Loop"
set done 0
vwait done
```

The output generated by this application looks like this:

```
Completed Loop
0 Elapsed Time: 2
3 HEAD Elapsed Time: 4
2 HEAD Elapsed Time: 6
1 HEAD Elapsed Time: 8
0 HEAD Elapsed Time: 10
1 Elapsed Time: 12
2 Elapsed Time: 14
3 Elapsed Time: 16
```

The first script placed on the stack is `showInfo 0`. The event queue will be processed before the next `thread::send` is evaluated. When the event queue is processed the thread will remove the `showInfo 0` message from the event queue and start evaluating it.

While the thread is processing that script, the primary thread is sending the rest of the `showInfo` commands. The `thread::send` messages using the `-head` option are placed at the head of the list,

while those without the -head option are placed at the end. This makes the order of evaluation the last -head message first, and the last default order message last.

The previous example ended with a vwait command. This command was used to force the Tcl interpreter to enter the event loop, rather than exiting immediately, before the thread had time to evaluate any of the scripts. You need to abort this tclsh application from the operating system with a **Control-C** or by closing the task's parent window.

A better solution is to wait for the secondary thread to complete and then exit the primary thread.

The thread::create -joinable, thread::join and thread::release commands provide this facility.

The -joinable flag to thread::create command creates a thread that will have its fate joined with another thread. The thread::join command joins the current thread to a joinable thread.

Syntax: thread::join *threadID*

Wait for the identified thread to exit before continuing execution.

threadID A thread identifier, as returned by thread::create

Once a thread has evaluated the thread::join command, it will block until the joined task exits.

Like Tcl variables, a Tcl thread contains a reference counter. When the reference counter reaches 0, the thread exits. The thread::release command releases a reference to the thread by decrementing the reference counter.

The next example uses the thread::send with and without the -head option to print a sentence in a non-obvious order. The thread::release command is sent last, putting it at the end of the scripts to be evaluated.

After all the calls to showString have been evaluated, the command is evaluated and the secondary thread releases itself and exits. This allows the thread::join command to continue and the primary thread evaluates the final puts "end." command before it exits.

Example 9

Joining Two Threads

```
package require Thread

set t [thread::create –joinable]

set procDef {proc showString {string} {
 after 1000
 puts –nonewline "$string "
}
}

thread::send $t $procDef

foreach word {Atropos in the} {
  thread::send –async $t "showString $word"
}

foreach word {threads all cuts } {
```

```
    thread::send -head -async $t "showString $word"
  }
  puts "The Greeks believed that"
  thread::send $t {thread::release}
  thread::join $t
  puts "end."
```

The output of this example looks like this:

```
The Greeks believed that
Atropos cuts all threads in the end.
```

In order for a thread to use `thread::send`, it needs to know the thread ID of the target thread. This is simple when the primary thread is sending data to a secondary thread, since the thread ID is returned by the `thread::create` command when the thread is created.

However, a secondary thread does not receive any information to identify the thread that created it. In order to send data from a secondary thread to a primary thread, the script needs to discover the thread ID of the primary thread.

The `thread::id` and `thread::names` commands will return the thread identifier of the current thread, and a list of all threads.

Syntax: `thread::id`
Return the identifier of the current thread

Syntax: `thread::names`
Return a list of all the thread identifiers

The next example shows how a primary thread can send its thread ID to secondary threads so that they can can communicate back with a primary thread. The example creates a simple GUI with a button to start new threads, a label to show how many threads currently exist, and a set of labels that display the contents of a variable. The value of the variable is modified by the secondary threads.

The calls to `thread::create` include a script to be evaluated in the secondary thread. The script contains a `for` loop that pauses and sends a message to the primary thread, using the thread ID provided by the primary thread. The message to the primary thread modifies the contents of the global variable that is used as a `-textvariable` for the labels. When the `for` loop is complete, the secondary thread exits.

The `updateThreadCount` is scheduled to run every second using the `after` command (see Section 16.1.3). It uses the `thread::names` command to count the currently active threads, again updating a global variable that is used as a `-textvariable` in the label.

Example 10

Sending Information to a Primary Thread

```
package require Thread

proc createThread {} {
  global tnum var
  incr tnum
  label .l$tnum -textvariable var($tnum)
```

```
    set cmd [format {for {set i 0} {$i < 10} {incr i} {
      thread::send %s "set var(%s) $i"
      after 1000
      }} [thread::id] $tnum]
    set id [thread::create $cmd]
    label .ltn$tnum -text $id
    grid .ltn$tnum .l$tnum
  }

  proc updateThreadCount {} {
    global threadCount
    set threadCount [llength [thread::names]]
    after 1000 updateThreadCount
  }

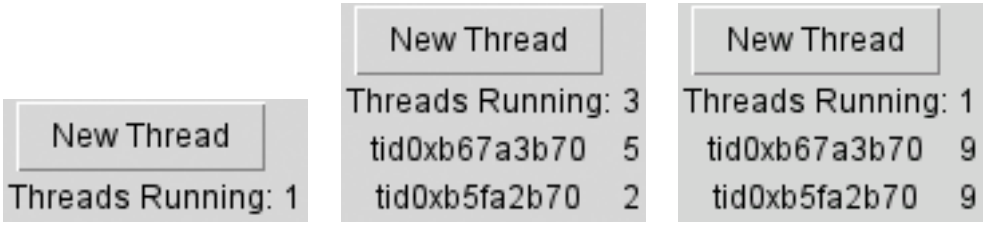
  button .b -command {createThread} -text "New Thread"
  grid .b

  label .lt -text "Threads Running:"
  label .ltc -textvariable threadCount
  grid .lt .ltc
  updateThreadCount

  set tnum 1
```

These three images show the initial display, what the display looks like while two threads are running, and what it looks like after the `for` loops in the threads have been completed and the thread has exited.

The count of threads is always at least 1, since the primary thread will always be present.



Start Running Complete

16.3 EMBEDDED TCL INTERP AND THREADING

This section will introduce the basics for creating threads within a compiled application that embeds the Tcl interpreter. It will introduce basic thread concepts but is not a complete guide to thread programming. There are many styles of threaded application architectures. If you are adding Tcl threads to an existing application you will probably need to study the application's thread behavior as well as Tcl's thread support.

Many compiled systems use multiple threads. The Tcl interpreter has been thread-safe and the Tcl library has supported creating new threads since version 8.1.

Each thread that uses a Tcl interpreter must include its own copy of an interpreter. You can't load a script into one thread and then run it in another. A thread may contain multiple interpreters.

Some applications (including sendmail and the Apache web server) create threads within the mainline compiled code and pass a thread identifier to a user-provided function. In this case, you can use the `Tcl_CreateInterp` function (described in Section 15.5) to create an interpreter within that thread.

If your application is standalone it can create a new thread with the `Tcl_CreateThread` function.

Syntax: `Tcl_CreateThread` *`(idPtr, func, clientData, stack, flags)`*

Creates a new thread in a platform-neutral manner. Passes control to the defined function to initialize the thread.

`idPtr`	A pointer to a `Tcl_ThreadId` variable which will receive the thread id of the new thread. Tcl_ThreadId is a platform neutral datatype that can be used with other `Tcl_Thread` calls.
`func`	A compiled function that will initialize the thread.
`clientData`	A pointer to arguments that will be passed to the initialization function.
`stack`	Some systems allow run-time definition of the stack size. The safest and most platform-neutral value to provide here is `TCL_THREAD_STACK_DEFAULT`.
`flags`	A set of flags to define the new thread. The two supported options are:

`TCL_THREAD_NOFLAGS`	There is no special behavior for this thread.
`TCL_THREAD_JOINABLE`	This thread can be joined to another thread to control execution. One common use of this is to have one thread wait for a joined thread to complete execution and release a shared resource.

When `Tcl_CreateThread` is called, it creates a new thread and passes control to `func` the function declared in the arguments. The `func` function will be running within the newly created thread. This function should create and initialize the Tcl interpreter as described in Section 15.5.

Once a new thread is created the main thread need not interact with it. If the main thread does nothing, it will simply exit after creating the new thread, at which point the entire application, including the new thread, is gone.

One solution to this issue is for the main thread to wait for the subordinate threads to exit. The `Tcl_JoinThread` command will wait for a given thread to exit, or return immediately if that thread has already exited.

Syntax: `void Tcl_JoinThread` *`(threadID, result)`*

Waits for the thread defined by threadID to exit.

`Tcl_ThreadId threadID`	The thread id set by `Tcl_CreateThread()`.
`int *result`	A pointer to an integer for the threads exit value. This may be NULL if you don't need to examine the return.

The next example extends the embedded application described in Section 15.5 to run a number of threads. The number of threads to create is defined by the user at run time using the command line argument. The bulk of the previous `main` function has been moved to the `createThread` function and replaced with loops to create new threads and wait for them to exit.

Example 11

mainThread.c

```
#include <stdlib.h>
#include <math.h>
#include <tcl.h>
#include <tk.h>

#include "./factorcountInt.h"

/*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
* int createThread (char *argv[])--
*    Create a new thread with a Tcl interpreter in it
*    Initialize the interp and start it running a
*    predefined script.
* Arguments:
*   argv    array of arguments from the command line
*
* Results:
*   Returns TCL_OK or TCL_ERROR
*
* Side Effects:
*   New thread, etc.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
int createThread (char *argv[]) {
  Tcl_Interp *interp;     /* Interpreter for application. */
  int rtn;

  // Create the interp and initialize it.
  Tcl_FindExecutable(argv[0]);

  interp = Tcl_CreateInterp();

  if (Tcl_Init(interp) == TCL_ERROR) {
      printf("FAIL TCL_INIT\n");
      return TCL_ERROR;
  }

  if (Tk_Init(interp) == TCL_ERROR) {
      printf("FAIL TK_INIT\n");
      return TCL_ERROR;
  }

  // Add the factorcount command
```

```
  // factorcount_Cmd is defined in factor.c
  Tcl_CreateObjCommand(interp, "factorcount",
      Factorcount_Cmd, (ClientData) NULL, NULL);

  // Run the Tcl script file – hardcoded for myScript.tcl

  rtn = Tcl_Eval(interp, "source myScript.tcl");

  if (rtn != TCL_OK) {
      printf("Failed Tcl_Eval: %d \n%s\n", rtn,
         Tcl_GetVar(interp, "errorInfo", TCL_GLOBAL_ONLY));
      exit(-1);
  }
}


main(int argc, char *argv[]) {
  // function variables
  int i;                  // Counter
  int threadCount;        // Number of threads to creats
                          // read from command line
  Tcl_ThreadId *threadID;          // Returned by Tcl_CreateThread

  sscanf(argv[1], "%d", &threadCount);

  threadID = calloc(threadCount, sizeof(Tcl_ThreadId));

  for (i=0; i< threadCount; i++) {
    if (TCL_OK != Tcl_CreateThread(&threadID[i],
        createThread, argv, TCL_THREAD_STACK_DEFAULT,
        TCL_THREAD_JOINABLE)) {
        printf("Failed to create thread #%d\n", i);
        Tcl_Exit(-1);
    }
  }

  // Wait for all threads to exit.

  for (i=0; i< threadCount; i++) {
    Tcl_JoinThread(threadID[i], NULL);
  }

  // Use Tcl_Exit() instead of exit() to
  //   ensure Tcl closes cleanly.
  Tcl_Exit(0);
}
```

The script for this example uses the `Thread` package to display the thread identifier in the decoration.

Example 12

myScript.tcl

```
lappend auto_path /opt/ActiveTcl-8.5/lib/tcl8.5/
package require Thread

wm title . [thread::id]

set done 0
load ../factorcount/libfactorcount.so

label .l1 -text "Enter Number:"
entry .e1 -textvar product

button .b1 -text "go" -command {set answer [factorcount count $product]}
button .b2 -text "done" -command {set done 1}

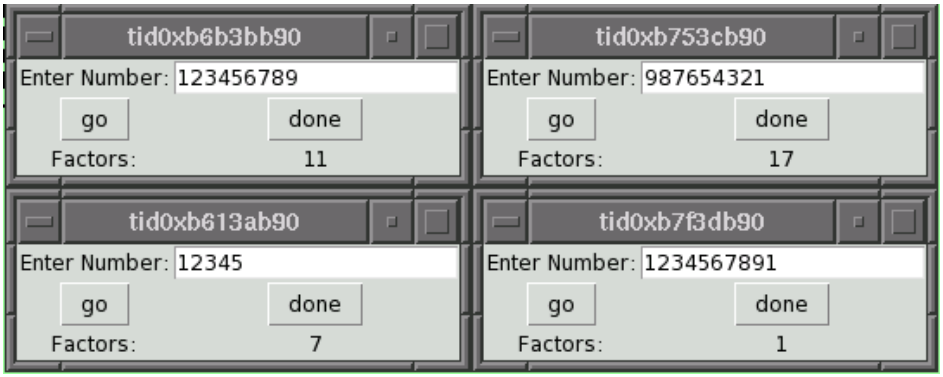
label .l2 -text "Factors:"
label .l3 -textvar answer

grid .l1 .e1
grid .b1 .b2
grid .l2 .l3

vwait done
```

Script Output

```
./factorsThread 4
```



The example requests 4 threads, which run concurrently (within constraints of your CPU). The `for` loop in the `main` function keeps all of the windows present until the last thread has exited.

16.4 BOTTOM LINE

- Tcl supports several techniques for controlling several application environments within a single interpreter.
- The event loop can be used to invoke Tcl procedures with information about the environment that caused the event.
- Slave interpreters contain a private copy of a Tcl interpreter.
- Tcl supports creating threads at both the script and compiled code layer.
- The `fileevent` command will invoke a script when data is available on an IO channel.
  `fileevent` *`channel direction ?script?`*
- The `upvar` command can map a global state variable into a procedure scope.
  `upvar` *`?level? variableName newName`*
- The `trace` command can be used to evaluate a procedure when a variable is modified.
  `trace add` *`type name ops script`*
- The `bind` command will bind a script to a widget and will invoke that script when a specific event occurs.
  `bind` *`tag event script`*
- A slave interpreter is created with the `interp create` command.
  `interp create` *`?-safe? ?--? ?name?`*
- Tcl interpreters can be full functioned or *s*afe. A safe interpreter has no capability to interact outside of the application.
- Code is evaluated within a slave interpreter with the `interp eval` command.
  `interp eval` *`name script`*
- A safe interpreter can be given access to a channel created in a full-featured interpreter with the `interp transfer` command.
  `interp transfer` *`primaryInterp channel destInterp`*
- Script level threading is supported with the `Thread` package. Commands within the `Thread` package exist within the `thread::` namespace.
- A new thread can be created within a script with the `thread::create` command.
  `thread::create` *`?-joinable? ?-preserved? ?script?`*
- A script can be sent to a thread for evaluation with the `thread::send` or `thread::eval` commands.
  `thread::send` *`?-async? ?-head? threadID script ?variableName?`*
- A new thread can be created at the C layer using the `Tcl_CreateThread` function.
  `Tcl_CreateThread` *`(idPtr, func, clientData, stack, flags)`*
- One thread can wait for another thread to exit with the `Tcl_JoinThread` function.
  `void Tcl_JoinThread` *`(threadID, result)`*

16.5 PROBLEMS

The following numbering convention is used in all Problem sections:

Number Range	Description of Problems
100–199	Short comprehension problems review material covered in the chapter. They can be answered in a few words or a 1–5-line script. These problems should each take under a minute to answer.
200–299	These quick exercises require some thought or information beyond that covered in the chapter. They may require reading a man page, or making a web search. A short script of 1–50 lines should fulfill the exercises, which may take 10–20 minutes each to complete.
300–399	Long exercises may require reading other material, or writing a few hundred lines of code. These exercises may take several hours to complete.

- *100* What types of events will the `fileevent` command process?
- *101* Can `after` be used to schedule a script to be evaluated in the future?
- *102* Can a Tcl application have more than one interpreter?
- *103* Can multiple environments be implemented with namespaces or TclOO class?
- *104* Can a new thread be created within a Tcl script?
- *105* Why would an application need to support multiple sets of state?
- *200* Write an IP server that will accept two socket connections and let two persons play a game of Tic-Tac-Toe against each other. The server should accept commands like:

  `X #` Place an X at location #

  `O #` Place an O at location #

  It should return the opposing players turn with a similar response, and tell players who has won when the game is complete.

  Use `fileevent` to read input from the players. Use an extra argument to the `getData` script to distinguish the two players.
- *201* Rework the previous example to support multiple pairs of players. This will require a way to track multiple games. Use the `upvar` technique to map global state variables to individual games.
- *202* Rework Exercise 200 to support multiple pairs of players. This will require a way to track multiple games. Use a separate slave interpreter to hold each game.
- *300* Add a Tk client to the game built in Exercise 200 to show the game board and allow the player to click the location for their move.

- *301* Rework Exercise 200 to support multiple pairs of players. This will require a way to track multiple games. When a pair of players joins create a new thread for that game using the `Thread` package.
- *302* Write a Tcl script which will spawn N threads. Each thread will get a unique random number seed and will then simulate a game of Tic-Tac-Toe with two players choosing random locations for the moves. Display the boards and seed using a GUI similar to that developed in Exercise 300.
- *303* Modify the scripts written in Exercise 302 to be run from a compiled `main.c` which will create N threads and then play the simulated games of Tic-Tac-Toe.

CHAPTER

17 Extensions and Packages

The previous chapters described building Tcl packages and C language extensions to Tcl. Using these techniques, you can create almost any application you desire.

In many cases your applications will require features that have already been developed. You can save time by using existing extensions and packages for these features. Using existing extensions not only saves you development time but reduces your maintenance and testing time.

This chapter provides an overview of a few extensions and packages. It is not possible to cover all of the extensions and packages written for Tcl. The Tcl FAQ (frequently asked questions) lists over 700 extensions, and more are being written. Before you write your own extension, check to see if one already exists.

The primary archive for Tcl extensions and packages is *www.sourceforge.net*. Most extensions are also announced in `comp.lang.tcl` or `comp.lang.tcl.announce`. You can use the Google News Search engine (*www.google.com/advanced_group_search*) to limit your search to those newsgroups. Finally, many of the extensions are described in the Tcl/Tk FAQ (*www.purl.org/NET/Tcl-FAQ*), and discussed on the Tcler's Wiki (*http://wiki.tcl.tk/*).

Many packages and extensions are grouped together in the `tcllib` group. The `tcllib` collection is provided as part of the ActiveState Active-Tcl distribution and is also available at `http://www.sourceforge.net/projects/tcllib`.

Although extensions can save you a great deal of development time, there are some points to consider when using extensions and packages.

- Extensions tend to lag behind the Tcl core.

  Most of these packages are developed by Tcl advocates who work in their spare time to maintain the packages. It can take weeks or months before they have time to update their package after a major change to the Tcl internals.

  Commercial extensions can also lag behind the core. Frequently, a commercial user of Tcl will decide to freeze their application at a particular level, rather than try to support multiple levels of code at their user sites.

  In particular, the change from Tcl 7.6 to Tcl 8.0 introduced some API changes requiring enough rewriting that some less popular extensions have not yet been ported beyond revision 7.6.

  Most popular and commercial packages, however, have been updated to take advantage of the 8.0 improvements.

Tcl/Tk: A Developer's Guide. DOI: 10.1016/B978-0-12-384717-1.00017-8

- Not all extensions are available on all platforms.

  Many packages were developed before Tcl was available on Macintosh and Windows platforms, and they may have some UNIX-specific code in them. Similarly, there are Windows-specific and Mac-specific extensions that interface with other platform-specific applications.
- All extensions may not be available at your customer site.

  Installing extensions takes time, and keeping multiple versions of libraries and extensions consistent can create problems for system administrators. Thus, the more extensions your application requires, the more difficult it may be for users to install your application.

  If you are writing an application for in-house use (and your system administrators will install the extensions you need), this is not a problem. However, if you write an application you would like to see used throughout your company or distributed across the Internet, the more extensions it requires and the less likely someone is to be willing to try it.

If you need an extension, by all means use it. However, if you can limit your application to using only core Tcl or a single extension, you should do so.

The ActiveTcl (or *Batteries Included*) Tcl distribution from ActiveState provides many of the popular extensions, and is becoming the de facto standard Tcl distribution. If your application requires these extensions, odds are good that it will run on your client systems.

Wrapping an application with the Tcl interpreter and required extensions to make a single executable module is another distribution option. Some of these options are discussed in the next chapter. Packages written in Tcl are more easily distributed than extensions written in C.

Whereas many sites may not have an extension, any site that is interested in running your Tcl application can run Tcl packages. You can include the Tcl packages your script needs with the distribution and install them with your application. For example, the HTML library discussed in Chapter 11 has been included in several other packages, including TclTutor and tkchat.

This chapter introduces the following packages.

`[incr Tcl]` The previous chapters have shown how object-oriented techniques can be used with pure Tcl. The `[incr Tcl]` extension provides a complete object-oriented extension for the Tcl developer. It supports classes; private, public, and protected scopes; constructors; destructors; inheritance; and aggregation.
The newest `[incr tcl]` also includes features from `snit` including `type`, `widget`, and widget adapters.

`expect` `expect` automates procedures that have a character-based interaction. It allows a Tcl script to spawn a child task and interact with that task as though the task were interacting with a human.

`TclX` This is a collection of new Tcl commands that support system access and reduce the amount of code you need to write. Many of the TclX features have been integrated into the more recent versions of Tcl.

`tdbc` The `tdbc` (Tcl DataBase Connectivity) package provides a server-neutral view of the various SQL database engines. This package replaces the older single-engine extensions like `OraTcl`, `SybTcl`, `mysqltcl`, `tclsqlite`, etc.

Rivet	Rivet is an Apache module that supports generating dynamic content with Tcl scripts.
BWidgets	The Bwidgets package provides several compound widgets and versions of standard Tk widgets with extended features. This is a pure Tcl package that can be used on multiple platforms.
Img	This extension adds support for more image file formats to the Tcl image object.

17.1 [incr Tcl]

Language	C
Primary Site	*http://incrtcl.sourceforge.net/*
Contact	*arnulf@wiedemann-pri.de*, Arnulf Wiedemann
Tcl Revision Supported	Tcl: 7.3–8.6 and newer Tk: 3.6–8.6 and newer
Supported Platforms	UNIX, MS Windows, Mac OSX
Mailing List	*incrtcl-users-request@lists.sourceforge.net* with a subject of: subscribe
Other Book References	*Tcl/Tk Tools* *[incr Tcl/Tk] from the Ground Up*

[Incr Tcl] is to Tcl what C++ is to C, right down to the pun for the name. It extends the base Tcl language with support for the following.

- Namespaces.
- Class definition with:
  - Public, private, and protected class methods
  - Public, private, and protected data
  - Inheritance
  - Delegation
  - Aggregation
  - Constructors
  - Destructors
  - Variables attached to a class rather than a class instance
  - Procedures attached to a class rather than a class instance

The namespace commands were first introduced into Tcl by [incr Tcl]. The [incr Tcl] namespace command has the same basic format as the namespace command in Tcl 8.0. One difference is that [incr Tcl] allows you to define items in a namespace to be completely private. A pure Tcl script can always access items in a Tcl namespace using the complete name of the item.

Previous chapters showed how you can use subsets of object-oriented programming techniques with Tcl. With [incr Tcl] you can perform pure object-oriented programming. The following

example is adapted from the [incr Tcl] introduction (itcl-intro/itcl/tree/tree2.itcl). It implements a tree object.

There are man pages and MS Windows help files included with the [incr Tcl] distribution. A couple of commands used in this example include class and method. The class command defines a class. In the example that follows, this defines a Tree class with the private data members key, value, parent, and children.

Syntax: class *className* { *script* }
Define an [incr Tcl] class.
className A name for this class.
{*script*} A Tcl script that contains the commands that define this class. These commands may include procedure (method) definitions, variable declarations, and inheritance and access definitions.

The class methods are defined with the method command and can be defined either in line (as the methods add, clear, and parent are defined) or similarly to function prototypes (as the get method is defined). When a method is defined as a function prototype, the code that implements the method can be placed later in the script or in a separate file.

Syntax: method *name* { *args* } { *body* }
Define a method for this class.
name The name of this method.
args An argument list, similar to the argument list used with the proc command.
body The body of this method. Similar to the body of a proc.

Example 1

Script Example

```
puts [package require Itcl]

itcl::class Tree {
  private variable key ""
  private variable value ""
  private variable parent ""
  private variable children ""

  constructor {n v} {
    set key $n
    set value $v
  }

  destructor {
    clear
  }
```

```
  method add {obj} {
    $obj parent $this
    lappend children $obj
  }

  method clear {} {
    if {$children != ""} {
      eval delete object $children
    }
    set children {}
  }

  method parent {pobj} {
    set parent $pobj
  }

  method contents {} {
    return $children
  }

  public method get {{option —value}}

}

itcl::body Tree::get {{option —value}} {
    switch —— $option {
      —key  { return $key }
      —value { return $value }
      —parent { return $parent }
    }
    error "bad option \"$option\""
  }

#
# USAGE EXAMPLE
#

# Create a Tree
#
Tree topNode key0 Value0

#
# Add two children
#
topNode add [Tree childNode1 key1 Value1]
topNode add [Tree childNode2 key2 Value2]
```

```
#
# Display some values from the tree.
#
puts "topNode's children are: [topNode contents]"
puts "Value associated with childNode1 is \
  [childNode1 get -value]"
puts "Parent of childNode2 is: [childNode2 get -parent]"
```

Script Output

```
topNode's children are: childNode1 childNode2
Value associated with childNode1 is Value1
Parent of childNode2 is: ::topNode
```

17.2 expect

Language	C
Primary Site	*http://expect.nist.gov/*
Contact	*libes@nist.gov*, Don Libes
Tcl Revision Supported	Tcl: 7.3–8.6 and newer Tk: 3.6–8.6 and newer
Supported Platforms	UNIX, Windows
Other Book References	*Exploring Expect*, *Tcl/Tk Tools*

expect adds commands to Tcl that make it easy to write scripts to interact with programs that use a character-based interface. This includes programs such as telnet and FTP that use a prompt/command type interface or even keyboard-driven operations like extracting usage information from a large router.

expect can be used for tasks ranging from changing passwords on remote systems to converting an interactive hardware diagnostics package into an automated test system. When you load both the expect and Tk extensions simultaneously you have a tool that is ideal for transforming legacy applications that use dialogs or screen menus into modern GUI-style applications. There are three indispensable commands in expect.

spawn	Open a connection to an application to interact with.
expect	Define input strings expected from the external application.
exp_send	Send output to the external application as if it were typed at a keyboard.

Syntax: spawn *options commandName commandArgs*

Starts a new process and connects the process's stdin and stdout to the expect interpreter.

options The spawn command supports several options, including:

- -noecho — The spawn will echo the command line and arguments unless this option is set.
- *-open fileID* — The -open option lets you process the input from a file handle (returned by open) instead of executing a new program. This allows you to use an expect script to evaluate program output from a file as well as directly controlling a program.

commandName The name of the executable program to start.

commandArgs The command line arguments for the executable program being spawned.

Example 2

spawn Example

```
# Open an ftp connection to the Ubuntu CD repository
spawn ftp cdimage.ubuntu.com
```

Syntax: expect *?-option? pattern1 action1 ?-option? pattern2 action2 ...*

Scan the input for one of several patterns. When a pattern is recognized, evaluate the associated action.

-option Options that will control the matching are:

- -exact — Match the pattern exactly.
- -re — Use regular expression rules to match this pattern.
- -glob — Use glob rules to match this pattern.

pattern A pattern to match in the output from the spawned program.

action A script to be evaluated when a pattern is matched.

Example 3

expect Example

```
expect {
  {Name } {
    puts "Received the Name prompt"
}
```

Syntax: `exp_send` *`string`*

Sends *`string`* to the slave process.

`string` The string to be transmitted to the slave process. Note that a `newline` character is not appended to this text.

Example 4

exp_send Example

```
# Download the readme file
exp_send "get README\n"
```

The `exp_send` command must explicitly add a newline if the command requires one. This is the opposite of the `puts` command that appends a newline unless requested otherwise.

When `expect` finds a string that matches a pattern, it stores information about the match in the associative array `expect_out`. This array contains several indices, including the following.

`expect_out(buffer)` All characters up to and including the characters that matched a pattern.

`expect_out(0,string)` The characters that matched an `exact, glob`, or regular expression pattern.

`expect_out(#,string)` The indices `(1,string)-(9,string)` will contain the characters that match regular expressions within parentheses. The characters that match the first expression within parentheses are assigned to `expect_out(1,string)`, the characters that match the next parenthetical expression are assigned to `expect_out(2,string)`, and so on.

The Ubuntu project maintains a website from which iso images can be downloaded. The next example will log into that site, change to the folder for the UbuntuStudio `natty` release and download any iso images that aren't already in the folder.

The `dialog` procedure simplifies the main program flow by extracting the `timeout` and `eof` tests that would otherwise be needed with each `expect` command.

Example 5

Script Example

```
#!expect

# Open a connection to Ubuntu
spawn ftp cdimage.ubuntu.com

# Set the timeout for 10 seconds.
set timeout 10000

##############################################################################
# proc dialog {prompt send }--
```

```
#    Generalize a prompt/response dialog
# Arguments
#   prompt: The prompt to expect
#   send:    The string to send when the prompt is received
# Results
#   No valid return
#   Generates an error on TIMEOUT or EOF condition.

proc dialog {prompt send } {
  expect {
    $prompt {
        sleep 1;
    exp_send "$send\n"
    }
    timeout {
    expect *
    error "Timed out" \
      "Expected: '$prompt'. Saw: $expect_out(buffer)"
    }
    eof {
    expect *
    error "Connection Closed" \
      "Expected: '$prompt'. Saw: $expect_out(buffer)"
    }
  }
}

# Expect "Name", send "anonymous"
dialog "Name" anonymous

# Expect the password prompt, and send a mail address
dialog "assword" "user@example.com"

# Change to the natty release folder
dialog "ftp>" "cd cdimage/ubuntustudio/releases/natty/release"

# Get a list of the available files
dialog "ftp>" "ls"

set files ""

# Loop until a break condition is reached
# Read lines looking for words (file names)
#   that end in .rpm

while {1} {
    expect {
```

```
        -re {([^ ]*iso)} {
           # Recognized an iso file.  Check to see if we
           #  already have it.

        if {![file exists $expect_out(1,string)]} {
            # No, we don't have this one, append it
            #  to a list to download.

            lappend files $expect_out(1,string)
        }
        }
        ftp> {
          # At the end of the file list, this is a new prompt
          break;
        }
        timeout {
          error "Connection Timed out!"
        }
        eof {
          error "Connection Closed!"
        }
      }
}

# We collected the prompt to find the end of the
#   'ls' data.
# Send a newline to generate a new prompt.

exp_send "\n"

# For each file in our list of files to collect,
#  Get the file.

foreach file $files {
    dialog "ftp>" "get $file"
}
```

Script Output

```
spawn ftp cdimage.ubuntu.com
Connected to cdimage.ubuntu.com.
220 Ubuntu FTP server (vsftpd)
Name (cdimage.ubuntu.com:clif): anonymous
331 Please specify the password.
Password:
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> cd cdimage/ubuntustudio/releases/natty/release
```

```
250 Directory successfully changed.
ftp> ls -latr
229 Entering Extended Passive Mode (|||48099|).
150 Here comes the directory listing.
ubuntustudio-11.04-alternate-amd64.template
ubuntustudio-11.04-alternate-amd64.iso
ubuntustudio-11.04-alternate-amd64.list
ubuntustudio-11.04-alternate-i386.template
ubuntustudio-11.04-alternate-i386.iso
ubuntustudio-11.04-alternate-i386.list
ubuntustudio-11.04-alternate-amd64.jigdo
ubuntustudio-11.04-alternate-amd64.iso.zsync
ubuntustudio-11.04-alternate-i386.jigdo
ubuntustudio-11.04-alternate-i386.iso.zsync
ubuntustudio-11.04-alternate-amd64.iso.torrent
MD5SUMS.gpg
MD5SUMS
SHA1SUMS.gpg
SHA256SUMS
ubuntustudio-11.04-alternate-i386.iso.torrent
SHA1SUMS
HEADER.html
FOOTER.html
SHA256SUMS.gpg
.htaccess
ubuntustudio-11.04-alternate-amd64.metalink
MD5SUMS-metalink
MD5SUMS-metalink.gpg
ubuntustudio-11.04-alternate-i386.metalink
226 Directory send OK.
ftp>
ftp> get ubuntustudio-11.04-alternate-i386.iso
local: ubuntustudio-11.04-alternate-i386.iso \
    remote: ubuntustudio-11.04-alternate-i386.iso
229 Entering Extended Passive Mode (|||57878|).
150 Opening BINARY mode data connection for \
    ubuntustudio-11.04-alternate-i386.iso (1560033280 bytes).
```

17.3 TclX

Language	C
Primary Sites	*http://tclx.sourceforge.net/* *http://wiki.tcl.tk/tclx/* *www.maths.mq.edu.au/~steffen/tcltk/tclx/* (TclX for Macintosh)

Contact	*markd@grizzly.com*, Mark Diekhans
Tcl Revision Supported	`Tcl: 7.3–8.6 and newer`
	`Tk: 3.6–8.6 and newer`
Supported Platforms	`UNIX, MS Windows, Mac Os`
Other Book References	*Tcl/Tk Tools*

The `TclX` extension is designed to make large programming tasks easier and to give the `Tcl` programmer more access to operating system functions such as `chmod`, `chown` and `kill`. `TclX` contains a large number of new commands. Many of the best features of Tcl (sockets, time and date, random numbers, associative arrays, and more) were introduced in `TclX`. There are still many features provided by `TclX` that are not in the Tcl core. `TclX` features include the following.

- Extended file system interaction commands
- Extended looping constructs
- Extended string manipulation commands
- Extended list manipulation commands
- Keyed lists
- Debugging commands
- Performance profiling commands
- System library interface commands
- Network information commands
- Message catalogs for multiple language support (compliant with *X/Open Portability Guide*)
- Help
- Packages

Using `TclX` can help you in the following ways.

- `TclX` gives you access to operating system functions that are not supported by the Tcl core. Using core Tcl, you would need to write standalone C programs (or your own extensions) to gain access to these.
- `TclX` scripts are smaller than pure Tcl scripts, because `TclX` has built-in constructs you would otherwise need to write as procedures.
- `TclX` commands run faster than the equivalent function written as a Tcl procedure, because the `TclX` command is written in C and compiled instead of interpreted.

The following example uses the recursive file system looping command `for_recursive_glob` to step through the files in a directory, and the three text search commands `scancontext`, `scanmatch`, and `scanfile`.

The `fileDict` procedure in chapter 6 used the `foreach`, `glob`, and `file type` commands to build a recursive directory search procedure. The `for_recursive_glob` command provides these features in a single loop.

Syntax: `for_recursive_glob` *`var dirlist globlist code`*

Recursively loops through the directories in a list looking for files that match one of a list of `glob` patterns. When a file matching the pattern is found, the script defined in *`code`* is evaluated with the variable *`var`* set to the name of the file that matched the pattern.

var	The name of a variable that will receive the name of each matching file.
dirlist	A list of directories to search for files that match the *globlist* patterns.
globlist	A list of glob patterns that will be used to match file names.
code	A script to evaluate whenever a file name matches one of the patterns in *globlist*.

Many applications require searching a large number of files for particular strings. A pure Tcl script can perform this operation by reading the file and using `string first`, `string match` or `regexp`. This type of solution uses many interpreter steps and can be slow. The `TclX scancontext`, `scanmatch`, and `scanfile` commands work together to optimize file search applications.

Syntax: `scancontext create`

Create a new scan context for use with the `scanfile` command. Returns a *contextHandle*.

The *contextHandle* returned by the `scancontext` create command can be used with the `scanmatch` command to link in a pattern to an action to perform.

Syntax: `scanmatch` *contextHandle ?regexp? code*

Associate a regular expression and script with a *contextHandle*.

contextHandle	A handle returned by `scancontext create`.
regexp	A regular expression to scan for. If this is blank, the script is assigned as the default script to evaluate when no other expression is matched.
code	A script to evaluate when the *regexp* is matched.

When a regular expression defined with `scanmatch` is recognized, information about the match is stored in the associative array variable `matchInfo`, which is visible to the `code` script. The `matchInfo` variable has several indices with information about the match.

The `scanfile` command will examine a file's contents and invoke the appropriate scripts when patterns are matched.

Syntax: `scanfile` *contextHandle fileId*

Scan a file for lines that match one of the regular expressions defined in a context handle. If a line matches a regular expression, the associated script is evaluated.

contextHandle	The handle returned by the `scancontext create` command.
fileId	A file channel opened with read access.

Example 6

Script Example

```
package require Tclx

proc scanTreeForString {topDir pattern matchString \
```

```
    filesWith filesWithout} {
  upvar $filesWith with
  upvar $filesWithout without

  set with ""

  # Create a scan context for the files that will be scanned.

  set sc [scancontext create]

  # Add an action to take when the pattern is recognized
  # in a file. If the pattern is recognized, append the
  # file name to the list of files containing the pattern
  # and break out of the scanning loop. Without the "break",
  # scanfile would process each occurrence of the text that
  # matches the regular expression.

  scanmatch $sc "$matchString" {
    lappend with [file tail $filename]
    break;
  }

  # Process all the files below $topDir that match the
  # pattern

  for_recursive_glob filename $topDir $pattern {
    set fl [open $filename RDONLY]
    scanfile $sc $fl
    close $fl
    # If there were no lines match the $matchString,
    # there will be no $filename in the list "with".
    # In that case, add $filename to the list of files
    # without the $matchString.
    if {[lsearch $with [file tail $filename]] < 0} {
      lappend without [file tail $filename]
    }
  }

  # Clean up and leave

  scancontext delete $sc
}
scanTreeForString /usr/src/tcl *.h Tcl_Obj hasObj noObj

puts "These files have 'Tcl_Obj' in them: $hasObj"
puts "These files do not: $noObj"
```

Script Output

```
These files have 'Tcl_Obj' in them: tclOOInt.h
tclFileSystem.h tclOODecls.h tclOO.h tclIntPlatDecls.h
tclCompile.h tclInt.h tclOOIntDecls.h tclRegexp.h
tclIntDecls.h tclIO.h tclDecls.h tcl.h

These files do not: tclTomMathDecls.h regerrs.h
tclTomMathInt.h regcustom.h tclTomMath.h tommath.h
tclPort.h tclPlatDecls.h regguts.h regex.h tclUnixThrd.h
tclUnixPort.h tommath_superclass.h tommath.h
tommath_class.h tclWinInt.h tclWinPort.h dirent2.h
limits.h stdlib.h fake-rfc2553.h dlfcn.h float.h
dirent.h string.h unistd.h mpi.h mpi-types.h
logtab.h mpi-config.h inffixed.h gzguts.h inftrees.h zlib.h
deflate.h zconf.h trees.h inffast.h crc32.h inflate.h
zutil.h gzlog.h blast.h zfstream.h puff.h ioapi.h
mztools.h iowin32.h crypt.h zip.h unzip.h inftree9.h
inflate9.h infback9.h inffix9.h zstream.h zfstream.h
```

17.4 TDBC

Language	TclOO and C
Primary Site	*http://core.tcl.tk/tdbc*
Contact:	*kennykb@acm.org*, Kevin Kenny
Tcl Revision Supported	Tcl: 8.6 and newer
Supported Platforms	UNIX, MS Windows, MacOSX

The tdbc package provides a single common interface to many different database engines. Connections to multiple database engines can be open at once.

The tdbc package requires a driver to map from a specific database engine's command set to the tdbc commands. Drivers are provided for Oracle, Postgres, Mysql, and Sqlite3. A wide variety of database engines are supported via the generic ODBC driver. New drivers can be written in pure Tcl (using TclOO) or as compiled extensions.

The tdbc package is part of the standard distribution, but the base tdbc package and drivers are not automatically loaded into a Tcl application. An application must require the tdbc driver packages it will need. The base tdbc package will be loaded automatically. Using package require to load both the base tdbc and tdbc::*driver* packages is not an error.

Example 7

Script Example

```
# Load the base tdbc
package require tdbc
```

```
# Load sqlite3 support
package require tdbc::sqlite3

# Load mysql support
package require tdbc::mysql
```

After the driver has been loaded an application can create a connection to the database engine or engines. Each time you create a connection to a database engine a new command is created to interact with that database engine. The new command can be used to select specific databases within the engine and perform SQL operations on those databases.

The `tdbc::connection` command creates a connection to the database engine.

Syntax: `tdbc::`*`driver`*`::connection create` *`dbCmd ?-option value?`*

Open a connection to a database engine.

`driver`	The driver for this database engine — `sqlite3`, `mysql`, etc.
`dbCmd`	The name of the new command to use to interact with this database.

The options supported for the `tdbc::connection` command vary depending on the requirements of the underlying engine. All of the drivers support these options:

`-readonly boolean`	If the boolean argument is true, the database cannot be modified. The default is false, to allow a database to be modified by the script.
`-timeout ms`	A timeout value in milliseconds. The default value is 0 which will wait forever.

Options that are related to specific drivers include:

Sqlite.

`fileName`	The name of the sqlite database file.

MySql, Oracle.

`-host addr`	The address of the host running the database server.
`-port portNum`	The port number at which the database server is accepting connections.
`-socket path`	The path to a Unix socket or named pipe if the server and application are running on the same host.
`-db name`	The database to attach.
`-user name`	The login for this database.
`-passwd pwd`	The password for this user and database.

ODBC.

connectionString A db-server specific set of connection values. Check the documentation for the database engine's ODBC driver for the format of this string.

One driving principle of tdbc was to make it easier to write safe code than unsafe code. Most database applications are vulnerable to injection attacks or other issues from malformed data. The code to prevent attacks and sterilize inputs can be cumbersome and may need to be repeated throughout an application.

tdbc provides a mechanism for creating an SQL command with references to data and then allowing the tdbc driver to substitute the actual values into the final SQL INSERT or SELECT commands. During the substitution phase all inputs and returns are sterilized and escaped safely and properly.

Values are referenced using a *bound variable*. A *bound variable* is defined within an SQL command by starting the variable with a single colon. When this command is evaluated, the *bound variable* is replaced with the contents of a Tcl variable with the same name.

Example 8

Script Example

```
set data "This needs cleanin'"
set sql "INSERT INTO atable VALUES (:data)"
```

An SQL command can be evaluated directly, but the preferred method is to use the connection's prepare subcommand to perform substitutions and sterilize the inputs. The prepare subcommand accepts an SQL statement and creates a new command (actually a TclOO object) which can be used to evaluate the command later. Any number of statements can be prepared and retained until they are needed.

Syntax: *dbCmd* prepare *statement*

Prepare an SQL statement for later evaluation.

statement A valid SQL command. This statement may include bound variables.

Example 9

Script Example

```
# An SQL statement can include all data:

set insert [dbCmd prepare {
    INSERT INTO author (firstname, lastname)
    VALUES ('Clif', 'Flynt')}]

# The prefered technique is to use bound variables:
```

```
set insert [dbCmd prepare {
    INSERT INTO author (firstname, lastname)
    VALUES (:first, :last)}]
set first "Clif"
set last "Flynt"
```

A prepared command can be evaluated with the statement's `execute` subcommand. The `execute` subcommand can map bound variables using a provided `dict` or from existing Tcl variables.

Syntax: *`statementCmd`* `execute` *`?dict?`*

Evaluate a prepared SQL statement.

`statementCmd`	The command created by a *`dbCmd`* `prepare` command.
`dict`	A dictionary in which dictionary keys match the names of bound variables. If this is provided, the bound variables will be mapped to the dictionary values. If it is not provided bound variables will be mapped to local variables.

Example 10

Script Example

```
set insert [dbCmd prepare {
    INSERT INTO author (firstname, lastname)
    VALUES (:first, :last)}]

# Add Clif Flynt to author table
$insert execute [dict create first Clif last Flynt]

# Add Mark Twain to author table
set first Mark
set last Twain
$insert execute
```

Depending on the application and data sizes it can be simpler to retrieve all the results of a `SELECT` command at once, or to iterate through the returns one at a time. The *`statementCmd`* has two subcommands to support this. The `allrows` subcommand returns all the results as a single dict or list. The `foreach` subcommand defines a loop for processing results in a row-by-row manner.

Syntax: *`statementCmd`* `allrows` *`?-option value? ?dict?`*

Execute a statement and return all results.

`-as value`	Defines whether the return should be as a list or a dict. The value must be one of `lists` or `dicts`. The default is to return the data as a dict.

-columnsvariable varName	If this option is included the list of column names will be placed into the named variable in the order that the data appears in the return. This is useful when data is returned as lists.
dict	A dictionary to use when mapping bound variables to values.

Example 11

Script Example

```
set data [$get allrows —as lists —columnsvar colNames]

foreach rtn $data {
  foreach val $rtn name $colNames {
    puts "$name: $val"
  }
}
```

Script Output

```
id: 1
firstname: Clif
lastname: Flynt
id: 2
firstname: Mark
lastname: Twain
```

If your application will be processing the results in a loop, the `foreach` command may be more appropriate.

Syntax: *statementCmd* `foreach` *?-option value? ?dict? varName script*

Execute a statement and iterate through the returned rows in a script. The script is evaluated once for each row.

-as value	Defines whether the return should be as a list or a dict. The value must be one of `lists` or `dicts`. The default is to return the data as a dict.
-columnsvariable name	If this option is included the list of column names will be placed into the named variable in the order that the data appears in the return. This is useful when data is returned as lists.
dict	An optional dictionary to use when mapping bound variables to values.
varName	The name of the variable to contain the returned row while iterating through the list.
script	A script to evaluate once for each row of data returned.

Example 12

Script Example

```
set report [dbCmd prepare {SELECT * FROM author}]
$report foreach -as dicts author {
  puts $author
}
```

Script Output

```
id 1 firstname Clif lastname Flynt
id 2 firstname Mark lastname Twain
```

The next example uses these commands to create a library database with books and authors. Note that the single quote in this book's title and the embedded `SELECT` command in the `bookInsert` statement work correctly, while Nasty Hacker's attempt at confusing the database with an injection attack fails.

Example 13

Script Example

```
#package require tdbc
package require tdbc::sqlite3

# Remove the database if it already exists.
catch {file delete bookfile.sql}

# Open a sqlite database named bookfile.sql
# The new command for this conenction is dbCmd

::tdbc::sqlite3::connection create dbCmd bookfile.sql

# The database is empty,
# Use the allrows command to create tables.

dbCmd allrows {CREATE TABLE book (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  authorid INT REFERENCES author,
  title VARCHAR(50))}

dbCmd allrows {CREATE TABLE author (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  firstname VARCHAR(20),
  lastname VARCHAR(20))}

# Insert a row into the author table.

set insert [dbCmd prepare {
```

```
    INSERT INTO author (firstname, lastname) VALUES ('Clif', 'Flynt')}]
  $insert execute

# Multiple entries are simpler using bound variables :fn, :ln

set insert [dbCmd prepare {
    INSERT INTO author (firstname, lastname) VALUES (:fn, :ln)}]

foreach {fn ln} {John Ousterhout Brent Welch Mark Twain Nasty Hacker} {
  $insert execute
}

# A bound variable can be used within an embedded SELECT

set bookInsert [dbCmd prepare {
    INSERT INTO book (authorid, title) VALUES
        ((SELECT id FROM author WHERE firstname = :first), :title)}]

set authorTitleList {
  Clif "Tcl/TK: A Developer's Guide"
  Brent "Practical Programming in Tcl/Tk"
  Mark  "Tom Sawyer"
  Mark  "Huckleberry Finn"
  Nasty "(SELECT firstname FROM author limit 1)"
  John  "Tcl and the Tk Toolkit"
}
foreach {first title} $authorTitleList {
  $bookInsert execute
}

set report [dbCmd prepare {SELECT * FROM book}]
set getAuthor [dbCmd  prepare \
    {SELECT firstname, lastname FROM author WHERE id=:authorid}]

$report foreach -as lists bookList {
  lassign $bookList id authorid title
  set author {*}[$getAuthor allrows -as lists]
  puts "$author"
  puts "   $title\n"
}
```

Script Output

```
Clif Flynt
   Tcl/TK: A Developer's Guide

Brent Welch
   Practical Programming in Tcl/Tk

Mark Twain
```

```
   Tom Sawyer

Mark Twain
   Huckleberry Finn

Nasty Hacker
   (SELECT firstname FROM author limit 1)

John Ousterhout
   Tcl and the Tk Toolkit
```

17.5 Rivet

Language	Tcl and C
Primary Site	*http://tcl.apache.org/rivet/*
Mailing List	rivet-dev-subscribe@tcl.apache.org
Original Authors:	David Welton, Damon Courtney, Karl Lehenbauer
Contact:	*mxmanghi@apache.org*, Massimo Manghi
Tcl Revision Supported	Tcl: 8.4 and newer
Supported Platforms	UNIX, MacOSX

There are many tools available for generating web pages dynamically with Tcl. These range from the simple CGI techniques that the Web started with to full HTTP servers like tclhttpd, Wub and AOLServer.

Rivet is an Apache plugin that allows you to embed Tcl into an HTML page or generate a complete page dynamically. It has support for creating and processing forms, works with Ajax, XML-based applications, JQuery and other Web 2.0 Technologies as well as with traditional HTML.

This industrial-strength package is in use at many public and commercial sites. It is included with the SuSE, Redhat, Centos, Debian and Ubuntu distributions.

The simple way to use Rivet is to install mod_rivet in your Apache directory (probably /usr/lib/apache2/modules). If you use an RPM or apt package this will happen automatically. If you build from source, the make install will perform this. After installing Rivet, it must be enabled by adding the following lines to the Apache configuration file (probably httpd.conf on SuSE or RedHat installations or apache2.conf on Debian or Ubuntu installations).

httpd.conf

```
LoadModule rivet_module /usr/lib/apache2/modules/mod_rivet.so
AddType application/x-httpd-rivet rvt
AddType application/x-rivet-tcl   tcl
AddType "application/x-httpd-rivet; charset=utf-8" rvt
```

If you have the Apache mod_dir installed you can use an index.rvt file instead of index.html as the default file to load when a browser requests a folder. This is defined by adding a line like the following to the httpd.conf file.

httpd.conf

```
DirectoryIndex index.rvt index.html
```

You will need to restart your Apache server after these changes.

You can add Tcl code to an HTML page by naming the page with a `.rvt` suffix and enclosing the Tcl code within a pair of angle bracket-question mark pairs. New text is added to the web page with the `puts` command as shown below.

Example 14
Simple Rivet Example

```
<html>
  <body>
    <H1> NOW:
    <? puts [clock format [clock seconds] -format \%H:\%M] ?>
    </H1>
  </body>
</html>
```

A page can be a mixture of HTML and Tcl scripts, as shown above, or completely generated by Tcl scripts.

The `html` command encloses a string with one or more HTML tags.

Syntax: `html` *`string tag1 tag2 ...`*

Returns a string with opening and closing tags around it.

`string`	String to display in the HTML output.
`tag1, tag2 ...`	Tags to put around the string. The first tag will be the outermost tag and the last will be the innermost.

The previous page can also be written as:

Example 15
Script Example

```
<?
html \
    "NOW: [clock format [clock seconds] —format %H:%M] \
    html body h1
?>
```

`Rivet` includes several configuration options that can be set in the `httpd.conf` file. These options support setting scripts to be run before and after a page is loaded, when Apache starts or ends a process, how to allocate resources and more. Check the current Rivet documentation under `directives` for more details on these.

Configuring Rivet to automatically run scripts when a new process is created or a new page is loaded is a powerful way to localize code that defines the way a website performs. This makes a site simpler to maintain than having duplicated text in each web page.

These features are enabled by adding `RivetServerConf` lines to the `httpd.conf` file. The `ChildInitScript` will be evaluated when `apache` starts a new process. The `BeforeScript` will be evaluated before each page is loaded. The next example defines a script `perProcess.tcl` to be sourced when a new Apache process is created, and `perPage.tcl` to be sourced when each `.rvt` page is loaded. The following configuration will evaluate the `perPage.tcl` script when pages with a `.rvt` or `.tcl` suffix are loaded, but not when plain `.html` pages are accessed.

httpd.conf with Tcl Initialization Scripts

```
LoadModule rivet_module /usr/lib/apache2/modules/mod_rivet.so
AddType application/x-httpd-rivet rvt
AddType application/x-rivet-tcl   tcl
AddType "application/x-httpd-rivet; charset=utf-8" rvt

RivetServerConf ChildInitScript "source /var/www/lib/perProcess.tcl"
RivetServerConf BeforeScript    "source /var/www/lib/perPage.tcl"
```

The `ChildInitScript` script is only guaranteed to be evaluated when Apache starts. The `BeforeScript` is evaluated for each page. It's simpler to develop special code in a `BeforeScript`, and then move the debugged code to a `ChildInitScript` when it's stable.

It's best to do persistent setup like creating common procedures in a `ChildInitScript` file. This reduces the overhead of creating the procdures when each page needs them.

perProcess.tcl

```
proc time {} {
  return [clock format [clock seconds] -format "%H:%M"]
```

The next example show how the `perPage.tcl` file can use the `time` procedure to insert boilerplate text into each page.

Example 16

perPage.tcl

```
puts "<h1>Now: [time]</h1>"
```

With these definitions and files, all `.rvt` pages will display `Now: 12:34` at the top of the page.

Rivet includes many commands to make it easier to generate web pages including generating forms, tracking cookies and session management.

Rivet's form support is handled in an object-oriented manner, similar to Tk objects. A web page can have multiple forms, and each form is connected to a unique set of data and has a unique command name. Once a form object is created, text entries, labels, radiobuttons, etc. can be added to it.

Syntax: `form` *`formName ?-OPTION? ?VALUE?`*

Create a form object to be populated and displayed.

`formName` The name of this form. A new command with this name will be created.

The `form` command supports several options, including:

`-method` Selects the submission method for this form the *`VALUE`* may be `post` or `get`. The default is `post`.

`-name` Sets the name to be used in the HTML forms `<form name="VALUE">` tag.

`-action` Defines the URL to receive the data when the submit button is clicked. If this is blank, the URL of the page generating the form is used.

Example 17

Rivet Create form Example

```
form tstForm –method get –name test
```

Once a form object is created, it needs to be placed in the HTML page and populated with the form elements.

The `start` command inserts the form tag into your document. This must be placed after the form is created and before elements are created, but is not otherwise constrained.

Example 18

Rivet form Start Example

```
form tstForm –method get –name test
html "Fill in this form" h2
tstForm start
```

Creating form elements follows a pattern:

`formName elementType name ?-option value`

`formName` The name of the form to associate this element with.

`elementType` The type of the element. May be one of `text`, `radiobutton`, `checkbutton`, `select`, `submit`, etc.

`name` The name associated with this element. This name is linked to the user-provided value when the form is submitted.

?-option value Whatever options and values are appropriate to the element being added. The option/-value pair will be added to the HTML as `option="value"`.

```
form tstForm –method get –name test
tstForm start

# Prompt for a name
html "Enter your name: " b
tstForm text name –size 20

# Add a hidden field
tstForm hidden displayed –value 1

# Add the submit button
tstForm submit submit –value "Submit"
```

The final step in creating a form is to end it.

The `end` command adds the `</form>` tag to complete the form.

```
form tstForm –method get –name test
tstForm start

html "Enter your name: " b
tstForm text name –size 20

tstForm submit submit –value "Submit"

tstForm end
```

Once a form is submitted, a page needs to process it. The `var` command returns information about variables submitted with a form. This command supports several subcommands.

Syntax: `var exists` *varName*

Returns true if the named variable exists in the `post` (or `get`) data.

varName The name of the variable, as defined in the `form` element.

```
# displayed is a hidden variable that exists
# if this page has been displayed already.
if {[var exists displayed]} {
...
}
```

Syntax: `var get` *varName*

Returns the value associated with the named variable.

varName The name of the variable, as defined in the `form` element.

```
# Check the required input

if {![var exists name]
     ([var get name] eq "")} {
    set fail 1
    ...
}
```

Syntax: `var all`

Returns the variables as a varname/value list, suitable for using with `array set` or the `dict` commands.

```
array set values [var all]
```

The next example creates a form requesting three pieces of information. When the user clicks `Submit`, the page is called again. The second time the page is loaded, the form variables have been populated.

The code that creates the form is placed at the end of the page definition. This allows the script to check for valid data before generating any display output. If the inputs are not complete, the form processing is aborted and control passes down to the form display section. If the inputs are complete, the values can be processed and the user can be redirected to a new page or a new set of html can be displayed.

If the inputs are complete this example displays a message and cancels the form display with the `return` command. The page could also redirect the user to a new page once the processing is complete.

Example 19

Script Example

```
<?
package require form

# displayed is a hidden variable that exists
# if this page has been displayed already.
if {[var exists displayed]} {

  # If displayed exists, check the values.
  # Assume OK
  set ok 1

  # Check the required inputs.
  # Generate a prompt if a field is missing.

  foreach required {name quest language} {
    if {![var exists $required]
         ([var get $required] eq "")} {
```

```
      set ok 0
      html "Fill in $required" br
    }
  }

  # If ok, check the answer and return
  # else fall through and redisplay the form.
  if {$ok} {
    # Destroy tstForm object - we're done with it.
    tstForm destroy

    if {[var get language] eq "Tcl"} {
      html "Good Answer" h1
    } else {
      html "Into the Gorge with 'im" h1
    }
    return
  }
}

# Only get here if something fails or first time

html "Form Demo" title

# Create a form object
# Earlier versions of Rivet used
# form create tstForm -method get -name test

form tstForm -method get -name test

# Start the output

tstForm start

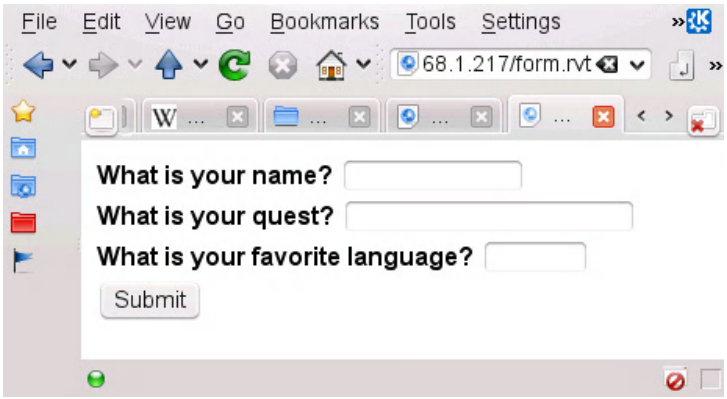
# Add fields to the form

foreach {prompt field size} {
    "What is your name?" name 15
    "What is your quest?" quest 25
    "What is your favorite language?" language 8
    } {
  if {[var exists $field]} {
    set val [var get $field]
  } else {
    set val ""
  }
  html $prompt b
```

```
    tstForm text $field -size $size -value $val
    html <br>
  }

  # Add the last hidden field and submit button
  # then finish the form
  tstForm hidden displayed -value 1
  tstForm submit submit -value "Submit"
  tstForm end
  ?>
```

Script Output



Web pages with static pages require frequent manual modifications when new material is added. The `ChildInitScript` and `BeforeScript` directives can be used to dynamically create pages. The dynamically created pages can be created from a database or by examining a folder.

The next example shows how to build a set of scripts that will look in the current folder for files that end with `quiz.tcl`. The file names will be used to create a navigation bar list of selections. When a user has selected a quiz, that quiz will be displayed in the main window.

To add a new quiz to the website, you simply copy the new ***foo***`quiz.tcl` file to the folder with the other quizzes and it is automatically added to the list of quizzes to select from.

The `ChildInitScript` has a single procedure that uses the `glob` and `string` commands to construct an HTML list of names with links to the appropriate quiz file.

Example 20

ChildInitScript

```
proc makeMenu {} {
  set rtn "<ul>\n"
  foreach f [glob –nocomplain *quiz.tcl] {
    append rtn "<li> <a href=\"/clif/quizes.rvt?page=$f\">"
```

```
    set fileName [file rootname [file tail $f]]
    append rtn "[string range $fileName 0 end-4]"
    append rtn "</a>\n"
  }
  append rtn "</ul>\n"
  return $rtn
}
```

The `BeforeScript` generates the page content. It uses a small stylesheet to split the page into navbar and content windows. The `page` value is defined by the link in the `makeMenu` procedure. The contents of `page` file are the same as the previous form example, except that the opening `<?` and closing `?>` symbols are removed.

Example 21

BeforeScript

```
puts -nonewline {
<html>
 <head>
  <title>}
puts -nonewline Quizes
puts {</title>
  <link rel="stylesheet" type="text/css" href="/clif/simple.css" >
 </head>
}
puts {  <div id="navbar">}
puts [makeMenu]
puts "  </div>"
puts {  <div id="content">}

set page [var get page]
if {$page ne ""} {
  source $page
} else {
  puts "<h2>Select a quiz</h2>"
}
puts "    </div>"
puts "</html>"
```

simple.css

```
#navbar {
  float: left;
  width: 90px;
  padding: 20px 0px 10px 10px;
  font-size: 18px;
}
```

```
#content {
  width: 60em;
  margin: 0px 0px 0px 6em;
  padding: 1em 2em;
  border-bottom: 1px solid #ccc;
}
```

A larger, more complete example of using the `ChildInitScript` and `BeforeScript` directives is included on the companion website.

When the quiz folder has two files `timquiz.tcl` and `tclquiz.tcl` the browser displays this:

Before Selecting a Quiz

After Selecting the Tim Quiz

Rivet provides support for tracking cookies. The `cookie` command has two subcommands to `set` or `get` the value of a cookie.

Syntax: `cookie set` *`cookieName value ?-option value?`*

Sets a cookie in the user's browser.

`cookieName`	The name to associate with this cookie.
`value`	The value to associate with this cookie.

Several options are supported, including:

-days dayCount	A number of days until the cookie expires.
-hours hourCount	A number of hours until the cookie expires.
-minutes minuteCount	A number of minutes until the cookie expires.

Retrieving a cookie is done with the `cookie get` command.

Syntax: `cookie get` *cookieName*

Returns the value associated with the named cookie, or an empty string if the cookie has not been set.

cookieName	The name defined for this cookie with the `cookie set` command.

The Rivet package is in active development and use. It includes many more commands and packages with more being added frequently. The HTML documentation package that comes with the distribution describes many other features.

17.6 BWidgets

Language	`Tcl`
Primary Site	*http://sourceforge.net/projects/tcllib/*
Tcl Revision Supported	`Tcl: 8.0–8.6 and newer; Tk: 8.0–8.6 and newer`
Supported Platforms	`UNIX, MS Windows, Macintosh`

Chapter 14 described how to create complex megawidgets from the simple Tk widgets. The `BWidgets` package provides many of the commonly used megawidgets (such as tab notebooks and a tree display widget), and versions of standard Tk widgets with expanded features, including buttons and labels with pop-up help, resizeable frames, and more.

The `BWidgets` package includes extensive documentation as HTML files. The following examples demonstrate using a few of the widgets and commands.

A tabbed notebook has become a standard widget for GUI programs. It provides a good way to present different sets of related information to a user. The BWidgets package includes a `NoteBook` widget that can be configured to appear as you wish for your application.

Syntax: `NoteBook` *widgetName ?option value?...*

Create a tabbed notebook widget.

widgetName	The name for this widget, following normal Tk window naming rules.

option value A set of options and values to fine-tune the widget. The options include:

`-font` *fontDesc*	The font to use for the notebook tabs.
`-height` *height*	The height of the tab notebook.
`-width` *width*	The width of the tab notebook.
`-arcradius` *number*	Describes how round the corners of the tabs should be.

A `NoteBook` widget supports many subcommands to manipulate it, including the following.

Syntax: *notebookName* `insert` *index id ?option value?*

Insert a new tab page into the notebook. This command returns a frame to hold the content for the new page.

notebookName The name of a previously created tab `NoteBook` widget.

index The location of the new tab. May be an integer or `end`.

id An identifier for this page.

option value A set of options and values to fine-tune the widget. The options include:

`-text` *text*	The text to display in the tab.
`-createcmd` *script*	A script to evaluate the first time the tab is raised.
`-raisecmd` *script*	A script to evaluate each time the tab is raised.

Syntax: *notebookName* `raise` *id*

Raise a notebook page to the top.

notebookName The name of a previously created tab `NoteBook` widget.

id An identifier for this page.

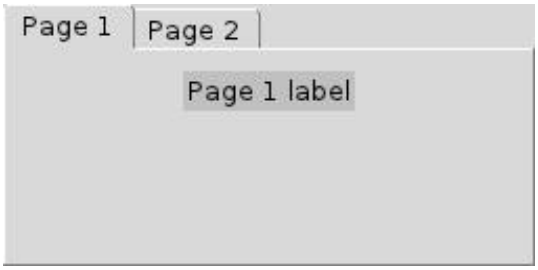
Example 22

Script Example

```
package require BWidget

set nb [NoteBook .nb -height 100 -width 250]
grid $nb
set p1 [$nb insert end page1 -text "Page 1"]
set p2 [$nb insert end page2 -text "Page 2"]
label $p1.l -text "Page 1 label" -bg #ccc
pack $p1.l
$nb raise page1
```

Script Output



In any form type application you end up with many sets of labels and associated `entry` widgets. The `LabelEntry` widget provides a simple way of creating label and entry pairs with automatic help displays.

Syntax: `LabelEntry` *`widgetName option value`*

Create a label and entry pair.

`option value` A set of options and values to fine-tune the widget. The options supported by the Tk `label` and `entry` widgets are supported, as well as the following.

`-label` *`text`*	The text to display in the label portion of this widget.
`-helptext` *`text`*	Text to display when the cursor rests on the label for a period of time.
`-font` *`fontDesc`*	Defines the font for the entry portion of this widget. This may be different from the font used by the `label` widget.
`-labelfont` *`fontDesc`*	Defines the font for the label portion of this widget. This may be different from the font used by the `entry` widget.

Example 23

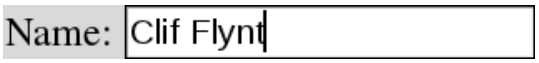
Script Example

```
package require BWidget

LabelEntry .le -font {arial 16} \
    -labelfont {Times 18} \
    -label "Name: " \
```

```
    -helptext "What is your name?"
grid .le
```

Script Output



Another common need is a window with a scrollbar. The BWidget `ScrolledWindow` widget provides a window in which the scrollbar appears when the data exceeds the display area, and vanishes if there is no need for a scrollbar.

Syntax: `ScrolledWindow` *`widgetName ?option value?`*

Create a window that will hold a scrollable widget (`canvas`, `text`, or `listbox` widget) and display scrollbars when needed.

`widgetName` The name for this widget, following normal Tk window naming rules.

`option value` A set of options and values to fine-tune the widget. The options include:

`-auto` *`type`* Specifies when to draw scrollbars. The `type` field may be:

`none`	Always draw scrollbars.
`horizontal`	Horizontal scrollbar is drawn as needed.
`vertical`	Vertical scrollbar is drawn as needed.
`both`	Both scrollbars are drawn as needed. This is the default.

The `setwidget` subcommand will assign a widget to be the scrolled widget in a `ScrolledWindow` megawidget.

Syntax: *`widgetName`* `setwidget` *`subWindowName`*

Assign *`subWindowName`* as the scrolled window in the *`widgetName`* `ScrolledWindow` widget.

`widgetName` The name for the `ScrolledWindow` widget, following normal Tk window naming rules.

`subWindowName` The name for the window to be scrolled. This window must be a child of *`widgetName`*.

Example 24

Script Example

```
package require BWidget

# Create the ScrolledWindow
```

```
set sw [ScrolledWindow .sw]

# Grid it and configure sw to change
# when the main window resizes.

grid .sw -sticky news -row 1 -column 1
grid rowconfigure . 1 -weight 1
grid columnconfigure . 1 -weight 1

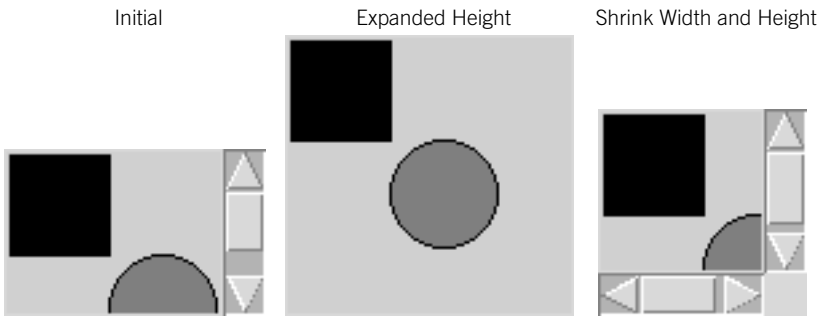
# Create a child to be held in the window
canvas $sw.cvs -height 60 -width 80 -background #ddd

# Map the child into the ScrolledWindow
$sw setwidget $sw.cvs

# Put some graphics into the canvas
$sw.cvs create rectangle 3 3 40 40 -fill black
$sw.cvs create oval 40 40 80 80 -fill gray50

# Define the scrollregion to cover everything drawn.
$sw.cvs configure -scrollregion [$sw.cvs bbox all]
```

Script Output



Using the BWidgets, a GUI for a modern library with books, music and videos might look like this:

Example 25

Script Example

```
package require BWidget

set nb [NoteBook .nb -height 100 -width 300]
grid $nb
```

```
set p1 [$nb insert end book  -text "Book"]
set p2 [$nb insert end music -text "Music"]
set p3 [$nb insert end video -text "Video"]

LabelEntry $p1.title -label "Title:     " \
    -textvariable book(title) -width 25
LabelEntry $p1.author -label "Author: " \
    -textvariable book(author) -width 25
LabelEntry $p1.subject -label "Subject:" \
    -textvariable book(subject) -width 25

button $p1.search -text "Search" -command bookSearch

grid $p1.title
grid $p1.author
grid $p1.subject
grid $p1.search

LabelEntry $p2.title -label "Title: " \
    -textvariable music(title) -width 25
LabelEntry $p2.artist -label "Artist:" \
    -textvariable music(artist) -width 25
LabelEntry $p2.genre -label "Genre:" \
    -textvariable music(genre) -width 25

button $p2.search -text "Search" -command musicSearch

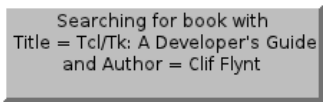
grid $p2.title
grid $p2.artist
grid $p2.genre
grid $p2.search

proc bookSearch {} {
  global book
  set top [toplevel .t]
  set notice "Searching for book with\n"
  set leader ""
  foreach in [array names book] {
    if {$book($in) ne ""} {
      append notice "$leader [string totitle $in] = $book($in)\n"
      set leader "and"
    }
  }
  label $top.l -text $notice -bg gray -borderwidth 3 \
      -relief raised
  grid $top.l
}
```

Script Output

Book Tab

Toplevel Search window



17.7 GRAPHICS EXTENSIONS: `Img`

Language	`C`
Primary Site	*`http://sourceforge.net/projects/tkimg/`*
Contact	*`Jan.Nijtmans@wxs.nl`*, Jan Nijtmans
Tcl Revision Supported	`Tcl: 7.6p2-8.6 and newer; Tk: 4.2p2-8.6 and newer`
Supported Platforms	`UNIX, MS Windows`

The `Img` extension adds support for BMP, XBM, XPM, GIF (with transparency), PNG, JPEG, TIFF, and postscript images to the Tk image object (described in Chapter 10). You must have the *libtiff*, *libpng*, and *libjpg* libraries available on your system to read TIFF, PNG, or JPEG image files. These libraries are all public domain and easily acquired from the Sourceforge `tkimg` site, or from these sites.

- TIFF library code: *ftp://ftp.sgi.com/graphics/tiff/*
- PNG library code: *ftp://ftp.uu.net/graphics/png*
- JPEG library code: *ftp://ftp.uu.net/graphics/jpeg*

17.8 BOTTOM LINE

- There are many extensions to enhance the base Tcl/Tk functionality.
- Not all extensions are available for all platforms and all Tcl revisions.
- Tcl extensions tend to lag behind the core Tcl releases.
- The primary archive for Tcl code is *core.tcl.tk*.
- Extensions are frequently announced in `comp.lang.tcl` or `comp.lang.tcl.announce`.
- Many frequently used extensions are mentioned in the FAQ.

CHAPTER

18 Programming Tools

One of the tricks to getting your job done efficiently is having the right tools for the job. There are plenty of tools available for developing Tcl applications.

This chapter provides a quick description of several program development tools that are in use in the Tcl community. It is not a complete listing of tools. If the tool you need is not mentioned here, try checking the Tcl Resource Center at *www.tcl.tk/resource/*, the announcements in `comp.lang.tcl`, the FAQs at *www.purl.org/NET/Tcl-FAQ*, or the Tcler's Wiki *http://wiki.tcl.tk/*.

Development tools for Tcl are rapidly changing and improving. This chapter discusses the versions of the tools that were available in August 2011. New versions of many of these packages will be available before the book and companion website are published.

The Open Source community is generating new ideas and packages, and commercial developers such as ActiveState, Neatware, and others have released full-featured toolkits that are constantly being improved. These options provide the Tcl developer with tools that can be configured to your needs and high-quality, commercially supported tools.

General purpose tools like `vim` and `emacs` have smart modes for editing Tcl files. The `Eclipse` (`http://blogsai.wordpress.com/2009/10/15/configuring-eclipse-as-tcltk-ide/`) and NetBeans (`http://netbeans.org`) IDE also have Tcl/Tk plugins.

Neatware (*www.neatware.com/*) and ActiveState (*www.activestate.com*) have each taken portions of the TclPro suite and enhanced them for new commercial products. Neatware has developed a Tcl-oriented IDE that includes a context-sensitive editor, debugger, code checker, byte-code compiler, and more. ActiveState has enhanced the Tcl Pro suite of tools and renamed it Tcl Dev Kit. The Tcl Dev Kit extends the TclPro suite by

- providing support for

Tcl 8.6	TkTable	BWidgets	Tk 8.6
Tcllib	TclXML	TclX	TclDOM
TclSOAP	[incr Tcl]	[incr Tk]	Img
IWidgets	Snack	TkHTML	TkCon
Tcom	Expect	TclOO	TDBC

- adding GUI front ends to the TclPro compiler and wrapper
- providing code coverage and hot spot profiling support in the debugger
- providing binaries for Linux, HP-UX, Solaris, and MS Windows

Tcl/Tk: A Developer's Guide. DOI: 10.1016/B978-0-12-384717-1.00018-X

The Tcl Dev Kit is designed to work with the commercial product Komodo Integrated Development Environment. ActiveState also supports the Komodo Editor, a free context-sensitive editor which supports Tcl, Perl, Python, Ruby, C, C++, HTML, and many other languages.

The complete Komodo Integrated Development Environment includes real-time syntax checking, GUI debugger, packaging and distribution tools, support for large projects, support for Tcl/Tk, Perl/Tk, Python/Tk, XML, HTML and JQuery, a regular expression wizard, and more.

This chapter briefly covers the following free and commercial tools.

Code Formatters

`frink`	Reformats code into a standard style for easy comprehension.

Code checkers

`tclCheck`	Checks for balanced brackets, braces, and parentheses.
`nagelfar`	Checks for syntax errors, unset or nonexistent variables, incorrect procedure calls, and more.

Debuggers

`Don Libes's Debugger`	This is a text-oriented package with support for setting breakpoints, examining data, and so on.

Packaging Tools

`FreeWrap`	Wraps a `tclsh` or `wish` interpreter with your application for distribution.
`Starkit`	Starkit wraps the entire Tcl/Tk distribution into a single file with hooks to evaluate scripted documents. This provides a middle ground between a fully wrapped and fully scripted package.

Exercising and Regression Testing

`TkTest`	Records GUI events and internal state as an application is run and then replays the events and checks that the internal state matches the previous state.

Tcl Extension Generators

`swig`	`swig` creates Tcl extensions from libraries of C functions by reading the function and data from an include file.
`CriTcl`	`CriTcl` generates a loadable Tcl extension from C code embedded into a Tcl script.

Integrated Development Environments

`Komodo`	The `Komodo` IDE supports Tcl and other languages on UNIX and Windows platforms.
`MyrmecoX`	The `MyrmecoX` IDE is optimized for Windows platform and supports several packages.

18.1 CODE FORMATTER

18.1.1 frink

Despite our best efforts, after several hours of refining our understanding of the problem (i.e., hacking code), we usually end up with badly formatted code. Incorrectly formatted code makes the logic difficult to follow and hides logical or syntax errors caused by misplaced braces. Reformatting the code can frequently help you find those errors. The `frink` program will reformat a Tcl script to make it more comprehensible and can check syntax.

Language	`C`
Primary Site	*ftp://catless.ncl.ac.uk/pub/frink.tar.gz*
Contact	*Lindsay.Marshall@newcastle.ac.uk*
Tcl Revision Supported	`Tcl: 7.3-8.6 and newer; Tk: 3.6-8.6 and newer`
Supported Platforms	`UNIX, MS Windows, Mac`

`Frink` will convert your script to a format that closely resembles the recommended style for Tcl scripts described in the *Tcl Style Guide*. As added benefits, `frink` will check the script for syntactic errors while it is reformatting the scripts, and the reformatted script will run faster. `Frink` supports several command line options to define how the code will be formatted.

`-a` put spaces around -command code in {} and "". (default = OFF)

`-A` turns OFF processing of expr calls.

`-b` add braces (see manual page for details) (default = OFF)

`-B` turns OFF processing of code with bind calls.

`-c <n>` set further indent for continuations to n. default = 2

`-C <f>` generate proc specs for use by `frink`. default = OFF

`-d` remove braces in certain (safe) circumstances (default = OFF)

`-D` warn about dynamic names. default = OFF

`-e` produce "else". (default = OFF)

`-E` extract constant strings. The parameter is the locale for which the strings are currently written. If the -f flag is also used then only the constant strings that are rewritten will be output. Output goes to a file called `<locale>.msg`. (default = OFF)

`-f` rewrite strings for msgcat (default = OFF)

`-F` selectively control heuristics. Currently the parameter is a single hex coded number with each bit representing a test. The values you need to know are:

- `00001` var parameter testing
- `00002` parameter number testing
- `00004` parameter value testing
- `00008` regexp parameter testing
- `00010` return checks

```
            00020    check for : or ::: in names
            00040    expr checks
            00080    foreach var checking
            00100    check for omitted parameters
            00200    check switches on commands
            00400    check for abbreviated options
            00800    check for unusedness
            01000    check for bad name choice
            02000    check for array usage
            04000    check for possible name errors
-g          indent switch cases (default = OFF)
-G          generate compiler style error messages (default = OFF)
-h          print this message
-H          turn on all heuristic tests and warnings (default = OFF)
-i <n>      set indent for each level to n (default = 4)
-I          treate elseif and else the same way (default = OFF)
-j          remove non-essential blank lines (default = OFF)
-J          just do checks, no output (default = OFF)
-k          remove non-essential braces
-K <f>      specify file of extra code specs
-l          try for one-liners (not yet implemented)
-m          minimize the code by removing redundant spacing (default =
            OFF)
-M          warn if there is no – on a switch statement (default = OFF)
-n          do not generate tab characters (default = OFF)
-N          do not put a newline out before elseif (default = OFF)
-o          obfuscate (not implemented yet) : default = OFF
-O <t>      don't format lines starting with token "t"
-p <v>      if v is a number produce that many blank lines after each
            proc definition, otherwise produce whatever format the code
            indicates. No codes are defined yet..... (default = do nothing)
-P          turn off processing of "time" command (default = OFF)
-q          put spaces round conditions (default = OFF)
-Q          warn about unquoted constants - not fully operational (default =
            OFF)
-r          remove comments (default = OFF)
-s <c>      format according to style "c:"
-S          don't preserve end of line comments (default = OFF)
```

-t <n>	set tabstops every n characters (default = 8)
-T	produce "then" (default = OFF)
-u	safe to remove brackets from elseif conds
-U	hardline checking enabled (default = OFF)
-v	put round variable names where appropriate
-V	the current version number
-w <n>	set line length (default = 80)
-W	halt on Warnings as well as errors
-x	produce "xf style" continuations
-X	recognize tclX constructs
-y	don't process -command code (default = OFF)
-Y	try to process dynamic code (default = OFF)
-z	put a single space before the character on continuations
-Z	control heuristics that are tested (-H turns on ALL tests)

`Frink` is distributed as source code with a `configure` file for Posix-style systems. You can compile and link this program under Microsoft Visual C++ with the following procedure.

1. Get a copy of Gnu `getopt.c` and `getopt.h`. (I found a good copy in the Gnu C library and as part of the Gnu `m4` distribution.)
2. Copy `getopt.c` and `getopt.h` to the `frink` source code directory.
3. Within `Visual C++` create a Win32 Console Application (File/New/Projects).
4. Add the files (Project/Add To Project/Files).
5. Build all.

The following example shows `frink` being used to reformat a poorly formatted piece of code.

Example 1

Badly Formatted Script

```
proc checksum \
  {
    txt
  } \
  {
    set sum 0;    foreach l [split $txt ""] \
    {
        binary scan $l c val
        set sum [expr {($sum>>1)+($sum^$val)}]
    };    return $sum
}

set data "12345678901234567890123456789012345678901234567890"
puts [time {checksum $data} 100]
```

***After** `frink`*

```
proc checksum {txt} {
    set sum 0
    foreach l [split $txt ""] {
        binary scan $l c val
        set sum [expr {($sum>>1) +($sum^$val)}]
    }
    return $sum
}

set data "12345678901234567890123456789012345678901234567890"
puts [time {checksum $data} 100]
```

The `frink` program can also perform syntax checking. You can enable various tests with the `-F` flag, and add more command syntax definitions with the `-K` flag. The format for a syntax definition is as follows.

Syntax: `cmdName { arg1 ...argN }`

`cmdName` The name of the command.

`arg*` An argument definition. May be one of:

- `var` Argument is the name of a variable.
- `any` Argument may be any value.
- `args` Any number of arguments. This value can only be the last element in the argument list.

The following example shows `frink` being used to examine a buggy code fragment.

Example 2

badcode.tcl

```
# The next line is missing a closing quote
proc foo {a b} {puts "a $b}

# The next line has too few arguments
foo b
```

syntaxdef

```
foo {any any}
```

`frink` command

```
frink -K syntaxdef badcode.tcl
```

Output

```
# The next line is missing a closing quote
*** /tmp/badcode.tcl Warning : Missing " in proc foo (line 2)
proc foo {a b} {
```

```
    puts "a $b"
}

# The next line has too few arguments
*** /tmp/badcode.tcl Warning : Call of foo with too few parameters (line 5)
foo b
```

18.2 CODE CHECKERS

One drawback of interpreted languages, as opposed to compiled languages, is that a program can run for months without evaluating a line of code that has a syntax error. This situation is most likely to occur in exception-handling code, since that is the code least often exercised. A syntax error in this code can cause your program to do something catastrophic while attempting to recover from a minor problem.

There are several code checkers that will examine your script for syntax errors, and more are being developed. Although none of these is perfect (Tcl has too many ways to confuse such programs), these will catch many bugs before you run your code.

18.2.1 tclCheck

Language	C
Primary Site	*http://catless.ncl.ac.uk/Programs/tclCheck/*
Contact	*Lindsay.Marshall@newcastle.ac.uk*
Tcl Revision Supported	Tcl: 7.3-8.6 and newer;Tk: 3.6-8.6 and newer
Supported Platforms	UNIX

This program checks for matching parentheses, braces, and brackets and a few other common programming errors. Most Tcl syntax errors are caused by these easy to catch errors (or syntax problems that are identified by `frink`). `TclCheck` and `frink` can save you hours of time you would otherwise spend staring at your code and counting braces.

`TclCheck` supports several command line flags to fine-tune its behavior. These include the following.

- `-c` By default `tclCheck` attempts to recognize comments so as to permit unmatching brackets in them. This flag turns this behavior off.
- `-e` This flag enables checking for lines that have a \ followed by spaces or tabs at their end. By default, this test is not carried out.
- `-g` By default, `tclCheck` pops its stack of brackets to a find a match with } `>`] `>`). This flag turns this off.
- `-i` This flag stops the printing of error messages beginning "Inside a string".
- `-j` Generates a compressed skeleton printout where indentation is ignored when matching brackets (see –l).

- `-l` Generates a compressed skeleton printout where nested matching lines are paired up and removed. Matching includes any indentation.
- `-m` Removes from the skeleton printout bracket pairs that match up directly on lines.
- `-q` Do not generate any output unless exceptions are detected.
- `-s` Generate a printout of the bracket skeleton of the entire program.
- `-t` Tiger mode. This will flag any single that occurs in places where Tcl would not detect the start of a string. By default this is turned off, but it can be very useful for tracking down some problems that are difficult to find.

`TclCheck` is distributed as C source code with a `Makefile`. Like `frink`, it will compile under Windows if you get a copy of Gnu `getopt.c` to link with it.

Example 3

Script Example

```
# The next line is missing a closing quote
proc foo {a b} {puts "a $b}
# The next line has too few arguments
foo b
```

Script Output

```
File ../buggy.tcl:
Inside a string: unmatched } on line 3 char 27
"missing, opened on line 3 char 22
} missing, opened on line 3 char 16
```

18.2.2 nagelfar

Language	`Tcl`
Primary Site	*http://nagelfar.berlios.de/*
Contact	*peter.spjuth@gmail.com*
Tcl Revision Supported	`Tcl: 8.4-8.6 and newer;Tk: 8.4-8.6 and newer`
Supported Platforms	`UNIX, Windows, Mac OS/X`

`Nagelfar` does a deep examination of a Tcl script and reports coding errors including mismatched braces, parentheses, quotes, incorrect argument counts, undefined commands. It also provides some code coverage analysis. It can be extended to understand new commands (your own library of procedures) and can provide varying levels of reports.

`-help`	Show usage.
`-gui`	Start with GUI even when files are specified.
`-s <dbfile>`	Include a database file. (More than one is allowed.)
`-encoding <enc>`	Read script with this encoding.
`-filter <p>`	Any message that matches the glob pattern is suppressed.

`-severity <level>`	Set severity level filter to N/W/E (default N).
`-html`	Generate html-output.
`-prefix <pref>`	Prefix for line anchors (html output).
`-novar`	Disable variable checking.
`-WexprN`	Sets expression warning level to N.
`2 (def)`	Warn about any unbraced expression.
`1`	Don't warn on single commands. "if [apa] ..." is ok.
`-WsubN`	Sets subcommand warning level to N.
`1 (def)`	Warn about shortened subcommands.
`-WelseN`	Enforce else keyword. Default 1.
`-strictappend`	Enforce having an initialized variable in (l)append.
`-tab <size>`	Tab size, default is 8.
`-header <file>`	Create a "header" file with syntax info for scriptfiles.
`-instrument`	Instrument source file for code coverage.
`-markup`	Markup source file with code coverage result.
`-quiet`	Suppress non-syntax output.
`-glob <pattern>`	Add matching files to scriptfiles to check.
`-H`	Prefix each error line with file name.
`-exitcode`	Return status code 2 for any error or 1 for warning.

The code below has many errors, which `nagelfar` detects and describes. Finding mismatched quotes is tricky, but even these can be found if you read the output carefully.

Example 4

Bad Script Example

```
# bb is an array, not a string variable
array set bb {one two}
puts $bb

# Reference an undefined variable
puts $a

# Forgot the value to assign to a
set a

set a "common error" # Should start with a semicolon

puts "mismatched quotes'

# Oops, caps lock slipped...
```

```
proc testProc {arg1 arg2] {
  puts "$arg1 and $arg2 are arguments"
}


# failed to close an internal list
set aList {
  {element pair 1}
  {element pair 2
}
```

Script Output

```
Checking file nagerr.tcl
Line   3: E Is scalar, was array
Line   6: E Unknown variable "a"
Line   9: E Unknown variable "a"
Line  11: E Wrong number of arguments (8) to "set"
Line  13: E Unknown variable "arg"
Line  13: E Wrong number of arguments (5) to "puts"
            Argument 2 at line 17
            Argument 3 at line 17
            Argument 4 at line 17
            Argument 5 at line 17
Line  17: E Extra chars after closing quote.
            Opening quote of above was on line 13.
Line  17: E Extra chars after closing quote.
            Opening quote of above was on line 13.
Line  17: E Unknown variable "arg2"
Line  17: N Unescaped quote
Line  18: E Unbalanced close brace found
Line  22: E Could not complete statement.
            One close brace would complete the first line
            One close brace would complete at end of line 25.
            One close brace would complete the script body at line 27.
            Assuming completeness for further processing.
```

If a procedure is defined within the Tcl script before it's used, `nagelfar` will report errors if the procedure is used incorrectly, as shown in the next example.

Example 5

Bad Script Example

```
proc copy {a b} {
  # Do copy
}
    set apa 1
```

```
# It should not warn about apa below
copy apa bepa
# Bepa should be known now
set cepa $bepa

# Detect $ mistake
copy apa $bepa
copy $apa bepa

copy a
copy a b c
```

Nagelfar Output

```
Checking file test2.tcl
Line   6: W Found constant "apa" which is also a variable.
Line   8: E Unknown variable "bepa"
Line  11: E Unknown variable "bepa"
Line  11: W Found constant "apa" which is also a variable.
Line  14: E Wrong number of arguments (1) to "copy"
Line  15: E Wrong number of arguments (3) to "copy"
```

If the procedures are not defined within the script that is being analyzed, they can be defined in a `.syntax` file that is loaded when the scan is performed.

When the next example is scanned, the procedure `copy` is reported as an unknown command.

Example 6

Script Example

```
set apa 1
# It should not warn about apa below
copy apa bepa
# Bepa should be known now
set cepa $bepa

# Detect $ mistake
copy apa $bepa
copy $apa bepa

copy a
copy a b c
```

Script Output

```
Line   3: W Found constant "apa" which is also a variable.
Line   3: W Unknown command "copy"
Line   5: E Unknown variable "bepa"
Line   8: E Unknown variable "bepa"
```

```
Line   8: W Found constant "apa" which is also a variable.
Line   8: W Unknown command "copy"
Line   9: W Unknown command "copy"
Line  11: W Unknown command "copy"
Line  12: W Unknown command "copy"
```

The syntax file to define copy looks like this:

```
##nagelfar syntax copy v n
```

It can be included in the code analysis by running `nagelfar` with the `-s` *`file.syntax`* option like this:

```
tclsh ls nagelfar.tcl -s fileName.syntax testFile.tcl
```

With the syntax file included, the output resembles this:

Nagelfar Output

```
Line   8: N Suspicious variable name "$bepa"
Line   9: N Suspicious variable name "$apa"
Line  11: E Unknown variable "a"
Line  11: E Wrong number of arguments (1) to "copy"
Line  12: E Unknown variable "a"
Line  12: E Wrong number of arguments (3) to "copy"
```

18.3 DEBUGGERS

Again, despite our best efforts, we spend a lot of time debugging code. These debuggers range from lightweight, compiled extensions such as Don Libes' `debug` package to full-featured GUI-based debuggers.

There are many debuggers not covered here, some of which are tailored to work with specific Tcl extensions. If the debuggers described here do not meet your requirements, use the search engine at *http://www.tcl.tk*.

18.3.1 `debug`

Language	`C`
Primary Site	*http://expect.nist.gov/*
Contact	*libes@nist.gov*
Tcl Revision Supported	`Tcl: 7.3 and newer; Tk: 3.6 and newer`
Supported Platforms	`UNIX`
Other Book References	*Exploring Expect*

This text-oriented debugger is distributed by Don Libes as part of the `expect` extension. Your script can load the `expect` extension to gain access to this feature without using the rest of the expect commands. The code for the debugger is in the files `Dbg.c`, `Dbg.h`, and `Dbg_cf.h`. With a little tweaking you can merge this debugger into other extensions.

This extension adds a `debug` command to the Tcl language. The command `debug 1` allows you to interact with the debugger; the command `debug 0` turns the debugger off. This debugger supports stepping into or over procedures and viewing the stack. This debugger supports setting breakpoints that stop on a given line and breakpoints with a command to evaluate when the breakpoint is hit. The complete Tcl interpreter is available for use while in debugger mode, so you can use normal Tcl commands to view or modify variables, load new procedures, and so on. The on-line help lists the available commands in this debugger.

```
dbg1.1> h
s [#]       step into procedure
n [#]       step over procedure
N [#]       step over procedures, commands, and arguments
c           continue
r           continue until return to caller
u [#]       move scope up level
d [#]       move scope down level
            go to absolute frame if # is prefaced by "#"
w           show stack ("where")
w -w [#]    show/set width
w -c [0|1]  show/set compress
b           show breakpoints
b [-r regexp-pattern]  [if expr] [then command]
b [-g glob-pattern]    [if expr] [then command]
b [[file:]#]           [if expr] [then command]
      if pattern given, break if command resembles pattern
      if # given, break on line #
      if expr given, break if expr true
      if command given, execute command at breakpoint
b -#  delete breakpoint
b -   delete all breakpoints
```

While in debugging mode, the normal prompt resembles `Dbg2.3>`, where the digit `2` represents the current stack level and the `3` represents the number of interactive commands that have been executed. In the sample debugging session shown next, the optional patterns and `if/then` statements are used with the break command to display the content of the variable `lst` when the breakpoint is encountered.

Example 7

Script Example

```
% cat test.tcl
proc addList {lst} {
set total 0;
foreach num $lst {
set total [expr $total + $num]
}
return $total
}
proc mean {lst} {
set sum [addList $lst]
set count [llength $lst]
set mean [expr $sum / $count]
return $mean
}
set lst [list 2 4 6 8]
puts "Mean $lst : [mean $lst]"
```

Debugging Session

```
$> expect
expect1.1> debug 1
0
expect1.2> source /tmp/test.tcl
1: history add {source /tmp/test.tcl
}
dbg1.1> b -g "*set count*" if {[llength $lst] < 10} {puts "LIST: $lst"}
0
dbg1.2> c
LIST: 2 4 6 8
3: set count [llength $lst]
dbg3.3> w
 0: expect
*1: mean {2 4 6 8}
 3: set {count} {4}
dbg3.4> n 2
3: set mean [expr $sum / $count]
dbg3.5> w
 0: expect
*1: mean {2 4 6 8}
 3: set {mean} {5}
dbg3.6> c
Mean 2 4 6 8 : 5
```

18.3.2 Graphic Debuggers

There are several GUI-oriented debuggers available for Tcl, ranging from free packages to the commercial debuggers included with the ActiveState Tcl Dev Kit and Komodo IDE or the Neatware MyrmocoX IDE.

Most of the debuggers are now packaged as part of an IDE like the Komodo IDE or MyrmocoX.

18.4 EXERCISING AND REGRESSION TESTING

Testing a GUI is usually a manual exercise with a lot of places where an almost-clicked button or misselected menu item can change results. There are many commercial tools to regenerate a set of user interactions for GUI testers.

18.4.1 TkTest

Language	Tcl/Tk
Primary Site	*http:/www.cwflynt.com/tktest/*
Contact	*clif@cflynt.com*
Tcl Revision Supported	Tcl: 8.4-8.6 and newer Tk: 8.4-8.6 and newer
Original Author	Charles Crowley, University of New Mexico

TkTest is an application that records events that occur to Tk widgets (buttons, entry, menu, etc.) and will then replay these events. While recording events it can also record internal state such as variable contents, database records, etc. During the replay it can check that the state is the same as the recorded state.

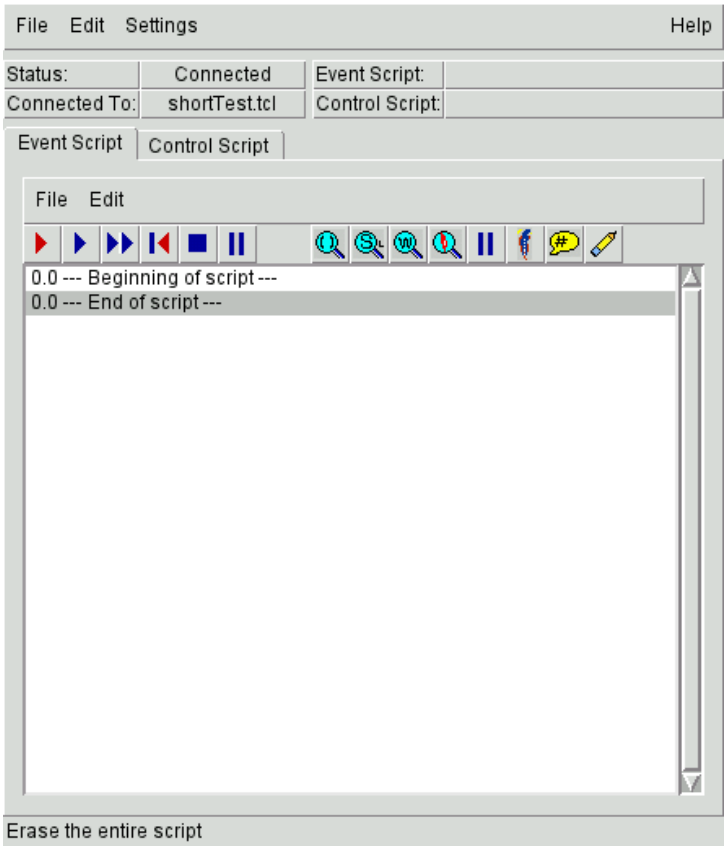
TkTest uses a small set of socket procedures to send and receive events from the application being tested. This requires a stub that checks that the application is being tested and has an IP address associated with it to be added to the test application. A stub resembles this:

Example 8

Script Example

```
if {[set pos [lsearch $argv -test]] >= 0} {
  set pos2 [expr {$pos+1}]
  set TkTestAddress [lindex $argv $pos2]
  source socksend/socksend.tcl
  sockappsetup tktest.tcl 3010  $TkTestAddress
  # Remove the -test IPAddr args
  set argv [lreplace $argv $pos $pos2]
}
```

This stub initiates the connection to TkTest.
When TkTest is connected to an application, it looks like this:



The `shortTest.tcl` script being examined is shown below:

Example 9

Script Example

```
if {[set pos [lsearch $argv -test]] >= 0} {
  set pos2 [expr {$pos+1}]
  set TkTestAddress [lindex $argv $pos2]
  source socksend/socksend.tcl
  sockappsetup tktest.tcl 3010  $TkTestAddress
  # Remove the -test IPAddr args
```

```
  set argv [lreplace $argv $pos $pos2]
}

# A simple application for testing/exercising tktest.

proc clearData {} {
  global data
  foreach nm [array names data] {
    set data($nm) ""
  }
  .lf1.t delete 0.0 end
}

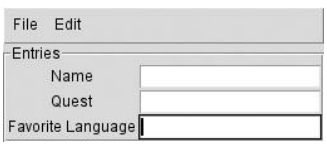
proc GUI {} {
  set f0 [menu .bf -type menubar]
  . configure -menu $f0
  $f0 add cascade -label "File" -menu $f0.file
  set m [menu $f0.file]
  $m add command -label "exit" -command exit

  $f0 add cascade -label "Edit" -menu $f0.edit
  set m [menu $f0.edit]
  $m add command -label "clear" -command clearData

  set lf1 [labelframe .lf1 -text "Entries"]
  grid $lf1

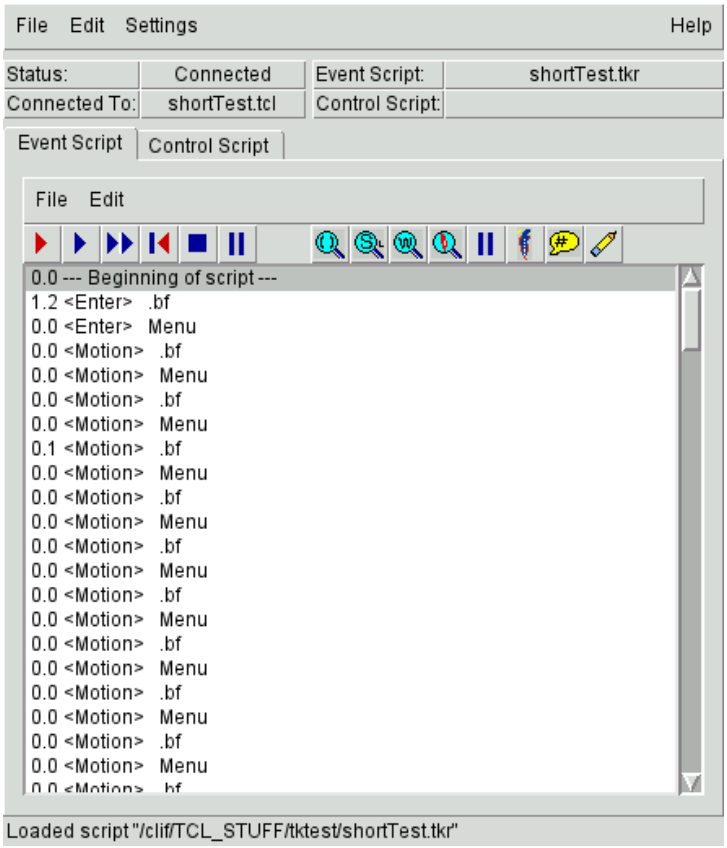
  set row 0
  foreach {ttl var} {Name name Quest quest "Favorite Language" language} {
    incr row
    set w [label $lf1.l_$var -text $ttl]
    grid $w -row $row -column 1
    set w [entry $lf1.e_$var -textvar data($var)]
    grid $w -row $row -column 2
  }
}

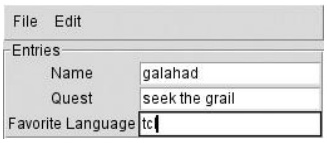

GUI
```



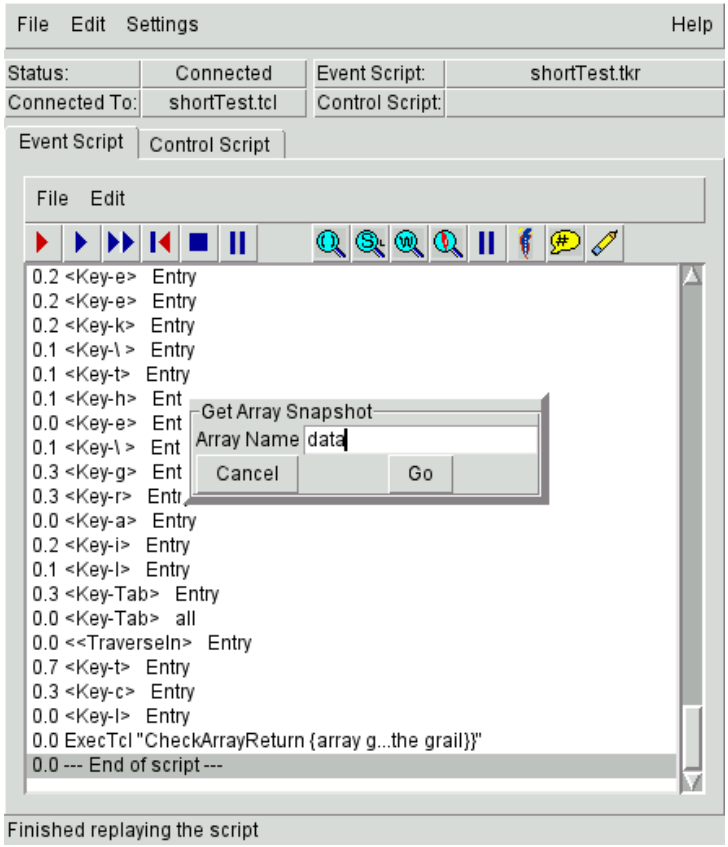
A recording session is started by clicking the leftmost triangle button in TkTest, and then performing a set of actions on the test application. As the actions are performed, the events are displayed in the TkTest's large window.

After filling in the fields the applications resemble this:



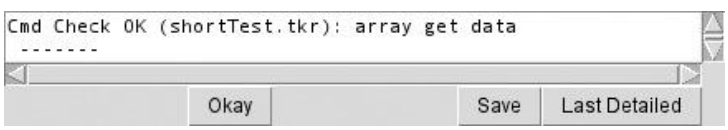


Clicking the leftmost magnifying glass button brings up a dialog that lets you enter the name of a global array to be examined.



This records the contents of the array for later comparison when the test is rerun.

After a set of events is replayed, if there are any tests, a display resembling the following is created showing the results of tests:



18.5 PACKAGING TOOLS

One of the other problems with interpreted applications is distribution and installation. Most corporations do not want to distribute their product in an easily read script format, and interpreting the scripts requires that the end user has all the required interpreters and extensions on their system.

Wrapping an application with the Tcl interpreter solves both of these problems. The script is converted to a compiled or g-zipped format, making it difficult to read, and the correct revision of the interpreter is distributed with the application. The size of a wrapped Tcl application is very similar to a compiled application linked with the appropriate libraries.

18.5.1 *freewrap*

Language	`executable`
Primary Site	*http://sourceforge.net/projects/freewrap/*
Contact	*freewrapmgr@users.sourceforge.net*
Tcl Revision Supported	`Tcl: 8.0-8.6 and newer; Tk: 8.0-8.6 and newer`
Supported Platforms	`UNIX, MS Windows, Mac, Mac OS-X`

The `freewrap` wrapper uses the zip file feature of including an executable preamble. The `freewrap` application adds support for treating a zip file as a file system to the Tcl interpreter, and then adds that interpreter to a zip file, creating a compressed executable program.

One nice feature of this technique is that if you have a copy of `freewrap` that was compiled on a target platform, you can use that copy to create an executable for that platform from any other platform. For instance, you can create Windows and Solaris executables on a Linux platform.

18.5.2 Starkit and Starpack

Language	`C, Tcl`
Primary Site	*www.equi4.com/starkit/*
Contact	*jcw@equi4.com*, *steve@digital-smarties.com*
Tcl Revision Supported	`Tcl: 8.1-8.6 and newer; Tk: 8.1-8.6 and newer`
Supported Platforms	`UNIX, MS Windows, Mac, Mac OS-X`

The standard Tcl installation includes two executable programs (`tclsh` and `wish`), a few configuration files, and several Tcl script files. The normal Tcl distribution installs all of these files in a painless fashion. However, if you are intending to distribute a simple application, you may not wish to make your end user install Tcl first.

The tclkit package merges all of the Tcl and Tk executable programs and support files into a single module using a virtual file system that mirrors a directory tree within a metakit database. The package provides tools to wrap all files involved in an application into a single file called a Starkit (STandAlone Runtime Kit). This allows you to deploy an application with just two files (the TclKit interpreter and your kit), instead of many files.

Starpack extends this idea and allow you to wrap both the TclKit executable and your Starkit into a single executable file. The program that generates Starkit and Starpack is `sdx`, which is available from *www.equi4.com/starkit*. Like many Tcl commands, the `sdx` program supports several subcommands.

Syntax: `sdx qwrap` *`file.tcl ?name ?`*

Wraps the *`file.tcl`* script into a starkit named `file.kit`, suitable for evaluating with a `tclkit` interpreter.

file.tcl A file with a Tcl script in it.

name An optional name to use instead of *file* for the name of the resulting Starkit.

```
$> cat test.tcl
package require Tk
grid [button .b -text ‘‘QUIT’’ -command exit]
$> sdx qwrap test.tcl
3 updates applied
$> ls -latr test*
-rw-r--r-- 1 clif users         64 Aug 17 16:44 test.tcl
-rwxr-xr-x 1 clif users        746 Aug 17 16:44 test.kit
$> ./test.kit   ;# Displays a button
```

Syntax: `sdx unwrap` *file.kit*

Unwraps a Starkit into a directory named `file.vfs`.

file.kit The name of a previously wrapped TclKit.

```
$> sdx unwrap test.kit
$> ls -ltr test*
-rw-r--r-- 1 clif users         64 Aug 17 16:44 test.tcl
-rwxr-xr-x 1 clif users        746 Aug 17 16:44 test.kit
test.vfs:
total 16
-rw-r--r-- 1 clif users         109 Aug 17 16:44 main.tcl
drwxr-xr-x 2 clif users        4096 Aug 17 16:56 lib
```

A standalone starpack can be built from the files in a `.vfs` directory tree with the `sdx wrap` command. The easiest way to construct this tree is to use `sdx qwrap` to wrap a file into a starkit, and then unwrap the starkit using the `sdx unwrap` command, and finally wrap a full standalone starpack with the `sdx wrap` command.

Syntax: `sdx wrap` *dirName* `-runtime` *tclkitName*

Wrap the files in *dirName.vfs* into a Starpack standalone, executable program.

dirName The prefix of a directory named *dirName*`.vfs`. All files in the *dirName*`.vfs/lib` directory will be placed in the virtual file system built into the Starpack.

tclkitName The name of a TclKit executable. This file can be for a platform other than the platform creating the wrap. Thus, you can create MS Windows or Linux executable programs on a Solaris platform, and so on.

```
$> sdx wrap test -runtime tclkit.linux
2 updates applied
$> ls -ltr test*
```

```
-rw-r--r-- 1 clif users         64 Aug 17 16:44 test.tcl
-rwxr-xr-x 1 clif users        746 Aug 17 16:44 test.kit
-rwxr-xr-x 1 clif users 2874580 Aug 17 17:03 test
test.vfs:
total 8
-rw-r--r-- 1 clif users        109 Aug 17 16:44 main.tcl
drwxr-xr-x 2 clif users       4096 Aug 17 16:56 lib
$> ./test ;# Displays button widget
```

If an application requires several `.tcl` files (and perhaps a `pkgIndex.tcl`), they can all be placed in the *`dirName`*`.vfs/lib` directory. Note that the tclkit provided in the `-runtime` option should not be the same tclkit used to evaluate `sdx`. This should be copy of the standalone tclkit.

The `sdx` application has been reworked and expanded in the ActiveState `TclApp` wrapper. An ActiveTcl distribution includes TclKits named `base_...` which can be used with either `sdx` or `TclApp`.

18.6 TCL EXTENSION GENERATORS

Chapter 15 described how to build a Tcl extension. Although the Tcl interface is very easy to work with, building an extension around a large library can mean writing a lot of code. The `SWIG` and `CriTcl` packages allow you to construct Tcl extensions without needing to learn the details of the Tcl interface. The `SWIG` package is designed to create a Tcl extension from a library of existing code, whereas the `CriTcl` package is ideal for smaller, special-purpose extensions that provide one or two new commands.

18.6.1 SWIG

Language	`C++`
Primary Site	*www.swig.org*
Contact	*beazley@cs.uchicago.edu*
Tcl Revision Supported	`Tcl: 8.0 and newer; Tk: 8.0 and newer`
Supported Platforms	`UNIX (gcc), MS Windows, Mac (PowerPC)`

The `SWIG` (Simplified Wrapper and Interface Generator) program reads a slightly modified include file and generates the Tcl interface code to link to the library API that the include file defined. `SWIG` will create Tcl commands that can access pointers and structures, as well as the standard Tcl data types. With a command line option, `SWIG` can put the new commands into a private namespace. `SWIG` can generate interfaces for almost any data construct. The package is very rich and complete.

The following example demonstrates how easily an extension can be created with `SWIG`. This example shows a subset of what `SWIG` can do. If you have an image analysis library, it would probably have a function to translate points from one location to another. The function and data definitions in the include file might resemble the following.

Example 10

```
typedef struct point_s {
  float x;
  float y;
} point;
int translatePoint(float xOffset, float yOffset,
    point *original, point *translated);
```

To convert the structure and function definition into a SWIG definition file, you would add the following items to an include (.h) file.

- A module definition name.

```
%module three_d
```

- An optional command to generate the AppInit function.

```
%include tclsh.i
```

- In-line code for the include files that will be required to compile the wrapper.

```
%{
# include "imageOps.h"
%}
```

- Dummy constructor/destructor functions for structures for which SWIG should create interfaces.

  When this is done, the definition file would resemble the following.

```
%module imageOps
%include tclsh.i
%{
#include "imageOps.h"
%}
typedef struct point_s {
float x;
float y;
point();
~point();
} point;
int translatePoint(float xOffset, float yOffset,
point *original, point *translated);
```

This input is all SWIG needs to generate code to create the structures and invoke the translate-Point function. The last steps in creating a new extension are compiling the wrapper and linking with the Tcl libraries.

```
$> swig imageOps.i
$> cc imageOps_wrap.c -limageOps -ltcl -ldl -lm
```

When this is done, you are ready to write scripts to use the new commands. The new extension has the following new commands.

Syntax: `new_point`
Create a new point structure, and return the handle for this structure.

Syntax: `point_x_set` *`pointHandle value`*

Syntax: `point_y_set` *`pointHandle value`*
Sets the value of the *`member`* field of the point structure.

`pointHandle`	The handle returned by the `new_point` command.
`value`	The value to assign to this member of the structure.

Syntax: `point_x_get` *`pointHandle`*

Syntax: `point_y_get` *`pointHandle`*
Returns the value of the *`member`* member of the point structure.

`pointHandle`	The handle returned by the `new_point` command.

Syntax: `delete_point` *`pointHandle`*
Destroys the point structure referenced by the handle.

`pointHandle`	The handle returned by the `new_point` command.

Syntax: `translatePoint` *`xOffset yOffset originalPoint translatedPoint`*
Translates a point by `xOffset` and `yOffset` distances and puts the new values into the provided structure.

`xOffset`	The distance to translate in the X direction.
`yOffset`	The distance to translate in the Y direction.
`originalPoint`	The handle returned by `new_point` with the location of the point to translate.
`translatedPoint`	The handle returned by `new_point`. The results of the translation will be placed in this structure.

A script to use these new commands would resemble the following.

Example 11

Script Example

```
# Define the original point
point original
original configure -x 100.0 -y 200.0
# Create a point for the results
point translated
# Perform the translation
translatePoint 50 -50 original translated
# Display the results
```

```
puts "translated position is: \
[translated cget -x] [translated cget -y]"
```

Script Output

```
translated position is: 150.0 150.0
```

18.6.2 CriTcl

Language	Tcl
Primary Site	*www.equi4.com/critlib/*
Contact	*jcw@equi4.com*, *steve@digital-smarties.com*
Tcl Revision Supported	Tcl: 8.0-8.6 and newer; Tk: 8.0-8.6 and newer
Supported Platforms	UNIX (gcc), MS Windows (Mingw)

One of the advantages of Tcl is that you can do your development in an easy-to-use interpreter, profile your code, and then recode the computer-intensive sections in C. CriTcl (Compiled Runtime In Tcl) makes this very easy.

CriTcl adds a new command `critcl::cproc` to allow a script to define in-line C code. When the script needs to use the C function defined by the `cproc` command, it is either generated and compiled on the fly, or loaded from a shared library generated when the application was run previously.

Syntax: `critcl::cproc` *name arguments return body*

Define an in-line C procedure.

name	The name of the new Tcl command to create from the C code.
arguments	A C function argument definition.
return	The return type of the C function.
body	The body of a C function.

```
critcl::cproc add {int x int y} int {
  return x + y;
}
```

The next example shows the same procedure for generating a simple 8-bit checksum written in both C and Tcl. Note in the next example that the char pointer is defined as a `char*`. CriTcl expects single-word data types.

Example 12

Script Example

```
package require critcl
proc checksum {txt } {
  set sum 0
  foreach l [split $txt ""] {
    binary scan $l c val
    set sum [expr ($sum >> 1) + ($sum ^ $val)]
```

```
  }
  return $sum
}
critcl::cproc checksum_c {char* txt} int {
  int sum = 0;
  char *l;
  l = txt;
  while (*l != 0) {
    sum = (sum >> 1) + (sum ^ *l);
    l++;
  }
  return sum;
}
set data "12345678901234567890123456789012345678901234567890"
puts "Tcl Checksum: [checksum $data]"
puts "C Checksum: [checksum_c $data]"
puts "Pure Tcl takes [time {checksum $data} 1000]"
puts "C code takes [time {checksum_c $data} 1000]"
```

Script Output

```
Tcl Checksum: 53
C Checksum: 53
Pure Tcl takes 1484 microseconds per iteration
C code takes 2 microseconds per iteration
```

CriTcl is deployed as a single cross-platform Starpack, which is interpreted using TclKit. CriTcl is also able to generate packaged extensions (with cross-platform support) suitable for inclusion in Starkits or other wrapping mechanisms. There is more information about these packages on the companion website.

18.7 INTEGRATED DEVELOPMENT ENVIRONMENTS

The Tcl language is easy to work with, and the interpreted nature of the language makes the edit/test sequence quick. This leads many folks to stick with traditional development tools such as the Emacs Tcl mode or other simple text editors.

There are several open source IDEs (for example, Eclipse) that support Tcl, as well as at least two commercial IDEs. Again, if you do not find anything listed here that suits your needs, check for new developments at *www.tcl.tk or comp.lang.tcl.announce*. These packages have some unique and some similar features. The following table provides an overview of the functionality.

Feature	Komodo	MyrmecoX
Highlighted commands	Y	Y
Command syntax hint	Y	Y
Command completion	Pull-down	Pull-down
Automatic quote/brace/bracket	Y	N

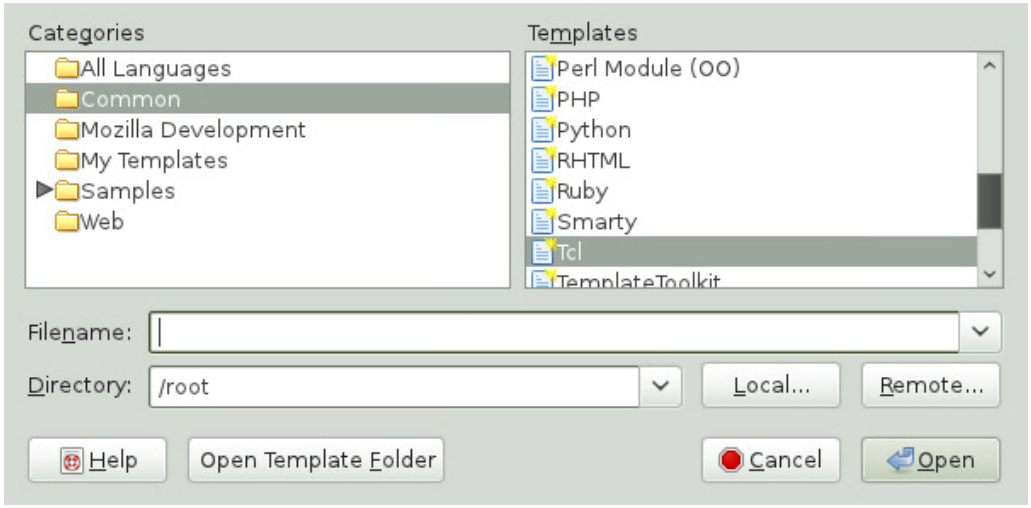
Show matching quote/brace/bracket	Y	N
Auto indent	Y	N
Highlight runtime errors	Y	N
Syntax checker	Y	Y
Code browse	N	N
Expand/shrink code segments	Y	N
Integrated console	N	Y
Integrated debugger	Y	Y

18.7.1 KomodoEdit

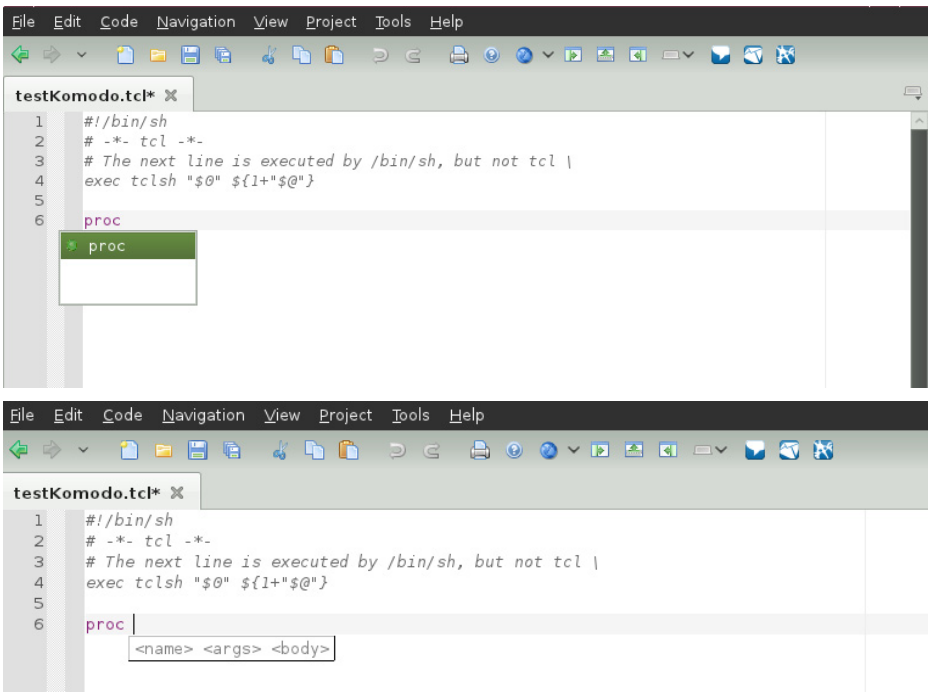
Language	`compiled`
Primary Site	*www.activestate.com/komodo-edit/*
Contact	*Komodo-feedback@ActiveState.com*
Tcl Revision Supported	`Tcl: 8.0-8.6 and newer; Tk: 8.0-8.6 and newer`
Supported Platforms	`Linux, MS Windows, Mac OS/X`

Komodo Edit is the smart editor portion of the full Komodo IDE. It lacks many of the features of the full IDE, while still providing enough features to be useful. The included features include command completion, command hints, syntax highlighting, quote/brace/bracket matching, user defined macro expansions and running a script from within the editor. The command hints make it especially useful to the person learning Tcl/Tk.

When you open a new edit session, Komodo Edit will prompt you for the type of file you are editing. This selection loads the template that will be used for name completion, popup hints and informational colors.



While entering new text, Komodo Edit supplies completion hints and syntax reminders as shown in the next two images.



18.7.2 Komodo

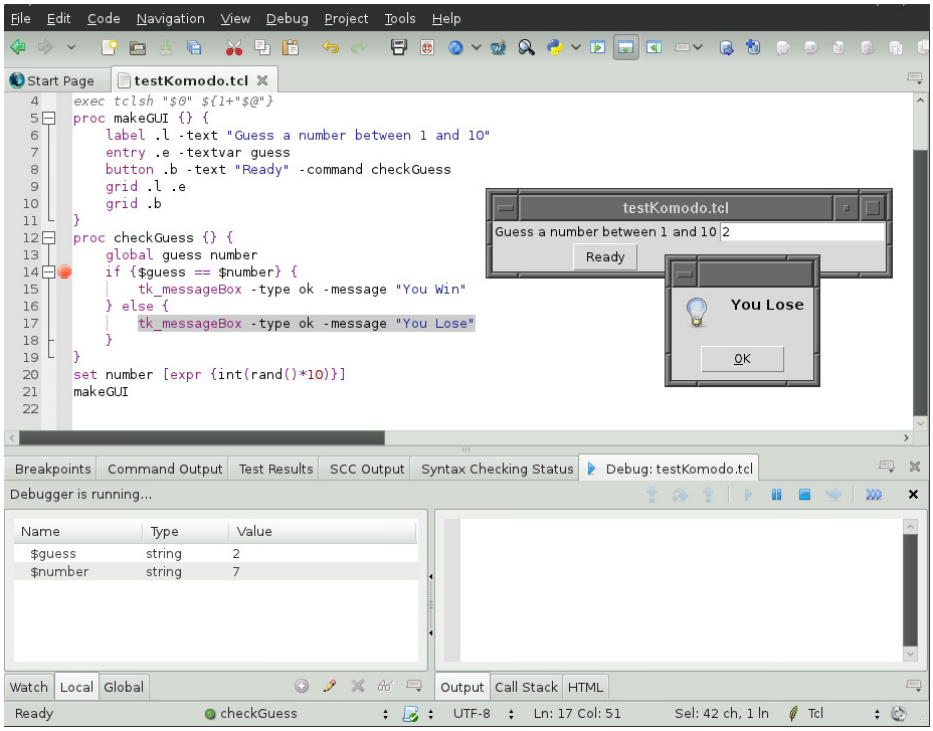
Language	`compiled`
Primary Site	*www.activestate.com/komodo-ide/*
Contact	*Komodo-feedback@ActiveState.com*
Tcl Revision Supported	`Tcl: 8.0-8.6 and newer; Tk: 8.0-8.6 and newer`
Supported Platforms	`Linux, MS Windows, Mac OS/X`

The Komodo IDE supports many languages, including Perl, Python, Tcl, PHP, XSLT, JavaScript, Ruby, Java, and others. It includes an integrated debugger, GUI layout tool, and a tool for testing and tuning regular expressions.

The context-sensitive editor will underline sections of code with syntax errors while you are typing them, to provide immediate feedback about problems. If you pause, the GUI will open a pop-up syntax reminder, or provide a pull-down menu to select subcommands. The built-in debugger supports breakpoints, stack display, variable monitoring, and other features.

The following image shows Komodo being used to debug an application.

The `mean` procedure is rolled up to save space in the edit window. The execution is currently halted at the breakpoint on line 9, with the bottom debug window displaying the current state of the program.



18.7.3 MyrmecoX

Language	Compiled
Primary Site	*www.neatware.com*
Contact	*changl@neatware.com*
Tcl Revision Supported	Tcl: 8.0-8.6 and newer;Tk: 8.0-8.6 and newer
Supported Platforms	MS Windows

The MyrmecoX IDE extends the TclPro tool set (including the byte-code compiler) to work within the framework of an IDE. The IDE supports context-sensitive editing with knowledge of the following packages.

Kernel/Util	
GUI	Tcl/Tk8.3.4, TclX/TkX 8.3, Tcllib0.8 BLT 2.4, BWidget 1.2.1, IWidget 3.0, Img 1.2, Tktable 2.6, tkWizard1.0m xWizard 2.0
Object	[incr Tcl] 3.2, [incr Tk] 3.2
Web	TclSoap 1.6.1, TclXML 2.0, TclDom 2.0
Database	TclODBC 2.1, Oratcl 3.3.

COM	`TCOM (Windows only)`
Automation	`Expect 5.2.1 (Windows only)`
Java	`TclBlend`
Security	`TLS 1.4`

The package is developed for MS Windows and supported on all flavors of Windows, including XP. Being compiled, this IDE is suitable for low-end Pentium systems, as well as 64-bit Itaniums.

18.8 BOTTOM LINE

- There are many tools available to help you develop Tcl/Tk applications. Information about tools and extensions can be found at *www.tcl.tk*, the announcements in the newsgroup `comp.lang.tcl`, the FAQs at *www.purl.org/NET/Tcl-FAQ*, and the Tcler's Wiki at *http://mini.net/tcl/*.
- You can reformat your Tcl code with `frink`.
- You can check your code for syntax errors with `tclCheck`, `frink` or `nagelfar`.
- You can debug Tcl scripts with `debug`, MyrmecoX or ActiveState Tcl Dev Kit `Debugger`.
- You can convert scripts to executable binary code with Freewrap, or TclKit.
- You can develop extensions quickly with the `SWIG` extension generator.
- You can improve an application's speed by embedding C code using CriTcl.
- Tcl applications can be developed using the Komodo, or MyrmecoX IDE tools.

CHAPTER

19 Tips and Techniques

Every language has its strengths and weaknesses, and every programmer develops ways to take advantage of the strengths and work around the weaknesses. This chapter covers some of the ways you can use Tcl more effectively to accomplish your tasks. This chapter discusses debugging, some common mistakes, some techniques to make your code more maintainable, and a bunch of odds and ends that have not been covered in previous chapters.

19.1 DEBUGGING TECHNIQUES

Testing and debugging a program are some of the most tedious parts of computer programming. The testing and debugging phase of a project can easily take more time than it took to write the application.

Testing includes both checking that the code runs at all, that it runs correctly under all circumstances, and that it runs the same way it did before you made changes.

The debuggers discussed in Chapter 18 are very useful tools but are not the only way to debug code. Tcl has some features that make it easy to debug code. The interpreted nature of Tcl makes it easy to do much of your debugging without a debugger. Since there is no separate compilation stage in Tcl, you can easily add a few debugging lines and rerun a test. The following sections discuss techniques for debugging code without a debugger.

19.1.1 Reading the Error Messages

The first step to debugging a Tcl script is to examine the Tcl error output closely. Tcl provides a verbose error information that can lead you to the exact line where a coding error occurs.

Tcl error messages consist of a set of lines. The first line will describe the immediate cause of the error (incorrect number of arguments, invalid argument, undefined variable, etc.). The rest of the message describes more details about where the error occurs.

For example, this procedure has a fairly common error—the closing brace and bracket are in the wrong order:

```
proc hasError {a} {
  return [expr {$a+2]}
}
```

The error message is:

```
missing close-bracket
    while executing
```

Tcl/Tk: A Developer's Guide. DOI: 10.1016/B978-0-12-384717-1.00019-1

```
"return [expr {$a+2]}"
    (procedure "hasError" line 2)
    invoked from within
"hasError 1"
```

The first line describes the error (missing curly bracket), and the rest of the lines show the exact line in the program

```
return [expr {$a+2]}
```

and where that line occurs (second line in the `hasError` procedure).

Here are a few of the common error messages and a description of the code that generates them.

- `wrong # args: should be ...`

  A command or procedure has been called with too few or too many arguments. This is commonly caused by forgetting to put braces or quotes around a string.

  This error is also common in code that's attached to a button, an after event or some other command that allows delayed evaluation.

  This code shows a button that will generate an error when clicked.

  ```
  set result "a b"
  button .b -text set -command "set x $result"
  ```

  This example looks reasonable, but the command to be evaluated when the button is clicked is created by substituting the `$result` and concatenating the values into a string resembling this: `set x a b`.

  The recommended way of creating a button like this is to use the `list` command to maintain grouping:

  ```
  set result "a b"
  button .b -text set -command [list set x $result]
  ```

- `missing "`
- `missing close-brace`
- `missing close-bracket`

  Your code has an unterminated string that starts with a double-quote. The usual causes of this are typos (not hitting shift fast enough and having a double quote at one end of a string and a single quote at the other), having a space after a back-slash line continuation character, or mismatching the open/close pairs of a set of nested quotes, braces and brackets.

  Here are some examples of lines that would generate one of these errors.

  ```
  set rtn "there is a space after the backslash \
           causing this to generate an error"
  puts "Missing close quote
  puts "mismatched quote and brace}
  puts "should be double quote to close'
  ```

- `can't read "...": no such variable`

  There is a `$varName` in your code for which `varName` has never been set.

This can happen:

- because you were reworking code and forgot to initialize a variable.
- because a global scope variable is referenced in a procedure without declaring it global.
- because a procedure invoked from a widget did not declare the widget's `-textvariable` to be in global scope.
- because code is being evaluated outside the scope in which it was created. Code connected with a button, after event, etc. is evaluated in the global scope (unless a namespace or class scope is assigned) even if the widget is created within a procedure or method. Procedure scope local variables will no longer exist after the GUI has taken control of the application.

```
set globalVar 1

# Fails because of missing "global globalVar"
proc hasError {} {
  puts "$globalVar"
}

# The entry widget is OK,
# but clicking the button will cause an error.
entry .e -textvariable globalVar
button .b -text "generateError" -command hasError

# This button references a local variable.

proc makeBadButton {} {
  set xx "local variable"
  button .b -text "throw error" -command "puts $xx"
  grid .b
}
```

- `invalid command name "..."`
  The first word on a command line is not a valid command or procedure name. This is most often caused by mis-typing a command name, or forgetting to `source` or `package require` the Tcl code that defines a command or procedure.
- `syntax error in expression "...":  variable references require preceding $`

  `invalid bareword "..."`

  This error is generated by the `expr` command. It's caused when you try to do arithmetic on a string. The usual causes are that you forgot to put a dollar-sign on a variable name or that a variable that should hold a number was assigned a string value.

19.1.2 Instrumenting Code to Generate Log Files

The Tcl error return provides information about syntax errors and sudden death failures. The more insidious problems are the ones that only occur after an application has run for a while.

If the problem appears the first time a condition occurs, it can be easy to diagnose with a GUI debugger. If it occurs after a period of running, you may need to track the program's internal state and generate log files to figure out when a data corruption or an incorrect procedure call occurs.

Examine the Call Stack with the `info level` *Command*

Sometimes a bug appears in a procedure that is invoked with bad arguments. In order to fix the bug, you need to know which code invoked the procedure. You can learn this with the `info level` command.

Syntax: `info level` *`?levelValue`*

If invoked with no *`levelValue`* argument, `info level` returns the stack level currently being evaluated. If invoked with a *`levelValue`* argument, `info level` returns the name and arguments of the procedure at that stack level.

`levelValue` If *`levelValue`* is a positive value, it represents the level in the procedure stack to return. If it is a negative value, it represents a stack location relative to the current position in the stack.

The next example shows `info level` being added to a procedure where an error is seen. The actual cause of the error is the procedure call in the `causesError` procedure where a dollar-sign is missing.

Example 1

Script Example

```
proc reportsError {a} {
  # Added info level to track how this procedure is called.
  puts "reportsError called as: [info level -0]"
  puts "reportsError called from: [info level -1]\n"

  return [expr {$a + 2}]
}

proc causesError {b} {
  reportsError b
}

proc worksOK {c} {
  reportsError $c
}

worksOK 2
causesError 3
```

Script Output

```
reportsError called as: reportsError 2
reportsError called from: worksOK 2
```

```
reportsError called as: reportsError b
reportsError called from: causesError 3

can't use non-numeric string as operand of "+"
    while executing
"expr {$a + 2}"
    (procedure "reportsError" line 6)
    invoked from within
"reportsError b"
    (procedure "causesError" line 2)
    invoked from within
"causesError 3 "
    (file "foo.tcl" line 19)
```

Examine Variables When Accessed with `trace` *Command*

Sometimes you know that a variable has an invalid value by the time it is used in a procedure and you know its value was valid when it was defined. Between the time the variable was set and when it was used, something changed the value but you do not know what. The `trace` command will let you evaluate a script whenever a variable is accessed. This makes it easy to track which procedures are using (and changing) variables.

Syntax: `trace add variable` *`name operations command`*

Puts a `trace` on a variable that will cause a procedure to be evaluated whenever the variable is accessed.

`name` The name of the variable to be traced.

`operations` Whenever a selected operation on the variable occurs, the *`command`* will be evaluated. `Operation` may be one of:

- `array` Evaluate the command whenever the variable is accessed via the `array` command.
- `read` Evaluate the command whenever the variable is read.
- `write` Evaluate the command whenever the variable is written.
- `unset` Evaluate the command whenever the variable is unset.

`command` A command to evaluate when the variable is accessed. The command will be invoked with the following three arguments.

- `name` The name of the variable.
- `index` If the variable is an associative array, this will be the index of the associative array. If the variable is a scalar variable, this will be an empty string.
- `operation` A single letter to denote the operation that triggered this evaluation.

The following example shows how the `trace` command can be used with a procedure (`traceProc`) to display part of the call stack and the new value when a variable is modified. If you find yourself using traces frequently, you may want to examine the `OAT` extension and `TclProp` package from the University of Minnesota (*www.cs.umn.edu/research/GIMME/tclprop.html*). The `OAT` extension enhances the `trace` command, allowing all Tcl and Tk objects (such as widgets) to be traced. The `TclProp` package (among other features) lets you watch for variables being changed with a single line command instead of needing to declare a script.

Example 2

Script Example

```
# The procedure to invoke from trace

proc traceProc {varName index operation} {
  upvar $varName var
  set lvl [info level]
  incr lvl -1;
  puts "Variable $varName is being modified in:
    `[info level $lvl]'"
  if {$lvl > 1} {
    incr lvl -1;
    puts "  Which was invoked from: `[info level $lvl]'"
  }
  puts "The current value of $varName is: $var\n"
}

# Example Script using traceProc


# A procedure to modify the traced Variable

proc modifyVariable {newVal} {
  global tracedVariable
  set tracedVariable $newVal
}

# A procedure to call the variable changing procedure,
#  to demonstrate getting information further up the
#  call stack.

proc otherProc {newVal} {
  modifyVariable $newVal
}

# A variable to watch.
```

```
global tracedVariable

# Create a trace on the tracedVariable variable

trace variable tracedVariable w traceProc

# Now, modify the variable, twice within procedures,
# and once from the global scope.

# This will modify the variable two levels deep
# in the call stack.

otherProc One

# This will modify the variable one level down the call stack.

modifyVariable Two

# Notice that the change from global scope reports itself as
# coming from the 'traceproc' command

set tracedVariable Three
```

Script Output

```
Variable tracedVariable is being modified in:
    'modifyVariable One'
  Which was invoked from: 'otherProc One'
The current value of tracedVariable is: One

Variable tracedVariable is being modified in:
    'modifyVariable Two'
The current value of tracedVariable is: Two

Variable tracedVariable is being modified in:
    'traceProc tracedVariable {} w'
The current value of tracedVariable is: Three
```

19.1.3 Run Script in Interactive Mode

You can invoke a script from a `tclsh` or `wish` interpreter with the `source` command. When a script is invoked with the `source` command, the interpreter returns you to the interactive prompt instead of exiting on an error. The `tkcon` program described in Chapter 2 is an excellent tool to use for interactive mode debugging. The ability to recover previous commands and edit them slightly makes it easy to run multiple tests and observe how a procedure behaves.

If your script requires command line arguments, you can set the `argv` and `argc` variables before you source the script, as shown in the following example. The example shows a simple debugging session that tries to open a nonexistent file, and then evaluates the procedure that would be called if a valid file name were provided.

Example 3

Script named `interactive.tcl`

```
proc processLineProcedure {line} {
  puts "Processing \"$line\""
}

puts "There are $argc arguments to this script"
puts "The arguments are: $argv"

set fileName [lindex $argv 0]
set infile [open $fileName "r"]
while {![eof $infile]} {
  set len [gets $infile line]
  if {$len > 0} {

    processLineProcedure $line
  }
}
```

Interactive Session Output

```
% set argv BadFile
BadFile
% set argc 1
1
% source interactive.tcl
There are 1 arguments to this script
The arguments are: BadFile
couldn't open "BadFile": no such file or directory
% set argv interactive.tcl
interactive.tcl
% source interactive.tcl
There are 1 arguments to this script
The arguments are: interactive.tcl
Processing "proc processLineProcedure {line} {"
Processing "  puts "Processing \"$line\""""
Processing "}"
Processing " "
Processing "puts "There are $argc arguments to this script""
Processing "puts "The arguments are: $argv""
Processing "set fileName [lindex $argv 0]"
```

```
Processing "set infile [open $fileName "r"]"
Processing "while {![eof $infile]} {"
Processing "  set len [gets $infile line]"
Processing "  if {$len > 0} {"
Processing "    processLineProcedure $line"
Processing "  }"
Processing "}  "
```

19.1.4 Use `puts` to Print the Value of Variables or Lines to Be Evaluated

This technique may not be elegant, but it works in all environments. You may need to print the data to a file instead of the screen if you are trying to debug an application that runs for a long time, does something strange, recovers, and continues running. Some of the variants on using `puts` to track program flow and variables include:

A Conditional `puts` in a Procedure

This technique is useful while you are in the latter stages of debugging a system. With this technique you can leave your debugging code in place, while confirming the behavior without the extra output.

```
proc debugPuts {args} {
  global DEBUG
  if { [info exists DEBUG] && $DEBUG } {
    puts "$args"
}
```

A Bitmapped Conditional

A bitmap can be used to control how much information is printed while the script is running. This is another technique that is useful when you do not always want to see a lot of output, but for tracking down some bugs you need all the help you can get.

The next example shows a procedure that will print different amounts of information, depending on the value of `stateArray (debugLevel)`. If bit 8 is set, a call stack is displayed. Note that a positive (absolute) level is used for the `info level` command rather than a negative (relative) level. This is a workaround for the fact that Tcl generates an error when you try to access level 0 via a relative level offset but will accept 0 as an absolute level.

For example, a procedure that is called from the global level is at level 1 (as returned by `[info level]`). If the `info level` command is invoked within that procedure as `info level -1`, Tcl will generate an error. However, if the `info level` command is invoked as `info level 0`, Tcl will return the procedure name and argument that invoked the procedure.

Example 4

Script Example

```
proc debugPuts {args} {
    global stateArray
```

```
    # If low order bit set, print the message(s)

    if {$stateArray(debugLevel) & 1} {
        puts "DEBUG Message: $args"
    }

    # If bit two is set, print the proc name and args that
    #  invoked debugPuts

    if {$stateArray(debugLevel) & 2} {
        set level [info level]
        incr level -1

        puts "DEBUG Invoked from:: [info level $level]"
    }

    # If bit four is set, print the contents of stateArray

    if {$stateArray(debugLevel) & 4} {
        foreach index [array names stateArray] {
            puts "DEBUG: stateArray($index): $stateArray($index)"
        }
    }

    # If bit 8 is set, print the call stack

    if {$stateArray(debugLevel) & 8} {
        set level [info level]
        for {set l 1} {$l < $level} {incr l} {
            puts "DEBUG CallStack $l: [info level $l]"
        }
    }
}

## Example of use

proc topProc {arg1 arg2} {
  lowerProc three four five
}

proc lowerProc {args} {
  debugPuts "Message from a proc called from a proc"
}

set stateArray(debugLevel) 15
topProc one two
```

Script Output

```
DEBUG Message: {Message from a proc called from a proc}
DEBUG Invoked from:: lowerProc three four five
DEBUG: stateArray(debugLevel): 15
DEBUG CallStack 1: topProc one two
DEBUG CallStack 2: lowerProc three four five
```

Printing Every Line

To debug some nasty problems, you may want to display each line before it is evaluated. Sometimes, seeing the line with the substitutions done makes the script's behavior obvious. You can modify your script to display each line before executing it by duplicating each line of code and adding a string resembling

```
puts "WILL EVAL:
```

to the beginning of the duplicated line, and a close quote to the end of the line. This modification is easily done with an editor that supports macros (such as `Emacs`). Alternatively, you can modify your script with another Tcl script. The caveat with this technique is to be careful that you do not accidentally perform an action within your `puts`. Consider the following example.

Example 5

```
# This will evaluate the modifyDatabase procedure.
#  It will print the status return, not the procedure call.
puts "WILL EVAL set status [modifyDatabase \$newData]"
# This prints the command and the value of newData
puts "WILL EVAL: set status \[modifyDatabase $newData\]"
set status [modifyDatabase $newData]}
```

19.1.5 Extract Portions of Script for Unit Testing

Since Tcl does not have a separate compilation stage, unit testing can be more easily performed in Tcl than in languages such as C, which might need to have more of the program infrastructure in place in order to link a test program.

19.1.6 Attach a `tkcon` Session to Application

The `tkcon` application can attach itself to another running a `wish` interpreter, providing a console interface to a running application. With this console you can examine global variables, check the status of `after` events, interact with windows, and so on. This is an excellent way of examining the state of an application running outside a debugger that displays an unexpected error. The following illustration shows how to attach a TkCon console to another `wish` application.

The menu to attach to another application is under the Console menu selection. The image below shows TkCon being attached to the `taskbar.tcl` application.

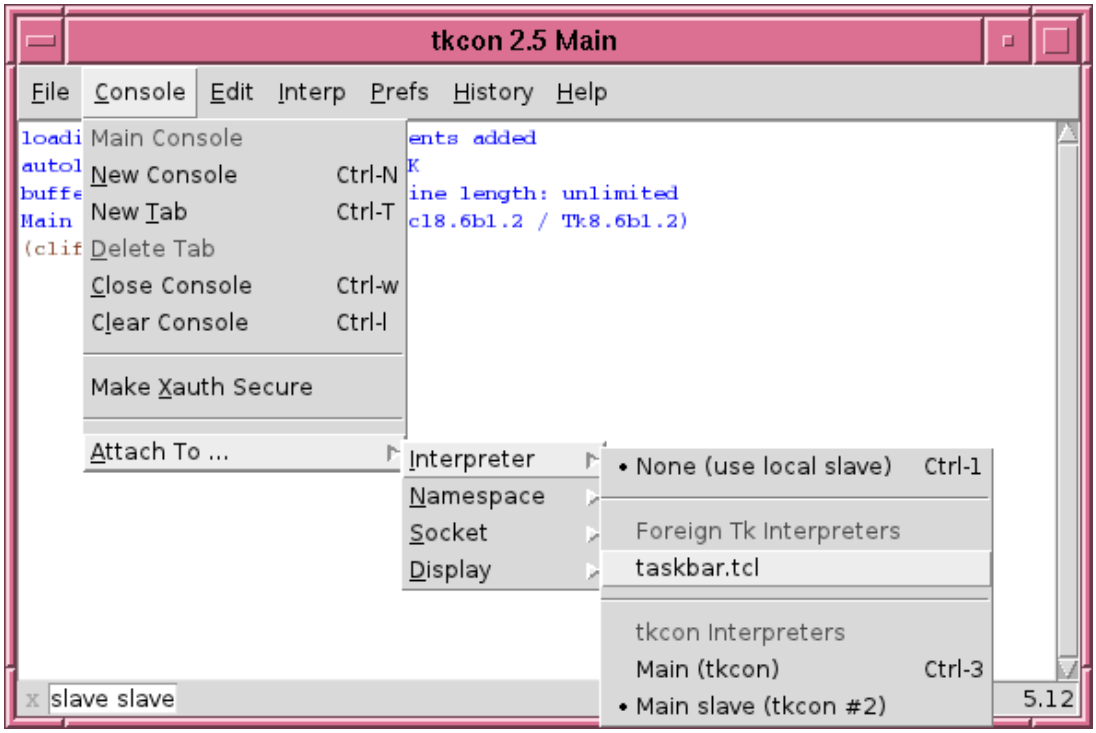


FIGURE 19.1

TkCon being attached to the taskbar.tcl application.

Note that TkCon can only attach itself to a running task when using X-Windows. When using Microsoft Windows, the `console` command can be run within an application to get similar control.

19.1.7 Create a Console Window under Windows

The Windows distributions for Tcl/Tk include a `console` command.

The command `console show` will create a console window, and `console hide` will remove the window.

19.1.8 Create a Command Window to Interact with Your Application

If you cannot use the `tkcon` program or the `console` command, you can add code to a script that will allow you to evaluate commands within the running application and display results.

The `makeInteractionWindow` procedure in the following example creates a new top-level window that accepts Tcl commands, evaluates them, and displays the results. The images show this procedure being used to examine a `radiobutton` application.

Example 6
Interaction Window Example

```
# Create a window to use to interact with the running script

proc makeInteractionWindow {} {
  # Can't have two toplevels open at once.
  if {[winfo exists .topInteraction]} {return}

  # Create a toplevel window to hold these widgets
  set topWindow [toplevel .topInteraction]

  # Create an entry widget for commands to execute,
  #   and a label for the entry widget

  set cmdLabel [label $topWindow.cmdLabel -text "Command"]

  set cmd [entry $topWindow.cmd -textvariable cmdString \
    -width 60]

  grid $cmdLabel -row 0 -column 0
  grid $cmd -row 0 -column 1 -columnspan 3

  # Create a scrolling text window for the command output

  set outputLabel [label $topWindow.outputLabel \
    -text "Command Result"]

  set output [text $topWindow.output -height 5 -width 60 \
    -relief raised]

  set sb [scrollbar $topWindow.sb -command "$output yview"]
  $output configure -yscrollcommand "$sb set"

  grid $outputLabel -row 1 -column 0
  grid $output -row 1 -column 1 -columnspan 3
  grid $sb -row 1 -column 4 -sticky ns

  # Create buttons to clear the command,
  #  execute the command and
  #  close the toplevel window.

  set clear [button $topWindow.clear -text "Clear Command" \
```

```
        -command { set cmdString "" }]

  # The command for the $go button is placed in quotes to
  #  allow "$output" to be replaced by the actual output
  #  window name. "$output see end" scrolls the window to
  #  the bottom after each command (to display the command
  #  output).

  set go [button $topWindow.go -text "Do Command" \
    -command "$output insert end \"\[uplevel #0 \
        \[list eval \$cmdString\]\n\]\"; \
        $output see end; \
       "]

  set close [button $topWindow.close -text "Close" \
    -command "destroy $topWindow"]

  grid $clear -row 2 -column 1
  grid $go  -row 2 -column 2
  grid $close -row 2 -column 3

  # bind the Return key in the command entry widget to
  #  behave the same as clicking the $go button.
  #  NOTE: <Enter> is the "cursor enters widget"
  #  event, NOT the Enter key

  bind $cmd "<Return>" "$go invoke"
}


#
# Example of interaction window
#

if {[lsearch $argv -debug] != -1} {
    makeInteractionWindow
}

# Update the displayed text in a label

proc updateLabel {myLabel item} {
  global price;
  $myLabel configure -text \
  "The cost for a potion of $item is $price gold pieces"
}

# Create and display a label
```

```
set l [label .l -text "Select a Potion"]
grid $l -column 0 -row 0 -columnspan 3

# A list of potions and prices

set itemList [list "Cure Light Wounds" 16 \
    "Boldness" 20 \
    "See Invisible" 60]
set position 0

foreach {item cost} $itemList {
  radiobutton .b_$position -text $item -variable price \
      -value $cost -command [list updateLabel $l $item]
  grid .b_$position -column $position -row 1
  incr position
}
```

Script Output

19.1.9 Use a Second `wish` Interpreter for Remote Debugging

The `send` command allows you to send a command from one `wish` interpreter to another. The command will be evaluated in the target interpreter and the results will be returned to the source interpreter. This feature lets you use one `wish` interpreter in interactive mode to debug a script running in another interpreter. You can display variables, evaluate procedures, load new procedures, and so on.

Syntax: `send` *`interp cmd`*

Send a command to another `wish` interpreter, and return the results of that command.

interp The name of the destination interpreter. You can get a list of the `wish` interpreters currently running on a system with the `winfo interps` command.

cmd A command to be evaluated in the destination interpreter. Remember to enclose the `cmd` string in curly braces if you want substitutions to be performed in the destination interpreter, not the source interpreter.

All of the preceding techniques can be done in a remote interpreter using the `send` command. You can print out data, invoke procedures, check the state of the `after` queue, or even `source` new versions of procedures. Note that using the `send` command on a UNIX system requires that you use `xauth` security (or build `wish` with a reduced security). Under Windows, versions of Tk prior to 8.1 do not support the `send` command.

19.2 TCL AS A GLUE LANGUAGE: THE `exec` COMMAND

Although almost any application can be written in Tcl, not all applications should be written in Tcl. Some applications should be written using Tcl with an extension, and some should be written using Tcl as a glue language to join other standalone programs.

Scripting languages are extremely useful for applications that require gluing several pieces of functionality into a new application. They are less well suited to creating base functionality such as the following.

- Arithmetic-based algorithms (generating checksums or compressing files)
- Large data applications (subsampling images)
- Controlling hardware (device drivers)

Tcl/Tk's strength is how easy it makes merging several libraries and standalone programs into a complex application. This merging can be done by creating new Tcl extensions or by using Tcl to glue several standalone programs into a new application. If the functionality you need involves several functions and is available in an existing library, it is probably best to make a Tcl extension wrapper to access the library. See Chapter 15 on writing extensions and Chapter 18 on the `SWIG` and `CriTcl` packages for automating creating extensions from libraries.

The extensions you create can either be linked into a new interpreter or merged at runtime with the `load` command. Note that you can use the `load` command only if your operating system supports dynamically loaded libraries (`.so` under Linux and Solaris, `.dll` under Windows). If standalone applications exist that perform a subset of what you need, you can use these existing applications with a Tcl script to control and extend them to perform the tasks you need done.

Many applications are easily written using Tcl and extensions for some functionality, and invoking standalone programs for other functionality. For example, I've used a Tk script to control the PPP connections on a UNIX system. It used the `BLT` extension to create an activity bar chart, invoked several standalone programs to initiate PPP connections and monitor the activity, and had some Tcl code to collect the data from various sources and report the number of connection hours.

The caveat with calling standalone programs from your script is that it can limit the portability of the script. For instance, a script that uses `ps` to monitor active processes and display the top CPU users

will require different ps arguments for BSD and System V-based UNIXes and will not run at all on a Macintosh or Windows platform.

The command that lets you merge existing standalone programs into a new application is the exec command. The exec command will invoke new programs in a subprocess and return the program output to the Tcl interpreter as the result of the command. If the subtask exits with a non-zero status (an error status), the exec command will generate an error. You can invoke a single program with the exec command, or a series of commands where the output of one program becomes the input to the next program with the UNIX pipe symbol (|). The argument to the exec command is either an exec option, a command to execute as a subprocess, an argument for the commands, or a pipeline descriptor.

Syntax: exec *?-options? arg1 ?arg2...argn?*

Execute arguments in a subprocess.

-options The exec command supports two options:

- *-keepnewline* Normally, a trailing newline character is deleted from the program output returned by the exec command. If this argument is set, the trailing newline is retained.
- -- Denotes the last option. All subsequent arguments will be treated as subprocess program names or arguments.

*arg** These arguments can be either a program name, a program argument, or a pipeline descriptor. There are many pipeline descriptors. Some of the commonly used ones are:

- | Separates distinct programs in the command line. The standard output of the program on the left side of the pipe symbol (—) will be used as the standard input for the program on the right of the pipe symbol.
- < *fileName* The first program in the list will use the content of fileName as the standard input.
- > *fileName* The standard output from the last program in the list will be written to *fileName*.

The following examples create compressed archives of files in a directory under UNIX or Windows using the exec command.

19.2.1 Creating a G-zipped tar Archive under UNIX

Example 7

Script Example

```
# This script is called with the name of a directory to
#  archive as the first argument in the command line,
```

```
#  and the name of the archive as the second argument.

set directory [lindex $argv 0]
set archive [lindex $argv 1]

# Get a list of the files to include in the archive.

set fllst [glob $directory/*]

# Create the tar archive, and gzip it.

eval exec tar -cvf $archive $fllst
exec gzip $archive
```

19.2.2 Creating a Zip Archive under Windows

Example 8

```
# This script is called with the name of a directory to
#  archive as the first argument in the command line,
#  and the name of the archive as the second argument.

set directory [lindex $argv 0]
set zipfile [lindex $argv 1]

# The file "distfiles" will contain a list of files for
#  this archive.

set outfl [open distfiles "w"]

# Get a list of the files to include in the archive.

set fllst [glob $directory/*]

# And write that list into the contents file, one
# filename per line

foreach fl $fllst {
 puts $outfl "$fl"
}

close $outfl
# Execute the winzip program to make an archive.
eval "exec C:/winzip/winzip32.exe -a $zipfile @distfiles"
```

19.3 COMMON MISTAKES

There are several common mistakes programmers make as they learn Tcl. Cameron Laird has collected many of these from the questions and answers posted in `comp.lang.tcl`, at *http://phaseit.net/claird/comp.lang.tcl/fmm.html*. This section is a sampling of common errors not discussed in the previous chapters.

19.3.1 Problems Using the `exec` Command

The previous description of the `exec` command looks straightforward, but there are some common mistakes people make trying to use the `exec` command.

The `tclsh` Shell Is Not the UNIX Shell

Many problems appear when people try to use shell expansions or shell escapes with the `exec` command. For example, the wrong way to write the `exec tar...` command in the previous example would be:

```
exec tar -cvf $archive $directory/*
```

In this case, the `tclsh` shell would substitute the name of the directory (for example, `foo`) for `$directory` and would pass that string (`foo/*`) to the `tar` program. The `tar` program would fail to identify any file named "*" in that directory. When you type a command with a "*" at the shell prompt (or in an `sh` script), the shell automatically expands the * to the list of files. Under Tcl, this expansion is done by the `glob` command.

In the same way, if you try to group an argument to `exec` with single quotes, it will fail. The single quote has meaning to the UNIX shell interpreter (disable substitutions) but is just another character to the Tcl interpreter.

The `tclsh` Shell Is Not `COMMAND.COM`

This is the Windows equivalent of the previous mistake. Remember that the `DIR`, `COPY`, and `DEL` commands are part of `COMMAND.COM`, not standalone programs. You cannot get a directory within a Tcl script with a command such as the following.

```
# Won't work
set filelist [exec dir C:*.*]
```

There is no `DIR.EXE`, or `DIR.COM` for the Tcl shell to execute. The best solution is to use the Tcl `glob`, `file copy`, `file delete`, and `file rename` commands. If you really need the `DIR` output you can `exec` the `COMMAND.COM` program with the `/C` option. This option tells the `COMMAND.COM` program to evaluate the string after the `/C` as though it had been typed in at a command prompt.

```
# This will get a list of files
set filelist [glob C:/*.*]
# This will get the output of the dir command
set filelist [exec COMMAND.COM /C dir C:*.*]
```

Note that `filelist` will contain all the output from the `DIR`, not just a list of the files.

A Tcl List Is Passed as a Single Argument to a Procedure

Notice the `eval` in the `eval exec tar -cvf $archive $fllst` line in the previous UNIX example. If you simply used

```
# won't work.
exec tar -cvf $archive $fllst
```

The content of `fllst` would be passed as a single argument to the `exec` command, which would pass them as a single argument to the `tar` program. Since there is no file named `file1 file2 file3..`, this results in an error.

The `eval` command concatenates all of the arguments into one string and then evaluates that string. This changes the list from a single argument to as many arguments as there are file names in the list.

```
# will work.
eval exec tar -cvf $archive $fllst
```

A new operator was introduced with Tcl 8.5. The {*} operator splits a list into its component parts. This is the preferred way to split a list into individual elements. Using the {*} operator limits expansions to the variable it's associated with, avoiding potential errors when all the variables in a command string lose a level of grouping.

```
# will work.
exec tar -cvf $archive {*}$fllst
```

Changing the Directory Is Done within a `tclsh` (or `sh`) Shell; It Is Not a Standalone Program

The command `exec cd $newDirectory` is probably not what you want.

The `change directory` command is a part of a shell. It changes the current directory within the shell. It is not an external program. Use the built-in `Tcl` command `cd` if you want to change the working directory for your script.

19.3.2 Calculating the Time: Numbers in Tcl

Numbers in Tcl can be represented in octal, decimal, or hexadecimal. The base of a number is determined by the first (or first two) digits. If the first digit is not a 0, the number is interpreted as a decimal number. If the first digit is a 0 and the second character is x, the number is interpreted as a hexadecimal number. If the first digit is a 0 and the second digit is between 1 and 7, the number is interpreted as an octal number. If the first digit is a 0 and the second digit is 8 or 9, this is an error.

In versions of Tcl without the `clock` command, it was common to split a time or date string and try to perform calculations on the parts of the time. This would work most of the time. Note that at 08:00 the command

```
set minutes [expr [lindex [split $time ":"] 0]*60]
```

will generate an error, since `08` is not a valid octal number. If you need to convert a time/date to seconds in versions of Tcl more recent than 7.5, you should use the `clock format` and `clock scan` commands.

If you need to process time and date in an older version of Tcl, or if you are reading data that may have leading zeros (the number of cents in a commercial transaction, for instance), use the `string`

trimleft command to remove the leading zeros. In the following example, note the test for an empty string. If the initial value is all zeros (000, for instance), the string trimleft command will remove all the zeros, leaving an empty string. An empty string will generate an error if you try to use it in a calculation.

```
set value [getNumberWithLeadingZeros]
set value [string trimleft $value "0"]
if {$value eq ""} {set value 0}
```

19.3.3 set, lset, lappend, append, and incr Are the Only Tcl Commands That Modify an Argument

The commands lreplace, string, and expr all return a new value without modifying any original argument.

```
# This WON'T remove the first list entry
lreplace $list 0 0 ""
# This WILL remove the first list entry
set list [lreplace $list 0 0 ""]
# This WON'T shorten the string
string range $myString 5 end
# This WILL shorten the string
set myString [string range $myString 5 end]
# This WON'T change the value of counter
expr $counter + 1
# These commands WILL change the value of counter
set counter [expr $counter + 1]
incr counter}
```

19.3.4 The incr Command Works Only with Integers

Some commands (for example, commands that return locations and widths of objects on a canvas) return float values that incr will not handle. You can convert a floating-point number to an integer with the expr command's round() and int() functions, or you can use the expr command to increment a variable. Remember that you must reassign the new value to the variable when using the expr command.

```
set variable [expr $variable + 1.0]
```

19.3.5 The upvar Command Takes a Name, Not a Value

When you invoke a procedure that uses the upvar command, you must pass the variable name, not *$varName*.

```
proc useUpvar {variableName} {
  upvar $variableName localVariable
  set localVariable 0
}
# This will set the value of x to 0
useUpvar x
```

```
# This will not change the value of x
useUpvar $x
```

19.3.6 Changes to Widgets Do Not Appear Until the Event Loop Is Processed

If your `wish` script is making many modifications to the display within a procedure, you may need to add the `update` command to the looping. This will not only update the screen and show the progress but also scan for user input (such as someone clicking an abort button). This is discussed in detail in Chapter 11.

19.3.7 Be Aware of Possible % Sign Reduction

The `bind` and `format` commands both use the `%` as a marker for special characters. If you combine the two commands, the `%` will be substituted twice. For example, a `bind` command that should set a value when an entry field is entered might resemble the following.

```
bind .entry <Enter> {set defaultVal [format %%6.2f $value]}
```

Note that there are two percent signs. When the cursor enters `.entry`, the script registered with the `bind` command will be evaluated. The first phase of this evaluation is to scan the script for `%` patterns and substitute the appropriate values. The `%%` pair will be replaced with a single `%`. When the script is evaluated, the `format` command resembles the following, which is a valid command.

```
format %6.2f $value
```

If only a single `%` had been used, it would have been discarded during the first evaluation, and the argument for the `format` command would have been simply `6.2f`.

19.4 CODING TIPS AND TECHNIQUES

There are many ways to write a program. Some techniques work better with a particular language than others. The following describe techniques that work well with Tcl.

19.4.1 Use the Interpreter to Parse Input

Tcl is one of the very few languages that provide the script writer with access to a parse engine. This can save you time (and help you write more robust code) in several ways.

Use Procedures Instead of `switch` *Statements to Parse Input*

If you are accustomed to C programming, you are used to constructs such as the following.

```
while {[gets stdin cmdLine]} {
  set cmd [lindex $cmdLine 0]
  switch $cmd {
    "cmd1"  {cmd1Procedure $cmdLine}
    "cmd2"  {cmd2Procedure $cmdLine}
    ...
  }
}
```

Whenever you add a new command using a `switch` statement, you need to add a new pattern to the `switch` command. In Tcl, you can write a loop to parse input such as the following.

```
while {[gets stdin cmdLine]} {
  set cmd [lindex $cmdLine 0]
  set cmdName [format "%sProcedure" $cmd]
  # Confirm that the command exists before trying to eval it.

  if {[info command $cmdName] != ""} {
    eval $cmdName $cmdLine
  }
}
```

When the requirements change and you need to add a new command, you simply add a new procedure without changing the command parsing code.

Use the `info complete` Command to Match Quotes, Braces, and Brackets

The `info complete` command will return a 1 if a command is complete, and a 0 if there are opening quotes, braces, or brackets without the appropriate closing quotes, braces, or brackets. The `info complete` is designed to determine whether a Tcl command has unmatched quotes, braces, or brackets, but it can be used to test any line of text that may include nested braces, and so on.

Use a Single Array for Global Variables

Rather than use simple variables for global variables, group all shared variables a set of procedures will need in a single array variable. This technique reduces the possibility of variable name collisions when you merge sets of code and makes it easy to add new variables when necessary without requiring changes to the `global` commands in all procedures that will use the new value.

Bad Style

```
# Don't use this technique
proc initializeParameters {
  global height width linecolor

  set height 100
  set width 200
  set linecolor blue
}
```

Good Style

```
# Use this technique
proc initializeParameters {

  global globalParams
  set globalParams(height) 100
  set globalParams(width) 200
  set globalParams(linecolor) blue
}
```

Better Style

```
# Use this technique
proc initializeParameters {

  global globalParams
  array set globalParams {
      height    100
      width     200
      linecolor blue
}
```

Using a single associative array lets you save the program's state with a single command, as in the following.

```
puts $saveFile "array set myStateArray \
    [list [array get myStateArray ]]"
```

This also lets you reload the settings with a simple `source` command. Saving the state in a data-driven loop allows you to add new indices to the state variable without needing to modify the save and restore state functions.

Declare Globals in a Single Location

If you use more than a few global variables, it can become difficult to keep all procedures that use them in sync as new variables are added. The names of the global variables can be kept in a single global list, and that list can be evaluated to set the global variables.

```
set globalList {errorInfo errorCode argv argc \
    stateArray1 stateArray2}

proc someProcedure {args} {
  global globalList ; global {*}$globalList
  # Perform processing.
}
```

Generate a GUI from Data, Not from Code

Some parts of a GUI need to be generated from code, but others can be generated on the fly from lists. For example, a menu in which some items are optional can be generated with a set of commands such as the following.

```
set cmdButton [menubutton .cmdButton \
    -text "Select cmds" \
    -menu .cmdButton.mnu]
set cmdMenu [menu $cmdButton.mnu]

if {[supportedFeature One]} {
  $cmdMenu add command -label {Selection One} \
      -command ProcOne
}
if {[supportedFeature Two]} {
```

```
  $cmdMenu add command -label {Selection Two} \
      -command ProcTwo
}
```

This menu could be generated with a loop and a list as shown in the next example.

```
set cmdMenuList [list \
    {Selection One} {procOne} \
    {Selection Two} {procTwo}]

set cmdButton [menubutton .cmdButton -text "Select cmds" \
    -menu .cmdButton.mnu]

set cmdMenu [menu $cmdButton.mnu]

foreach {label cmd} $cmdMenuList {
  if {[selected feature $cmd]} {
    $cmdMenu add command -label $label -command $cmd
  }
}
```

The list-driven code is simpler to maintain, because the list of labels and commands is more tightly localized and the test need not be repeated.

If the list can be generated at runtime from other data, it can reduce the amount of code that needs to be modified when the program needs to be changed as shown in the next code snippet that uses the contents of the `liquidConversion` to populate the menu.

Example 9

Script Example

```
array set liquidConversion {
  liter             3.8
  quart             4
  {US gallon}       1
  {imperial gallon} 0.833
}

set unitButton [menubutton .unitButton -text "Units" \
    -menu .unitButton.mnu]

set unitMenu [menu $unitButton.mnu]

foreach nm [array names liquidConversion] {
  $unitMenu add command -label $nm \
      -command "set globalConversion $liquidConversion($nm)"
}
```

The menu contents are defined by the contents of the `liquidConversion` array. When new conversion factors are added to the array, the GUI automatically displays the new selections.

19.4.2 Handling Platform-Specific Code

Tcl has excellent multiple platform support. In many circumstances you can use built-in Tcl commands that are identical across all platforms. However, sometimes there are spots where you need to write platform-specific code. For instance, in TclTutor, you can click on a word to get on-line help. Under UNIX, this is done using `exec` to run `TkMan` as a subprocess. Under Windows, the `exec` command invokes `winhlp32.exe`.

The `tcl_platform(platform)` array variable (discussed in Chapter 5) contains the name of the platform where your code is being evaluated. Once you have determined the platform, there are several options for evaluating platform-specific code.

- Place platform-specific scripts in a variable, and evaluate the content of the variable in the mainline code. This works well when the platform-specific script is short.

```
switch $tcl_platform(platform) {
  "unix"  {
      set helpString "exec TkMan $topic"
  }
  "win"  {
       set helpString "exec winhlp32 -k$topic $helpFile"
  }
}
...
if {[userRequestsHelp]} {eval $helpString}
```

- Place platform-specific code within `tcl_platform(platform)` test. If the platform-specific code is more complex, you may prefer to put a script within the test, as follows.

```
if {[userRequestsHelp]} {
  switch $tcl_platform(platform) {
    "unix"  {
        exec TkMan $topic
     }
    "win"   {
        exec winhlp32 -k$topic $helpFile
     }
  }
}
```

- If there are large amounts of platform-specific code, those scripts can be placed in separate files and loaded at runtime, as follows.

```
# A platform-specific procedure named HelpProc is
# defined in each of these script files.
switch $tcl_platform(platform) {
  "unix" {
      source "unix_Procedures.tcl"
```

```
    }
  "win"  {
       source "win_Procedures.tcl"
    }
  "mac"  {
       source "mac_Procedures.tcl"
    }
}
if {[userRequestsHelp]} {HelpProc}
```

19.5 OPTIMIZATION

The first question after "does it work?" is usually "can you make it run faster" The largest speed improvements come from either using a different algorithm or converting a section of Tcl script to compiled code. However, there are some techniques that will improve simple script performance.

The first step at optimizing code is to understand which section of code needs to be optimized. In many cases the code that most obviously needs to be changed is not what's slowing an application down. Two tools for analyzing speed are the `profiler` package and the `time` command.

When updating code from one revision of Tcl to another it can be worthwhile to analyze subsets of the code. There has been a consistent effort at improving Tcl performance since Tcl 8.0. Some of these improvements have resulted from improving the behavior of individual commands and some have been implemented with new commands.

The `profiler` package is part of Tcllib. It's available at *http://sourceforge.net/projects/tcllib/* or it can be installed on your system with `teacup` if you are using an ActiveState distribution. Profiler reports the number of times a procedure is invoked, the length of time a set of code spends in each procedure, how long it took to compile the procedure and a bit more.

The profiler package supports being suspended and resumed to limit the report to a specific section of code and generates both human readable reports and machine readable lists.

To use the profiler your code must:

1. Load the profiler package with `package require profiler`
2. Initialize the profiler with `profiler::init`
3. Access the results with `profiler::print` or `profiler::dump`

The next example shows the profiler being used to compare two procedures that generate the same result when `num` is less than 10.

Example 10

Script Example

```
package require profiler

profiler::init
```

```
proc p1 {num} {
  append num $num
  return [expr $num+1]
}

proc p2 {num} {
  return [expr {$num*10+$num+1}]
}

for {set i 0} {$i < 100} {incr i} {
  p1 1
  p2 1
}

puts [profiler::print]
```

Script Output

```
Profiling information for ::p1
============================================================
            Total calls:  100
    Caller distribution:
  GLOBAL:  100
             Compile time:  1176
            Total runtime:  12367
          Average runtime:  123
            Runtime StDev:  110
           Runtime cov(%):  89.4
  Total descendant time:  0
Average descendant time:  0
Descendants:
  none

Profiling information for ::p2
============================================================
            Total calls:  100
    Caller distribution:
  GLOBAL:  100
             Compile time:  135
            Total runtime:  9052
          Average runtime:  90
            Runtime StDev:  19
           Runtime cov(%):  21.1
  Total descendant time:  0
Average descendant time:  0
Descendants:
  none
```

```
Profiling information for ::tcl::clock::add
============================================================
            Total calls:  0

Profiling information for ::tcl::clock::format
============================================================
            Total calls:  0

Profiling information for ::tcl::clock::scan
============================================================
            Total calls:  0
```

The `time` command is part of Tcl. It returns the length of time that it takes to evaluate a single command. The command might be a procedure call.

Syntax: `time` *`script ?count?`*

Returns the number of microseconds it takes to evaluate a script.

`script` The script to evaluate.

`count` An optional count of times to run the script and return the average.

The next example demonstrates using `time`. Using the `time` command doesn't require a list to collect average times.

Example 11

Script Example

```
proc p1 {num} {
  append num $num
  return [expr $num+1]
}

proc p2 {num} {
  return [expr {$num*10+$num+1}]
}

puts [time {p1 1} 1000]
puts [time {p2 1} 1000]
```

Script Output

```
20.557 microseconds per iteration
4.486 microseconds per iteration
```

19.6 TECHNIQUES FOR IMPROVING PERFORMANCE

- Use `option` command

  Tcl widgets can be customized when they are created with `-option value`. If you have a consistent set of options you are always using (perhaps a non-default font), it's faster to use the `option` command (discussed in Chapter 14).

- Avoid shimmering

  A Tcl variable's value is maintained internally in a `Tcl_Obj` as described in Chapter 15. The `Tcl_Obj` maintains two copies of the data, the string representation and the native representation. When the value is accessed as a string the native representation is used to create a string representation and the native representation is cleared. When the value is accessed for a native-mode operation, the native representation is created from the string and the string representation is cleared.

  The value representation is converted in a "lazy" fashion. The conversion is only done when a command needs the representation that is currently cleared.

  Lists, numbers and dicts have a non-string native representation. If you can restrict accessing the values to a single type of operation, Tcl will not perform extra conversions.

Code that shimmers

```
proc p1 {num} {
  # num is converted to a string.
  append num $num
  # num is converted to a number.
  return [expr $num+1]
}

proc shimmer {val} {
  # val is treated as a string to append the ".0",
  # then converted to a number to divide it by 2.
  return [expr $val.0 / 2]
}
```

Code that does not shimmer

```
proc p2 {num} {
  return [expr {$num*10+$num+1}]
}

proc noShimmer {val} {
  return [expr {double($val) / 2}]
}
```

- Use most specific `string` commands for tests

  The `string equal` command is faster than `string match` which is faster than `regexp`. The pattern recognition code in `string match` and `regexp` is slower than the simple equality comparisons in `string equal`. The `eq` and `ne` operators for the `if` command are as fast as `string equal`.

The `string map` command is much faster than `regsub` for the same reason.

When the code requires patterns, you'll need to use `regexp` and `regsub`, but code that can use the simpler commands will run faster.

- Use *in place* commands instead of modify and assign

Tcl 8.4 introduced the `lset` command to modify a list in place. Earlier versions of Tcl used `set` and `lreplace` to create a new copy of a modified list.

Syntax: `lset` *?indexList? value*

Modify a list by replacing the entire contents or a single list element with a new value.

indexList A list describing the element to replace. If this is missing or an empty list, replace the entire list. If indexList is a list or a set of numbers the values represent indices with the first value being the index into the list and subsequent numbers being indices of sublist elements.

value The new value to put in the list.

In a trivial case, (shown in the example below) using `lset` takes about 2/3 the time of the equivalent `set` and `lreplace` commands.

For complex cases of replacing elements of internal lists, the ability to step into a list of lists and replace an individual element is much simpler than code to extract an element and replace it.

Note that while `lset` without an *indexList* and `set` each completely replace a list, they differ in that `lset` requires that the list already exist while `set` will create a new variable.

Example 12

Script Example

```
set lst {a b c}
puts "Start list $lst"
lset lst  B
puts "After: lset lst  B"
puts "$lst"

set lst {a b c}
lset lst 1 B
puts "After: lset lst 1 B"
puts "$lst"

set lst {a b c}
set lst [lreplace $lst 1 1 B]
puts "After: set lst \[lreplace $lst 1 1 B\]"
puts "$lst"

set lst {a {b c} {d {e f} } }
puts "\nStart list $lst"
lset lst 1 B
```

```
puts "After: lset lst 1 B"
puts $lst

set lst {a {b c} {d {e f} } }
lset lst 1 0 B
puts "After: lset lst 1 0 B"
puts $lst

set lst {a {b c} {d {e f} } }
lset lst {1 0} B
puts "After: lset lst {1 0} B"
puts $lst

set lst {a {b c} {d {e f} } }
lset lst {2} D
puts "After: lset lst {2} D"
puts $lst

set lst {a {b c} {d {e f} } }
lset lst {2 0} D
puts "After: lset lst {2 0} D"
puts $lst

set lst {a {b c} {d {e f} } }
lset lst {2 1 0} E
puts "After: lset lst {2 1 0} E"
puts $lst
```

Script Output

```
Start list a b c
After: lset lst  B
B
After: lset lst 1 B
a B c
After: set lst [lreplace a B c 1 1 B]
a B c

Start list a {b c} {d {e f} }
After: lset lst 1 B
a B {d {e f} }
After: lset lst 1 0 B
a {B c} {d {e f} }
After: lset lst {1 0} B
a {B c} {d {e f} }
After: lset lst {2} D
a {b c} D
```

```
After: lset lst {2 0} D
a {b c} {D {e f}}
After: lset lst {2 1 0} E
a {b c} {d {E f}}
```

- Use `lassign` to extract individual variables from a list

  It's quite common for a script to receive data as a list of values and then need to deal with them individually. Traditionally this was done with a sequence of commands like this:

  ```
  set var0 [lindex $lst 0]
  set var1 [lindex $lst 1]
  ```

  When the `foreach` command was extended to support multiple loop variables it became possible to replace code that had multiple calls to `lindex` with a single `foreach` loop with a `break` command as the body.

  This was faster than multiple assignments in early releases of Tcl 8, and then became slower when the byte-code compiler was extended to compiling the contents of a loop. The compilation makes a loop run faster but when a loop is only evaluated once, the time for the optimization exceeds the time saved. Using `foreach` in the example below is actually slower than the three `lindex` calls.

  The `lassign` command was added in Tcl 8.5 to optimize assigning elements of a list to a set of variables. It's smaller and faster than the repeated set of `set` and `lindex` commands and almost twice as fast as a `foreach` command.

  The next example shows three ways of extracting the values from a list. They each perform the same function.

Example 13

Script Example

```
set lst {liter 3.8 gallon}

set startUnit [lindex $lst 0]
set factor [lindex $lst 1]
set endUnit [lindex $lst 2]

foreach {startUnit factor endUnit} $lst {break;}

lassign $lst startUnit factor endUnit

puts "Multiply by $factor to \
  convert from $startUnit to $endUnit"
```

Script Output

```
Multiply by 3.8 to convert from liter to gallon
```

- Use lists instead of arrays with numeric indices

  If you are familiar with C or FORTRAN type arrays you might be tempted to organize a set of values as an array with a numeric index and then use a `for` loop to access the elements.

  If your script will always access the data in a linear manner, use a list instead.

  The next example shows two procedures that return the sum of a set of numbers. The `arraySum` procedure accepts an array in which each value to add is assigned to an array with a numeric index. The second procedure accepts a set of values as a list.

Example 14

Script Example

```
proc arraySum {arrName} {
  upvar $arrName arr;
  set sum 0
  for {set i 0} {$i < 1000} {incr i} {
    set sum [expr {$sum + $arr($i)}]
  }
  return $sum
}
proc listSum {lst} {
  set sum 0
  foreach val $lst {
    set sum [expr {$sum + $val}]
  }
  return $sum
}

for {set i 0} {$i < 1000} {incr i} {
  set valueArray($i) $i
  lappend valueList $i
}

puts "Time to use array"
puts [time {arraySum valueArray} 10]

puts "Time to use list"
puts [time {listSum $valueList} 10]
```

Script Output

```
Time to use array
2008.4 microseconds per iteration
Time to use list
1522.6 microseconds per iteration
```

- Place repeated code in a procedure or loop

  A script inside a loop or within a procedure is compiled to a byte-code the first time it's encountered. The compiled code is significantly faster than uncompiled code and the time for compiling is short. In most cases it's a worthwhile trade-off.

19.7 BOTTOM LINE

This chapter has discussed several tricks, techniques, and tips for writing efficient and maintainable Tcl scripts. These include the following.

- You can use other `wish` interpreters or new windows to debug a `wish` application.
- Some applications should be written as separate programs, invoked with `exec`, rather than as extensions or Tcl scripts.
- The Tcl interpreter is not the UNIX command shell or `COMMAND.COM`.
- Be aware of possible leading zeros in numbers.
- Most Tcl commands do not modify the content of their arguments. `append`, `lappend`, `set`, `lset`, and `incr` are exceptions.
- Use the Tcl interpreter to parse data whenever possible, rather than writing new code to parse data.
- Use a single global array for shared data instead of many variables. If you must use many variables, group them into a global list and evaluate that list to declare the globals in procedures.
- Use data rather than code to control GUI contents whenever practical.
- Use the `profiler` package or `time` command to analyze code before you try to optimize it.

Index

Bold page references point to definitions.

A

B

C

D

E

G

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Z