14

Object-Oriented Programming

This chapter describes Tcl features that support object oriented programming. It does *not* go into detail about what constitutes object oriented programming, what its benefits are, or how your classes should be designed. The answers generally depend on who you ask and there have been enough words written on the topic.

Nevertheless, as we go along we will briefly describe some basic concepts for the benefit of the reader who really is completely unexposed to OO programming.

A bit of history
One of the knocks against Tcl in its early days was that it did not support object oriented programming. This criticism was both incorrect and unfair because Tcl did in fact support not one, but several, OO implementations. This misconception was at least partly due to the fact that these OO systems did not come as part of the core language, but rather were implemented as extensions or packages. In fact, writing an OO system in Tcl became a rite of passage for many Tcl programmers.

Some of these systems became fairly widely used and remain so today:

- *IncrTcl*'s name was a take-off on C++ and so is its design. It was intended to make programmers used to that language feel at home. It was one of the earliest Tcl-based OO extensions to be widely used.
- *Snit* (Snit's Not Incr Tcl) is a popular OO implementation which is particularly useful towards building Tk widgets.
- *XoTcl* and its successor *nx* are OO implementations designed for research into dynamic OO programming.

The experience gained from these system led to the implementation of a OO system in the Tcl core — TclOO. This became part of the Tcl 8.6 release and is also available as an extension for Tcl 8.5. TclOO can be used as a standalone OO system by itself. However, one of its goals was also to provide the base facilities required for layering other OO systems on top.

The Tcl based OO programming described in this book is based on TclOO.

Most of the example code in this chapter is based on a framework for modeling banks. Our bank has accounts of different types, such as savings and checking, which allow operations like deposits and withdrawals. Some of these are common to all account types while others are unique. We have certain privileged customers who get special treatment and we have to also follow certain directives from Big Brother.

No, Citibank cannot run its operations based on our framework but it suffices for our illustrative purposes.

14.1. Objects and classes

The core of OO programming involves, no surprise, *objects*. An object, often a representation of some real world entity, captures state (data) and behaviour which is the object's response to *messages* sent to it. In most languages, implementation of these messages involves calling *methods* which are just function calls with a context associated with the object. For example, the state contained in an object representing a bank account would include items

such as the current balance and account number. The object would respond to messages to deposit or withdraw funds from the account.

A *class* is (loosely speaking) a template that defines the data items and methods (collectively called members) encapsulated by objects of a specific type. More often than not, creating a object of the class, often known as instantiating an object, is one of the duties of the class.

Not every OO system has, or needs, the notion of a class. *Prototype* based systems instead create objects by "cloning" an existing object — the prototype — and defining or modifying members.

TclOO provides facilities to support both the classy and the classless [1] models.

TclOO actually exposes sufficient internal functionality to allow you to develop your own vision of what object-oriented programming models should look like. The ITcl 4.0 implementation for instance is layered on the services provided by the TclOO. For most folks though who are not into experimentation with OO as a way of life, the base functionality provided by TclOO suffices for most purposes.

14.2. Classes

We will start off with first describing classes as most OO programming code you will encounter in Tcl is based on the use of classes.

14.2.1. Creating a class

Classes are created in TclOO using the `oo::class create` command. Let us create a class, `Account`, that models a banking account.

```
% oo::class create Account
→ ::Account
```

This creates a new class `Account` that can be used to create objects representing bank accounts.

The class `Account` is actually just another Tcl command and could have been created in any namespace we choose, not necessarily the global one. For example, either

```
oo::class create bank::Account
namespace eval bank {oo::class create Account}
```

would create a new class `Account` in the bank namespace, entirely unrelated to our `Account` class in the global namespace.

There is however no class definition associated with our `Account` class and therefore there is as yet no state or behaviour defined for objects of the class. That is done through one or more *class definition scripts*. We will look at the contents of these definition scripts throughout this chapter, but for now, simply note that class definitions can be built up in incremental fashion. A definition script can be passed as an additional argument to the `oo::class create` command, in the form

```
oo::class create CLASSNAME DEFINITIONSCRIPT
```

and also through `oo::define` commands which take the form

```
oo::define CLASSNAME DEFINITIONSCRIPT
```

[1] No value judgement intended.

Thus the statements

```
oo::class create CLASSNAME DEFINITIONSCRIPT
```

and

```
oo::class create CLASSNAME
oo::define CLASSNAME DEFINITIONSCRIPT
```

are equivalent. As is generally the case, Tcl has the flexibility to fit your programming style.

One advantage of the `oo::define` command is that it may be used multiple times for the same class to define it in incremental fashion. We will see this we work through this chapter. Moreover, the command also has a second syntactic form:

```
oo::define CLASSNAME SUBCOMMAND ARG ?ARG ...?
```

Here *SUBCOMMAND* is one of the commands that may be used in a class definition script. The following listing shows the two equivalent forms of `oo::define`.

```
oo::define Account {
    method foo {} {}
    method bar {} {}
}

oo::define Account method foo {} {}
oo::define Account method bar {} {}
```

This second form of `oo::define` can be useful when one of the arguments to the subcommand is to be taken from a variable, a situation that often arises in metaprogramming with objects.

14.2.2. Destroying classes

A class, as we shall see later, is also an object and like all objects can be destroyed by invoking its `destroy` method.

```
% Account destroy
```

Classes can also be destroyed by renaming the corresponding command to the empty string:

```
rename Account ""
```

The above will erase

- the definition of the `Account` class,
- any classes that inherit from (see Section 14.4), or mix-in (see Section 14.6), the `Account` class
- all objects belonging to all destroyed classes.

Not commonly used in operational code, this ability to destroy classes is sometimes useful during interactive development and debugging to reset to a known clean state.

We will be using the `Account` class throughout the chapter so let us recreate it before we move on.

```
% oo::class create Account
→ ::Account
```

14.2.3. Defining data members

In our simple example, the state for an account object includes an account number that uniquely identifies it and the current balance in the account.

We will need data members to hold this information and we define them through the `variable` command within a class definition script.

```
% oo::define Account {
    variable AccountNumber Balance ❶
}
```

❶ The author uses mixed case for data members to avoid conflicts with names of arguments and local variables.

This defines the data members for the class as per-object variables. `AccountNumber` and `Balance` are then visible within all methods of the class and can be referenced there without any qualifiers or declarations.

There can be multiple `variable` statements, each defining one or more data members. These append new variable definitions to those existing. However, if the `-set` option is specified for the command, the current set of variable definitions is **replaced** by the ones defined in the current command. You can also remove all existing variable definitions without creating new ones by specifying the `-clear` option.

Data members do not have to be declared using `variable` in a class definition script. They can also be declared within a method using the `my variable` command which we show later.

Note the difference between the `variable` statement in the context of a class definition and the `variable` command used to define namespace variables. They both have very similar function but the former only defines data member names, not their values whereas the latter defines names of variables within a namespace as well as their initial values.

14.2.4. Defining methods

Having defined the data members, let us move on to defining the methods that comprise the behaviour of an `Account` object. An `Account` object responds to requests to get the current balance and to requests for depositing and withdrawing funds.

Methods are defined through the `method` command which, like `variable`, is executed as part of a class definition script.

```
oo::define Account {
    method UpdateBalance {change} {
        set Balance [+ $Balance $change]
        return $Balance
    }
    method balance {} { return $Balance }
    method withdraw {amount} {
        return [my UpdateBalance -$amount]
    }
    method deposit {amount} {
        return [my UpdateBalance $amount]
    }
}
```

As you see, a method is defined in exactly the same manner as `proc` defines a Tcl procedure. Just like in that case a method takes an arbitrary number of arguments including a variable number of trailing arguments collected as the `args` variable. The difference from a procedure lies in how it is invoked and the context in which the method executes.

14.2.4.1. Method visibility

Another point about method definitions concerns method visibility. An *exported* method is a method that can be invoked from outside the object's context. A *private* method on the other hand, can only be invoked from within another method in the object context. Methods that begin with a lower case letter are exported by default. Thus in our example, `deposit` and `withdraw` are exported methods while `UpdateBalance` is not. Method visibility can be changed by using the `export` and `unexport` commands inside a `oo::define` class definition script. Thus

```
oo::define Account export UpdateBalance
```

would result in the private `UpdateBalance` method being exported. Conversely,

```
oo::define Account unexport UpdateBalance
```

will remove it from the exported list.

The `export` and `unexport` commands are not limited to the methods defined in that particular class. They can also be applied to methods inherited from superclasses or mix-ins. This is useful, for example, when a derived class wants to limit functionality supported by a base class.

14.2.4.2. Deleting methods

Method definitions can be deleted at any time with the `deletemethod` command inside a class definition script.

The following code snippet will crash Tcl versions prior to 8.6.2 due to a bug in the Tcl implementation.

```
% oo::class create C {method print args {puts $args}}
→ ::C
% C create c
→ ::c
% c print some nonsense
→ some nonsense
% oo::define C {deletemethod print}
% c print more of the same
Ø unknown method "print": must be destroy
```

Deletion of methods from classes is rarely used. However, deletion of methods from objects is sometimes useful in object specialization (see Section 14.5).

14.2.4.3. Renaming methods

The `renamemethod` command is used within a class definition script to rename an existing method.

```
oo::class create C {method print args {puts $args}}
C create c
oo::define C renamemethod print output
```

Once renamed, the method must be invoked by its new name, even for existing objects.

```
% c print foo
Ø unknown method "print": must be destroy or output
% c output foo
→ foo
```

We have not discussed class inheritance as yet, but we will just note here that renaming a method in a class will not rename methods of the same name in any ancestors or descendents for that class.

14.2.5. Constructors and destructors

There is one final thing we need to do before we can start banking operations and that is to provide some means to initialize an `Account` object when it is created and perform any required clean up when it is destroyed.

These tasks are performed through the special methods named `constructor` and `destructor`. These differ from normal methods in only two respects:

- They are not explicitly invoked by name. Rather, the `constructor` method is automatically run when an object is created. Conversely, the `destructor` method is run when the object is destroyed.
- The `destructor` method definition differs from other methods in that it only has a single parameter — the script to be run. It does not have a parameter corresponding to arguments.

For our simple example, these methods are straightforward.

```
oo::define Account {
    constructor {account_no} {
        puts "Reading account data for $account_no from database"
        set AccountNumber $account_no
        set Balance 1000000
    }
    destructor {  ❶
        puts "[self] saving account data to database"
    }
}
```

❶ Note the syntax of the `destructor` definition

Both constructors and destructors are optional. They do not have to be defined in which case TclOO will simply generate empty methods for them.

14.2.6. The unknown method

Every object has a method named unknown which is run when no method of that name is found in the method chain (see Section 14.9) for that object.

The definition of the unknown method takes the form

```
oo::define CLASSNAME {
    method unknown {target_method args} {.. implementation ..}
}
```

The unknown method is passed the name of the invoked method as its first argument followed by the arguments from the invocation call.

The default implementation of this method, which is inherited by all objects from the root `oo::object` object, raises an error. Classes and objects can override the default implementation method to take some other action instead.

An example of its use is seen in the COM client implementation in TWAPI[2]. The properties and methods exported from a COM component are not always known beforehand and in fact can be dynamically modified. The TclOO based wrapper for COM objects defines an unknown method that looks up method names supported by a COM component the first time a method is invoked. If found, the lookup returns an index into a function table that can then be invoked through the `ComCall` method. The implementation of unknown looks like

[2] http://twapi.sf.net

```
oo::define COMWrapper {
    method unknown {method_name args} {
        set method_index [COMlookup $method_name]
        if {$method_index < 0} {
            error "Method $method_name not found."
        }
        return [my ComCall $method_index {*}$args]
    }
}
```

(This is a greatly simplified, not entirely accurate or correct, description for illustrative purposes.)

14.2.7. Modifying an existing class

As we have seen in previous sections, you can incrementally modify a class using `oo::define`. Practically nothing about a class is sacred — you can add or delete methods, data members, change superclasses or mix-ins, and so on.

The question then arises as to what happens to objects that have already been created if a class is modified. The answer is that existing objects automatically "see" the modified class definition so for example any new methods can be invoked on them. Or if you add a mix-in or a superclass, the method lookup sequence for the object will be appropriately modified.

However, some care should be taken when modifying a class since existing objects may not hold all state expected by the new class. For example, the new constructors are (obviously) not run for the existing objects and thus some data members may be uninitialized. The modified class code has to account for such cases.

14.3. Working with objects

Having defined our model, we can now begin operation of our bank to illustrate how objects are used.

14.3.1. Creating an object

An object of a class is created by invoking one of two built-in methods on the class itself. The `create` method creates an object with a specific name. The new method generates a name for the created object.

```
% set acct [Account new 3-14159265]
→ Reading account data for 3-14159265 from database
  ::oo::Obj217
% Account create smith_account 2-71828182
→ Reading account data for 2-71828182 from database
  ::smith_account
```

Creating an object also initializes the object by invoking its constructor.

The created objects are Tcl commands and as such can be created in any namespace.

```
% namespace eval my_ns {Account create my_account 1-11111111}
→ Reading account data for 1-11111111 from database
  ::my_ns::my_account
% Account create my_ns::another_account 2-22222222
→ Reading account data for 2-22222222 from database
  ::my_ns::another_account
```

Note that `my_account` and `my_ns::my_account` are two distinct objects.

14.3.2. Destroying objects

Objects in Tcl are **not** garbage collected as in some other languages and have to be explicitly destroyed by calling their built-in `destroy` method. This also runs the object's `destructor` method.

```
% my_ns::my_account destroy
→ ::my_ns::my_account saving account data to database
```

Any operation on a destroyed object will naturally result in an error.

```
% my_ns::my_account balance
Ø invalid command name "my_ns::my_account"
```

Objects are also destroyed when its class or containing namespace is destroyed. Thus

```
% namespace delete my_ns
→ ::my_ns::another_account saving account data to database
% my_ns::another_account balance
Ø invalid command name "my_ns::another_account"
```

generates an error as expected.

14.3.3. Invoking methods

An object in Tcl behaves like an ensemble command of the form

```
OBJECT METHODNAME args...
```

This is the form used to invoke a method on the object from code "outside" the object.

```
% $acct balance
→ 1000000
% $acct deposit 1000
→ 1001000
```

As discussed in Section 14.2.4, when calling a method from another method **in the same object context**, the alias `my` is used to refer to the current object. So the `deposit` method we saw earlier calls the `UpdateBalance` method as:

```
my UpdateBalance $amount
```

14.3.3.1. Method contexts

A method runs in the context of its object's namespace (see Section 14.11.2.1). This means the object data members such as `Balance`, defined through `variable`, are in the scope of the method and can directly be referenced without any qualifiers as seen in the method definition above.

The method context also makes available several commands — such as `self`, `next` and `my` — which can only be called from within a method. These cannot be invoked from outside the method body even by prefixing with the object's command name.

We will see various uses of these commands as we proceed but for now note the use of `my` to refer to a method of the object in whose context the current method is running.

14.3.4. Accessing data members

Data members are not directly accessible from outside the object. Methods, such as `balance` in our example, have to be defined to allow callers to read and modify their values. Many OO-purists, and even non-purists like the author, believe this to be desirable.

However, Tcl being Tcl, it is always possible to add a variable access capability using the fact that each object has a private namespace that can be retrieved through introspection with the `info object namespace` command. Thus,

```
% set [info object namespace $acct]::Balance 5000
→ 5000
% $acct balance
→ 5000
```

This practice breaks encapsulation and is **not** recommended. However, some OO systems layered on top of TclOO do offer this feature in a structured manner that does not explicitly expose internal object namespaces. These are however not discussed here.

A good alternative is to automatically define *accessor* methods for public variables without the programmer having to explicitly do so. One such implementation is described in the Lifecycle Object Generators paper.

14.4. Inheritance

The defining characteristic of OO systems is support for *inheritance*. Inheritance refers to the ability of a *derived* class (also refered to as a *subclass*) to specialize a class — called its *base* class or *superclass* — by extending or modifying its behaviour.

Thus in our banking example, we may define separate classes representing savings accounts and checking accounts, each inheriting from the base account and therefore having a balance and methods for deposits and withdrawal. Each may have additional functionality, for example check writing facilities for the checking account and interest payments for the savings account.

The intention behind inheritance is to model *is-a* relationships. Thus a checking account *is a* bank account and can be used at any place in the banking model where the behaviour associated with a bank account is expected. This *is-a* relation is key when deciding whether to use inheritance or some other facility such as mix-ins.

Let us define our `SavingsAccount` and `CheckingAccount`. Instead of using `oo::define` as before, we will provide the full class definition as part of the `oo::class` command itself.

```
oo::class create SavingsAccount {
    superclass Account
    variable MaxPerMonthWithdrawals WithdrawalsThisMonth
    constructor {account_no {max_withdrawals_per_month 3}} {
        next $account_no
        set MaxPerMonthWithdrawals $max_withdrawals_per_month
    }
    method monthly_update {} {
        my variable Balance
        my deposit [my MonthlyInterest]
        set WithdrawalsThisMonth 0
    }
    method withdraw {amount} {
        if {[incr WithdrawalsThisMonth] > $MaxPerMonthWithdrawals} {
            error "You are only allowed $MaxPerMonthWithdrawals withdrawals a month"
        }
        next $amount
    }
    method MonthlyInterest {} {
        my variable Balance
        return [format %.2f [* $Balance 0.005]]
    }
}
oo::class create CheckingAccount {
    superclass Account
    method cash_check {payee amount} {
```

```
        my withdraw $amount
        puts "Writing a check to $payee for $amount"
    }
}
→ ::CheckingAccount
```

The `superclass` command in the class definition establishes that `SavingsAccount` and `CheckingAccount` inherit from `Account`. This statement by itself means they will behave exactly like the `Account` class, with the same methods and variables defined. Further declarations will extend or modify the class behaviour.

14.4.1. Methods in derived classes

Methods available in the base class are available in derived classes as well. In addition, new methods can be defined, such as `cash_check` and `monthly_update` in our example, that are only present on objects of the derived class.

If the derived class defines a method of the same name as a method in the base class, it overrides the latter and will be called when the method is invoked on an object of the derived class. Thus the `withdraw` method of the `SavingsAccount` class overrides the `withdraw` method of the base `Account` class. However, we are just modifying the original method's functionality with an additional condition, not replacing it. Therefore, after making the check we want to just pass on the request to the base class method and not duplicate its code. This is done with the command `next` which invokes the superclass method with the same name as the current method. This *method chaining* is actually only an example of a broader mechanism we will explore in detail in Section 14.9.

Constructors and destructors are also chained. If a derived class does not define a constructor, as is true for the `CheckingAccount` class, the base class constructor is invoked when the object is created. If the derived class does define a constructor, that is invoked instead and it is up to that constructor to call the base class constructor using `next` as appropriate. Destructors behave in a similar fashion.

Note that `next` may be called at any point in the method, not necessarily in the beginning or the end.

14.4.2. Data members in derived classes

Derived classes can define new data members using either `variable` in the class definition or `my variable` within a method as in `withdraw`.

Because data members are **always** defined in the namespace of the object, you have to careful about conflicts between variables of the same name being defined in a base class and a derived class if they are intended to represent different values.

Data members defined in a parent (or ancestor) class are also accessible within a derived class **but** they have to be brought within the scope of the method through the `variable` declaration in the derived class definition or the `my variable` statement within a method as is done in the implementation of `MonthlyInterest`. Although we use a direct variable reference there for expository purposes, in the interest of data hiding and encapsulation, direct reference to variables defined in ancestors should be avoided if possible. It would have been better to write the statement as

```
my deposit [format %.2f [* [my balance] $rate]]
```

Let us try out our new accounts.

```
% SavingsAccount create savings S-12345678 2
→ Reading account data for S-12345678 from database
  ::savings
% CheckingAccount create checking C-12345678
→ Reading account data for C-12345678 from database
  ::checking
% savings withdraw 1000
```

```
→ 999000
% savings withdraw 1000
→ 998000
% savings withdraw 1000 ❶
Ø You are only allowed 2 withdrawals a month
% savings monthly_update
→ 0
% checking cash_check Payee 500 ❷
→ Writing a check to Payee for 500
% savings cash_check Payee 500 ❸
Ø unknown method "cash_check": must be balance, deposit, destroy, monthly_update or withdraw
```

❶ Overridden base class method
❷ Method defined in derived class
❸ Check facility not available for savings

14.4.3. Multiple inheritance

Imagine our bank also provides brokerage services. Accordingly we define the following class:

```
oo::class create BrokerageAccount {
    superclass Account
    method buy {ticker number_of_shares} {
        puts "Buying $number_of_shares shares of $ticker"
    }
    method sell {ticker number_of_shares} {
        puts "Selling $number_of_shares shares of $ticker"
    }
}
→ ::BrokerageAccount
```

The company now decides to make it even more convenient for customers to lose money in the stock market. So we come up with a new type of account, a Cash Management Account (CMA), which combines the features of the checking and brokerage accounts. We can model this in our system using *multiple inheritance*, where the corresponding class inherits from more than one parent class.

```
oo::class create CashManagementAccount {
    superclass CheckingAccount BrokerageAccount
}
```

Be careful when using multiple `superclass` statements as the earlier declarations are overwritten if the `-append` option is not specified. The above example written using multiple `superclass` commands would be written as:

```
oo::class create CashManagementAccount {
    superclass CheckingAccount
    superclass -append BrokerageAccount
}
```

Our CMA account can do it all.

```
% CashManagementAccount create cma CMA-00000001
→ Reading account data for CMA-00000001 from database
  ::cma
% cma cash_check Payee 500
→ Writing a check to Payee for 500
% cma buy GOOG 100
```

```
→ Buying 100 shares of GOOG
```

Use of multiple inheritance is a somewhat controversial topic in OO circles. Be as it may, TclOO offers the facility, and also an alternative using mix-ins (see Section 14.6), and leaves the design choices for programmers to make.

14.5. Specializing objects

The next thing we talk about, object specialization, may be new to readers who are more familiar with class-based OO languages such as C++ where methods associated with objects are exactly those that are defined for the class(es) to which the object belongs.

In TclOO on the other hand, we can further "specialize" an *individual* object by overriding, hiding, and deleting methods defined in the class or even adding new ones. In fact, the potential specialization includes features such as forwarding, filters and mix-ins but we leave them for now as we have not discussed them as yet. As we will see, we can even change an object's class.

Specialization is done through the `oo::objdefine` command which is analogous to the `oo::define` command for classes except that it takes an object as its argument instead of a class. With the obvious exception of the commands `constructor`, `destructor` and `superclass`, all commands, like `method`, `variable`, `export` etc., available within `oo::define` scripts can also be used inside the script passed to `oo::objdefine`. The difference is that they act on a specific object instead of a class.

Like `oo::define`, `oo::objdefine` also has two syntactic forms.

```
oo::objdefine OBJ DEFINITIONSCRIPT
oo::objdefine OBJ SUBCOMMAND ARG ?ARG ...?
```

14.5.1. Object-specific methods

Let us illustrate with our banking example. Imagine our banking system had the requirement that individual accounts can be frozen based on an order from the tax authorities. We need to define a procedure we can call to freeze an account so all transactions on the account will be denied. Correspondingly, we need a way to unfreeze an frozen account. The following code accomplishes this.

```
proc freeze {account_obj} {
    oo::objdefine $account_obj {
        method UpdateBalance {args} {
            error "Account is frozen. Don't mess with the IRS, dude!"
        }
        method unfreeze {} {
            oo::objdefine [self] { deletemethod UpdateBalance unfreeze }
        }
    }
}
```

When the `freeze` procedure is passed an `Account` object, it uses `oo::objdefine` to override the `UpdateBalance` method that was part of the object's class definition with a object specific `UpdateBalance` method that raises an error instead.

It then defines a new method `unfreeze` that can be called on the object at the appropriate time to restore things back to normal. We could have actually defined an `unfreeze` procedure instead of a `unfreeze` method as follows:

```
proc unfreeze {account_obj} {
    oo::objdefine $account_obj deletemethod UpdateBalance
}
```

This would have accomplished the same job in a clearer manner. We chose to implement an `unfreeze` method instead to illustrate that we can actually change an object's definition even from **within** the object.

There are a couple of points that need to be elaborated:

- The `self` command is only usable within a method and returns the name of the current object when called without parameters. Thus the `oo::objdefine` command is instructed to modify the object itself. We will see other uses of the `self` command later.
- Although not required in our example, it should be noted that variables defined in the class are not automatically visible in object-specific methods. They need to be brought into scope with the `my variable` command.
- When called from within a `oo::objdefine` script, the `deletemethod` erases the specified object-specific methods. It does **not** affect methods defined in the class so the original `UpdateBalance` will still be in place and will no longer be overridden.

Let us see how all this works. At present Mr. Smith can withdraw money freely from his account.

```
% smith_account withdraw 100
→ 999900
```

So far so good. Now we get a court order to freeze Mr. Smith's account.

```
% freeze smith_account
```

Mr. Smith tries to withdraw money and run away to the Bahamas.

```
% smith_account withdraw [smith_account balance]
Ø Account is frozen. Don't mess with the IRS, dude!
```

Have we affected other customers?

```
% $acct withdraw 100
→ 4900
```

No, only the `smith_account` object was impacted.

Cornered Mr. Smith pays up to unfreeze the account.

```
% smith_account unfreeze
% smith_account withdraw 100
→ 999800
```

Notice that the class definition of `UpdateBalance` was not lost in the process of adding and deleting the object-specific method.

This ability to define object-specific methods can be very useful. Imagine writing a computer game where the characters are modeled as objects. Several characteristics of the objects, such as the physics determining movement, are common and can be encapsulated with a class definition. The special "powers" of each character cannot be part of this class and defining a separate class for each character is tedious overkill. The special power of a character can instead be added to the character's object as a object-specific method. Even modeling scenarios like temporary loss of a power without a whole lot of conditionals and bookkeeping becomes very simple using the object specialization mechanisms.

14.5.2. Changing an object's class

Being a true dynamic OO language, TclOO can even change the class of an object through `oo::objdefine`. For example, one might change a savings account to a checking account.

```
% set acct [SavingsAccount new C-12345678]
→ Reading account data for C-12345678 from database
  ::oo::Obj228
% $acct monthly_update
→ 0
```

So far so good. Let us attempt to cash a check.

```
% $acct cash_check Payee 100
Ø unknown method "cash_check": must be balance, deposit, destroy, monthly_update or withdraw
```

Naturally that fails because it is not a checking account. Not a problem, we can fix that by morphing the object to a `CheckingAccount`.

```
% oo::objdefine $acct class CheckingAccount
```

We can now cash checks successfully

```
% $acct cash_check Payee 100
→ Writing a check to Payee for 100
```

but monthly updates no longer work as the account is no longer a `SavingsAccount`.

```
% $acct monthly_update
Ø unknown method "monthly_update": must be balance, cash_check, deposit, destroy or withdraw
% $acct destroy
→ ::oo::Obj228 saving account data to database
```

Needless to say, you have to be careful when "morphing" objects in this fashion since data members may differ between the two classes.

Note the optional form of the oo::objdefine command that we have used in the above code fragment. When the script passed to `oo::define` or `oo::objdefine` contains only one command, it can be directly specified as additional arguments to `oo::define` or `oo::objdefine`.

Lifecycle Object Generators describes an example of when such morphing might be used. Consider a state machine where each state is represented by a class that implements the state's behaviour. When a state change occurs, the state machine object changes its class to the class corresponding to the target state. See the abovementioned reference for implementation details.

14.6. Using mix-ins

Earlier we looked at the use of inheritance to extend a class. We will now look at another mechanism to extend or change the behaviour of classes (and objects) — mix-ins.

The literature on the subject describes mix-ins in several different ways, often depending on language-specific capabilities. From this author's perspective, a mix-in is a way to package a bundle of related functionality such that it can be used to extend one or more classes or objects. In some languages, multiple inheritance is used for this purpose but we will postpone that discussion until after we have seen an example of a mix-in.

Let us go back to our banking model. Imagine we have an Electronic Fund Transfer (EFT) facility that provides for transferring funds to other accounts. We will not worry about how this is done but just assume some global procedures are available for the purpose. This facility is available to all savings accounts but only to selected

checking accounts. There are several ways we could implement this but our preference in this case is for mix-ins over the alternatives for reasons we discuss later.

In TclOO a mix-in is also defined as a class in exactly the same manner as we have seen earlier. In fact, in theory any class can be a mix-in. What sets a mix-in apart is the conceptual model and how the class is **used**. In our example, the EFT facility would be modeled as a class that implements two methods, `transfer_in` and `transfer_out`. Conceptually, the class does not represent an *object*, but rather a *capability* or, as is termed in some literature, a *role*. It adds functionality to a "real" object.

```
oo::class create EFT {
    method transfer_in {from_account amount} {
        puts "Pretending $amount received from $from_account"
        my deposit $amount
    }
    method transfer_out {to_account amount} {
        my withdraw $amount
        puts "Pretending $amount sent to $to_account"
    }
}
→ ::EFT
```

Since we want all checking accounts to have this facility, we will add `EFT` to the `CheckingAccount` class as a mix-in. This is accomplished with the `mixin` command within a class definition script.

```
% oo::define CheckingAccount {mixin EFT}
% checking transfer_out 0-12345678 100
→ Pretending 100 sent to 0-12345678
% checking balance
→ 999400
```

We are now able to do electronic transfers on all checking accounts.

Note that modifying the class definition in **any** manner, in this case adding a mix-in, also impacts **existing** objects of that class. Thus the `checking` object automatically supports the new functionality.

In the case of savings accounts, we only want **select** accounts to have this facility. Assuming our `savings` object represents one of these privileged accounts, we can add the mix-in to just that object through `oo::objdefine`.

```
% oo::objdefine savings {mixin EFT}
% savings transfer_in 0-12345678 100
→ Pretending 100 received from 0-12345678
  1003090.0
% savings balance
→ 1003090.0
```

Notice that the `EFT` class does not really know anything about accounts. It encapsulates features that can be added to **any** class or object that defines the methods `deposit` and `withdraw` required to support the mix-in's functionality. So if we had a `BrokerageAccount` class or object, we could mix it in there as well.

14.6.1. Using multiple mix-ins

A class or object may have multiple classes mixed in. So for example if we had a facility for electronic bill presentment implemented as a mix-in class `BillPay`, we could have added it along with `EFT` as a mix-in in a single statement

```
oo::define CheckingAccount {mixin BillPay EFT}
```

or as multiple statements

```
oo::define CheckingAccount {
    mixin EFT
    mixin -append BillPay
}
```

By default, the `mixin` command overwrites existing mix-in configuration so without the `-append` option when using multiple `mixin` statments, only class `BillPay` would be mixed into `CheckingAccount`.

We could also clear out all mix-ins from the class by specifying the `-clear` option to the `mixin` command.

14.6.2. Mix-ins versus inheritance

Because one of its goal is to provide the required infrastructure for additional OO system to be built on top, TclOO offers a wide variety of capabilities that sometimes overlap in their effect. The question then arises as to how to choose the appropriate feature for a particular design requirement. One of these design choices involves mix-ins and inheritance.

We offer the author's thoughts on the matter. Luckily, these tend to be few and far between so a couple of paragraphs is sufficient for this purpose.

Instead of mixing our `EFT` class into `CheckingAccount`, we could have made it a superclass and used multiple inheritance instead. Or even modified or derived from the `CheckingAccount` class to add transfer methods. Why did we choose to go the mix-in route?

Not directly inheriting or modifying the `CheckingAccount` class was a no-brainer for obvious reasons. The functionality is something that could be used for other account types as well and it does not make sense to duplicate code and add it to every class that needs those features. That leaves the question of multiple inheritance.

There were several considerations:

- Inheritance implies an *is-a* relationship between classes. Saying a checking account is-a "account that has transfer features" sounds somewhat contrived.
- The above stems from the fact that EFT does not really reflect a real object. It is more like a set of features or capabilities that accounts have. In the real world, it would be a checkbox on a account opening form for a checking account. The general thinking is that such classes are better modeled as mix-ins.
- Perhaps most important, when implemented as a mix-in, we can provide the feature sets to individual accounts, for example to specific savings accounts. You cannot use multiple inheritance to specialize individual objects in this manner.

For these reasons, mix-ins seemed a better choice in our design (aside from the fact that we needed **some** example to illustrate mix-ins).

There is one practical aspect of TclOO design that may drive your decision. Methods implemented via mix-ins appear in the method chain **before** methods defined on the object whereas inherited methods appear **after**. This was not relevant to our example because the mix-in only added new methods. It did not override exising ones.

14.7. Method forwarding

A method can be *forwarded* to another command, its *target*, so that when the method is invoked on the object, the target command is invoked instead. Forwarded methods may be defined on the class or on an object.

```
forward METHOD TARGET ?ARG ...?
```

This command may appear in the definition script for either classes or objects. *METHOD* is the name of the method to be defined. *TARGET* is the command to be invoked when that method is invoked. There are no restrictions on what *TARGET* may be. It may be a Tcl procedure or command, another object, a coroutine etc. It is invoked within

the context of the object so that name resolution occurs in that context. So for example, if *TARGET* is my, the method is effectively forwarded to another method defined for the same object.

When *METHOD* is invoked, any optional *ARG* arguments specified in the forward declaration are prepended to the list of arguments provided by the caller before it is passed to the target command.

Method forwarding is used in many patterns in object-oriented programming, for example composition. Earlier we defined the cash management account using multiple inheritance, in effect treating a CashManagementAccount as being a CheckingAccount and a BrokerageAccount. We could instead have thought of it as a consolidated account that **contained** the two account types. The class would then be defined as

```
oo::class create ConsolidatedAccount {
    constructor {acct_no} {
        CheckingAccount create checking_account $acct_no
        BrokerageAccount create brokerage_account $acct_no
    }
}
→ ::ConsolidatedAccount
```

We want the same operations available for this new account type as before. We can do this by forwarding methods to the appropriate contained account. For example, cash withdrawals would happen from the checking account.

```
oo::define ConsolidatedAccount {
    forward buy brokerage_account buy
    forward sell brokerage_account sell
    forward cash_check checking_account cash_check
    forward withdraw checking_account withdraw ❶
}
```

❶ Note we want withdrawals to be from the checking account

When the methods are forwarded, the target commands are resolved within the object's context. Thus brokerage_account and checking_account refer to the account objects we created within the object's context in the constructor.

This now has behaviour very similar to our CashManagementAccount based solution.

```
% ConsolidatedAccount create consolidated CONS-0000001
→ Reading account data for CONS-0000001 from database
  Reading account data for CONS-0000001 from database
  ::consolidated
% consolidated cash_check Payee 500
→ Writing a check to Payee for 500
% consolidated buy GOOG 100
→ Buying 100 shares of GOOG
```

Forwarding may also supply arguments to the targeted command as shown below.

```
oo::objdefine consolidated {
    forward quick_cash my withdraw 100
}
consolidated quick_cash
→ 999400
```

For illustrative purposes, this last definition differs from previous ones in the following respects:

- The forwarded method is only defined for the object, not the class.
- It is forwarded to another method in the same object through the use of the my command.

- It supplies an argument within the forwarding definition.

14.8. Filter methods

Imagine Mr. Smith is suspected of being up to his old tricks again and we need to monitor his accounts and log all activity. How would we do this? We could specialize every method for his accounts via `oo::objdefine` and log the activity before invoking the original method. We would have to do this for every method available to the object — those defined in the object, its class (and superclasses), object mix-ins and class mix-ins. This would be tedious and error prone. Moreover, since Tcl is a dynamic language, we would have to make sure we do that any time new methods were defined for the object or any ancestor and mix-in.

Filter methods offer a easier solution. A filter method is defined in the same manner as any method in the class or object. It is marked as a filter method using the `filter` command. Any method invocation on the object will then result in the filter method being invoked first.

We can add a filter method to the account object whose activity we want to track.

```
oo::objdefine smith_account {
    method Log args {
        my variable AccountNumber
        puts "Log([info level]): $AccountNumber [self target]: $args"
        return [next {*}$args]
    }
    filter Log
}
```

Now all actions on the account will be logged.

```
% smith_account deposit 100
→ Log(1): 2-71828182 ::Account deposit: 100
  Log(2): 2-71828182 ::Account UpdateBalance: 100
  999900
```

Notice from the output that all method invocations, even those called internally from `deposit` are recursively logged. The filter method must be aware that it may be recursively entered. We use the `info level` command to show the stack level. When methods are chained, the filter is called for **every** method in the chain.

Some additional notes on filter methods:

- Many times, the filter method needs to know what method the caller is actually trying to invoke. The `self target` command is useful for this purpose.
- Multiple filters may be present and are chained like any other method.
- Because filter methods are called for **all** method invocations, they are generally defined with a variable number of arguments.
- Filter methods may be defined on an object, as in our example, or on a class, in which case they will affect all objects belonging to the class.
- Our filter method is not exported because it starts with an upper-case letter. This means it will not be called accidentally by clients of the object. However, there is no requirement that filter methods must be private.
- The filter method normally passes on the call to the target method via `next` which can be called at any point in the filter method. Moreover, the filter is not required to call the target method at all.
- The filter method may choose to pass on the target method result, or something else entirely.
- Filter methods are bypassed when invoking constructors, destructors or the `unknown` method of a class.

14.8.1. Defining a filter class

The `filter` declaration need not occur in the same class that defines the filter method. This means you can define a generic class for a filter which can be mixed into a "client" class or object which can install or remove the filter at appropriate times as desired.

Let us rework our previous example. To start with a clean slate, let us get rid of the `Log` method defined earlier.

```
% oo::objdefine smith_account {
    filter -clear ❶
    deletemethod Log
}
```

❶ Note `-clear` option to clear any currently defined filters

Then we define a class that does the logging.

```
% oo::class create Logger {
    method Log args {
        my variable AccountNumber
        puts "Log([info level]): $AccountNumber [self target]: $args"
        return [next {*}$args]
    }
}
→ ::Logger
```

Since we only want transactions for that account to be logged, we mix it into the object and add the filter.

```
% oo::objdefine smith_account {
    mixin Logger
    filter Log
}
% smith_account withdraw 500
→ Log(1): 2-71828182 ::Account withdraw: 500
  Log(2): 2-71828182 ::Account UpdateBalance: -500
  999400
```

As you can see, we have the same behaviour as before. The advantage of course is that defining a class allows a collection of additional behaviours to be abstracted and easily added to any class or object without repeating the code.

In our example above, we used the `-clear` option to `filter` to first remove any filters from the object before adding the `Log` filter later. This is because `filter` by default will append to any filters already installed. Alternatively, we could have specified the `-set` option to `filter` which would **replace** any existing filters.

14.8.2. When to use filters

Filters could be replaced by other techniques such as overriding and then chaining methods. Conversely, method overrides, such as in our account `freeze` example, could be replaced by filters. Usually though it is clear which one makes the most sense. Some general rules are

- If we need to hook into multiple methods, it is easiest to use a filter method rather than override individual methods. If necessary, `self target` can be used within the filter to selectively hook specific methods as illustrated in Section 14.11.3.4.
- When a method behaves more as an "observer" on an object as opposed to being a core part of the object's function, a filter method is a better fit.
- Filter methods are always placed at the front of the method chain so that can be a factor as well in deciding to use a filter.

14.9. Method chains

Throughout this chapter we have seen that when a method is invoked on an object, the code implementing the method for that object may come from several different places — the object, its class or an ancestor, a mix-in, forwarded methods, filters or even unknown method handlers. TclOO locates the code to be run by searching the potential implementations in a specific order. It then runs the first implementation in this list. That implementation **may** choose to chain to the next implementation in the list using the `next` command and so on through the list.

14.9.1. Method chain order

For the exact search order and construction of this method chain, see the reference documentation of the `next` command. Here we will simply illustrate with an example where we define a class hierarchy with multiple inheritance, mix-ins, filters and object-specific methods. Note our method definitions are empty because we are not actually going to call them.

```
oo::class create ClassMixin { method m {} {} }
oo::class create ObjectMixin { method m {} {} }
oo::class create Base {
    mixin ClassMixin
    method m {} {}
    method classfilter {} {}
    filter classfilter
    method unknown args {}
}
oo::class create SecondBase { method m {} {} }
oo::class create Derived {
    superclass Base SecondBase
    method m {} {}
}
→ ::Derived
```

Having defined our classes, let us create an object and add in some object-specific methods and mix-ins.

```
Derived create o
oo::objdefine o {
    mixin ObjectMixin
    method m {} {}
    method objectfilter {} {}
    filter objectfilter
}
```

We have created an object of class `Derived` that inherits from two parent classes, all of which define a method `m`. Further we have mix-ins for both a class and directly into the object. To confuse matters further, we have filters defined at both the class and object levels.

What will the method chain for method `m` look like? Luckily, we do not have to work it out while reading the manpage. We can do it through introspection via the `info object call` command.

```
% print_list [info object call o m]
→ filter objectfilter object method
  filter classfilter ::Base method
  method m ::ObjectMixin method
  method m ::ClassMixin method
  method m object method
  method m ::Derived method
  method m ::Base method
```

```
  method m ::SecondBase method
```

The output shows the method chain so we can see for example that the filter methods are first in line.

The `info object call` command returns a list that contains the method chain for a particular method invocation for a particular object. Each element of the list is a sublist with four items:

- the type which may be `method` for normal methods, `filter` for filter methods or `unknown` if the method was invoked through the unknown facility
- the name of the method which, as noted from the output, may not be the same as the name used in the invocation
- the source of the method, for example, a class name where the method is defined
- the *implementation* type of the method which may be `method` or `forward`

We reiterate that not every method in the chain is automatically invoked. Whether a method occuring in the list is actually called or not will depend on preceding methods passing on the invocation via the `next` command.

14.9.2. Method chain for unknown methods

What does the method chain look like for a method that is not defined for the object? We can find out the same way.

```
% print_list [info object call o nosuchmethod]
→ filter objectfilter object method
  filter classfilter ::Base method
  unknown unknown ::Base method
  unknown unknown ::oo::object {core method: "unknown"}
```

As expected, the unknown method, where defined, is called. Note the root `oo::object` object which is the ancestor of **all** TclOO objects, has a predefined unknown method.

14.9.3. Retrieving the method chain for a class

The above example showed the method chain for an object. There is also a `info class call` command that works with classes instead of objects.

```
% print_list [info class call Derived m]
→ filter classfilter ::Base method
  method m ::ClassMixin method
  method m ::Derived method
  method m ::Base method
  method m ::SecondBase method
```

14.9.4. Inspecting method chains within method contexts

Within a method context, the command `self call` returns more or less the same information for the current object as `info object call`.

In addition, you can use `self call` from within a method context to locate the current method in the method chain. This command returns a pair, the first element of which is the same as the method chain list as returned by `info class call` command. The second element is the index of the current method in that list.

An example will make this clearer.

```
% catch {Base destroy} ❶
→ 0
% oo::class create Base {
    constructor {} {puts [self call]}
    method m {} {puts [self call]}
```

```
}
→ ::Base
% oo::class create Derived {
    superclass Base
    constructor {} {puts [self call]; next}
    method m {} {
        puts [self call]; next
    }
}
→ ::Derived
% Derived create o
→ {{method <constructor> ::Derived method} {method <constructor> ::Base method}} 0
  {{method <constructor> ::Derived method} {method <constructor> ::Base method}} 1
  ::o
% o m
→ {{method m ::Derived method} {method m ::Base method}} 0
  {{method m ::Derived method} {method m ::Base method}} 1
```

❶ Clean up any previous definitions

Note the special form `<constructor>` for constructors. Destructors similarly have the form `<destructor>`.

Constructor and destructor method chains are only available through `self call`, not through `info class call`.

14.9.5. Looking up the next method in a chain

At times a method implementation may wish to know if it is the last method in a method chain and if not, what method implementation will be invoked next. This information can be obtained with the `self next` command from within a method context.

We illustrate by modifying the `m` method of the `Derived` class that we just defined.

```
% oo::define Derived {
    method m {} { puts "Next method in chain is [self next]" }
}
% o m
→ Next method in chain is ::Base m
```

As seen, `self next` returns a pair containing the class or object implementing the next method in the method chain and the name of the method (which may be `<constructor>` and `<destructor>`). In the case the current method is the last in the chain, an empty list is returned.

Notice that although the next method in the method chain is printed out, it does not actually get invoked because the `m` method in `Derived` no longer calls `next`.

Do not confuse `self next` with `next`. The latter invokes the next method in the method chain while the former only tell you what the next method is.

There is one important issue solved by `self next` that we will illustrate with an example. Imagine we want to package some functionality as a mix-in class. The actual functionality is immaterial but it is intended to be fairly general purpose (for example, logging or tracing) and mixable into any class.

```
% oo::class create GeneralPurposeMixin {
    constructor args {
        puts "Initializing GeneralPurposeMixin";
```

```
        next {*}$args
    }
}
→ ::GeneralPurposeMixin
% oo::class create MixerA {
    mixin GeneralPurposeMixin
    constructor {} {puts "Initializing MixerA"}
}
→ ::MixerA
% MixerA create mixa
→ Initializing GeneralPurposeMixin
  Initializing MixerA
  ::mixa
```

So far so good. Now let us define another class that also uses the mix-in.

```
% oo::class create MixerB {mixin GeneralPurposeMixin}
→ ::MixerB
% MixerB create mixb
Ø Initializing GeneralPurposeMixin
  no next constructor implementation
```

Oops. What happened? If it is not clear from the error message, the issue is that the GeneralPurposeMixin class naturally calls `next` so that class that mixes it in can get initialized through its constructor. The error is raised because class `MixerB` does not have constructor so there is no "next" method (constructor) to call.

This is where `self next` can help. Let us redefine the constructor for GeneralPurposeMixin.

```
% oo::define GeneralPurposeMixin {
    constructor args {
        puts "Initialize GeneralPurposeMixin";
        if {[llength [self next]]} {
            next {*}$args
        }
    }
}
% MixerB create mixb
→ Initialize GeneralPurposeMixin
  ::mixb
```

It all works now because we only call `next` if there is in fact a next method to call.

14.9.6. Controlling invocation order of methods

As we have seen in our examples, a method can use the `next` command to invoke its successor in the method chain. With multiple inheritance, mix-ins, filters involved, it may sometimes be necessary to control the order in which inherited methods are called. The `next` command, which goes strictly by the order in the method chain, is not suitable in this case.

The `nextto` command allows this control. It is similar to `next` except that it takes an argument that specifies the name of the class that implements the next method to be called.

```
nextto CLASSNAME ?args?
```

Here *CLASSNAME* must be the name of a class that implements a method appearing later in the method chain.

When might you use this ? Well, imagine you define a class that inherits from two classes whose constructors take different arguments. How do you call the base constructors from the derived class? Using `next` would not work because the parent class constructors do not take the same arguments.

That's where `nextto` rides to the rescue as illustrated below.

```
oo::class create ClassWithOneArg {
    constructor {onearg} {puts "Constructing [self class] with $onearg"}
}
oo::class create ClassWithNoArgs {
    constructor {} {puts "Constructing [self class]"}
}
oo::class create DemoNextto {
    superclass ClassWithNoArgs ClassWithOneArg
    constructor {onearg} {
        nextto ClassWithOneArg $onearg
        nextto ClassWithNoArgs
        puts "[self class] successfully constructed"
    }
}
→ ::DemoNextto
```

We can now call it without conflicts.

```
% [DemoNextto new "a single argument"] destroy
→ Constructing ::ClassWithOneArg with a single argument
  Constructing ::ClassWithNoArgs
  ::DemoNextto successfully constructed
```

14.10. Programming without classes

Our exposition of object-oriented programming in Tcl has so far been centered around classes. Behaviours, object construction, inheritance and other relationships are all expressed in terms of classes.

However, not all object-oriented programming models involve classes. Another style, refered to in various forms as *classless* or *prototype-based programming*, dispenses with classes completely. Instead, objects are *cloned* from other objects, called prototypes, with the same methods and configuration. The inheritance features in class-based systems are replaced by the ability to add, remove or otherwise modify methods and delegate others.

As we stated in our introductory chapter, one of the defining features of TclOO is its flexibility in being adaptable to different programming models. Here we present classless object-based programming as one example of this.

We have already seen one of the requirements for such a system — the ability to define methods at an individual object level. We now describe how TclOO fulfils two additional ones — creating objects outside of classes, and cloning of objects.

The first of these is very straightforward, given our earlier discussion of `oo::object`. Given its dual nature as a class as well as an object, we can use it for creating a classless object.

```
oo::object create oa → ::oa
```

Strictly speaking, this object does have a class as seen below.

```
info object class oa → ::oo::object
```

For practical purposes though we can still treat this as classless as we did not have to go through an explicit class definition.

We now have what is essentially a shell of an object. The only method available for the object is `destroy` which it inherits from `oo::object`. There are no object-specific methods defined on it yet.

```
info object methods oa → (empty)
```

We now need to fill the object with methods and data. We know how to do that with `oo::objdefine`.

```
oo::objdefine oa {
    variable x
    method setx {val} {set x $val}
    method getx {} {set x}
}
oa setx 100
→ 100
```

We now have a functioning object. The last requirement is the ability to clone the object. The `oo::copy` command does this for us.

```
oo::copy OBJ ?NEWOBJ?
```

The command creates a new object named *NEWOBJ* that is a copy of *OBJ*. If *NEWOBJ* is not specified, the new object is created with an automatically generated name. The new object is of the same class as the source object, includes any object-specific methods that were defined on it and contains the same variables with values as of the time of copying.

```
oo::copy oa ob → ::ob
ob getx        → 100
```

As seen, we now have a new object ob that is the copy of oa with the same methods and data.

We are now free to extend (or not) this new object as we wish.

```
oo::objdefine ob method doublex {} {incr x $x} → (empty)
ob doublex                                     → 200
```

Notice that we now have similar functionality to method inheritance and overriding in class-based systems. A full-blown prototype-based object system would need more machinery than described here but that is all implementable with the mechanisms we have described so far and the introspection capabilities we will describe in the next section.

One final point about `oo::copy` that is worth noting. In some cases, copying across methods and variable definitions may not suffice for cloning an object. For example, the source object may have a variable that is being traced or a file that is open. To deal with such cases, the object may define a `<cloned>` method. This will be invoked at the time the object is cloned and is passed the source object as its sole argument. This method can then do any additional work required to create a full clone of the original.

14.11. OO introspection

Introspection of classes and objects from any context is primarily accomplished through the `info class` and `info object` ensemble commands. These have subcommands that return different pieces of information about a class or an object. In addition, the `self` command can be used for introspection of an object from inside a method context for that object.

The Tcl object browser[3] is a useful tool for peering inside classes, objects and namespaces. It displays the class hierarchy, objects, namespaces and method definitions in an easily navigable interface.

14.11.1. Introspecting classes

14.11.1.1. Enumerating classes

Classes are also objects in TclOO and therefore the same command `info class instances` used to enumerate objects can be used to enumerate classes.

```
% info class instances oo::class
→ ::oo::object ::oo::class ::oo::Slot ::Account ::SavingsAccount ::CheckingAccount
   ↳ ::BrokerageAccount ::CashManagementAccount ::EFT ::ConsolidatedAccount ::Logger
   ↳ ::ClassMixin ::ObjectMixin ::SecondBase ::Base ::Derived ::GeneralPurposeMixin ::MixerA
   ↳ ::MixerB ::ClassWithOneArg ::ClassWithNoArgs ::DemoNextto
```

We pass `oo::class` to the command because that is the class that all classes (or class objects, if you prefer) belong to. The returned list contains two interesting elements:

- `oo::class` is returned because as we said it is a class itself (the class that all class objects belong to) and is therefore an instance of itself.
- If that were not confusing enough, `oo::object` is also returned. This is the root class of the object hierarchy and hence is an ancestor of `oo::class`. At the same time it is a class and hence must be an instance of `oo::class` as well.

This circular and self-referential relationship between `oo::object` and `oo::class` seems strange but it is what allows all programming constructs in TclOO to be work in consistent fashion. It is also a common characteristic of many OO systems.

As before, we can also restrict the classes returned by specifying a pattern that the name must match.

```
% info class instances oo::class *Mixin
→ ::ClassMixin ::ObjectMixin ::GeneralPurposeMixin
```

14.11.1.2. Checking if an object is a class

The `info object isa class` command returns 1 if its argument references a class and 0 otherwise. A class is an instance of `oo::class` or one of its subclasses.

```
info object isa class SavingsAccount → 1
info object isa class savings        → 0 ❶
info object isa class clock          → 0 ❷
```

❶ An object but not a class
❷ A command and not a class

A related command is `info object isa metaclass` which returns 1 if the passed argument is a class that can create classes.

```
info object isa metaclass oo::class → 1
info object isa metaclass Account   → 0 ❶
```

❶ A class but not one that can create classes

[3] https://chiselapp.com/user/eugene.mindrov/repository/tcl-class-browser/home

14.11.1.3. Inspecting class relationships

The `info class superclasses` command returns the **direct** superclasses of a class.

```
% info class superclasses CashManagementAccount
→ ::CheckingAccount ::BrokerageAccount
% info class superclasses ::oo::class
→ ::oo::object
```

Notice that `oo::object` is a superclass of `oo::class`.

Conversely, the `info class subclasses` will return the classes directly inheriting from the specified class.

```
% info class subclasses Account
→ ::SavingsAccount ::CheckingAccount ::BrokerageAccount
```

As one might expect, there is also a command, `info class mixins` for listing mix-ins.

```
% info class mixin CheckingAccount
→ ::EFT
```

14.11.2. Introspecting objects

As for classes, there are several commands that let us introspect on TclOO objects.

14.11.2.1. Object identity

Under some circumstances, an object needs to discover its own identity from within its own method context. A method may need to know the command name of the object

- when an object method has to be passed to a command callback
- when an object is redefined "on the fly" from within an method, its name must be passed to `oo::objdefine`. See Section 14.5.1 for an example.

This can be accomplished with the `self object` command, which can also be called as simply `self`. We have seen this used in several instances in this chapter.

In addition to the command used to access it, an object may also be identified by the unique namespace in which the object state is stored. This is obtained through the `self namespace` command within a method context or with the `info object namespace` command elsewhere.

```
% oo::define Account {method get_ns {} {return [self namespace]}}
% savings get_ns
→ ::oo::Obj223
% set acct [Account new 0-0000000]
→ Reading account data for 0-0000000 from database
  ::oo::Obj256
% $acct get_ns
→ ::oo::Obj256
% info object namespace $acct
→ ::oo::Obj256
```

Notice that when we create an object using new, the namespace matches the object command name. This is an artifact of the implementation and this should not be relied on. In fact, like any other Tcl command, the object command can be renamed.

```
% rename $acct temp_account
```

```
% temp_account get_ns
→ ::oo::Obj256
```

As you can see, the object command and its namespace name no longer match. Also note that the namespace does not change when the object is renamed.

14.11.2.2. Checking if a command is an object

The `info object isa object` command can be used to check if a command is actually an object.

```
info object isa object savings       → 1 ❶
info object isa object clock         → 0 ❷
info object isa object nosuchcommand → 0 ❸
```

❶ A OO object
❷ A command but not an object
❸ No such command

14.11.2.3. Enumerating objects

The `info class instances` command returns a list of objects belonging to the specified class.

```
% info class instances Account
→ ::oo::Obj217 ::smith_account ::temp_account
% info class instances SavingsAccount
→ ::savings
```

As seen above, this command will only return objects that directly belong to the specified class, not if the class membership is inherited.

You can optionally specify a pattern argument in which case only objects whose names match the pattern using the rules of the `string match` command are returned. This can be useful for example when namespaces are used to segregate objects.

```
% info class instances Account ::oo::*
→ ::oo::Obj217
```

14.11.2.4. Inspecting class membership

You can get the class an object belongs to with the `info object class` command.

```
info object class savings → ::SavingsAccount
```

The same command will also let you check whether the object belongs to a class, taking inheritance into account.

```
info object class savings SavingsAccount  → 1
info object class savings Account         → 1
info object class savings CheckingAccount → 0
```

The `info object isa typeof` is an alternate means of getting at the same information.

```
info object isa typeof savings Account → 1
```

For enumerating the classes mixed-in with an object, use `info object mixins`, analogous to `info class mixins`.

```
info object mixins savings → ::EFT
```

Conversely, to check if a class is **directly** mixed into an object, use `info object isa mixin`.

```
info object isa mixin savings EFT → 1
```

From within the method context of an object, the command `self class` command returns the class defining the currently executing method. **Note this is not the same as the class the object belongs to as the example below shows.**

```
% catch {Base destroy} ❶
→ 0
% oo::class create Base {
    method m {} {
        puts "Object class: [info object class [self object]]"
        puts "Method class: [self class]"
    }
}
→ ::Base
% oo::class create Derived { superclass Base }
→ ::Derived
% Derived create o
→ ::o
% o m
→ Object class: ::Derived
  Method class: ::Base
```

❶ Clean up any previous definitions

The `self class` command will fail when called from a method defined directly on an object since there is no class associated with the method in that case.

14.11.3. Introspecting methods

14.11.3.1. Enumerating methods

The list of methods implemented by a class or object can be retrieved through `info class methods` and `info object methods` respectively. Options can be specified to control whether the list includes inherited and private methods. The `-private` option controls whether non-exported methods are also included in the returned list.

```
% info class methods CheckingAccount ❶
→ cash_check
% info class methods CheckingAccount -private ❷
→ cash_check
% info class methods CheckingAccount -all ❸
→ balance cash_check deposit destroy get_ns transfer_in transfer_out withdraw
% info class methods CheckingAccount -all -private ❹
→ <cloned> UpdateBalance balance cash_check deposit destroy eval get_ns transfer_in
   ↳ transfer_out unknown variable varname withdraw
% info object methods smith_account -private ❺
```

❶ Lists all methods defined and exported by `CheckingAccount` itself
❷ Lists both exported and private methods defined by `CheckingAccount`
❸ Lists all methods defined and exported by `CheckingAccount`, its ancestors, or mix-ins
❹ Lists both exported and private methods defined by `CheckingAccount`, its ancestors, or mix-ins
❺ Lists exported and non-exported methods defined in the object itself

The list of methods returned includes forwarded methods as well as shown by our example class for method forwarding.

```
% info class methods ConsolidatedAccount ❶
→ cash_check sell buy withdraw
% info object methods consolidated ❷
→ quick_cash
```

❶ Methods forwarded in the class
❷ Methods forwarded in the object

You can distinguish between "normal" methods and forwarded methods in the returned list by querying its type with the `info class methodtype` and `info object methodtype` commands. The commands will return `method` for the former and `forward` for the latter.

```
info class methodtype Account withdraw              → method
info class methodtype ConsolidatedAccount withdraw → forward
info object methodtype consolidated quick_cash      → forward
```

14.11.3.2. Retrieving method definitions

To retrieve the definition of a specific method that is not forwarded, use `info class definition`. This returns a pair consisting of the method's arguments and its body.

```
% info class definition Account UpdateBalance
→ change {
          set Balance [+ $Balance $change]
          return $Balance
      }
```

The method whose definition is being retrieved has to be defined in the specified class, not in an ancestor or a class that is mixed into the specified class.

Similarly, `info object definition` will return the definition of a method **directly** defined on an object. It will raise an error if passed a method name that is defined on the object's class.

Constructors and destructors are retrieved differently via `info class constructor` and `info class destructor` respectively.

```
% info class constructor Account
→ account_no {
          puts "Reading account data for $account_no from database"
          set AccountNumber $account_no
          set Balance 1000000
      }
```

For methods that are forwards, the `info class forward` and `info object forward` return information about the forward definition in a class and object respectively.

```
% info class forward ConsolidatedAccount buy
→ brokerage_account buy
```

```
% info object forward consolidated quick_cash
→ my withdraw 100
```

14.11.3.3. Inspecting method chains and contexts

The `info class call` command retrieves the method chain for a method. From a method context, the `self call` command returns similar information while `self next` identifies the next method implementation in the chain. We have already discussed these in detail in Section 14.9. There are two additional related commands that provide further information within a method context.

- The `self method` command returns the name of the current method being executed.
- The `self caller` command returns information of the caller of the method **when called from another method**. This is in the form of a list of three elements — the class, the object and the calling method.

```
oo::class create C {
    method m {} {
        lassign [self caller] cls obj meth
        puts "In [self method], called from method $meth in object $obj of class $cls"
    }
    constructor {} { my m }
}
C create c
→ ::c
  In m, called from method <constructor> in object ::c of class ::C
```

14.11.3.4. Inspecting filters

The list of methods that are set as filters can similarly be obtained with `info class filters` or `info object filters`.

```
% info object filters smith_account
→ Log
```

Note that `info object filters` will return a list of filter methods **directly** defined on the object. It will not include filters defined on the object's class.

When a method is run as a filter, it is often useful for it to know the real target method being invoked. This information is returned by `self target` which can only be used from within a filter context. Its return value is a pair containing the declarer of the method and the target method name.

For example, suppose instead of logging every transaction as in our earlier example, we only wanted to log withdrawals. In that case we could have defined the `Log` command as follows:

```
oo::define Logger {
    method Log args {
        if {[lindex [self target] 1] eq "withdraw"} {
            my variable AccountNumber
            puts "Log([info level]): $AccountNumber [self target]: $args"
        }
        return [next {*}$args]
    }
}
```

We would now expect only withdrawals to be logged.

```
% smith_account deposit 100
→ 999500
% smith_account withdraw 100
```

```
→ Log(1): 2-71828182 ::Account withdraw: 100
  999400
```

The other piece of information that is provided inside a filter method is about the filter itself and is available through the `self filter` command.

Let us redefine our Log filter yet again to see this.

```
% oo::define Logger {
    method Log args {
        puts [self filter]
        return [next {*}$args]
    }
}
% smith_account withdraw 1000
→ ::smith_account object Log
  ::smith_account object Log
  998400
```

As seen above, the `self filter` command returns a list of three items:

- the name of the class or object where the filter is declared. Note this is **not** necessarily the same as the class in which the filter method is defined. Thus above, the filter was **defined** in the `Logger` class but **declared** in the `smith_account` object.
- either `object` or `class` depending on whether the filter was declared inside an object or a class.
- the name of the filter.

You will see two output lines in the above example. Remember the filter is called at *every* method invocation. Thus the `Log` method is invoked twice, once before the `withdraw` method, and then again when that method in turn calls `UpdateBalance`.

14.11.4. Enumerating data members

The command `info class variables` returns the list of variables that have been declared with the `variable` statement inside a class definition and are therefore automatically brought within the scope of the class's methods.

```
% info class variables SavingsAccount
→ MaxPerMonthWithdrawals WithdrawalsThisMonth
```

The listed variables are only those defined through the `variable` statement **for that specified class**. Thus the above command will not show the variable `Balance` as that was defined in the base class `Account`, not in `SavingsAccount`.

For enumerating variables for an object as opposed to a class, there are two commands:

- `info object variables` behaves like `info class variables` but returns variables declared with `variable` inside object definitions created with `oo::objdefine`. These may not even exist yet if they have not been initialized.
- `info object vars` returns variables currently **existing** in the object's namespace and without any consideration as to how they were defined.

Hence the difference between the output of the following two commands.

```
% info object variables smith_account
% info object vars smith_account
```

```
→ Balance AccountNumber
```

The first command returns an empty list because no variables were declared through `variable` for **that object**. The second command returns the variables in the object's namespace. The fact that they were defined through a class-level declaration is irrelevant.

14.12. Chapter summary

This chapter described the core Tcl facilities for object-oriented programming. What distinguishes these from many other languages is their dynamic nature, which permits classes and objects to be changed on the fly, and the extensive customizability that permits many different styles of object-oriented programming. The references at the end of the chapter explore these aspects in more detail.

14.13. References

DKF2005
TIP #257: Object Orientation for Tcl, Fellows et al, http://tip.tcl.tk/257. The Tcl Implementation Proposal describing TclOO.

WOODS2012
Lifecycle Object Generators, Woods, 19th Annual Tcl Developer's Conference, Nov 12-14, 2012. Describes a number of useful OO patterns built on top of TclOO.

DKF2013
Adventures in TclOO, Fellows, www.tclcommunityassociation.org/wub/proceedings/.../Adventures-in-TclOO.pdf. Discussion of some advanced TclOO techniques and applications.

15

The Event Loop

> Great minds discuss ideas; average minds discuss events; small minds discuss people.
>
> — Eleanor Roosevelt

Not possessing a great mind, and unwilling to accept the possibility of a small one, I'm left with no choice but to discuss events.

Most of the programming we have discussed so far involves a sequential style of program flow where the Tcl interpreter executes a script one command at a time and then terminates when the last command in the script is finished. For applications that are primarily command line utilities, like file search, this is adequate. For other "long running" applications whose function is to react to different types of external events, this is not suitable model. Common examples include GUI applications which react to events like mouse clicks and key presses and network services that respond to requests from multiple clients. These applications sit idle waiting for specific events to occur upon the occurence of which they execute *event handlers*. These event handlers may update the display in response to a key press, send back a web page to the requesting client and so on. These applications are said to be written in *event driven* style.

15.1. Event sources and types

Events may come from a variety of sources, such as input devices, network connections and so on. Correspondingly there are a number of different event types. Out of the box, Tcl supports the following event types:

- Timer events that are generated on the expiry of some time interval. The `after` command is one way for scheduling these.
- Channel events that indicate I/O activity on the channel. These events may be related to arrival of data on a serial port, a network connection request, the user entering a line on standard input and so on.
- Idle events, which are "pseudo-events" that are generated when the system has no other events to process. These are used by applications to schedule background tasks that are to be run when the system is otherwise idle. We sometimes refer to these as *idle tasks*.

In addition, extensions can add both new event sources and event types. The most commonly encountered ones are those added by the Tk graphical interface toolkit. These events include display events such windows becoming visible, mouse movement and clicks and keyboard input.

How you register event handlers for the different events is dependent on the event source generating the events. For example, use the `chan event` or `fileevent` commands to register for channel events, the `after` command for scheduling timer events, the Tk `bind` command for Tk user interface events and so on.

We will look at the `after` command in Section 15.3 and the channel related commands in Chapter 17 but first let us look at the event loop.

15.2. The Tcl event loop

Tcl provides built-in support for event-driven applications through its *event loop*. The basic flow of event driven applications in Tcl is as follows:

1. At startup the application registers event handlers to be invoked on events of interest.
2. The event loop is entered. This sits in wait for events to occur and on the occurence of such an event, invokes the associated handler(s).
3. Each handler takes whatever actions are necessary in response to the event. As part of its operation, a handler may register itself or other handlers for additional events or unregister existing handlers.
4. After the handler completes, control returns to the event loop and the cycle is repeated.

The second step above, entering the event loop, may be done by the application's native code outside of the Tcl interpreter, or from within a Tcl script through any of several commands like `update` or `vwait`. Notice that the latter implies that event loops can be nested, something that we will look at in more detail later.

As we will see in later chapters, a Tcl application may consist of multiple threads each running multiple Tcl interpreters. In this case, each thread runs an independent event loop which is shared among all interpreters running in that thread.

15.2.1. The event and idle task queues

Internally, the event loop implementation runs off of two queues:

- The event queue: on the occurence of a physical event, such as a mouse click or arrival of data on a channel, the driver or module responsible will place an entry in this queue with details of the event and the handler to be called for the event.
- The idle task queue: this second queue holds tasks to be executed when the system is idle, i.e. has no events to process. At the script level, tasks can be placed on this queue through the `after idle` command.

15.2.2. Event loop operation

When the event loop runs,

1. It first checks the event queue for any handlers that have been queued up to be run and executes them if any. Note the handlers may themselves add new entries to the event queue.
2. If the event queue is empty, it checks with various registered event sources if new events have occurred. Any new events are placed on the event queue and the event loop goes back to the first step.
3. If there are no entries to process on the event queue, the event loop runs **all** the tasks present on the idle task queue.
4. If the idle task queue is empty, the event loop sits in wait for the next event to occur.

15.2.3. Entering the event loop

An application may itself enter the event loop from the C level. For example, the `wish` application evaluates the contents of any script file supplied as an argument and then enters the event loop processing user interface events. Thus in a `wish` based application, the event loop is effectively always running.

Alternatively, the Tcl script itself can initiate the event loop from script. This is commonly done from long running programs like network servers that are not GUI programs and therefore use `tclsh` to host the interpreter rather than `wish`.

Since we do not delve into C programming in this book, we will only concern ourselves with the latter method — running the event loop from a script.

15.2.3.1. Waiting on a variable: `vwait`

The most common way of entering the event loop from a script is the `vwait` command.

```
vwait VARNAME
```

The *VARNAME* variable name is resolved in the **global** context, and not in the context of the caller of `vwait`. You can also use a fully qualified namespace variable instead.

The command enters the Tcl event loop calling the handlers for events as they occur until the variable *VARNAME* is written at which point the command returns.

Unlike `wish`, the `tclsh` application does **not** automatically enter the event loop. You will therefore often find the following command at the end of a script that implements a `tclsh` based event-driven application such as a network server.

```
vwait forever
```

When the above command is executed, `tclsh` will enter the event loop processing network connection requests until one of the event handlers either calls `exit` to terminate the application or sets the variable `forever` at which point the command returns, the end of the script is reached and `tclsh` exits. We will see an example of such a server in Chapter 17.

Note that there is nothing special about the name `forever` above. It is commonly used as a hint that the command is supposed to loop forever and not return.

Here is a short example illustrating `vwait` that you can try out in the `tclsh` shell. The example uses the `after` command that we examine later to schedule timer events that trigger after a specified interval.

```
% after 1000 [list puts "1 second elapsed"]
→ after#0
% after 2000 [list puts "2 second elapsed"]
→ after#1
% after 3000 [list set ::done 1]
→ after#2
% vwait done
→ 1 second elapsed
  2 second elapsed
% puts $done
→ 1
```

Running the above sequence, you will see the `tclsh` prompt disappear for 3 seconds. Because the event loop is running, the timer events are triggered at one second intervals. After 3 seconds the variable done is set causing the `vwait` command (and event loop) to return.

15.2.3.1.1. Avoiding deadlocks with `vwait`

The `vwait` command should be used with some care from within code that runs within an event handler else deadlock can occur. Consider the following script.

```
proc handler {} {
    puts "handler enter"
    set ::varA 1
    vwait ::varB
    puts "handler exit"
}
proc demo {} {
    after 1000 handler
    vwait ::varA
    set ::varB 1
}
demo
```

If you run this fragment in a `tclsh` shell, you will find that the prompt does not return after printing `handler enter`. The expectation was that setting `varA` within the handlerA would permit the demo procedure to continue

which in turn would set varB allowing handler to also complete. However, what actually happens is that control will not return to demo until **after** handler completes. But handler cannot complete until varB is set. Hence deadlock.

This example is contrived but similar situations will arise in real life if you are not careful. The fundamental point to be made is that **multiple calls to vwait do not execute "in parallel", they are nested**. The outermost call will not return until inner calls have returned.

These situations generally arise as a result of trying to make asynchronous execution appear synchronous. See the vwait documentation in the Tcl reference pages for additional information and workarounds. In many cases coroutines, which we discuss in Chapter 21, are a better option.

As an aside, the above example also illustrates that it is possible to enter the event loop recursively.

15.2.3.2. Single invocation: update

The vwait command runs the event loop continuously until a variable is set, waiting for events if there are none to be processed currently. In contrast, the update command invokes the event loop once and returns when there are no pending events to be processed.

```
update ?idletasks?
```

If idletasks is not specified, the command enters the event loop and runs all event handlers that are pending including new ones that may be added while the event loop is running. The command returns when no more events remain in the event queue. The following short example illustrates its behaviour.

```
proc handler {} {
    puts "Event 0"
    after 0 [list puts "Event 1"]
}
after 0 handler
after 1000 [list puts "Event 2"]
update
→ Event 0
  Event 1
```

The after command schedules a timer to expire after a specified time. In this case 0 means the timer expires immediately while 1000 indicates expiry after one second. When a timer expires, a timer event is triggered and the corresponding handler becomes pending. When update is called, all **pending** handlers are run. This causes handler to run which again schedules another timer to expire immediately. Thus the event loop invocation runs that as well. On the other hand, the one second timer has not expired as yet so no more event handlers are pending and the event loop terminates and the update command returns. Thus we see Event 0 and Event 1 being printed and not Event 2. If you do run update (or the event loop by any other means) after a second has elapsed, you will see the one second timer handler being run as well.

If the idletasks argument is specified to the update command, only background tasks scheduled to run when the event loop is idle are processed.

```
after 0 [list puts "After 0"]
after idle [list puts "After idle"]
update idletasks
→ After idle
```

The after idle command registers a script to be run when the event loop is idle. Notice from the output that only the idle event handler was run. The timer event handler was not run even though it would have been pending. Moreover, the example shows that update idletasks forces any idle handler to be invoked **even if other events were pending** and the event loop was not actually idle.

Update considered harmful
The `update` command is often used to keep an application's user interface responsive during the course of a long computation. This is not considered good practice due to reentrancy and data consistency issues. The Tcler's Wiki [1] page http://wiki.tcl.tk/1255 discusses this in some detail.

In the author's experience, the only time an `update` has been mandated is for forcing window geometry calculation and propagation in the Tk GUI extension on some platforms that use native OS widgets.

15.2.4. Event handlers and the call stack

It is worthwhile taking a look at how the call stack (see Section 10.5) and the C stack (see Section 10.5.6) appear when an event handler is running. Figure 15.1 shows one such snapshot.

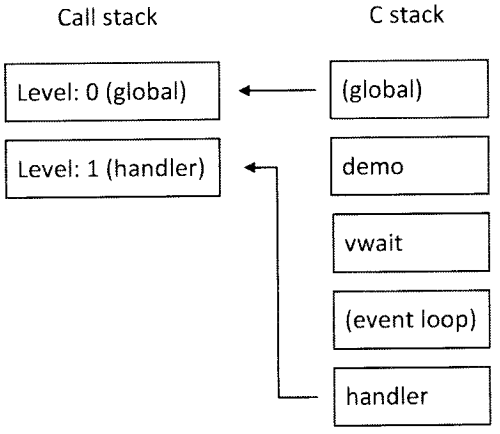

Figure 15.1. Call stack in an event handler

The above figure reflects the state of the two stacks when the `handler` procedure is running in the following code snippet.

```
proc handler {} {
    puts "handler level: [info level]"
    set ::done 1
}
proc demo {} {
    puts "demo level: [info level]"
    after 0 handler
    vwait ::done
}
demo
→ demo level: 1
  handler level: 1
```

[1] http://wiki.tcl.tk

Note the following points from the figure:

- From the perspective of the internal C stack, the call to vwait which in turn runs the event loop and executes the handler, all add a level. Further calls to vwait (or update which would behave similarly) from within the event handler would add more levels to the C stack.
- On the other hand, the call stack that maintains execution and variable contexts is reset to the global level. The output of our little script which showed both demo and handler executing at level 1 corroborates this. This illustrates that event handlers are run in the global context. You cannot expect to use upvar or uplevel to reach into the context of the demo procedure. Note that those contexts are not lost, they will be restored once the vwait command returns.

15.3. Scheduling execution of code: after

We now turn our attention to how we can add entries to the event and idle queues to be invoked via the event loop. There are many mechanisms through which this can happen as we will describe in later chapters. Here we only look at the simplest of these:

- Timer expiry events which can be used to run code after a specified interval
- Registering tasks to be run when the system is idle and no events are pending processing.

The after command is used for both purposes, and more.

As of Tcl 8.6, one limitation of the after command to keep in mind is that it depends on the system clock for time-keeping. If the system clock is inaccurate or jumps for whatever reason, after will not correctly measure intervals. There is work in progress to fix this behaviour which will likely show up in future releases of Tcl.

15.3.1. Suspending execution

Most systems have a sleep call which effectively suspends the caller for a specified interval. The same can be done in Tcl with the after command.

```
after MILLISECS
```

This call will halt the execution of all code, including the event loop, in the current thread for the specified number of milliseconds. For example,

```
after 250
```

will put the current thread to sleep for a quarter of a second.

15.3.2. Scheduling code

The next form of the after command schedules code to run after a specified time interval has elapsed.

```
after MILLISECS SCRIPT ?SCRIPT ...?
```

The command sets up a timer that will expire after *MILLISECS* milliseconds and returns right away. The result of the command is an identifier for the timer that can be passed to after cancel to cancel it if desired.

When the timer expires, an entry is added to the event queue with a handler that will invoke the script formed by concatenating the *SCRIPT* arguments separated by a space in the same manner as the concat or eval commands.

We have already seen examples of this command for setting up timers. Here is a slightly more realistic example that uses the http package to retrieve a Web page in conjunction with a timeout within which the transaction must complete.

```
package require http
proc http_data_sink {token} {
    set ::status done
}
proc geturl_with_timeout {url ms} {
    after $ms {set ::status timeout}
    set http_token [http::geturl $url -command http_data_sink]
    vwait ::status
    if {$::status eq "timeout"} {
        http::cleanup $http_token
        error "Operation timed out."
    }
    set data [http::data $http_token]
    http::cleanup $http_token
    return $data
}
```

Let us try to retrieve a Web page.

```
% geturl_with_timeout http://www.example.com 10000
→ <!doctype html>
  <html>
  <head>
...Additional lines omitted...
```

That works fine, but if we give it a short time to do its work,

```
% geturl_with_timeout http://www.example.com 10
Ø Operation timed out.
```

the timer event fires first and the operation is timed out.

A special case of scheduling an task is when 0 is specified as the value for the *MILLISECS* argument. In this case the timer expires immediately and the handler script is appended to the event queue. This idiom is often used when the programmer wants some piece of code to run only after the current handler completes execution. Another use is to break up a long computation into smaller pieces while allowing other handlers to run. We will say more about this in Section 15.3.3.1.

The `cron` package from Tcllib[2] provides a layered interface to the `after` command that may be more convenient. For example, it permits absolute times to be specified for executing code and can schedule recurring events. It is also potentially more efficient when a large number of timer events are to be scheduled as it collapses multiple timers that expire at the same timer.

15.3.3. Running on idle: `after idle`

The `after` command can also be used to add background tasks to the idle task queue.

```
after idle SCRIPT ?SCRIPT …?
```

This command behaves almost identically to the `after 0` form of the command except that the script formed by concatenating the *SCRIPT* arguments is added to the idle task queue instead of the event queue. Here is a short example that illustrates the relation between the two queues.

[2] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

```
% after idle [list puts "Idle task executed"]
→ after#11
% after 0 [list puts "Event handled"]
→ after#12
% update ❶
→ Event handled
  Idle task executed
```

❶ Run all pending event and idle tasks

Notice that the idle handler runs after the event handler even though it was queued first.

As before, the command returns an identifier that can be used to remove the task from the queue with the `after cancel` command.

15.3.3.1. Avoiding event queue starvation

Running a long computation prevents the event loop from running and responding to user input, network activity and so on. A common technique used to avoid this is to break up the computation into pieces and run each part in turn from the event loop.

We will illustrate with our prior example of summing the first N natural numbers. Our procedure will do one addition operation and then queue itself back on the event queue to continue with the next stage of computation.

```
proc background_sum {n {sum 0}} {
    if {$n <= 0} {
        puts "Sum is $sum"
    } else {
        puts "Calculating..."
        incr sum $n
        after 0 [list background_sum [incr n -1] $sum]
    }
}
```

Notice when we call the command nothing is printed because the computation happens via the event loop (this assumes you are running in `tclsh` and not `wish` where the event loop is already running):

```
% after 0 background_sum 2
→ after#13
```

We can run the event loop to finish the computation.

```
% update
→ Calculating...
  Calculating...
  Sum is 3
```

The `update` command keeps executing handlers as long as the event queue is not empty. Since we keep adding a timer (with a 0 expiration) the event loop will keep running until the computation is completed. Morever, while this computation is going on, other events that may arrive in the meanwhile will also be queued and executed in the order of arrival. The user interface will stay responsive, network connections will be accepted and so forth.

However, there is one flaw in the above. As long as there are entries in the event queue, their handlers will be executed and the **event loop will never move on to the idle task queue**. The idle task queue will be starved of any execution cycles and activities like updating of windows which happen at idle time will not happen.

To demonstrate this, let us write a script that will keep running in the background on the idle queue.

```
proc idler {{n 2}} {
    puts Idle!
    if {$n > 0} {
        after idle [list idler [incr n -1]]
    }
}
```

Now we fire up the two scripts.

```
% after idle idler
→ after#16
% after 0 background_sum 2
→ after#17
% update
→ Calculating...
  Calculating...
  Sum is 3
  Idle!
  Idle!
  Idle!
```

As shown by the output, the idle task does not get to run until the computation was done.

Queuing our computation on the idle task queue would not solve the problem either because once the event loop starts processing the idle task queue, it will continue to do so until it is empty. We will therefore starve the event queue instead.

The solution to this is to modify our procedures as follows

```
proc idler {{n 2}} {
    puts Idle!
    if {$n > 0} {
        after 0 [list after idle [list idler [incr n -1]]] ❶
    }
}
proc background_sum {n {sum 0}} {
    if {$n <= 0} {
        puts "Sum is $sum"
    } else {
        puts "Calculating..."
        incr sum $n
        after idle [list after 0 [list background_sum [incr n -1] $sum]] ❷
    }
}
```

❶ Modified line
❷ Modified line

Now if we run our previous code, the script output is interleaved indicating neither queue is starved.

```
% after idle idler
→ after#22
% after 0 background_sum 2
→ after#23
% update
→ Calculating...
  Idle!
  Calculating...
  Idle!
```

```
Sum is 3
Idle!
```

In essence, instead of rescheduling itself directly, the code "bounces" itself off the other queue ensuring entries on that queue get to run as well.

This technique applies to computations that directly or indirectly reschedule themselves continuously. If **new** independent events are arriving at a rate faster than the rate at which the queues can be emptied, this would obviously not help.

15.3.4. Cancelling tasks: after cancel

Any timer events and idle tasks that have been scheduled with the after command can be cancelled with after cancel which can take one of two forms:

```
after cancel ID
after cancel SCRIPT ?SCRIPT ...?
```

In the first form, the command's argument is an identifier returned with the after or after idle commands. The corresponding event is cancelled.

```
set id1 [after 0 puts Timer1] → after#32
set id2 [after 0 puts Timer2] → after#33
after cancel $id1             → (empty)
update                        → Timer2
```

Only the second timer fires as the first one has been cancelled.

In the second form, instead of specifying the timer identifier, the caller can specify the actual script that was scheduled. Repeating the above example but using the idle task queue and the second form of the command,

```
after idle puts Timer1   → after#34
after idle puts Timer2   → after#35
after cancel puts Timer1 → (empty)
update                   → Timer2
```

Again, only the second timer fires as the first was cancelled.

Note that the command does not raise an error if no matching timer is found in either case.

15.3.5. Querying after handlers

The after info command can be used to query the current event handlers registered with the after commands.

```
after info ?ID?
```

If no *ID* argument is specified, the command returns a list of identifiers of the currently active handlers.

```
after 1000 {puts "Timer"} → after#36
after idle {puts "Idle"}  → after#37
after info                → after#37 after#36
```

If *ID* is specified, it returns a pair whose first element is the associated handler script and second is either timer or idle depending its type.

```
% foreach id [after info] {
    lassign [after info $id] script type
    puts "$id ($type): $script"
    after cancel $id
}
→ after#37 (idle): puts "Idle"
  after#36 (timer): puts "Timer"
```

Note that timers that have already been triggered or that have been canceled do not show up in the results returned by `after info`.

15.4. Event loop error handling

When an error exception is raised it is propagated up the call stack as described in Section 11.2.2 until it is handled at some call level. If it reaches all the way to the global level in a Tcl application that is not event-driven, the result is application dependent. The `tclsh` shell in interactive mode will trap the error and display it to the user. In non-interactive mode, `tclsh` will by print the error to standard output and exit.

Things work differently with event driven applications. If an error is raised during the execution of an event (or idle task) handler and propagates up to the event loop, it is reported through a background exception handler. The event loop invokes this handler with two additional arguments: the interpreter result and a dictionary of return options. These are exactly the same values that are captured by a `catch` command and are described in Section 11.4.1 and Section 11.4.2.

The `tclsh` and `wish` shells provide their own default background exception handlers which display the error message on the `stderr` channel and through a window dialog respectively.

Here is a demonstration of the `tclsh` default background error handler.

```
% proc demo {arg} {}
% after 0 demo ❶
→ after#2
% catch update
→ wrong # args: should be "demo arg"
      while executing
  "demo"
      ("after" script)
  0
%
```

❶ Intentionally call demo with wrong number of arguments

Notice that `catch` command returned a 0 indicating the `update` command ran without errors. The generated error exception was handled by the event loop and not propagated to `update`. The default background exception handler invoked by the event loop printed the error stack to the standard error channel.

15.4.1. Custom background error handling: `interp bgerror`

An application can choose to customise default handling of background exceptions by calling `interp bgerror`.

```
interp bgerror INTERPRETER CMDPREFIX
```

The handling of background errors can be customized on a per-interpreter basis and the *`INTERPRETER`* argument specifies the path of the interpreter to be customized. We will take a deeper look at interpreter paths in Chapter 20 but for the moment we just mention that the empty string refers to the current interpreter.

The *CMDPREFIX* argument is a command prefix that will be called with two additional arguments, the error result and return options dictionary as earlier described.

To illustrate, let us define our own background exception handler for our above example.

```
% proc bghandler {message ropts} { puts stderr "MyApp error: $message" }
% interp bgerror {} bghandler
→ bghandler
% after 0 demo
→ after#2
% catch update
→ MyApp error: wrong # args: should be "demo arg"
  0
%
```

Our error handler does essentially the same thing as the default one except adding an application name and only printing the error message instead of the whole stack. In a real application, you might choose to log the error to a file or some other more sophisticated action.

In old versions of Tcl, the global `bgerror` command was used for customizing background error handling. This is now deprecated in favour of `interp bgerror`.

15.5. Chapter summary

In this chapter we have introduced the event loop which forms the basis of asynchronous programming facilities required for almost any long running application. We also looked at the basic use of the event loop to run scheduled and background tasks and described how error exceptions in these are managed. In subsequent chapters, we will examine the use of the event loop for advanced I/O, networking and interprocess communications.

16

Processes and Pipelines

The most basic form of integration with other programs and applications is the exchange of data by coupling the standard inputs and outputs of processes. Tcl provides two commands for this purpose.

- The exec command, starts one or more child processes and returns any content written to their standard output as the result of the command.
- The open command on the other hand provides more flexibility by returning a channel that can be used to communicate with child process(es).

Both commands also support *process pipelines* wherein multiple processes are chained via their standard input and output.

16.1. Executing child processes: exec

The exec command starts a *pipeline* of one or more processes.

```
exec ?-keepnewline? ?-ignorestderr? ?--? ARG ?ARG ...? ?&?
```

Here the *ARG* arguments specify one or more programs to run along with their parameters or special character sequences that separate the programs and indicate redirection of input and output. If the last argument is not the & character, the command result is the data written to standard output by the last process in the pipeline. Any trailing newline character in the output is discarded unless the `-keepnewline` option is specified. The significance of the & character and the `-ignorestderr` option are discussed later. As always, `--` can be used to indicate the end of options.

In the simplest case, the command starts a single process and returns the data written to its standard output as the result of the command. For example, here we run the netstat program and collect its output.

```
% set connections [exec netstat -n]
→
  Active Connections

    Proto  Local Address          Foreign Address        State
    TCP    192.168.1.128:53143    40.100.136.18:443      ESTABLISHED
...Additional lines omitted...
```

In the general form, the arguments can specify multiple programs comprising a process pipeline where each program and its parameters is separated from the other programs by a | or |& character sequence. In the former case, the standard output of the preceding process in the pipeline is fed into the standard input of the next process. In the latter case, both standard output and standard error of the preceding process are piped into the standard input of the next.

In the absence of any I/O redirection or errors, the output of the last process in the pipeline is returned as the result of the exec command.

Here is a pipeline where we filter the output of `netstat` through the `findstr` program on Windows to only retrieve UDP connections.

```
% set udp_connections [exec netstat -an | findstr UDP]
→   UDP    0.0.0.0:3544           *:*
    UDP    0.0.0.0:3702           *:*
    UDP    0.0.0.0:3702           *:*
...Additional lines omitted...
```

16.1.1. Passing program arguments

When passing argument values to the executed programs, keep in mind that the arguments first undergo substitutions as per Tcl's quoting rules (see Section 3.2). Depending on the program being executed, they may then be subject to that program's quoting rules which in all likelihood differ from those of Tcl. Thus care must be taken to appropriately escape program arguments when special characters are part of the passed arguments. Unfortunately, different programs follow different conventions, particularly on Windows, so the escaping of special characters is necessarily program-specific.

The same also applies to file paths that may be passed to the child processes. On Windows for example, many programs do not accept `/` as a path separator. Thus attempting to produce a directory listing of the Tcl installation binary directory with Windows command shell's `dir` internal command will fail.

```
% exec cmd /c dir [file dirname [info nameof]]
Ø Parameter format not correct - "l".
```

The passed file path must be transformed into native form with the `file nativename` command.

```
% exec cmd /c dir [file nativename [file dirname [info nameof]]]
→  Volume in drive C is OS
   Volume Serial Number is E8C6-0D60

   Directory of c:\tcl\866\x64\bin

  03/30/2017  07:28 PM    <DIR>          .
  03/30/2017  07:28 PM    <DIR>          ..
...Additional lines omitted...
```

This transformation of file paths is only required for program arguments. If a directory path is specified for the program name itself, Tcl will automatically convert it to the native form. No explicit conversion is necessary.

Also to be noted is that Tcl's `exec` behaviour should not to be confused with that of Unix shells. The latter implicitly do glob-style expansion of wildcard patterns in arguments to programs whereas Tcl's `exec` command does not. Thus the following command executed in a Unix shell:

```
ls *.c
```

would not be

```
exec ls *.c
```

in Tcl but rather

```
exec ls {*}[glob *.c]
```

16.1.2. Locating programs

The program to be executed for each stage of the exec pipeline may be an executable image or a shell script (on Unix) or a batch file (on Windows). It may be specified as an absolute path, a relative path or just a file name with no directory component. The path will undergo tilde substitution (see Section 9.1.1.3) as appropriate.

If the program is specified purely by name (i.e. no directory components are present), exec will look for the file in an operating system dependent fashion. On Unix, it looks for the file in directories specified in the PATH environment variable. On Windows, the command will look in the directory containing the Tcl application, the current directory, the Windows system directories and finally the directories in the PATH environment variable.

Additionally, on Windows platforms if the search fails and no extension was specified in the program path, the command will repeat the search by appending .com, .exe, .bat and .cmd to the file name.

16.1.2.1. Locating internal commands: auto_execok

Some "programs" that are executed in command shells are not separate executables at all but are actually implemented internal to the shell. For example, the DIR command for listing directories at the Windows command prompt is internal to the Windows cmd.exe shell. Attempting to run this directly from Tcl will raise an error.

```
% exec dir *.*
→ couldn't execute "dir": no such file or directory
```

Tcl provides a command, auto_execok, for dealing with such commonly used commands that are built into the operating system command shells.

```
auto_execok PROGRAMNAME
```

The command returns a list of words to be passed to exec to run that program.

```
% auto_execok notepad
→ C:/WINDOWS/system32/notepad.EXE
% auto_execok dir
→ C:/Windows/System32/cmd.exe /c dir
```

Notice the difference in the two outputs. Since dir is an internal command of cmd.exe, auto_exec returns the command prefix to be used to invoke it. We can then run it as

```
% exec {*}[auto_execok dir] *.*
→  Volume in drive C is OS
   Volume Serial Number is E8C6-0D60

   Directory of C:\temp\book

...Additional lines omitted...
```

16.1.3. Redirecting I/O

By default, the output of each process in the pipeline is supplied as the input to the next process. This behaviour can be changed for each process in the pipeline for both input and output through special character sequences in the arguments to exec. This I/O redirection takes a form very similar to that used in Unix or Windows command shells.

16.1.3.1. Redirecting input

By default, the **first** process in an exec pipeline reads its standard input from the standard output of the parent Tcl application that invoked the exec command. This behaviour can be changed so that the process gets its input

- from a file
- from an open channel in the Tcl application
- from a value in the Tcl application

We will demonstrate all three techniques by recursively invoking `tclsh` as a separate process and printing its PID.

On Windows platforms, the examples will not work with the `wish` shell because GUI programs on Windows do not have a real operating system provided standard input or output.

Redirecting input from a file

To have the first process read its standard input from a file, prefix the file path with a < character. Let us first write out our sample file that we use as input.

```
set chan [file tempfile temppath] ❶
puts $chan { puts "My PID is [pid]" }
close $chan
```

❶ See Section 9.2.10

Now to recursively invoke ourselves with standard input for the child process redirected to this file, pass the file path prefixed with < to the `exec` command.

```
% exec [info nameofexecutable] <$temppath
→ My PID is 10328
```

Note that you can optionally separate the file path from the < character with whitespace so we could also have written the above as

```
% exec [info nameofexecutable] < $temppath
→ My PID is 4056
```

(Note the space character before the `temppath` reference.)

Redirecting input from a channel

As an alternative to redirecting input from a file, you can redirect input from a channel that is already open by prefixing the channel with the <@ character sequence. For example, a variation of the above example:

```
% set chan [open $temppath r]
→ file3787000
% exec [info nameofexecutable] <@$chan
→ My PID is 1732
% close $chan
```

As shown in the example, the channel must have been opened for reading for channel based redirection to work. As before, the child process will exit when it encounters an EOF on the input channel.

Not all channel types are supported for use with input redirection. In particular, on Windows platforms network socket based channels cannot be used unlike on Unix. Moreover, on all platforms reflected channels (see Section 17.3) also do not work with input redirection.

You have to keep two factors in mind when using the channel based input redirection. The first is that the file access pointer is shared between the parent and the child so the following sequence of commands will not yield the desired result.

```
set fd [open foo.txt w+]           → file3037340
puts $fd {puts "My PID is [pid]"}  → (empty)
exec [info nameofexecutable] <@$fd → (empty)
close $fd                          → (empty)
```

The reason why this does not work is that after the write to the channel, the file access pointer is positioned at the end. Consequently, when the child process reads from the channel it only sees the end-of-file marker and exits. This problem can be fixed by inserting a seek to reposition the access pointer to the front of the file.

```
set fd [open foo.txt w+]           → file3787000
puts $fd {puts "My PID is [pid]"}  → (empty)
chan seek $fd 0 start              → (empty)
exec [info nameofexecutable] <@$fd → My PID is 3220
close $fd                          → (empty)
```

Now the child process reads the file as desired.

The other point to remember is the potential for race conditions. If you change the order of operations to invoke the `exec` before the `puts`, it is possible that the child process will run before the write to the file, find it empty and exit.

For these reasons, the channel redirection mechanism is not recommended as a means for continuous communication between the parent and the child. We will see alternative means described in Section 16.2 and Section 16.3 that are more suitable for such scenarios.

Redirecting input from a Tcl value

The final option available for redirecting the standard input of the spawned process is through the << redirection operator. Instead of a file path or a channel, this redirects input from the specified value. Our example could be written as

```
% exec [info nameofexecutable] << {puts "My PID is [pid]"}
→ My PID is 4620
```

Tcl arranges for the argument following the << to be passed in to the child process in its standard input. We have specified the value in our example as a braced string literal. Of course it could also have been the result of a command or a variable reference as well.

16.1.3.2. Redirecting output

Just as for the input side, standard output and error of processes in an `exec` pipeline can also be redirected. There are more combinations possible here which can be confusing so we first lay out elements that are common to all.

- Only the standard output of the **last** process in the pipeline can be redirected. Other processes always send their output to the next process.
- On the other hand, the redirection of standard error output applies to **all** processes. It is not possible to only redirect it for any single process (not even the last in the pipeline).
- The single > character sequence always indicates writing to the channel at its current file access position. The double >> sequence on the other hand always appends.
- A prefix of 2 before any of the above character sequences indicates the redirection only applies to standard error and not to standard output.
- A suffix of & indicates the redirection applies to both standard output and standard error.
- The @ character specifies redirection to an open channel in the Tcl application calling the `exec`. The channel must have been opened for writing.

- If output is redirected, the result of the exec command is the empty string.

Sending standard output to a file

The > character followed by a file path will write the standard output of the last process in the pipeline to that file, overwriting it if it already exists.

```
exec netstat -an | findstr ESTABLISHED > connections.log
```

Note that the result of the exec command above is the empty string due to the redirection.

The << redirection works similarly except that it will append to the file instead of overwriting it.

Sending standard error to a file

The 2> character sequence followed by a file path will write the standard error of all processes in the pipeline to that file, overwriting it if it already exists. The standard outputs are unaffected. In the example below we spawn off another Tcl interpreter and have it print messages to standard output and error.

```
% exec [info nameofexecutable] 2> error.log << {
    puts stdout "This is standard output"
    puts stderr "This is standard error"
}
→ This is standard output
% print_file error.log
→ This is standard error
```

As seen above, only the standard error is redirected to the file while the standard output appears as the result of the exec command.

The standard error output can be appended to the file instead of overwriting it by using the 2>> redirection instead.

Redirecting the standard error modifies the error handling behaviour of exec. This is discussed in Section 16.1.4.

Sending output to a channel

Instead of sending standard output and error to a file, it can be sent to an open channel in the current Tcl interpreter by using >@ and 2>@ respectively.

```
% set chan [file tempfile temppath]
→ file420a2d0
% exec {*}[auto_execok date] /t >@ $chan
% close $chan
% print_file $temppath
→ Tue 07/04/2017
```

Note that there are no equivalent redirections that append to channels.

Conflating standard output and error

The redirections discussed so far have dealt with independently redirecting standard output and error. For example, one might write

```
exec grep Tcl_.*Init {*}[glob *.c] > matches.txt 2> errors.txt
```

In cases where you want both standard output and error to go to the **same** destination, append a & character to the appropriate operator. Thus >& and >>& will overwrite or append the standard output of the **last** process and the standard error of **all** processes in the pipeline to the specified file.

```
exec [info nameofexecutable] >& output.log << {
    puts stdout "This is standard output"
    puts stderr "This is standard error"
}
print_file output.log
→ This is standard output
  This is standard error
```

The >&@ works similarly except it writes to an open channel.

The use of the >& as above is **not** the same as separately redirecting the standard output and error to the same file as below.

```
exec [info nameofexecutable] > output.log 2> output.log << {
    puts stdout "This is standard output"
    puts stderr "This is standard error"
}
print_file output.log
→ This is standard error
```

As you see from the above, the output from one can overwrite or mangle output from the other.

One final form, 2>@1, redirects standard error to be included as part of the command result. Since the standard output is already included in the result, the following will return both as the result of the exec command.

```
set result [exec [info nameofexecutable] << {
    puts stdout "This is standard output"
    puts stderr "This is standard error"
} 2>@1]
puts $result
→ This is standard output
  This is standard error
```

16.1.4. Error handling in exec

Error handling for exec invocation is complicated by the fact that the error may come from different sources:

- The exec command itself may raise an error if one of the programs in the pipeline is not found or does not have execute permission for the user and so on.
- The program(s) run but terminate with some error condition.

Moreover, the executed programs may signal error conditions or abnormal termination in different ways:

- The process may exit with a non-0 exit code.
- The process may write to its standard error output.

The application needs to be able to recognize and distinguish all these different forms. A further complication is that not all programs follow the above conventions. For example, a search application may use the exit code as a result value returning the number of matches. Others may use standard error to record progress messages, not necessarily errors. For these reasons, error detection is very much dependent on the program(s) being executed. The discussion below is focused on distinguishing the various cases. Interpretation as errors or normal behaviour is up to the application.

We can interactively explore various scenarios by spawning a Tcl process that imitates application behaviour. For starters we assume that standard error has **not** been redirected as that affects error handling.

First, consider an attempt to execute a program that does not exist.

```
% catch {exec nosuchprogram} result ropts
→ 1
% dict get $ropts -errorcode
→ POSIX ENOENT {no such file or directory}
```

As expected, the `catch` command result indicates an error. The `-errorcode` element of the return options dictionary gives the details of the failure. Other error codes are also possible, such as insufficient permissions. These are generally returned as POSIX error codes.

Another possibility is that the program starts up but suffers abnormal termination via a segment violation, signal etc. In this case the error code will be of the form

```
CHILDKILLED PID SIGNALNAME MESSAGE
```

where *PID* is the process identifier of the terminated child process, *SIGNALNAME* indicates the signal (`SIGTERM`, `SIGSEGV` etc.) that forced the termination and *MESSAGE* is the human readable description of the reason for termination. For example, a null pointer access in the child process would result in an error code of

```
CHILDKILLED 12408 SIGSEGV {segmentation violation}
```

Finally, there is the possibility of the program itself signalling an error. Tcl considers a child process exiting with a non-0 exit code **or** writing to its standard error output to be an error. We can simulate both these conditions by recursively invoking the Tcl shell.

```
catch {
    exec [info nameofexecutable] << {
        puts "This is standard output"
        exit 3
    }
} result ropts
→ 1
```

The `catch` command returns an error because child exited with a non-0 exit code (3 in this case). The error code from the return options dictionary also includes this exit code.

```
puts [dict get $ropts -errorcode] → CHILDSTATUS 10092 3
```

Just as for the normal completion of an executed program, the result includes its standard output. In addition, the error message is also appended to this result.

```
% puts $result
→ This is standard output
  child process exited abnormally
```

The other situation considered as an error is if the child writes to its standard error. This is simulated by the following snippet.

```
catch {
    exec [info nameofexecutable] << {
        puts "This is standard output"
        puts stderr "This is standard error"
```

```
        puts "This is standard output again"
        exit 0
    }
} result ropts
→ 1
```

Again, the `catch` command result indicates an error exception even though the child process exited with an exit code of 0. This is because it wrote to its standard error. The error code however shows up as NONE.

```
puts [dict get $ropts -errorcode] → NONE
```

The result of the exception includes the standard output of the child followed by its standard error content. Note the latter always appears at the end no matter the order in which `puts` statements were executed.

```
% puts $result
→ This is standard output
  This is standard output again
  This is standard error
```

If the `-ignorestderr` option is specified, `exec` does not treat any output to standard error as an error condition.

```
% catch {
    exec -ignorestderr -- [info nameofexecutable] << {
        puts "This is standard output"
        puts stderr "This is standard error"
    }
} result ropts
→ 0
% puts $result
→ This is standard output
```

As before, the result includes both standard output and standard error but now the command does not raise an error exception as evinced by the 0 result of the `catch` command above.

Another way to accomplish the same thing is to redirect the standard error using any of the error redirectors like 2> etc.

The following pseudocode template summarizes handling of all these various cases using the `try` command described in Section 11.4.3. Depending on the program being executed, the application can take appropriate action depending on whether the condition signifies a real error or not. Errors for which trap clauses are not specified will be automatically propagated.

```
try {
    set result [exec command parameters ...]
} trap NONE output {
    # Application exited with a 0 exit code but wrote to standard error.
    # The variable output will contain the standard output content
    # followed by the standard error content
    ...do whatever...
} trap CHILDSTATUS {- ropts} {
    # Child exited with a non-0 exit code
    # Retrieve the PID and exit code
    lassign [dict get $ropts -errorcode] -> pid exit_code
    ...do whatever...
} trap CHILDKILLED {- ropts} {
    # Child terminated abnormally
    # Retrieve the PID, signal and message
    lassign [dict get $ropts -errorcode] -> pid signal reason
    ...do whatever...
```

```
} trap CHILDSUSP {_ ropts} {
    # Child suspended
    # Retrieve the PID, signal and message
    lassign [dict get $ropts -errorcode] -> pid signal reason
    ...do whatever...
} trap POSIX {- ropts} {
    # Other errors like permissions, file not existing etc.
    # Retrieve POSIX error mnemonic and reason
    lassign [dict get $ropts -errorcode] -> posix_code reason
    ...do whatever...
}
```

16.1.5. Running background processes

Normally the exec command waits for all processes in the pipeline to terminate and returns as its result the standard output of the last process in the pipeline. However, if the last argument to exec is &, the command returns immediately running the processes in the background. The return value is a list containing the process identifier (PID) of each process in the created pipeline.

In the following example, the exec command returns immediately without waiting for the `netstat` and `findstr` processes to finish executing.

```
% exec netstat -an | findstr ESTABLISHED > connections.log &
→ 9300 11952
```

The list of PID's returned is in the order of processes in the pipeline.

Tcl does not provide any built-in commands dealing with process monitoring and management. However, the cross-platform `processman` module in Tcllib[1] as well as the Windows-specific `process` module in TWAPI[2] provide commands for checking for process existence, terminating processes and so on. Our example above could be modified as follows to let us know when the background processes were done without blocking the exec command itself.

```
package require processman
set pids [exec netstat -an | findstr ESTABLISHED > connections.log &]
::processman::onexit [lindex $pids 1] { puts "Background processing done" }
```

Note that the above assumes the Tcl event loop is running.

If no I/O redirection is in effect, the standard output and standard error of the last process in the pipeline will go to the application's standard output and error.

16.1.6. Limitations in exec

There are certain limitations in exec for some scenarios:

- It does not provide for an "interactive" bidirectional data exchange with the child process. For example, we cannot use it fire up another copy of our Tcl applications and feed it a command, get back the result and then repeat that sequence.
- There is no control over encodings, line translations etc. with respect to data written to and read from the child process.
- Its syntax makes it not just difficult but impossible to pass arguments that match certain character sequences such as those used for redirection.

[1] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html
[2] http://twapi.sf.net

- It has certain platform-specific limitations such as not being able to execute child process with elevated privileges on Windows.

The first two of these are addressed in the next section and a proposal and accompanying implementation to fix the third in the next release of Tcl is already in place as Tcl Improvement Proposal #424 available at http://www.tcl.tk/cgi-bin/tct/tip/424.html.

To overcome the last limitation, you will need the help of third party extensions. For example, the TWAPI[3] extension has commands that offer these capabilities for the Windows platform.

16.2. Channels for process pipelines: open

We introduced the open command in Chapter 9 where we used it to create an I/O channel to a file on disk. The command is in fact more general in that it can be used to create channels of different types. Here we examine its use for creating channels to process pipelines.

Use of open to create channels for process pipelines has several advantages over exec at the cost of some slight complexity:

- It allows for arbitrary sequences of bidirectional data exchange with the child process.
- The data exchange can be asynchronous, a capability we will describe in Section 17.1.
- The encodings used for the data exchange can be controlled by appropriately configuring the channel.
- Being a channel, we are are able to use all capabilities of Tcl channels including applying channel transforms as described in Section 17.2. For example, we could transparently compress the data we are piping into the process pipeline.

A channel to a process pipeline is opened using the same syntax we saw in Section 9.3.2.

```
open PATH ?ACCESS? ?PERMISSIONS?
```

The *ACCESS* and *PERMISSIONS* arguments are the same as described there for file channels. The *PATH* argument on the other hand must begin with the | character. The rest of the *PATH* argument is treated in the same fashion as the arguments to the exec command.

Used in this form, open returns a channel that may be used to write to the standard input of the first process in the pipeline or read from the standard output of the last process (assuming no redirections are in effect). The returned channel must as always be released with the close command when done.

If the channel to a process pipeline is in blocking mode, the close will not return until all processes in the pipeline have ended.

The operations permitted on the returned channel depend on the *ACCESS* argument as shown in Table 16.1. The descriptions in the table assume that no redirections are in effect. For example, if the output of the pipeline is redirected to a file, no data will be read from the channel. Some illustrative examples follow the table.

[3] http://twapi.sf.net

Table 16.1. Access mode for pipelines using `open`

Mode	Description
r, rb	The channel is opened as read-only in text and binary modes respectively. The standard input of the first process in the pipeline is taken from the standard input of the current process. The standard output of the last process in the pipeline can be read from the created channel.
r+, rb+, r+b, w+, wb+, w+b	The channel is opened for reading and writing in text and binary mode respectively. Any writes to the channel will be fed into the standard input of the first process in the pipeline. The standard output of the last process in the pipeline can be read from the created channel.
w, wb	The channel is opened only for writing in text and binary mode respectively. Any writes to the channel will be fed into the standard input of the first process in the pipeline. The channel cannot be read and any output from the last process in the pipeline will go to the current standard output unless any redirection is in place.

A read-only pipe

Our first example is a variation of one of the `exec` based ones we saw earlier. We list network connections using the `netstat` program and pipe the output to `findstr` to filter them. Since we never need to pass any data **to** the process pipeline, we can open the channel in read-only mode.

```
% set chan [open "|netstat -an | findstr ESTABLISHED" r]
→ file3a9b030
% while {[gets $chan line] >= 0} {
    puts $line
}
→   TCP    192.168.1.128:53143    40.100.136.18:443      ESTABLISHED
    TCP    192.168.1.128:53226    40.100.138.18:443      ESTABLISHED
...Additional lines omitted...
% close $chan
```

Having a channel in hand, we can process the output data a line at a time should we choose unlike for `exec` where we got all the data in one lump. For our example, this may not matter much since the output data is limited in size and easy enough to split into lines. But in the general case, where the data is either very large or an continuous stream (we will see an example of this later), `exec` is not a viable option.

A write-only pipe

Our second example involves a write-only pipe where we will write to the `gzip` program to compress data that we will generate incrementally.

```
% set chan [open "|gzip - > foo.zip" w] ❶
→ file7
% chan configure $chan -encoding utf-8 -translation lf
% puts $chan "Line 1"
% puts $chan "Line 2"
% close $chan
```

❶ Specifying `-` as the input file causes `gzip` to read data from its standard input

Several additional points are illustrated by this example:

- Redirection operators like > above can be used with `open` in the same fashion as with `exec`.
- We can call `chan configure` to set various options on the channel. In this case, because most compression programs expect binary data, we configure the channel to transform our Unicode strings to binary data as

described in Section 9.3.8. Without this, the channel would use system encoding which may or may not be able to handle all characters.

- We do not need to collect all data and feed it to gzip in one step as for exec. We can write it piecemeal as it is generated.
- We need to close the channel when done. Otherwise, not only will we have a resource leak with the channel handle, it will also cause gzip will hang around waiting for more data.

16.2.1. Running tclsh in a pipeline

It is sometimes useful in real world applications to "drive" tclsh in a pipeline for executing ancillary tasks, parallelizing computation and so on. We demonstrate such usage here.

Additionally, this will also serve as an example of

- Using a bi-directional pipe where the application both writes to and reads from the child process.
- Using the list command to construct the program and argument parameters.
- Additional channel configuration that must be set up for some applications.

If on Windows, the code below must be run from tclsh or some other Tcl console application, not from wish or a GUI based one as the latter do not have standard input/output.

Our opening command itself looks different from what we have seen earlier.

```
% set chan [open |[list [info nameofexecutable] -encoding utf-8] r+]
→ file3ab1570
```

The r+ argument opens the pipe for both reading and writing. The -encoding option to tclsh informs that it should expect UTF-8 encoded data in its standard input. The list command is used to correctly form the arguments to the open command.

Consider if we had written the command as

```
set chan [open "|[info nameofexecutable] -encoding utf-8" r+]
```

Now, if our tclsh was installed in a directory with spaces in its path, say under Program Files on Windows, Tcl will attempt to execute the following after command substitution.

```
set chan [open "|C:/Program Files/Tcl/bin/tclsh.exe -encoding utf-8" r+]
```

This will fail as space in the path will cause the open command to treat C:/Program as the name of the program to run. Although some combination of quoting and escapes would also work, it is generally simpler and less error prone to use list to correctly form the arguments when command or variable substitution are involved as in our example.

As an aside, we could have placed the first argument in quotes

```
set chan [open "|[list [info nameofexecutable] -encoding utf-8]" r+]
```

but that is not necessary as long there is no whitespace between the | and [characters.

The next thing you need to be aware of relates to buffering in the channel (see Section 9.3.5.1). By default, the created channel is fully buffered as we can verify:

```
chan configure $chan -buffering → full
```

Fully buffered channels offer highest performance but are not convenient in scenarios like ours where commands we write to our child tclsh process need to be immediately sent across. We could do this by explicitly calling chan flush after every write but it is easier to just set the channel to be line buffered instead. At the same time we also set our channel to use UTF-8 encoding to match the encoding our child tclsh is expecting.

```
% chan configure $chan -buffering line -encoding utf-8
```

The child tclsh process is now running and since it was not passed a script file on the command line, it will loop reading commands from its standard input, i.e. the pipe, and executing them.

First, let us configure its buffering for the same reason listed above for our side of pipe. We write the appropriate command to the pipe.

```
% puts $chan {chan configure stdout -buffering line}
```

The child tclsh will then read the command and set its stdout channel configuration appropriately.

At this point, we list two important distinctions between running tclsh in a pipeline versus running it interactively.

- In interactive mode, tclsh will write a prompt to the standard output when it is ready for the next command. It does not do this when reading from a pipe.
- Secondly, unlike in interactive mode, the result of the evaluated command is **not** written to standard output.

We can check whether the child tclsh thinks it is in interactive mode by asking it to print the value of the tcl_interactive global variable in the child process. This value can then be read back from our end of the pipe.

```
% puts $chan {puts $tcl_interactive}
% gets $chan
→ 0
```

We now know that the child is non-interactive mode. We do not therefore have to worry about dealing with the tclsh prompt characters being read from the pipe and having to separate them from the actual data.

At the same time, we need to be aware that in non-interactive mode tclsh will not print the result of evaluated commands to its standard output. So if, instead of forcing an explicit write using puts as above, we had invoked the following commands

```
puts $chan {set tcl_interactive}
gets $chan
```

our shell would have appeared to hang. The child tclsh does not write the result of the set to its standard output. Consequently, our gets command would sit there waiting forever (since we have not discussed non-blocking I/O as yet).

If for some reason, you want the child tclsh to output prompts and display the result of evaluated commands, you can set the value of the tcl_interactive variable to 1 (obviously in the **child** tclsh, not in our parent shell).

When we are done with the child tclsh, we can either send it an exit command or simply close our end of the pipe which will cause it to exit. Here we will explicitly ask it to exit.

```
% puts $chan {exit}
```

Now our attempt to read from the pipe returns an empty string and eof indicates an end of file on the channel at which point we can close it.

```
gets $chan  → (empty)
eof $chan   → 1
close $chan → (empty)
```

16.2.2. Pipeline process ids: pid

The process identifiers of all processes present in a pipeline associated with a channel can be retrieved with the pid command.

```
pid CHANNEL
```

The command returns the list of PIDs for the processes in the pipeline and an empty list if the channel specified is not a process pipeline.

```
% set chan [open "|netstat -an | findstr ESTABLISHED" r]
→ file4154cd0
% pid $chan
→ 7896 9392
```

16.2.3. Error handling in pipelines

If any of the processes running in the pipeline signal an error either with the exit status or by writing to their standard error as described in Section 16.1.4, the close on the pipeline channel will throw an exception.

```
% set chan [open |[list [info nameofexecutable] -encoding utf-8] r+]
→ file4154cd0
% chan configure $chan -buffering line -encoding utf-8
% puts $chan {exit 2} ❶
% close $chan
Ø child process exited abnormally
% puts $::errorCode
→ CHILDSTATUS 11792 2
```

❶ Force child to exit with an error code

As displayed above, the error code corresponds to the child exiting with a non-0 status.

16.3. Standalone pipes: chan pipe

In the previous sections, we have seen various ways of creating and communicating with child processes through pipes and redirections. However there are some scenarios that are at best awkward to program for using these means.

One of these involves reading the standard output and standard error of a child process as separate data streams. This is not possible with the redirection forms we have seen where we can at most redirect the standard error into standard output and then somehow separate out the two, which may or may not be possible. Alternatively, standard error may be redirected to a file but the semantics of a file are very different when it comes to EOF and other aspects.

The chan pipe command provides a solution.

```
chan pipe
```

The command creates an operating system pipe and returns a list containing two channels, the first for reading from the pipe, and the second for writing to it. Any data written to the second channel will be read from the first.

Let us interactively observe how the channels work. First we create the pipe and assign the read and write channels to rchan and wchan respectively.

```
% lassign [chan pipe] rchan wchan
```

We will be sending simple text across so we configure them to be line buffered so writing a line will immediately flush the pipe making the line available on the read side.

```
% chan configure $rchan -buffering line
% chan configure $wchan -buffering line
```

Now we write data into the pipe via the write-side channel. As expected, it can be read from the read-side channel.

```
% puts $wchan "Testing 1 2 3..."
% gets $rchan
→ Testing 1 2 3...
```

Trying to write to the read-side channel is an error as is reading from the write-side channel.

```
% puts $rchan "Fail"
Ø channel "file41545d0" wasn't opened for writing
% gets $wchan
Ø channel "file3ba88d0" wasn't opened for reading
```

Closing the write side channel will be detected as end-of-file on the read side.

```
% close $wchan
% gets $rchan ❶
% chan eof $rchan
→ 1
% close $rchan
```

❶ Empty string returned because of EOF

Separating standard output and error

Having seen the basic working of a pipe, let us see how it might serve our purpose. As in previous examples, we will spawn a second copy of our Tcl shell as the child process.

Again, we start by creating a pipe.

```
% lassign [chan pipe] rchan wchan
```

Then as before, we use redirection into a channel to have the child process write its standard error to this pipe (see Section 16.1.3.2).

```
% set chan [open |[list [info nameofexecutable] 2>@ $wchan << {
    puts "This is standard output"
    puts stderr "This is standard error"
}]]
→ file41563d0
% close $wchan
```

The channel returned by open will be attached to the child's standard output. The write side of our pipe will be attached to the child's standard error through the 2>@ redirection.

Notice we then immediately close the write-side in our application. This is important because otherwise the write-side pipe file descriptor will have two references - one held by us and the other by the child process. When the child closes its end or exits, we will not see EOF on our read-side channel because the write-side will still be open due to the reference we hold. By closing it, we ensure that we see the EOF when the child closes the write side of the pipe.

Now we can independently read the child standard output and error.

```
gets $chan   → This is standard output
gets $rchan  → This is standard error
close $chan  → (empty)
close $rchan → (empty)
```

Filter programs

Another use of channel pipes arises in conjunction with programs that act as "filters". They read in their standard input, apply some transform to it and write the result to their standard output. We saw examples earlier in this chapter that used `findstr` as a filter.

However, the technique demonstrated there would not work with some filter programs because they do not write their output until they see an EOF on their standard input. For such programs, we are in something of a Catch-22. It will not return any data until sees an EOF. For that to happen, we have to close the the channel returned by open. But then once we close the channel, we cannot read back the data the program writes!

Standalone pipes provide a solution for this as well as demonstrated in the script below. We use the `filter_upcase.tcl` script to simulate these filters. It will keep reading standard input until EOF and then write the upper case form of the input data to its standard output.

```
# filter_upcase.tcl
fconfigure stdin -buffering line
fconfigure stdout -buffering line
set result ""
while {[gets stdin line] >= 0} {
    append result [string toupper $line]\n
}
puts stdout $result
exit 0
```

Now we need to call this script as a child process, feed it some data and read back the results. Although we could use open for our demo as in our previous example, we will instead use exec just to illustrate its use with standalone pipes. Unlike for open which returns a channel, here we will need to create **two** pipes; one to send data to the child as its standard input, and one to read data from the child as its standard output.

```
% lassign [chan pipe] childread mywrite ❶
% lassign [chan pipe] myread childwrite ❷
```

❶ Child's standard input
❷ Child's standard output

We then fire up the child with appropriate redirections. We also close the child's end of the channels for reasons explained in the previous example.

```
% exec [info nameofexecutable] scripts/filter_upcase.tcl <@ $childread >@ $childwrite &
→ 1640
% close $childread
% close $childwrite
```

We now write to the channel connected to the child's standard input.

```
% puts $mywrite "This is a test"
% puts $mywrite "This is only a test"
```

Then we close our write-side pipe so that the child will see EOF on its standard input and know it can stop reading and write out the transformed data.

```
% close $mywrite
```

The transformed data now appears on our side of the child's standard output channel.

```
% read $myread
→ THIS IS A TEST
  THIS IS ONLY A TEST

% close $myread
```

Now the dirty little secret is that we only used this example as a means of demonstrating pipe usage. Our simple example could have been done more easily via the method discussed in the next section.

16.4. Half-closing of channels

We described in the previous section one method of running filter programs that wait for EOF on standard input before writing to standard output. The `close` command provides an alternate, possibly simpler, means for working with such program.

We described the basic operation of the `close` command in Section 9.3.3. As described there, the command can take an optional second argument, `read` or `write`, that specifies that the channel is to be closed only for that direction of data transfer.

```
close CHANNEL ?DIRECTION?
chan close CHANNEL ?DIRECTION?
```

We can make use of this capability to close the standard input the child process. The code below makes use of this technique to run our example child process from the previous section.

We open a pipe to the child process for read and write.

```
% set chan [open |[list [info nameofexecutable] scripts/filter_upcase.tcl] r+]
→ file41545d0
```

We then write our data to the pipe and then close **only the write side** of the pipe which then results in the standard input of the child seeing an EOF.

```
% puts $chan "This is a test"
% puts $chan "This is only a test"
% chan close $chan write
```

The child will then write to its standard output and since the read side of the channel is still open, we can read its output.

```
% read $chan
→ THIS IS A TEST
  THIS IS ONLY A TEST

% close $chan
```

This technique is clearly simpler, but less general, than the one described in Section 16.3.

Half-closing of channels is not limited to pipes. You can also use it for network sockets where closing the client socket for writes would indicate to the server that no more data is forthcoming from the client while leaving the connection open in the reverse direction.

16.5. Passing environment to child processes

The process environment variables in a child process are inherited from its parent. If you want to pass a different environment to the child, you need to save the env array, modify it as per what is desired for the child, start the child process and then restore env from the saved copy.

This is further complicated in a multi-threaded environment since the env global values are reflected across all threads in a process. You would need to employ one of the synchronization mechanisms described in Section 22.11 to coordinate modification of env.

One possible work-around is to run the child process through an intermediary that allows setting of the environment such as `/usr/bin/env` on Unix or the command shell on Windows.

16.6. Interprocess communications

Exchanging data between processes through pipes is the simplest and most convenient mechanism available. However, for more structured communication, more sophisticated alternatives are needed, like COM on Windows or D-Bus on Unix platforms. Although not built into Tcl and described here, these are available as extensions. See Appendix A for a listing.

16.7. Chapter summary

In this chapter, we described Tcl's support for locating and running other applications, and exchanging data through the standard input and output mechanisms. One of the main features that gave Unix its flexibility and power was the ability to transform data through pipelines of processes that perform specialized tasks. The exec and open commands bring the same flexibility to Tcl.

We have now covered basic input and output, both with respect to files in Chapter 9 and to processes in this chapter. We will now move on in the next chapter to the more advanced I/O facilities available in Tcl.

17

Advanced I/O

In Chapter 9 we introduced the channel abstraction and basic operations in Tcl in the context of reading and writing files. We also saw the use of channels for I/O with child process pipelines in Section 16.2. We now expand on this topic and delve into more advanced topics related to I/O in Tcl:

- Asynchronous input and output operations
- Transforming data during I/O
- Defining new channel types using Tcl's reflected channel abstraction

17.1. Asynchronous I/O

All the I/O operations we have seen so far with files as well as process pipelines have involved *blocking* I/O where the invoked command, `gets`, `read` etc., will not return until the I/O operation is completed or the channel end-of-file is reached. For channels backed by files, this behaviour is acceptable for the most part. For channels where the incoming data is intermittent and not always immediately available, this mode of operation is undesirable since the process may be blocked from doing any useful work for long intervals.

For example,

- The application may fire off a process pipeline to carry out a long computation. If it is blocked while reading from the pipeline, it cannot respond to the user or do any other tasks until the computation completes.
- A network server waiting for data from a client will be blocked from communicating with other clients and effectively can only service one at a time.
- Serial port communication is slow to begin with and if there is a human at a terminal on the other end, there is a lifetime between character arrivals. Unless the application is dedicated to responding to that device, it is not feasible to block while waiting for data to show up on the port.

These are the types of situations for which *non-blocking* I/O is designed. When a channel is in non-blocking mode, the command will return immediately even if the I/O operation cannot be completed. The application can then attempt to try the operation at a later point.

Using threads for blocking I/O

Another solution, as in the original releases of the Java language, involves using threads but that not only adds unnecessary complexity but also has scalability issues. Although Tcl also provides threading capabilities at the script level as we will see in Chapter 22, threads are best used for computations that mostly independent or in cases where the underlying API itself does not support a non-blocking mode of operation, as in some database drivers.

There is one issue that arises with non-blocking I/O and that is with regard to when the application retries the I/O operation. Polling continuously is no better than blocking and polling at intervals is neither efficient nor responsive. It would be nice if there were a mechanism whereby the application is notified when a channel is ready for the required operation. As always, Tcl doesn't disappoint! Channels can deliver such notification events through the same machinery we described in Chapter 15.

Non-blocking channels and channel event notifications are almost always used in combination to perform *asynchronous I/O*. Channels are set to non-blocking mode and any I/O operations take place only in response to channel events.

For ease of exposition however, we will start off with a description of non-blocking I/O without involving channel events.

17.1.1. Non-blocking I/O

A channel's blocking mode is controlled with the `-blocking` option to the same `fconfigure` or `chan configure` commands that we saw in Section 9.3.4 for setting other channel configuration options.

```
chan configure CHAN -blocking BOOLEAN
fconfigure CHAN -blocking BOOLEAN
```

The channel *CHAN* is set to blocking mode if *BOOLEAN* is a boolean true value and to non-blocking otherwise. Note that setting the blocking mode affects both read and write operations. It is not possible to set them independently.

17.1.1.1. Non-blocking reads

A read operation on a channel may block because the input data buffers and device are empty or contain less data than what was requested. The effect of this condition on non-blocking read operations depends on the specific command invoking the operation.

17.1.1.1.1. Reading lines in non-blocking mode: `chan gets`, `gets`

The `chan gets` and equivalent `gets` commands retrieve a single line from a channel. If no complete line is available and blocking mode is in effect, the commands will wait unless end of file is reached.

```
chan gets CHANNEL ?VARNAME?
gets CHANNEL ?VARNAME?
```

In non-blocking mode, **if a complete line is available**, the command behaves the same as in blocking mode:

- If *VARNAME* is specified, the line is stored in the variable of that name and the command returns the number of characters in the line. Remember that end of line characters are neither stored nor included in the character count. Thus an empty input line — consecutive newlines with no other intervening characters — will result in the empty string being stored in *VARNAME* and the command returning 0.
- If *VARNAME* is not specified, the command returns the line (possibly as the empty string) as its result.

In the case where **a complete line is not available**, the command differs from blocking mode operation:

- If *VARNAME* is specified, the command returns `-1` as its result. The variable is not affected.
- If *VARNAME* is not specified, the command returns an empty string as its result.

When is a complete line not available? It could be for one of two reasons:

- The channel is at end of file.
- The channel is not at end of file but the received data does not (yet) contain a line terminator that would form a complete line.

The following commands can be used to distinguish the two cases:

- The `chan eof`, or equivalent `eof`, command returns 1 if the channel is at end of file and 0 otherwise.
- The `chan blocked`, or equivalent `fblocked`, command returns 1 if the channel is not at end of file but is blocked because the incoming data does not form a complete line. Otherwise it returns 0.

When an attempted read fails because a complete line is not available, it can be useful to see how many characters are present in the channel input buffer. That way, if it exceeds some maximum permitted line length, in a network

protocol for instance, we can take appropriate action such as terminating the connection. The `chan pending` command will return the number of buffered bytes (**not** characters) in a channel.

```
chan pending DIRECTION CHANNEL
```

Here *DIRECTION* may be `input` or `output` depending on whether we are interested in the input or output side of the channel. We'll see an example use below.

OK, enough of the theory. Let us experiment with non-blocking behaviour using a pipe channel as described in Section 16.3. You will recall that data written to the output end of the pipe can be read from the input end. We create the pipe and assign the input and output channels.

```
lassign [chan pipe] in out → (empty)
```

We put the output side into line buffering mode so it will flush automatically any time it sees a newline character.

```
chan configure $out -buffering line → (empty)
```

We are now ready to communicate over the pipe channel. First we will put the input end into non-blocking mode and attempt to read a line. Since we have not written to the pipe as yet, the input buffer will be empty.

```
chan configure $in -blocking 0 → (empty)
gets $in line                  → -1
```

The returned character count is `-1` as expected. Attempting a read using the second form of `gets` gets back an empty string.

```
chan gets $in → (empty) ❶
```

❶ Remember that `gets` and `chan gets` are equivalent. We can use either.

We can confirm that the channel is not at end of file and that there is no line in the input buffer.

```
chan eof $in     → 0
chan blocked $in → 1
```

Now let us write two **empty** lines to the pipe.

```
chan puts $out "" → (empty)
chan puts $out "" → (empty)
```

We use the first form of `gets` to attempt to read a line.

```
chan gets $in line → 0
```

The return value of 0 indicates that an empty line was read. Let us read the second empty line without passing the variable argument.

```
set line [gets $in] → (empty)
string length $line → 0
```

Again we get back an empty line. How do we know whether it is a "real" empty line, or an incomplete one, or end of file? We use `chan eof` / `eof` and `chan blocked` / `fblocked` to find out.

```
eof $in       → 0
fblocked $in → 0
```

Both return 0 so we know it was indeed a empty line sent by the "remote" end of the channel.

Let us then write data to the channel without an end of line character. Notice we need to do an explicit flush to make sure the data is sent to read side of the channel.

```
chan puts -nonewline $out "A few words" → (empty)
chan flush $out                         → (empty)
chan gets $in line                      → -1
chan gets $in                           → (empty)
chan eof $in                            → 0
chan blocked $in                        → 1
```

We see that even though there is data in the input buffer the two forms of the `chan gets` command returned `-1` and an empty string respectively, as the buffered data did not form a complete line. The `chan eof` and `chan blocked` calls confirmed as much.

Do not confuse the buffering mode we set with `-buffering line` with line completion handling. The `-buffering` option only controls when data is flushed from the output buffers.

We can in fact confirm that there are characters in the input buffer with `chan pending`.

```
chan pending input $in → 11
```

The command tells us there are 11 bytes pending in the input.

Let us now examine the end of file condition. We close the output side of the pipe and attempt another read.

```
close $out → (empty)
gets $in   → A few words
```

Notice that on end of file, the buffered input content is delivered even though no newline characters were present.

Subsequent reads then indicate end of file which we can check with `eof`.

```
chan gets $in line → -1
gets $in           → (empty)
eof $in            → 1
fblocked $in       → 0
close $in          → (empty)
```

Before moving on to its sibling — the `read` command — you may want to play further with the various scenarios and how they affect the results returned by `gets`, `eof` and `fblocked`.

17.1.1.1.2. Reading characters in non-blocking mode: `chan read`, `read`

We described the basic operation of the `chan read`, and the equivalent `read`, commands in Section 9.3.6.2. The commands have two forms, one where the number of characters to read is not specified.

```
chan read ?-nonewline? CHANNEL
read ?-nonewline? CHANNEL
```

In the other, the caller explicitly asks for a specific number of characters.

```
chan read CHANNEL NUMCHARS
read CHANNEL NUMCHARS
```

In blocking mode, the commands read a specified number of characters from a channel or all characters till end of file if no character count is specified. If the specified number of characters is not available, or end of file is not reached in the case of no count being specified, the `read` command will wait.

In non-blocking mode, both forms alter their behaviour to return whatever data is available even if less than requested. We will again use a pipe for demonstration. This time since we are writing a character stream, we will set the buffering mode to none to ensure the data is passed along right away.

```
lassign [chan pipe] in out          → (empty)
chan configure $out -buffering none → (empty)
chan configure $in -blocking 0      → (empty)
```

An attempt to read at this point returns empty strings. As before we can use `eof` and friends to check the cause.

```
chan read $in     → (empty)
chan read $in 1   → (empty)
chan eof $in      → 0
chan blocked $in → 1
```

Let us write a single character and attempt to read two.

```
puts -nonewline $out A → (empty)
chan read $in 2        → A
```

As you can see, the read returns a single character which is all that was available in the input buffer. The same holds true if we tried to read to end of file.

```
puts -nonewline $out B → (empty)
chan read $in          → B
eof $in                → 0
```

When closing the pipe, we take the opportunity to reiterate another point about end of file conditions. When we check for end of file after the pipe is closed, `eof` returns 0, **not** 1.

```
close $out → (empty)
eof $in    → 0
```

This is because `chan eof` / `eof` **only detect an end of file after a input command (`gets` or `read`) fails** because of an end of file condition. To prove that point,

```
read $in  → (empty) ❶
eof $in   → 1
close $in → (empty)
```

❶ Empty string returned due to EOF

17.1.1.2. Non-blocking writes: chan puts, puts

The non-blocking behaviour on the output side using `chan puts` or `puts` is a lot simpler than on the input side. In blocking mode when a write is made to a channel, if its internal buffers (if in use) are full, an attempt is made to flush them to the underlying device. If the device cannot accept the data, the `puts` command will block until such a time that the device is ready to accept more data.

In non-blocking mode, Tcl will accept the data and store it in its internal buffer, growing it as necessary. When the device is ready to accept more data, this internal buffer is flushed to the device behind the scenes. However, **this requires that the Tcl event loop be running**. This is not generally a problem because non-blocking I/O is almost always used in conjunction with channel events to implement asynchronous I/O.

In non-blocking mode, even the `flush` / `chan flush` commands do not (can not) flush the internal buffers to the device if it is not ready. They will be flushed in the background, potentially after the command returns. To force data to be written out immediately, the channel must be placed in blocking mode, flushed and then reverted to non-blocking mode. Of course, this means the application is blocked until the flush completes.

Although Tcl will accept any amount of data on output in non-blocking mode, it is advisable to use channel event notifications to only write when the channel is ready to accept data. Otherwise, there is a danger of the output buffers growing unacceptably large causing memory pressure.

And that leads us to a discussion of event-driven I/O.

17.1.2. Event driven I/O: chan event, filevent

As we discussed earlier, efficient asynchronous I/O requires some means for an application to be notified when a channel is ready to receive data on output or has data available for reading. The channel subsystem in Tcl provides these notifications by generating events when channels are ready for input or output. An application can then register callbacks to be invoked on the occurence of these events.

Channel events are tied into the same eventing infrastructure we described in Chapter 15 and therefore require the event loop to be running. On every iteration of the event loop, each channel driver is given an opportunity to add events to the event queue. If any channel event handlers are registered for a channel and it has notifications pending, the channel driver will enqueue an entry to event queue with the event details and handler information. When the event loop processes the event queue, it will invoke any queued handlers just as it does for timer events.

The handlers for channel events are registered with the `chan event`, or equivalent `fileevent` command.

```
chan event CHANNEL EVENT ?HANDLER?
fileevent CHANNEL EVENT ?HANDLER?
```

The *HANDLER* argument is the callback script that should be invoked in reaction to the notification event specified by *EVENT*. If there is an existing handler already registered, it is replaced. If *HANDLER* is not specified, the command returns the script currently registered for that channel for the *EVENT* event or an empty string if none is registered. If *HANDLER* is passed as the empty string, the currently registered handler, if any, is unregistered.

The *EVENT* argument indicates the type of event and must be either `readable` or `writable`.

A `readable` event is generated under either of two conditions:

- Data is available to be read from the channel.
- The channel is at end of file.

As we will see in our examples, the handler script generally uses the methods described in the previous sections to read data or detect end of file on the channel.

A writeable event is generated when the channel is ready to accept data from the application. A special case of this condition occurs on asynchronous network socket connections when the connection set up is completed and is open for transmitting data. We will look at this case further in Chapter 18.

As a first example of event driven I/O, let us revisit our pipe examples from the previous section except that instead of blindly reading from the pipe, we will only attempt to read when we are notified that data is available. The example also highlights some points you need to be aware of when programming asynchronous I/O so we will go through it slowly.

```
lassign [chan pipe] in out                          → (empty)
chan configure $out -buffering none -blocking 0 → (empty)
chan configure $in -blocking 0                      → (empty)
```

We have turned off buffering for the output channel so that data is immediately written to the pipe for reasons we will see later.

Next, we write a procedure that implements the event handler.

```
proc read_handler {chan} {
    set status [catch {gets $chan line} nchars]
    if {$status == 0 && $nchars >= 0} {
        puts "Received: $line"
        return
    }
    if {$status || [chan eof $chan]} {
        puts "All done!"
        chan event $chan readable {}
        set ::exit_flag 1
        return
    }
    puts "Incomplete line"
}
```

Our procedure will be invoked for every readable event on the channel of interest. It starts off with attempting to read a line from the channel. We use the catch command to trap any possible errors that the gets command might raise. This might happen, for example, if the remote end aborts a network connection.

Then, if there were no errors and the command read a line successfully (nchars is not negative), we print the line and return.

Otherwise, if either an error occured on the channel read or if the channel is at end of file, we remove the read handler from the channel, set an exit flag and return. Note that use of exit_flag is because we will be using vwait to run the event loop for our little example. In a real application which is already running the event loop, this line would not be necessary.

When an end of file is seen on a channel, it is crucial to either remove the read handler from the channel as we have done, or to close the channel in the handler itself before returning. Otherwise, the channel will continuously raise readable events because the channel is at end of file.

When none of the previous conditions are satisfied, the procedure falls through to the end to print the No data message. We will see in a bit the conditions under which this can happen.

Now we register our read handler for the input channel. Notice that we have to explicitly pass in the channel as an argument to the handler since the channel subsystem does not itself pass in this information. Just for kicks, we also confirm that it is registered.

```
chan event $in readable [list read_handler $in] → (empty)
chan event $in readable                          → read_handler file3b4bef0
```

Having set up the read handlers on the input channel, it is time to write to the pipe and see what happens. To simulate data arriving intermittently as from a remote network, we will split up the writes into partial writes with intervening delays, all scheduled through the event loop with the `after` command. (This is also the reason why we turned off buffering for the output channel earlier.)

```
after 50 [list puts $out "Hello World!"]          → after#38
after 100 [list puts -nonewline $out "Goodbye "] → after#39 ❶
after 150 [list puts $out "World!"]               → after#40
after 200 [list close $out]                       → after#41
```

❶ Note incomplete line

Finally, we get the event loop rolling with the `vwait` command. In a real application that expected to do asynchronous I/O the event loop would already be running but that is not true for our interactive shell. So we have to explicitly start it.

```
% vwait ::exit_flag
→ Received: Hello World!
  Incomplete line
  Received: Goodbye World!
  All done!
% close $in
```

Let us now examine the resulting output.

- When the first string written to the channel is received on the input side, our read handler is invoked as there is data available on the channel. Since an entire line is available, the `gets` call succeeds and the line is printed out.
- The second string written to the channel is an incomplete line. When the data arrives on the input side, our read handler is again invoked. This time the `gets` returns `-1` because a complete line is not available in the input buffer. The first `if` condition fails. Moreover, since the end of file is not reached, the second `if` condition also fails, and control passed to the bottom of the procedure resulting in `Incomplete line` being printed.
- The third write results in another invocation of the read handler. This time a complete line is available and printed.
- Finally the sending side closes its end of the pipe. The resulting end of file on the input side also triggers the read handler. This time the `gets` returns `-1` and `eof` indicates end of file. Consequently, the second `if` block is executed. **Here we are careful to remove the read handler from the channel otherwise we would be continuously invoked.** We could have closed the channel instead but we leave that for the main line code.
- Because the read handler set the `exit_flag`, the `vwait` command terminates the event loop and returns. We go on to close the input channel.

In our example, we used `gets` to read the channel a line at a time. We could also have used the `chan read` / `read` commands as well keeping in mind the differences with respect to `gets` that we described in Section 17.1.1.

The above example dealt with read handlers. We will not show an example of a write handler here as we will discuss further examples in Chapter 18.

One final note about channel event handlers. If the event handler raises an uncaught exception, Tcl will unregister it from the channel.

17.1.3. Closing non-blocking channels

There are a couple of considerations to keep in mind with regards to closing channels that are in non-blocking mode.

- The `close` command on a non-blocking channel returns immediately before any buffered data is written out. The data is then flushed in the background. This requires the event loop to continue to be active. Moreover, the

application should not assume the data is will be available on the underlying device when the `close` command returns.

- Any open non-blocking channels must be switched to blocking mode before exiting the process. Otherwise any buffered data may not be written out before the process exits.

17.1.4. An interactive command line

When you run `tclsh` without any arguments, it enters an interactive read-eval-print-loop (REPL) where it executes Tcl entered on the command line. This default REPL uses blocking I/O and does not have the event loop active. In order to have any event loop based functionality, such as asynchronous I/O, you need to enter the event loop through a command such as `vwait`. However, you then lose the ability to enter commands at the command line.

Another common situation that arises is in an event loop based application such as a network server. The script implementing the server usually has a `vwait` at the end that activates the event loop. Although these are generally "background" applications, it is nevertheless useful for them to be able to expose an interactive command line interface for purposes such as configuration, troubleshooting etc.

An event-driven REPL interface is useful in both the above scenarios. It collects interactive command input using non-blocking I/O permitting other event processing to proceed without interruption. A basic implementation is shown below.

```
namespace eval repl {}
proc repl::prompt {prompt} {
    puts -nonewline stderr $prompt
    flush stderr
}

proc repl::repl {} {
    variable command
    variable done

    set command ""
    prompt "% "
    fileevent stdin readable [namespace current]::repl_handler
    vwait [namespace current]::done
}
```

The `repl::repl` procedure is intended to be called from the main application to enter the event loop while also displaying a REPL for command input. It does some initialization, sets up a read handler on standard input and then enters the event loop.

The read handler `repl_handler`, shown below, is where the hard work is all done. It is invoked when a full line is available on the standard input or if the input channel is closed. In the latter case, it simply terminates the `vwait` loop after removing itself as the input handler. Otherwise, it appends the new line to any previously collected input. If the command is syntactically complete, it is executed in the global scope and the result printed. If the command is not complete, the secondary prompt is displayed.

```
proc repl::repl_handler {} {
    variable command
    if {[gets stdin line] < 0} {
        fileevent stdin readable {}
        set [namespace current]::done 1
        return
    }
    append command $line
    if {[info complete $command]} {
        fileevent stdin readable {} ❶
        set status [catch {uplevel #0 $command} result]
        fileevent stdin readable [namespace current]::repl_handler
```

```
        if {$result ne ""} {
            if {$status == 0} {
                puts $result
            } else {
                puts stderr $result
            }
        }
        set command ""
        prompt "% " ❷
    } else {
        append command \n
        prompt "> " ❸
    }
}
```

❶ Avoid nested calls
❷ Primary command prompt
❸ Secondary command prompt

One point to be noted above is that the read handler disables itself before calling `uplevel` and then restores itself afterward. This is a precautionary measure in case the script executed in the `uplevel` call itself recursively enters the event loop. In that case we do not want our read handler to be called if more input is available until the currently executing `uplevel` has finished execution.

The above implementation leaves out some details for pedagogic purposes. More complete implentations are available in the Tcler's Wiki[1], for example the one at http://wiki.tcl.tk/1968.

17.2. Channel transforms

In Section 9.3.8 we described the use of the `-encoding` option to transparently encode data written to a channel. This removes the burden from applications to remember to explicitly encode strings when writing to a file. This is very convenient when the output commands may be spread over multiple locations in the application.

Now consider an application writing to a log file where the data must be compressed to save disk space. Or perhaps encrypted in some form to hide passwords or other private details. Or both. Clearly the same kind of transparency afforded by the `-encoding` option would be very useful here as well.

Obviously Tcl cannot have built-in facilities for all the infinite ways that data might be transformed. So it provides something almost as good — a way for applications to implement their own *channel transforms* that can alter the data stream flowing through a channel. Moreover, multiple channel transforms can be applied to a channel in a stacked fashion so the output of one transform is fed into the next. Thus we can write a compression transform and an encryption transform to meet our needs and even combine them to implement a compress and encrypt "combo" channel.

17.2.1. Channel transform basic operation

Figure 17.1 shows data flow in a channel when the command `puts` is invoked with encoding set to UTF-8 and line endings to CRLF. The flow on the left side has no channel transforms applied. The Tcl channel top layer then encodes the input data, translating newlines to CR-LF pairs and emits a byte stream to the underlying device which may be a file, a network socket etc.

The right side of the figure shows the flow with two transforms pushed onto the channel, first the `base64` transform that we assume converts binary data into base64 encoding and a second transform which compresses[2] the data using the DEFLATE algorithm we discussed in Section 4.16. This is not necessarily a sensible combination of transforms but ... whatever. It serves our illustrative purpose.

[1] http://wiki.tcl.tk
[2] Short strings like in our example will actually land up being longer

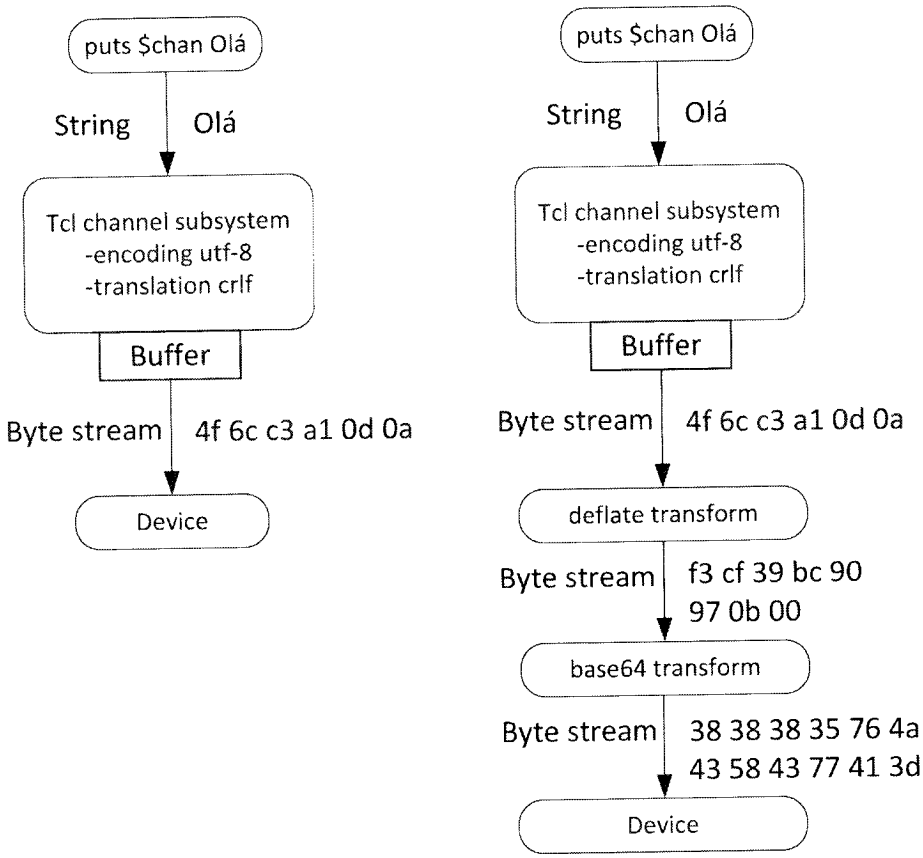

Figure 17.1. Basic channel operation

As shown in the diagram,

- The top layer of the Tcl channel subsystem produces a byte stream (or a binary string in Tcl parlance) and this is what the channel transforms see. There are no "characters" at this level so if the transform wants to do character based operations like upper-casing all text, it gets trickier than you might think. We will revisit this issue later.
- When multiple channels are pushed on to a channel, the later ones are placed on top of earlier ones. Hence the term "stacked" transforms.
- The channel subsystem calls each transform in turn, passing it the data returned by the previous transform and using the data returned by the transform as the input to the next. The data produced by the bottom most transform is written to the device.
- Although not depicted in the figure, a transform is free to return an empty value in which case the transform below is not called. An example of this behaviour would be exhibited by block oriented encryption algorithms which operate on fixed length sequences of bytes. In general, a channel transform may convert and pass on all, some or none of the data passed to it. The rest can be buffered internally for later processing.

The above discussion described operations on the channel output path. For the input path, the operations are very similar, except in reverse order, and we will not describe them separately.

17.2.2. Implementing channel transforms

The Tcl channel subsystem expects channel transforms to be implemented in the form of a command prefix that will be invoked with arguments such as `read` that are subcommands indicating the operation to carry out. Channel transforms may be implemented as namespace ensembles, TclOO object or even as a simple procedure whose first argument is treated as the operation to perform.

The subcommands that a channel transform must implement fall into three categories:

- Those that must be implemented by all transforms
- Those that must be implemented for transforms that affect read operations
- Those that must be implemented for transforms that affect write operations

The complete list of subcommands is shown in Table 17.1.

Table 17.1. Channel transform subcommands

Subcommand	Direction	Description
`initialize` *HANDLE MODE*	Both	Called to initialize the transform.
`finalize` *HANDLE*	Both	The last call on the transformer to permit it to clean up any allocated resources.
`clear` *HANDLE*	Both	Called to have the transformer clear internal buffers or state. This command is optional.
`drain` *HANDLE*	Read channels	Called to force the transform to return any internally buffered data to the channel subsystem so it can be passed to the higher layers. This subcommand is optional.
`limit?` *HANDLE*	Read channels	This subcommand is a way for the transform to tell the Tcl channel subsystem to limit the number of "read-ahead" bytes. This command is optional.
`read` *HANDLE BUFFER*	Read channels	Called to pass data from the device or another transformer below this transform.
`flush` *HANDLE*	Write channels	Called when internally buffered data in the transformer must be passed to the device or transformer below. This command is optional.
`write` *HANDLE BUFFER*	Write channels	Called to pass data into the transformer from the layer above.

Note that a channel transform need not be applicable to both input and output sides of a channel. For input-only and output-only transforms, the `write` and `read` subcommands respectively, need not be implemented.

Let us examine the implementation of these subcommands in detail by working through an example. We will implement a channel transformation for encrypting data using the RC4 cipher.

The use of RC4 in practice is not recommended because of vulnerabilities in the algorithm itself. We use it here because of the simplicity of the RC4 interface allows us to focus on the channel abstraction itself.

The RC4 implementation we use is from the `rc4` package of Tcllib[3].

```
package require rc4
```

[3] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

We will use Tcl's object oriented features instead of a namespace for implementing our transform as it makes it slightly easier to keep a context for keying material. We start by defining a class that implements the transform wherein the subcommands shown in Table 17.1 are methods of the class. For ease of explanation, we will construct the class piecemeal.

```
oo::class create RC4Transform {
    variable RC4Read
    variable RC4Write
    constructor {key} {
        set RC4Read [rc4::RC4Init $key]
        set RC4Write [rc4::RC4Init $key]
    }
}
```

Since encryption requires keying material, we pass it as an argument to the constructor. The constructor passes this to the RC4 package to get back handles that can be used for encryption operations on output and decryption on input. We store these away into member variables.

17.2.2.1. Initializing channel transforms

Next we define the mandated `initialize` subcommand.

```
oo::define RC4Transform {
    method initialize {transform_handle mode} {
        return {initialize read write finalize}
    }
}
```

The first parameter of all subcommands of a channel transformation is a handle to the channel transform (**not** the owning channel). Implementation based on namespaces or procedure calls generally make use of this handle, called `transform_handle` in our sample code, to distinguish between different channel instances applied to different channels. With a TclOO based implementation like ours, the context is implicit in the object instance because a different instance is pushed onto each channel. Thus this handle is not of much use to our implementation and it is ignored in all methods.

The second parameter to `initialize` is a list of one or two elements, from the strings `read` and `write`, indicating whether the attached channel is readable and writable. Our transformation does not care so we will ignore this parameter as well in our method.

The return value from `initialize` should be the list of subcommands supported by the transformation.

17.2.2.2. Finalizing channel transforms

The other mandated command is `finalize` which is called by the Tcl I/O system when the transformation is popped from the channel.

```
oo::define RC4Transform {
    method finalize {transform_handle} {
        rc4::RC4Final $RC4Read
        rc4::RC4Final $RC4Write
        [self] destroy
    }
}
```

The transformation handle is the only parameter passed and we ignore it for the reasons explained above. The purpose of the call is to allow the transformation to release any resources. In our case, we release the resources associated with the RC4 handles. No further calls are made to the transformation by the I/O system once this call returns.

We also choose to have the transformation commit suicide by calling its `destroy` method. By doing so, we free the application from having to keep track of the transformation object itself and destroying it at some suitable point. We could have also chosen to leave it to the application to destroy the object after the channel transformation is popped from the channel. This is useful when the channel transformation may hold some additional computed value based on the data passing through the channel. For example, a channel transformation may compute a checksum on data passing through a channel and allow it to be retrieved after the channel is closed.

17.2.2.3. Transforming data

Finally, we come to the actual data transfer and implementation of the `read` and `write` subcommands.

```
oo::define RC4Transform {
    method read {transform_handle bytes} {
        return [rc4::RC4 $RC4Read $bytes]
    }
    method write {transform_handle bytes} {
        return [rc4::RC4 $RC4Write $bytes]
    }
}
```

The `read` and `write` commands both take two argments, the channel transformation handle which we ignore as before, and the binary data to be transformed. Our implementation is very simple as we leave it up to the RC4 package to do the hard work.

Here then is the complete channel transformation in a condensed form.

```
package require rc4
oo::class create RC4Transform {
    variable RC4Read
    variable RC4Write
    constructor {key} {
        set RC4Read  [rc4::RC4Init $key]
        set RC4Write [rc4::RC4Init $key]
    }
    method initialize {transform_handle mode} {
        return {initialize read write finalize}
    }
    method finalize {transform_handle} {
        rc4::RC4Final $RC4Read
        rc4::RC4Final $RC4Write
        [self] destroy
    }
    method read {transform_handle bytes} {
        return [rc4::RC4 $RC4Read $bytes]
    }
    method write {transform_handle bytes} {
        return [rc4::RC4 $RC4Write $bytes]
    }
}
→ ::RC4Transform
```

We will look at how we could use our RC4 transform in a bit but first we take a short detour to discuss some buffering issues that are not relevant to our example because of its simplicity but need to be managed in more complex transforms.

17.2.2.4. Buffering in channel transforms

Our example was greatly simplified by the fact that we had no need for internal buffering. In particular, data coming in to the transform in either direction was immediately passed on.

We now look at issues that arise when this is not possible due to the characteristics of the transform. Examples include

- Transforms that are block oriented such as the base64 encoding scheme or the DES encryption algorithm. These require the data to be transformed in multiples of fixed size blocks. Since there is in general no control over how the Tcl I/O system and other transforms pass in data, partial blocks have to be buffered internally and then transformed when further calls complete the block.
- Some transforms such as `zlib` compression are stream oriented but perform better when the data is compressed in larger chunks. These may choose to internally buffer data for performance reasons.
- Some transforms may deal with variable length blocks. These are the most complex to deal with. A common example of this kind of transform is anything dealing with characters as opposed to bytes. Consider a transform that changes the character case of all data passing through to upper case. Depending on the character encoding in effect on the channel, each character may be encoded as a variable number of bytes. The transform then has to parse the byte stream to recover and transform individual characters. To further complicate matters, a multibyte character may be split across multiple `read` (or `write`) calls.

At some point, such as when the channel is closed or the channel transform is popped from the channel, Tcl has to ask the transformation to flush its internal buffers and return their content. It does this by calling the `flush` and `drain` subcommands. The former is called for the output side of the channel and the latter for the input. Both commands take the transformation handle as their only parameter and are expected to complete transforming the data in their internal buffers and return the result.

The other situation concerning internal buffers involves the application calling `chan seek` on the channel. Since this changes the access pointer for the file (or any channel that supports seeks), internal buffers must be cleared. In this situation, the Tcl I/O subsystem calls the `clear` subcommand. In response, the channel transformation is expected to clear its internal buffers, **both** input and output, throwing away all stored data in the process.

Channel transformations that do internal buffering generally need to implement these three subcommands.

Applications should avoid calling `chan seek` or `seek` on channels which have transforms applied. By their very nature, transforms like compression are not amenable to these operations.

We will not present any examples of channel transformations that do internal buffering. You can look at the source code for the `tcl::transform::base64` package in Tcllib[4] for a real world example.

17.2.2.5. Limiting read-ahead

For reasons of performance, in normal operation the Tcl I/O system will read more bytes from a device than are actually requested by the application. This can be a problem for transforms that expect data streams in some bounded format. Consider using a transform on network channel to decompress a compressed chunk of an HTTP stream. If the Tcl I/O system reads ahead and passes data beyond the compressed portion, the transform will process the compressed part but has no way of returning the remaining portion back to the I/O system if the transform is then removed from the channel. (The application in this case knows how many bytes are compressed and pops the transform after reading that number.)

The channel transformation can implement the `limit?` command to control the amount of read-ahead done by Tcl on input. It takes the channel transformation handle as argument and should return the upper limit for the number of bytes that Tcl is allowed to read ahead, returning a negative integer to indicate no upper limit. It is called by Tcl before every `read` subcommand invocation. The subsequent `read` subcommand invocation will never pass in data of length more than the (non-negative) value returned by the `limit?` call.

17.2.3. Using channel transforms

Let us now look at how an application utilizes a channel transform using `chan push` and `chan pop`.

[4] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

```
chan push CHANNEL CMDPREFIX
chan pop CHANNEL
```

The chan push command places a channel transform at the top of the channel transform stack, i.e. just below the Tcl I/O buffering layer as shown in Figure 17.1. Successive chan push commands result in a stack of transformations with the last one pushed on top.

The chan pop command removes the topmost channel transform from the specified channel. It is not possible to remove any transform except the topmost.

The *CMDPREFIX* is a command prefix callback that in effect identifies the channel transform. When carrying out various I/O operations, Tcl invokes *CMDPREFIX* with the subcommand arguments as described in the previous section.

We will use our RC4Transform channel transformation across a pipe for demonstrative purposes. First we create an instance of the transformation and the pipe.

```
set xform [RC4Transform new secret] → ::oo::Obj296 ❶
lassign [chan pipe] in out          → (empty)
```

❶ Warning: it is a terrible idea to use an ASCII string for keying but this is just a demo

We will apply this transform only on the **output** side of a pipe. On the other end of the pipe, we will simply read the channel in binary mode without the transform applied. In effect, this will tell us what bytes are being written to the underlying operating system pipe.

```
chan push $out $xform                  → file4102620
chan configure $in -translation binary → (empty)
puts $out "Just a demo"                → (empty)
close $out                             → (empty)
```

Reading the other end of the pipe tells us the raw bytes written from the output side. We will dump the data in hex using our bin2hex helper procedure.

```
bin2hex [set encrypted [read $in]] → a7 43 a1 68 a2 c5 f6 c2 57 a6 d4 d1 f9
close $in                          → (empty)
```

As you can see the data was encrypted, something we can verify by decrypting with the rc4::rc4 command.

```
rc4::rc4 -key "secret" $encrypted → Just a demo
```

Of course, in a real application both ends of the communication would have a transform applied. Obviously, the same shared secret has to be used for both.

```
set in_xform [RC4Transform new secret]  → ::oo::Obj297
set out_xform [RC4Transform new secret] → ::oo::Obj298
lassign [chan pipe] in out              → (empty)
chan push $out $out_xform               → file4134d50
chan push $out $in_xform                → file4134d50
puts $out "Just a demo"                 → (empty)
close $out                              → (empty)
read $in                                → Just a demo
close $in                               → (empty)
```

17.2.4. Applications of channel transforms

Our demonstration of channel transforms involved transforming the data passing through a channel. A number of such transforms are implemented by the `tcl::transform` package in Tcllib[5]. These include

- `base64` and `hex` for encoding binary data
- `otp` (one time pad) and `rot` encryption
- `zlib` for compression (although the built-in capabilities described in Section 17.2.5 make this superfluous).

However, transformation of data is not the only use for channel transforms. Several transformations provided Tcllib[6] do not modify the data but do computations on them instead.

- `crc32`, `adler32` compute checksums on the data passing through
- `counter` simply counts the bytes flowing through a channel

A short example illustrating their use in stacked fashion:

```
package require tcl::transform::crc32
package require tcl::transform::counter
lassign [chan pipe] in out
::tcl::transform::crc32 $out -write-variable crc
::tcl::transform::counter $out -write-variable counter
puts $out "Just a demo"
close $out
puts "Wrote $counter bytes with a CRC of $crc"
→ Wrote 13 bytes with a CRC of 3760966656
```

The count of bytes is the number of bytes, **not** characters, passing through the channel after any encoding, line ending translation etc.

As an aside, note from the example that there is no explicit `chan push` call. The Tcllib[7] transforms make this call internally on the channel passed to them. This is different from our implementation of `RC4Transform` and is a stylistic choice.

Yet another form of channel transform, again available in Tcllib[8] just observes the data passing through a channel. Since channel transforms can be pushed and popped dynamically, this can be very useful for troubleshooting and logging purposes. The `tcl::transform::observer` transformation implements this functionality.

17.2.5. Zlib channel transforms

In Section 4.16, we introduced the `zlib` command for compressing binary strings. In addition to the features described there, the command also directly supports use as a channel transform with the `zlib push` command.

```
zlib push ALGORITHM CHANNEL ?OPTIONS?
```

The command pushes a zlib transformation on to the specified channel. The transformation may be popped with the normal `chan pop` command. The *ALGORITHM* argument specifies the type of compression or decompression to be applied and must be one of the values `deflate`, `inflate`, `compress`, `decompress`, `gzip`, `gunzip`.

The command takes the options shown in Table 17.2.

[5] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html
[6] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html
[7] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html
[8] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

Table 17.2. `zlib push` command options

Option	Description
`-dictionary` *BINDATA*	See Table 4.20.
`-header` *HEADER*	See Table 4.20.
`-level` *LEVEL*	See Table 4.20.
`-limit` *LIMIT*	Specifies a limit for read-ahead of data. See Section 17.2.2.5. Defaults to 1.

Once the transform is pushed on a channel, the `-dictionary`, `-header` and `-limit` options can be read with the `fconfigure` or `chan configure` commands.

In addition, `chan configure` / `fconfigure` take two additional options. The first, `-checksum`, is a read-only option that returns the checksum for the uncompressed data seen on the channel up to that point. The second option, `-flush`, is a write-only option that takes values `sync` or `full`. See Section 4.16.2.9 for their effect.

A simple example of using the transform follows. We create a channel to a file, push the compression transform onto the channel, and write to it.

```
set chan [file tempfile path]              → file3b33750
zlib push compress $chan                   → file3b33750
puts $chan [string repeat "abcdefghij" 10] → (empty)
```

Let us also retrieve the checksum before we close the channel.

```
flush $chan                     → (empty)
chan configure $chan -checksum  → 525608894
close $chan                     → (empty)
```

Notice that we flush the channel before retrieving the checksum. Otherwise the transformation would not have seen all the data yet. If the buffering on the channel is turned off, this is not necessary.

Let us see if the file is compressed.

```
file size $path → 23
```

Indeed it is. But let us read it back to ensure it is compressed and not truncated. Along the way, we retrieve the checksum and see that it is correct.

```
set chan [open $path r]         → file4108640
zlib push decompress $chan      → file4108640
read $chan                      →
 abcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghij
chan configure $chan -checksum → 525608894
close $chan                     → (empty)
```

One point to note is that even though the compressed file is a binary file, not text, we do not open it in binary mode. The mode for open is the same mode used to write the file (defaulting to text in our example). The transformation sits **below** the text / binary translation layer of the Tcl I/O subsystem so it will always see the raw bytes from the file irrespective of how the channel is configured.

17.3. Reflected channels

Whereas channel transforms allow us to process data as it is being passed through channels, Tcl's reflected channel facility presents a way to define entire new channel types at the script level. With reference to Figure 17.1, reflected channels occupy the "device" box in the I/O stack.

Our discussion is mostly focused towards **implementation** of reflected channels as **using** them is for the most part no different from the built-in channels. However, there are some notable limitations that we describe in Section 17.3.3.

17.3.1. Implementing reflected channels

Just like channel transforms, reflected channels are implemented as command prefixes that are invoked by Tcl's I/O system in response to various I/O operations. These command prefixes may be in the form of namespace ensembles, TclOO objects or anything else that will accept subcommands that specify the operation.

The complete list of these subcommands is shown in Table 17.3. As for channel transforms, implementation of some of these subcommands is optional depending on whether the channel supports read or write operation.

Table 17.3. Reflected channel subcommands

Subcommand	Direction	Description
`initialize` *HANDLE MODE*	Both	Called to initialize the channel.
`finalize` *HANDLE*	Both	The last call on the channel to permit it to clean up any allocated resources.
`watch` *HANDLE EVENTS*	Both	Called to inform the channel as to the types of I/O events, read and/or write, that should be reported.
`read` *HANDLE COUNT*	Read channels	Called to read *COUNT* bytes from the channel.
`write` *HANDLE BYTES*	Write channels	Called to write the *BYTES* binary string to the channel.
`seek` *HANDLE OFFSET BASE*	Both, optional	This subcommand should move the file access pointer as specified.
`configure` *HANDLE OPTNAME OPTVAL*	Both, optional	Called to configure a channel configuration option.
`cget` *HANDLE OPTNAME*	Both, optional	Called to retrieve the value of a channel configuration option.
`cgetall` *HANDLE*	Both, optional	Called to retrieve the values of all channel configuration options.
`blocking` *HANDLE MODE*	Both, optional	Called to set the blocking mode of a channel

As before, we will look at these subcommands in detail as we go through the implementation of a simple reflected channel. Our illustrative sample implements a channel that will simply store data written to it and allow it to be read out. The `tcl::chan::fifo` module in Tcllib[9] already provides such a reflected channel but ours will differ in a couple of respects for pedagogic purposes.

Whereas we demonstrated channel transformations using TclOO, for reflected channels our implementation will use namespaces instead so they don't feel left out. All our subcommands will be implemented as procedures within this namespace which we will name `bureaucrat`. Why, you ask? Because

- It introduces unnecessary delays in data transfer without doing any useful work in the process. In our case, this is to better demonstrate event-driven I/O.
- It will sit on one task and refuse to accept another until the first one is done. This is for demonstrating flow control and how blocking is signaled to the Tcl I/O system.

[9] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

For reasons that will become clear later, our reflected channel requires the event loop to be running.

17.3.1.1. Initializing a reflected channel

On channel creation, the Tcl I/O system will call the `initialize` subcommand for the reflected channel.

```
namespace eval ::bureaucrat {
    variable channels
    array set channels {}
}
proc ::bureaucrat::initialize {chan mode} {
    variable channels
    set channels($chan) {
        State    OPEN
        Data     {}
        InFlight 0
        Delay    100
        Blocking 1
        Watch    {}
    }
    return {initialize finalize watch read write configure cget cgetall blocking}
}
```

The first parameter of `initialize` is the channel name itself. While a TclOO based implementation like the one we demonstrated for channel transforms, a namespace based one does not maintain an implicit per-channel context. We therefore store the per-channel state as a dictionary in an array `channels` indexed by the channel name. The semantics of the keys in the dictionary will be clarified as we go along.

The second parameter to `initialize` is a list of one or two elements `read` and/or `write` indicating whether the attached channel is readable and writable. We ignore this parameter as we do not care.

The return value from `initialize` should be the list of supported subcommands. We will not support the seek operation and therefore notice it is not included in the returned list.

17.3.1.2. Closing a reflected channel

When an application closes a reflected channel, Tcl invokes the `finalize` subcommand to inform the reflected channel that it is now history and clean up any allocated resources.

```
proc ::bureaucrat::finalize {chan} {
    variable channels
    unset -nocomplain channels($chan)
}
```

Our toy reflected channel is self contained so all we need to do is get rid of the state variable. In channels like network connections, the remote end would have to be notified and so on.

17.3.1.3. Configuring a reflected channel

A reflected channel may **optionally** define its own configuration options in addition to the standard ones defined by Tcl for all channels. The `configure` subcommand is called when an application wants to set the value of an option.

The standard configuration options on channels are handled by the Tcl I/O system itself and never passed down to the reflected channel.

In our example, we will allow the channel delay to be configured by an application. We only do minimal error checking for brevity.

```
proc ::bureacrat::configure {chan optname optval} {
    variable channels
    if {$optname ne "-delay"} {
            error "Unknown option \"$optname\"."
    }
    dict set channels($chan) Delay $optval
    return
}
```

The `configure` subcommand is optional and a reflected channel is not obliged to provide one.

Correspondingly, the channel may implement two subcommands `cget` and `cgetall` for retrieving configuration values. Again, these are optional. However, if either one is provided, the other one must be be provided as well. The `cget` command should return the value of the specified option. The `cgetall` command should return a dictionary of all options and their values.

```
proc ::bureacrat::cget {chan optname} {
    variable channels
    if {$optname ne "-delay"} {
        error "Unknown option \"$optname\"."
    }
    return [dict get $channels $chan Delay]
}

proc ::bureacrat::cgetall {chan} {
    variable channels
    return [list -delay [dict get $channels $chan Delay]]
}
```

17.3.1.4. Non-blocking mode and event driven I/O

The next two subcommands we discuss, `blocking` and `watch` pertain to non-blocking and event-driven I/O operations.

Channels always start out in blocking mode. When an application sets or resets the blocking mode of a channel with a call of the form

```
chan configure $chan -blocking 0
```

the Tcl I/O system makes a call down to the channel's `blocking` subcommand to inform it of the expected blocking mode of operation. In our case, we simply save the mode in the channel's state. We will make use of its setting when we implement the actual reads and writes.

```
proc ::bureaucrat::blocking {chan mode} {
    variable channels
    dict set channels($chan) Blocking $mode
}
```

While the `blocking` command is optional, the `watch` command is mandatory. It is called by the Tcl I/O system in response to the application indicating an interest in read and/or write events on the channel (see Section 17.1.2).

```
proc ::bureaucrat::watch {chan events} {
    variable channels
    set watched [dict get $channels($chan) Watch]
    dict set channels($chan) Watch $events
    if {"read" in $events && "read" ni $watched} {
        notify $chan read
    }
```

```
    if {"write" in $events && "write" ni $watched} {
        notify $chan write
    }
}
```

The `events` parameter to `watch` is a list, possibly empty, containing some combination of the values `read` and `write`. It indicates what kind of channel events the application has registered an interest in with the `chan configure` or `fconfigure` commands. The `Watch` element of our state dictionary keeps track of this information. We check if we are **adding** either event type as an event of interest. If so, the `notify` procedure, that we define next, is called to send an appropriate event to the application. We do this check to avoid sending extraneous events in case we were already notifying that event type.

The `notify` procedure is internal to our implementation and not called directly by the Tcl I/O system. Its purpose in life is to generate an event notification to the application if the conditions warrant it.

```
proc ::bureaucrat::notify {chan event} {
    variable channels
    dict with channels($chan) {
        if {$event ni $Watch} {
            return
        }
        if {$event eq "read"} {
            if {[string length $Data] == 0 && $State ne "EOF"} {
                return
            }
        } else {
            if {$State ne "OPEN"} return
        }
    }
    after idle [list after 0 [list chan postevent $chan $event]]
}
```

The `notify` procedure is short but there are several important points to be noted therein so we will go over it line by line.

- First, if the application has indicated no interest in the event we return.
- Next we have to check if the conditions implied by the event are present.
- For `read` events, the channel must have data available to read **or** it must be at end-of-file. If neither of these conditions exist, we will not send a `read` notification.
- For `write` events, we impose no restrictions except that the channel must be in the open state for the application to be able to write.
- If the proper conditions are met, we generate an appropriate event with the `chan postevent` command.

We will discuss this last point in a bit but first we need to touch on the channel state conditions. In our simple example, the channel is always in only one state — that we designate as `OPEN` — from the time it is created to the time it is closed. Channels in the real world tend to be more complex. An network connection for example may transition from a "connecting" state to an "open" state, to an "end of file" state if the remote end closes the connection. Our `watch` procedure assumes such state transitions for pedagogic purposes even though the channel itself does not go through these states.

Regarding the sending of events to the application, Tcl provides the `chan postevent` command to do the needful.

```
chan postevent CHANNEL EVENTLIST
```

Here *CHANNEL* is the channel for which the event is to be generated and and *EVENTLIST* is a list containing one or both of `read` or `write` denoting the events. In response, the Tcl I/O system will invoke any channel event handlers that were registered for that channel.

One final explanation is warranted for the manner in which `chan postevent` is called. Basically, `chan postevent` is scheduled through the event loop and not called directly. The reason for this is that `chan postevent` implementation will directly invoke the application event handler. That may in turn call back into any of the channel I/O API's and thereby back into our implementation even before we have completed the `notify` procedure. This reentrancy can get very tricky to deal with and we avoid it by scheduling through the event loop. That ensures that the current call is unwound completely before any application code calls into our code.

As to the strange `after idle after 0` idiom, it prevents starvation of the event queue. See Section 15.3.3.1 for a full explanation.

17.3.1.5. Implementing data output

We finally come to implementing the main functions of a reflected channel, viz. some form of data transfer. We will begin with the output side of data transfer.

The Tcl I/O system will call the `write` subcommand to write data to the device. Although not directly relevant to our implementation, remember from our discussion of channel transforms that this is binary data after all the channel encoding and translation have taken place. This subcommand need not be implemented for a read-only channel.

Our implementation of `write` has a couple of artifacts. First, it introduces an artificial delay in the data transfer so that data is not immediately available on the input side. Second, it restricts data flow such that while the data is "on the way" to the input side, additional data will not be accepted. These are for the purposes of illustrating non-blocking and event driven operation.

Here is the code itself.

```
proc ::bureaucrat::write {chan bytes} {
    variable channels
    if {[string length $bytes] == 0} {
        return 0
    }
    dict with channels($chan) {
        if {$InFlight} {
            if {$Blocking} {
                return -code error EAGAIN
            } else {
                return -code error EAGAIN
            }
        }
        set InFlight 1
        after $Delay [list [namespace current]::delayed_receive $chan $bytes]
    }
    return [string length $bytes]
}
```

We implement the data transfer delay by scheduling a callback to the `delayed_receive` procedure with the `after` command. The `InFlight` element in the channel's state dictionary is used as a flag to remember that data is "in flight" via the event loop. Before the actual scheduling of the callback, we check this flag and if set, we return the `EAGAIN` error code to the caller.

The Tcl I/O system treats an error code of `EAGAIN` in special fashion. It is taken as an indicator that the channel cannot accept more data at this time, because its output buffers are full, or the remote end has flow controlled the connection and so on. The `chan blocked` command would then return 1 to the application. Tcl itself will continue to retry sending any buffered data at a later point via the event loop.

The error code used for indicating the blocking condition **must** be the string EAGAIN. You cannot use the integer value of this POSIX error instead as the value differs from system to system.

You may noticed something strange in the bodies of the `if` clause that checks the `Blocked` variable. Both bodies are the same! Normally, when `Blocked` is `true` the channel implementation should block and not return until either more data is available or an end of file condition occurs. However, we have no means of blocking and therefore are forced to deal with the condition in the same way as the non-blocking case. For a full explanation, see Section 17.3.3. **In a reflected channel that was backed by a OS descriptor, the implementation should block on that descriptor when `Blocked` is true instead of returning a EAGAIN error.**

The return value from the `write` should be the number of bytes that the channel accepted. In our implementation, we accept the whole data and thus return its length. However, this need not be the case. A channel may choose to accept only part of the data for whatever reason and return just the length of the accepted part. Tcl will then hold on to the remaining data and include it in a future call.

We now come to the "second half" of the output implementation. The `delayed_receive` procedure can be thought of as the equivalent of the procedure that will receive data from the remote end of a network connection.

```
proc ::bureaucrat::delayed_receive {chan bytes} {
    variable channels
    if {! [info exists channels($chan)]} {
        return;
    }
    dict with channels($chan) {
        if {$State ne "OPEN"} {
            return
        }
        append Data $bytes
        set InFlight 0
    }
    notify $chan read
    notify $chan write
}
```

The first thing `delayed_receive` does is check that the channel has not been closed between the time the procedure was queued and the time it was invoked. It also checks if the channel is still in the OPEN state (as we said before, this is not really required for our simple implementation). If both conditions are met, it appends the new data to the incoming data buffer. The `InFlight` flag is reset since there is no data queued in transit. It then calls the `notify` procedure we defined earlier to generate a channel read event so any waiting event handlers will be invoked. Since we were limiting writes while the data was in transit, we also need to generate a write event in case any handlers were waiting on the output side as well.

The need to generate a `write` event is really a peculiarity of our example because data is being input on the same channel as it is output. In most real world cases, there would be no need to generate a `write` event when **receiving** data.

Before moving on to implementation of the input side, one final point to reiterate is that the data written with the `write` subcommand should be treated as a byte stream with no direct correspondence with `puts` commands at the application level. Due to encoding, translation and buffering by Tcl, not only might the number of bytes differ, but a single `puts` may be split into multiple `write` invocations, multiple `puts` may be combined into a single `write` and so on.

17.3.1.6. Implementing data input

The `read` subcommand is invoked by the Tcl I/O system when it needs to fetch more data from the channel. As for the output side, there is no immediate correspondence between `gets` or `read` command invocations at the application level and the `read` subcommand of the channel.

The `read` subcommand is passed the channel name and the number of input bytes to be read. If there are fewer bytes available than are requested, it is permitted to return a smaller number **but not more** than requested. Moreover, returning zero bytes will be treated as an end-of-file condition by Tcl. If there are no bytes currently available but the channel is **not** at EOF, the procedure must signal this by raising `EAGAIN` error just as for the output side as we described earlier.

Given the above, our `read` implementation should be mostly self-explanatory.

```
proc ::bureaucrat::read {chan count} {
    variable channels
    dict with channels($chan) {
        if {[string length $Data] == 0} {
            if {$Blocking} {
                return -code error EAGAIN
            } else {
                return -code error EAGAIN
            }
            return -code error EAGAIN
        }
        set bytes [string range $Data 0 $count-1]
        set Data [string range $Data $count+1 end]
    }
    notify $chan read
    return $bytes
}
```

We do call `notify` at the end so that if there is still more data pending, the application read event handlers will be notified appropriately.

With regards to both bodies of the `if` statement checking the `Blocking` condition being identical, see the explanation above for the `write` subcommand.

17.3.1.7. Reflected channel creation

We are almost done with our reflected channel implementation. Just two things remain:

- We have defined a bunch of procedures to implement the various operations that are expected of the reflected channel. However, the Tcl I/O system expects to invoke a single command prefix to which it appends the subcommand and arguments.
- We need to provide a way for the application to create our new channel type.

For the first task, we can simply create an ensemble command (see Section 12.6).

```
namespace eval ::bureaucrat {
    namespace export *
    namespace ensemble create
}
```

This will allow our procedures to be called as

```
bureaucrat initialize ...
bureaucrat read ...
```

and so on. Note our ensemble also makes visible our ancillary procedures like `notify` but there is no harm done. If this offends your sensibilities, you can be more specific about the exported subcommands.

The second task, a means of creating a `bureaucrat` channel, is already provided for us by Tcl with the `chan create` command.

```
chan create MODE CMDPREFIX
```

Here *MODE* is a list of one or two elements with values `read` or `write` that determine whether the channel is read-only, write-only or read-write. The *CMDPREFIX* parameter is the command prefix the Tcl I/O system should call to implement the various operations. Thus to create a `bureaucrat` channel, the application would call

```
set chan [chan create {read write} ::bureaucrat]
```

17.3.1.8. Seeking in a reflected channel

By its nature, just like the built-in sockets our reflected channel type is not amenable to seeking and thus will not actually implement the optional . However, we describe it here just so as to not get an incomplete grade on the book.

The signature for the `seek` subcommand is

```
seek CHANNEL OFFSET BASE
```

The command is called to position the access pointer from where the next read or write will take place. The arguments are exactly those we described for the application level `chan seek` command so we will not describe them again.

The subcommand return value should be the absolute position of the access pointer after it has been moved. The application level `chan tell` also maps to this `seek` subcommand. Tcl will call it with *OFFSET* set to 0 and *BASE* set to `current`, in effect getting the value of the access pointer without actually moving it.

17.3.1.9. The complete channel implementation

We can now put together our complete implementation (with some minor stylistic changes to put procedures inside the namespace block).

```
namespace eval ::bureaucrat {
    variable channels
    array set channels {}

    proc initialize {chan mode} {
        variable channels
        set channels($chan) {
            State    OPEN
            Data     {}
            InFlight 0
            Delay    100
            Blocking 1
            Watch    {}
        }
        return {initialize finalize watch read write configure cget cgetall blocking}
    }

    proc finalize {chan} {
        variable channels
```

```
    unset -nocomplain channels($chan)
}

proc configure {chan optname optval} {
    variable channels
    if {$optname ne "-delay"} {
        error "Unknown option \"$optname\"."
    }
    dict set channels($chan) Delay $optval
    return
}

proc cget {chan optname} {
    variable channels
    if {$optname ne "-delay"} {
        error "Unknown option \"$optname\"."
    }
    return [dict get $channels $chan Delay]
}

proc cgetall {chan} {
    variable channels
    return [list -delay [dict get $channels $chan Delay]]
}

proc blocking {chan mode} {
    variable channels
    dict set channels($chan) Blocking $mode
}

proc watch {chan events} {
    variable channels
    set watched [dict get $channels($chan) Watch]
    dict set channels($chan) Watch $events
    if {"read" in $events && "read" ni $watched} {
        notify $chan read
    }
    if {"write" in $events && "write" ni $watched} {
        notify $chan write
    }
}

proc notify {chan event} {
    variable channels
    dict with channels($chan) {
        if {$event ni $Watch} {
            return
        }
        if {$event eq "read"} {
            if {[string length $Data] == 0 && $State ne "EOF"} {
                return
            }
        } else {
            if {$State ne "OPEN"} return
        }
    }
    after idle [list after 0 [list chan postevent $chan $event]]
}

proc write {chan bytes} {
    variable channels
    if {[string length $bytes] == 0} {
```

```
            return 0
        }
        dict with channels($chan) {
            if {$InFlight} {
                if {$Blocking} {
                    return -code error EAGAIN
                } else {
                    return -code error EAGAIN
                }
            }
            set InFlight 1
            after $Delay [list [namespace current]::delayed_receive $chan $bytes] ❶
        }
        return [string length $bytes]
    }

    proc delayed_receive {chan bytes} {
        variable channels
        if {! [info exists channels($chan)]} {
            return;
        }
        dict with channels($chan) {
            if {$State ne "OPEN"} {
                return
            }
            append Data $bytes
            set InFlight 0
        }
        notify $chan read
        notify $chan write
    }

    proc read {chan count} {
        variable channels
        dict with channels($chan) {
            if {[string length $Data] == 0} {
                if {$Blocking} {
                    return -code error EAGAIN ❷
                } else {
                    return -code error EAGAIN
                }
                return -code error EAGAIN
            }
            set bytes [string range $Data 0 $count-1]
            set Data [string range $Data $count+1 end]
        }
        notify $chan read
        return $bytes
    }

    namespace export *
    namespace ensemble create
}
→ ::bureaucrat
```

❶ Postpone the actual write

❷ Same code as the false clause! See the text

17.3.2. Using reflected channels

Use of a reflected channel is along the same lines as a built-in Tcl channel. We demonstrate our `bureaucrat` channel here, more so to prove it works than to illustrate any new commands or techniques.

We start off creating the channel and configuring it to our liking. Except for channels backed by terminal devices, Tcl defaults to full buffering whereas we would like line buffering. Also, we want to test event driven I/O and further verify our custom option `-delay` is effective.

```
% set chan [chan create {read write} ::bureaucrat]
→ rc18
% chan configure $chan -buffering line -blocking 0 -delay 1000
```

We set up a handler to be called when the channel is writable and implement it using an anonymous procedure that we define using our `lambda` helper. Because of the artificial flow control we implemented, we expect it will be triggered every second or so based on the value we set for `-delay`. It then writes out the current time to the channel.

```
% set counter 0
→ 0
% chan event $chan writable [lambda {chan} {
    if {[incr ::counter] > 2} {
        close $chan
        set ::done 1
        return
    }
    puts $chan [clock format [clock seconds] -format %T]
} $chan]
```

Similarly, our read handler prints out the time the message is received. Strictly speaking we do not need the end-of-file check since we are writing and reading into the same channel. When the writer closes the channel, no handlers will be called anyways once it is closed. But we include the check for completeness.

```
% chan event $chan readable [lambda {chan} {
    set message [gets $chan]
    if {$message eq "" && [chan eof $chan]} {
        close $chan
        return
    }
    puts "Received \"$message\" at [clock format [clock seconds] -format %T]"
    return
} $chan]
```

Finally, assuming we are running in `tclsh` without the event loop active, we fire it up.

```
% vwait ::done
→ Received "11:45:52" at 11:45:53
  Received "11:45:53" at 11:45:54
```

As we hoped, messages are sent at one second intervals and received after our configured delay.

17.3.3. Reflected channel limitations

There are some limitations in the functionality of reflected channels that we now outline.

The first and most important is that reflected channels are not associated with operating system level descriptors or handles and are invisible to other processes or even other non-Tcl threads. Therefore they cannot be passed to child processes, used for `exec` redirection and so on.

Secondly, reflected channels cannot support the half-closing of read-write channels as can socket channels. This limitation is actually not specific to reflected channels as even file based channels do not support half-close operations.

The third limitation is more subtle and involve blocking operation. Moreover, it is only specific to reflected channel implementations in which the same channel was used for both reading and writing within a single Tcl thread. Since this is the the type of reflected channel we used for illustrative purposes, we demonstrate with the following short example code.

```
set chan [chan create {read write} ::bureaucrat] → rc19
```

Note the channel is a blocking channel. Given it has no content, we expect the following `read` to permanently block. What happens instead is that we get an empty string back and if we call `chan block`, it indeed indicates no data is available.

```
read $chan 1        → (empty)
chan blocked $chan → 1
chan eof $chan      → 0
```

As per the Tcl reference page for `read`, for blocking channels our `read` should have only returned the empty string if the channel was at end-of-file. If no data was available, the command should have blocked.

This behaviour could be indeed be seen as a bug in our implementation. However, it is also exhibited by similar reflected channel implementations like `fifo` in Tcllib[10]. The issue, and reason why it cannot be fixed, is as follows. When our channel implementation's `read` subcommand is invoked and no data is available, there is no way for it to block **while still allowing the same thread to run and write to the channel to unblock it**. Use of commands like `vwait` for this purpose would violate blocking semantics. We are thus stuck between a rock and a hard place resulting in the seen behaviour.

It is worth reiterating that this flaw is only present in specific types of reflected channels. Those backed by a real operating system device, or one in which data is always available, such as the `random` channel in Tcllib that sources random byte sequences, will not have this blocking issue. Moreover, even for those that do, an application can work around is by checking `chan blocked` on empty results and not assuming end-of-file.

Nevertheless, these channels are best utilized in non-blocking event-driven mode.

17.3.4. Applications of reflected channels

There are a number of useful applications of reflected channels available in Tcllib. We list some of these here.

- `tcl::chan::null` implements a write-only channel that simply discards anything written to it.
- `tcl::chan::zero` implements a read-only channel that returns a continuous stream of null characters.
- `tcl::chan::string` implements a read-only channel that returns the contents of a string. This is useful when a command only operates on channel content and you want it to process the data in a string.
- `tcl::chan::fifo` is similar to our `bureaucrat` without its delay and flow control artifacts.
- `tcl::chan::cat` implements a read-only channel that allows reading of multiple channels via a single channel. The following code will slurp all content of files with an extension of `txt` in the current directory.

```
package require tcl::chan::cat
set chan [tcl::chan::cat {*}[lmap fn [glob *.txt] {open $fn}]]
set content [read $chan]
close $chan
```

[10] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

- `tcl::chan::random` implements a read-only channel that returns a continuous stream of random bytes.

```
package require tcl::chan::random              → 1
package require tcl::randomseed                → 1
set chan [tcl::chan::random [tcl::randomseed]] → rc20
bin2hex [read $chan 4]                         → 08 04 6e 1f
bin2hex [read $chan 6]                         → 22 2a 50 32 45 f1
close $chan                                    → (empty)
```

- `tcl::chan::variable` is similar to `tcl::chan::string` above except that the channel is writable as well and reflects the content of a variable.

```
package require tcl::chan::variable → 1.0.3
set var "abc"                       → abc
set chan [tcl::chan::variable var]  → rc21
puts -nonewline $chan "def"         → (empty)
flush $chan                         → (empty)
set var                             → def
close $chan                         → (empty)
```

- `tcl::chan::textwindow` implements a write-only channel that appends any content written to it to a Tk text window.

Tcllib[11] provides some other channels not listed above as well. In addition, it provides a pair of TclOO classes, `tcl::chan::core` reflected channel class and `tcl::chan::events` reflected channel class, which form the basis of the above implementations and can be used for your reflected channels as well.

17.4. Chapter summary

We introduced Tcl I/O in Chapter 9 and delved into the more advanced aspects of asynchronous I/O, channel transforms and reflected channels in this chapter. What remains with respect to I/O is a discussion of communication facilities in Tcl and that is where we will go next.

17.5. References

TIP219
Tcl Channel Reflection API, Andreas Kupries, Tcl Improvement Proposal #219, http://www.tcl.tk/cgi-bin/tct/tip/219

TIP230
Tcl Channel Transformation Reflection API, Andreas Kupries, Tcl Improvement Proposal #230, http://www.tcl.tk/cgi-bin/tct/tip/230

[11] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

18

Networking and Communications

> Communication is everyone's panacea for everything.
>
> — Tom Peters

And what's true for humans is true for software. Communication with other systems, whether local or over the Internet, is part of almost every modern application. Tcl provides support for communications at multiple levels and protocols:

- Raw TCP and UDP
- Application level protocols like HTTP, FTP
- Communication over serial ports

We will cover all of these in this chapter.

18.1. Network communications

Tcl's support for networking is for the most part limited to the Internet (IP) suite of protocols though other protocols suites such as IPX are available through platform-specific packages. Even for IP, the core Tcl command set only supports communication over raw TCP. Support for UDP and application level protocols like HTTP, FTP etc. is provided through additional packages.

18.1.1. IP addresses

The `socket` command used to create a TCP communication channel will accept both IPv4 and IPv6 format addresses .

Applications that need to explicitly parse or otherwise manipulate IP addresses can use the `ip` package from Tcllib[1]. For example,

```
% package require ip
→ 1.3
% ip::normalize 2404:6800:4007:807::2004 ❶
→ 2404:6800:4007:0807:0000:0000:0000:2004
% ip::contract 2404:6800:4007:0807:0000:0000:0000:2004 ❷
→ 2404:6800:4007:807::2004
% ip::version 2404:6800:4007:807::2004 ❸
→ 6
% ip::version 216.58.197.36
→ 4
```

❶ Canonical form for an IPv6 address
❷ Compact form for an IPv6 address
❸ IP version

[1] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

The package provides a number of other commands as well that are particularly useful for applications focused on network or system administration.

18.1.2. DNS names

Instead of IP addresses, DNS names may also be used with the `socket` commands. Tcl will internally resolve them to addresses as required.

The `info hostname` command can be used to retrieve the name of the local system.

```
info hostname → ares
```

Note however that the name returned may not be fully qualified.

Tcl does not itself provide a means to map a DNS name to an IP address but you can use the `dns` package from Tcllib[2] for this purpose.

```
% package require dns
→ 1.3.5
% dns::nameservers ❶
→ 8.8.8.8 8.8.4.4 192.168.1.1
```

❶ Retrieve the list of name servers configured for the system

To retrieved information for a DNS name you have to first obtain a handle for it using `dns::resolve` and check for success with `dns::status`.

```
% set tok [dns::resolve www.microsoft.com]
→ ::dns::4
% dns::status $tok
→ ok
```

You can then retrieve all records received in the DNS response by passing the received handle to `dns::result`.

```
% dns::result $tok
→ {name www.microsoft.com type CNAME class IN ttl 849 rdlength 35 rdata
   ↳ www.microsoft.com-c-2.edgekey.net} {name www.microsoft.com-c-2.edgekey.net type CNAME
   ↳ class IN ttl 14534 rdlength 55 rdata
   ↳ www.microsoft.com-c-2.edgekey.net.globalredir.akadns.net} {name
   ↳ www.microsoft.com-c-2.edgekey.net.globalredir.akadns.net type CNAME class IN ttl 897
   ↳ rdlength 24 rdata e1863.dspb.akamaiedge.net} {name e1863.dspb.akamaiedge.net type A class
   ↳ IN ttl 17 rdlength 4 rdata 23.66.237.138}
```

Alternatively, you can just query it for specific fields.

```
% dns::address $tok ❶
→ 23.66.237.138
```

❶ Note return value is a **list**, possibly with a single element

The `dns::cleanup` command must be called afterwards to release resources.

[2] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

```
→ dns::cleanup $tok
```

The dns package will use UDP to communicate with the DNS server if the `tcludp` package is available and TCP otherwise. This can be controlled via the `-protocol` option to the `dns::resolve` command.

The dns package **only** looks up names via the DNS system. This may not be the same as how the host system resolves names. For example, a Windows system may use WINS in addition to DNS.

The package has a number of other commands and configuration options. See the reference documentation for details.

18.1.3. Text and binary protocols

It is particularly important to ensure the channel configuration settings are correct for the application protocol used over a network connection. Obviously, both the client and server ends must use the same settings.

Binary protocols

For binary application protocols that treat data as a sequence of bytes, the channel's `-translation` option must be set to `binary`. This will also have the desired side effect of setting the channel encoding to `binary` as well and turning off special handling of end-of-line and end-of-file characters. Usually

```
chan configure $chan -translation binary
```

suffices to set up the channel appropriately. Needless to say, the strings written to the channel must also be binary strings (see Section 4.13).

Text protocols

For text based protocols, the channel options to consider are `-translation`, `-encoding` and, rarely, `-eofchar`.

- The application protocol defines the character sequence for line endings and accordingly the `-translation` option must be set to `lf`, `cr` or as is most common, `crlf`.
- The default `-encoding` on the created channel is the system encoding. This is rarely correct since the systems at the two ends could very well be using different system encodings. Most modern protocols use "utf-8" while older ones like older ones like SMTP may use `iso8859-1`.
- For text protocols that are **line** oriented, it can be convenient to set the `-buffering` option to `line` mode. Otherwise, you need to remember to do a `chan flush` on the channel at appropriate times.
- Application protocols do not generally use character sequences to indicate end of file so the `socket` command's default of disabling this feature is usually appropriate.

A suitable configuration for HTTP might be

```
chan configure $chan -translation crlf -encoding utf-8
```

Note however that protocols like HTTP can require the encoding to be changed midstream, for example to transfer binary data.

18.1.4. Communicating over TCP

The *Transmission Control Protocol* (TCP) is a stream oriented protocol where the data being transferred is seen as sequence of bytes without any message boundaries. This is a natural fit for Tcl's I/O model and allows the standard channel commands like `puts`, `read` etc. to be used for communicating over TCP as well.

18.1.4.1. Writing TCP clients: `socket`

We will start off with the client end of TCP connections, describing the basic operation, asynchronous connects and various channel options specific to sockets.

The `socket` command is used by client applications to establish a TCP connection to a server.

```
socket ?-myaddr LOCALADDR? ?-myport LOCALPORT? ?-async? SERVER SERVERPORT
```

The command returns a read-write channel which can be used in the same fashion as was described for files and process pipes.

The *SERVER* parameter is the DNS name or IP address (either IPv4 or IPv6) of the server to which the connection is to be established and *SERVERPORT* is the port number on which the server is listening.

The address and port used for the local end of the connection will be automatically picked by the system. However, if required the application can specify the `-myaddr` and `-myport` options to bind the socket to a specific local address and port.

Like all channels, socket based channels must also be closed with the `close` / `chan close` command once the application is done with them. Moreover, as was described for process pipelines in Section 16.4, it is also possible to "half-close" a channel by shutting down only one direction of the two-way connection. For example,

```
close $so write
```

18.1.4.1.1. Connecting synchronously

If the `-async` option is not specified, the `socket` command will connect synchronously to the server and block until the connection is established.

Here is a simple example to retrieve a Web page.

```
% set http_req "GET http://www.example.com HTTP/1.1\n" ❶
→ GET http://www.example.com HTTP/1.1

% set so [socket www.example.com 80]
→ sock0000000003CE1EE0
% chan configure $so -encoding utf-8 -translation crlf -buffering line ❷
% puts $so $http_req
% close $so write ❸
% read $so
→ HTTP/1.1 200 OK
  Cache-Control: max-age=604800
  Content-Type: text/html
  Date: Mon, 01 May 2017 06:34:57 GMT
  Etag: "359670651+gzip+ident"
  ...Additional lines omitted...
```

❶ Note the extra newline to indicate end of HTTP headers
❷ HTTP is line-oriented with CR-LF line endings
❸ Half close the channel so server knows we are done

One point to be noted above is the half-closing of the channel. We described half-closing in Section 16.4. Its purpose here is the essentially the same as we described there. For HTTP 1.1, the server allows the client to send multiple requests. Our half-close indicates to the server that no more requests will be arriving, allowing it to then close its end of the connection. That in turn will result in an EOF on our end of the channel causing our `read` to complete. Without the half-close, our unconstrained `read` would block until the server timed out and closed its connection.

18.1.4.1.2. Connecting asynchronously

By default, the `socket` command will block until the connection to the server is fully established. To have the connection be established asynchronously, the application can specify the `-async` option. The command will then initiate the connection but return right away before the connection is fully established.

As of the time of writing, if *SERVER* specifies a host name as opposed to an IP address, the `socket` command is not completely asynchronous as the name lookup is done in synchronous fashion.

Note that even with `-async` specified, the channel is still in blocking mode so any attempt to do read from the socket will block until the connection is established **and** data is received from the other end.

Most commonly, an asynchronous connect is followed by a call to put the channel into non-blocking mode and registering read and write event handlers. When the connection is established, the write event is fired indicating that the channel may now be written. Conversely, the read handler will be invoked upon the arrival of data over the connection.

Let us rewrite our previous script in asynchronous style using non-blocking sockets. Like all our asynchronous and event driven I/O examples, this script also relies on the event loop and hence the `vwait` on the variable done. In a real application which already had the event loop active, this would not be required.

```
proc on_read so {
    set s [read $so]
    if {[string length $s] == 0} {
        if {[eof $so]} {
            close $so
            set ::done 1
        }
        return
    }
    puts $s
}

proc on_write so {
    puts $so "GET http://www.example.com HTTP/1.1\n"
    chan event $so writable {}
    chan close $so write
}

set so [socket -async www.example.com 80]
fconfigure $so -blocking 0 -encoding utf-8 -translation crlf -buffering line
chan event $so readable [list on_read $so]
chan event $so writable [list on_write $so]
vwait ::done
```

We have already seen similar code in Chapter 17 so we just point out the differences. The `on_write` handler will be called when the socket channel is ready to be written to, the first instance of which is when the connection to the remote server is established. It writes the `GET` request to the channel and then removes the write handler on the channel as we will not have anything more to write to it. We also close the socket for writes so the server knows we have nothing more to send.

When data arrives, the `on_read` handler is called which reads all data received so far and prints it, closing the socket on an end of file which indicates the connection is closed.

18.1.4.2. Writing TCP servers: `socket -server`

We now turn our attention to the other end of a network connection — the server side. A server starts listening on a port using a different form of the `socket` command.

```
socket -server COMMANDPREFIX ?-myaddr LOCALADDR? SERVERPORT
```

The command returns a channel for a socket that is listening on port *SERVERPORT*. If *SERVERPORT* is 0, the system will pick a random unused port whose value can be retrieved with the `-sockname` option to `chan configure`.

By default, incoming requests to any of the local system's addresses, both IPv4 and IPv6 (if supported by the system) will be accepted. The `-myaddr` option may be used to restrict accepted connections to those targeting a specific DNS name or IP address on the local system.

When a client's connection request is received, Tcl creates a **new** channel attached to the network connection for that client. It then invokes *COMMANDPREFIX* with three additional arguments: the new channel which is to be used to communicate with this client, the client's IP address and client port number.

Note the distinctions between the original channel returned by the `socket` command and the new channel:

- All communications occurs over the client channels created in response to received connection requests. The channel returned by a `socket -server` command cannot be used for any data transfer.
- Closing a client channel does not impact the listening channel. New connections will continue to be accepted. Likewise, closing the listening channel will prevent new connections from being accepted but will not impact existing connections in any way.
- The channel options, such as `-buffering`, `-encoding`, etc. have to be set individually for each client channel. They are **not** "inherited" from the listening socket.

Here is an example server that services multiple clients. It assumes a line oriented protocol and responds to each line from a client with the client's port number followed by the received line with the characters reversed.

We start off by defining the command that will invoked in response to every connection request. We want to be able to serve multiple clients so we make the accepted client channel non-blocking and attach a read handler.

```
proc on_accept {so client_ip client_port} {
    chan configure $so -buffering line -encoding utf-8 -blocking 0 -translation crlf
    chan event $so readable [list on_read $so $client_port]
}
```

Our read handler reverses the line and sends it back to the client. As a special case, an empty line from the client will end our server.

```
proc on_read {so client_port} {
    set n [gets $so line]
    if {$n > 0} {
        puts $so "$client_port: [string reverse $line]"
        return
    } elseif {$n == 0} {
        exit 0 ❶
    } elseif {[chan eof $so]} {
        close $so
    }
}
```

❶ Empty line → treat as exit server command (**any** client can shut down server)

Now we create the listening socket on a port. Instead of picking a port ourselves, we will let the system pick one for us by specifying 0 as the port number. Also we will only listen on our local loopback address. After creating the listening socket the server will sit waiting in the event loop.

```
set listener [socket -server on_accept -myaddr 127.0.0.1 10042]
vwait forever
```

Here is the entire server script.

```
# server.tcl
proc on_accept {so client_ip client_port} {
    chan configure $so -buffering line -encoding utf-8 -blocking 0 -translation crlf
    chan event $so readable [list on_read $so $client_port]
}

proc on_read {so client_port} {
    set n [gets $so line]
    if {$n > 0} {
        puts $so "$client_port: [string reverse $line]"
        return
    } elseif {$n == 0} {
        exit 0
    } elseif {[chan eof $so]} {
        close $so
    }
}

set listener [socket -server on_accept -myaddr 127.0.0.1 10042]
vwait forever
```

Let us run the server **in a separate process** from an interactive client.

```
% exec [info nameofexecutable] scripts/server.tcl &
→ 9072
```

When connecting, we take care to configure the client channels with the same options as the server end.

```
% set so [socket 127.0.0.1 10042]
→ sock0000000003F071B0
% chan configure $so -buffering line -encoding utf-8 -translation crlf
% puts $so abc
% gets $so
→ 53463: cba
```

And just to prove we can have multiple connections without the server getting blocked, start a second client connection.

```
% set so2 [socket 127.0.0.1 10042]
→ sock00000000041CB930
% chan configure $so2 -buffering line -encoding utf-8 -translation crlf
% puts $so2 def
% gets $so2
→ 53464: fed
%
% puts $so ""  ❶
% close $so
% close $so2
```

❶ Shut down the server

18.1.4.3. TCP connection state

The `chan configure` / `fconfigure` commands can return additional information for socket based channels. These are read-only options and shown in Table 18.1.

Table 18.1. Socket-specific configuration options

Option	Description

`-connecting`	Returns 1 if the socket is still in the process of connecting and 0 otherwise. This option is only supported for client side sockets and is intended to be used for asynchronous connects that specify the `-async` option.
	`% set so [socket -async 127.0.0.3 9999]`
	`→ sock00000000041CB930`
	`% chan configure $so -connecting`
	`→ 1`

`-error`	Returns any error status currently associated with a socket.
	`% set so [socket -async 127.0.0.3 9999]`
	`→ sock00000000041CB930`
	`% wait 1000` ❶
	`% chan configure $so -error`
	`→ connection refused`
	❶ Hang around a bit for the connect to fail

`-peername`	Returns a list of three elements containing the address, hostname, and port of the remote end of a connection. This option is not supported for a listening socket as it is not connected to a remote peer.
	`% set so [socket www.microsoft.com 80]`
	`→ sock000000000310BAC0`
	`% chan configure $so -peername`
	`→ 23.15.107.229 a23-15-107-229.deploy.static.akamaitechnologies.com 80`
	`% close $so`

`-sockname`	For connected sockets this returns a list of three elements containing the address, hostname and port of the local end of a connection. For listening sockets, the return value is a flat (i.e. not nested) list that may contain more than one such triple corresponding to each address and port that is being listened on.
	`% set so [socket -server accept 9999]`
	`→ sock000000000431E2F0`
	`% chan configure $so -sockname`
	`→ 0.0.0.0 0.0.0.0 9999 :: :: 9999`

18.1.5. Communicating over UDP

As implied by its name, the *User Datagram Protocol* (UDP) is not a stream protocol like TCP. Data transfer in UDP takes place in the form of individual messages or *datagrams*. Aside from the fact that message delivery and sequencing is not guaranteed, UDP is not a good fit for Tcl's channel-based I/O model which has no concept of message boundaries or means to demarcate them. This must be kept in mind when using channels to communicate over UDP.

Because Tcl has no built-in support for it, one of several available extensions must be used for communicating over UDP. The one we will describe is the `tcludp` package available from https://sourceforge.net/projects/tcludp/ as it is the one most widely used and available on both Windows and Linux/Unix platforms.

```
package require udp → 1.0.11
```

18.1.5.1. Creating a UDP socket

A UDP socket is created with the udp_open command.

```
udp_open ?PORT? ?reuse? ?ipv6?
```

The command creates a UDP network socket and returns a channel tied to it. Unlike TCP, there is no notion of a "connection" in UDP so from a UDP protocol perspective there is no command variation to create a "client" or "server" side socket per se. Of course, at the application protocol level, a DNS "client" may make a request to a DNS "server" but at the UDP level there is no real distinction.

The optional *PORT* parameter specifies the port on which the socket will receive packets. If 0 or unspecified, the system will select an unused port for the application. This can then be retrieved with the custom `-myport` option to the `fconfigure` or `chan configure` commands.

By default, the system will not permit the same address and port to be used on multiple sockets. Specifying the `reuse` keyword as an argument sets the `SO_REUSEADDR` flag on the socket permitting such sharing.

Finally the `ipv6` keyword will create an IPv6 socket instead of the default IPv4 one.

Let us create a few UDP sockets for us to play with.

```
set so1 [udp_open]                                          → sock788
set so2 [udp_open]                                          → sock820
set so3 [udp_open]                                          → sock816
chan configure $so1 -buffering none -translation binary → (empty)
chan configure $so2 -buffering none -translation binary → (empty)
chan configure $so3 -buffering none -translation binary → (empty)
```

For reasons we will delve into in Section 18.1.5.6, we turn off buffering on all channels and set the channels to binary mode.

Since we have not specified the port numbers, the system would allocate them for us. We can find the allocated ports via the `-myport` option.

```
set so1_port [chan configure $so1 -myport] → 57012
set so2_port [chan configure $so2 -myport] → 57013
set so3_port [chan configure $so3 -myport] → 57014
```

18.1.5.2. Sending UDP datagrams

Given that a UDP socket is not connected to a particular remote end point, it can be used to send to **any** remote UDP end point. This does mean that the receiving address and port have to be explicitly specified. Since Tcl's channel `puts` command does not have a means to specify this, it must be configured on the channel before doing the send. This is done by the `-remote` option to `fconfigure` / `chan configure`.

```
% chan configure $so2 -remote [list localhost $so1_port]
→ localhost 57012
```

The option takes a pair consisting of the remote hostname or IP address and port number. All outgoing datagrams on that channel will be sent to that end point until it is changed with another call to set the `-remote` option.

You can then send datagrams over the channel with the puts command. Each puts command results in a single datagram **as long as you follow the guidelines** in Section 18.1.5.6.

```
puts -nonewline $so2 "so2->so1 (0)"                → (empty) ❶
puts -nonewline $so2 "so2->so1 (1)"                → (empty)
fconfigure $so2 -remote [list localhost $so3_port] → localhost 57014
puts -nonewline $so2 "so2->so3 (0)"                → (empty)
```

❶ Note we are writing strings to a channel configured to be binary. That is OK as long as strings are all ASCII.

Just as a UDP socket may send to multiple end points, a socket may also receive from multiple end points. Thus so1 may receive data from so3 as well.

```
fconfigure $so3 -remote [list localhost $so1_port] → localhost 57012
puts -nonewline $so3 "so3->so1 (0)"                → (empty)
```

Note the use of -nonewline with all calls to puts. Like the buffering options, this is discussed in Section 18.1.5.6.

18.1.5.3. Receiving UDP datagrams

Datagrams received on a socket are received with the read command. The gets command is not recommended for reasons outlined in Section 18.1.5.6.

```
% puts "Received <[read $so1]> on so1"
→ Received <so2->so1 (0)> on so1
```

Each read returns a single datagram. Since a UDP socket may receive datagrams from multiple remote end points, we need to know the sender. This can be obtained with the -peer option to fconfigure. This will return a pair containing the remote address and port corresponding to the last read datagram.

```
chan configure $so1 -peer            → 127.0.0.1 57013
puts "Received <[read $so1]> on so1" → Received <so2->so1 (1)> on so1
chan configure $so1 -peer            → 127.0.0.1 57013
puts "Received <[read $so1]> on so1" → Received <so3->so1 (0)> on so1
chan configure $so1 -peer            → 127.0.0.1 57014
```

Our snippets above used blocking mode operation. As we described for some types of reflected channels in Section 17.3.3, UDP channels (as implemented by the tcludp package) also do not block if no data is available. They return an empty string instead. Calling fblock / chan blocked will then return 1.

```
read $so2         → (empty)
chan blocked $so2 → 1
```

As for reflected channels, it is then best to use UDP channels in event driven mode in the same fashion we described elsewhere for other channels.

18.1.5.4. Receiving broadcast datagrams

In order to receive datagrams that are broadcast on the network, the -broadcast option must be set to 1 on the UDP channel.

```
chan configure $so1 -broadcast 1 → 1
```

The above command will result in broadcast datagrams being received on that channel. Conversely, setting the option to 0 will stop reception of broadcasts on that channel.

18.1.5.5. Multicast operation

The `tcludp` package supports multicast operations for both IPv4 and IPv6 networks. To join and exit a multicast group, specify the `-mcastadd` and `-mcastdrop` configuration options respectively.

```
chan configure CHANNEL -mcastadd GROUPINFO
chan configure CHANNEL -mcastdrop GROUPINFO
```

Here *GROUPINFO* is a list of one or two elements, the first of which is the group address and the second optional element is a platform-specific network interface identifier.

The multicast groups that the channel is registered for can be obtained with the `-mcastgroups` option.

All commands return the multicast membership after execution of the command.

```
chan configure $so1 -mcastadd 224.1.1.1  → 224.1.1.1
chan configure $so1 -mcastadd 224.1.1.2  → 224.1.1.1 224.1.1.2
chan configure $so1 -mcastgroups         → 224.1.1.1 224.1.1.2
chan configure $so1 -mcastdrop 224.1.1.1 → 224.1.1.2
chan configure $so1 -mcastgroups         → 224.1.1.2
```

UDP channels have two additional options that are specifically useful for multicast operation. The `-mcastloop` option controls whether multicast datagrams **sent** on the socket are also placed on the channel's input. This option is `true` by default. The other option, `-ttl`, allows the *time to live* value to be specified. This controls the number of router hops the datagram may cross before it is dropped.

18.1.5.6. Best practices for UDP

Although not mentioned as such in the `tcludp` reference documentation, the author recommends you follow certain guidelines when using UDP channels as implemented by that package.

- Always configure the channel with `-buffering` set to none and `-translation` set to `binary`. Correspondingly, write only binary strings to the channel with any encoding etc. done explicitly via the `encoding` command.
- Do not register any channel transforms on a UDP channel.
- Use the `-nonewline` option when writing to the channel with `puts`. Each `puts` call will correspond to a single datagram.
- Receive datagrams from the channel with `read` command with no length argument specified. Each such `read` will return a single datagram.

The rationale for the above recommendations is summarized below.

UDP data transfer takes place through datagrams with are conceptually indepedent messages with clear boundaries. Tcl channels on the other hand are stream oriented where data is interpreted purely as a sequence of bytes. Channels therefore do not have, neither at the command level nor internally, any mechanisms to demarcate boundaries within the byte stream. Data written by the application is transformed, encoded, buffered and then sent on to the driver (`tcludp` in our case) as a logical stream of bytes through multiple "write" calls to the driver. There is no correspondence between the `puts` at the application level and the writes as seen by the driver. They may be merged or split in any fashion.

This then leads to the question of how are the datagram boundaries to be discerned by the `tcludp` driver. Having no other choice, it **assumes** that each write it receives from the Tcl I/O system is a single datagram. Since the application has no direct control over writes at that level, it has to somehow arrange for a single `puts` to map to a single `write`. Disabling buffering, encodings and line ending translation is the first step towards that. Even then, a `puts` call actually results in two writes from the Tcl I/O system to the channel driver with the first containing the data provided by the application and the second containing the implicit newline character appended by the `puts` command. This results in a spurious datagram containing the newline character by itself. The `-nonewline` option to `puts` avoids this problem.

On the input side, `read` is preferable to `gets` because the latter hides the boundaries between datagrams, something to which most datagram based protocols are not amenable.

18.1.6. Layered protocols

Protocols layered on TCP and UDP are supported by Tcl through additional packages and extensions. These include security protocols such as SSL/TLS as well as higher level application protocols, such as HTTP, FTP, SMTP etc. We will not cover them in this book but they are listed in Appendix A. There is also an entire book devoted to their use with Tcl — see Tcl 8.5 Network Programming.

Here is a short example of the most pervasive of these application protocols — HTTP. The supporting package, `http`, actually comes as part of Tcl itself though you do have to explicitly load it.

```
package require http → 2.8.9
```

The package only supports the client side of the HTTP protocol. The `http::geturl` command fetches a URL and returns a token to be used for further operations. It stores the results in an internal array whose elements can be retrieved with various commands.

```
% set tok [http::geturl http://www.example.com]
→ ::http::1
% http::status $tok ❶
→ ok
% http::code $tok ❷
→ HTTP/1.1 200 OK
% http::data $tok ❸
→ <!doctype html>
  <html>
  <head>
      <title>Example Domain</title>

  ...Additional lines omitted...

% http::cleanup $tok
```

❶ Whether the HTTP request completed
❷ The code returned by the HTTP server
❸ The body of the HTTP response

The package offers many other options for asynchronous operations, formatting queries, custom headers, proxy configuration and so forth.

18.2. Communication over serial ports

Before the Internet and Gigabit desktop connections, there were serial ports. In the time of dinosaurs, my son would say but the reality is that serial ports are still used in a wide variety of devices ranging from the humble RS-232 console ports still available on some servers to industrial control equipment. Tcl's channel-based I/O system includes support for communicating over these interfaces.

A channel to a serial port is opened with the same open command we described for opening disk based files.

```
open PATH ?ACCESS? ?PERMISSIONS?
```

Here *PATH* specifies the serial port. The *ACCESS* and *PERMISSIONS* parameters are exactly as we described for files in Section 9.3.2 so we will not describe them here. Similarly use of `puts` for output, `gets` and `read` for input and event driven I/O follows the same patterns we have described for the various channel types. We will therefore

stick only to those aspects like configuration options that are specific to serial ports. The discussion below assumes familiarity with serial port terminology and operation.

On Unix/Linux systems, serial port channels are opened with the by specifying a file system path that maps to a serial port. This is usually, but not necessarily of the form `/dev/ttyN` where *N* is the serial port number. On Windows systems, serial ports are specified either in the form `COMN:` where *N* is a serial port number in the range `1-9` or in the form `//./comN` where *N* is **any** serial port number. Platforms other than Windows and Unix may use some other syntax to identify serial ports.

18.2.1. Serial port speed, parity, and bit lengths

The `-mode` option to `chan configure` is used to modify or retrieve the speed (baud rate), parity bits, data bits and stop bits settings for a serial port. The argument to the option consists of 4 values separated by commas in the form *SPEED*,*PARITY*,*DATABITS*,*STOPBITS* corresponding to those four settings. The *PARITY* value should be one of `n`, `o`, `e`, `m` or `s` signifying none, odd, even, mark or space respectively. The *DATABITS* value should be a number in the range `5-8`. The *STOPBITS* value should be either `1` or `2`.

18.2.2. Serial port flow control

Serial ports may use one of several handshaking mechanisms for implementing flow control to ensure input buffers are not overrun. The mechanism used can be configured with the `-handshake` option to `chan configure`. The configured value may be one of those shown in Table 18.2.

Table 18.2. Values for handshake configuration

Value	Description
`none`	Turns off any form of flow control.
`rtscts`	Enables hardware based flow control using the RTS and CTS control signals.
`dtrdsr`	*Windows only*. Enables hardware based flow control using the DTR and DSR control signals.
`xonxoff`	Specifies software based handshake. The pair of characters used for flow control can be then be specified with the `-xchar` option. The value should be a pair of characters used to enable and disable data transfer. The default value is the ASCII standard XON/XOFF pair.

The handshake can also be implemented by directly setting the state of the RTS and DTR control lines and querying the status of the CTS, DSR, RING and DCD lines.

The state of the outgoing control lines, RTS and DTR, is controlled with the `-ttycontrol` option. It is recommended that you not directly control these with the `-handshake` option set to `rtscts` or `dtrdsr`. The value for the option is a dictionary keyed by the signal names `RTS` and `DTR` with the corresponding element value set to `0` or `1` to turn the signal off and on.

The `-ttycontrol` option also permits setting or resetting the BREAK condition on the serial port by setting the value of the `BREAK` key in the dictionary passed as the option value.

Conversely, detection of the input control signals is done with the `-ttystatus` option. This returns a dictionary with the keys `CTS`, `DSR`, `RING` and `DCD` with the corresponding values being `0` or `1` reflecting the signal states.

18.2.3. Serial port buffer and queue sizes

On Windows and Unix, the current number of bytes in the input and output queues can be obtained with the read-only option `-queue`. This returns a pair containing the number of characters currently present in the input and output queues.

Further, on Windows systems the maximum size of the input queue buffer can be set with the `-sysbuffer` option. The value a list of one or two integers. The first specifies the size of the input buffer and the second, if present, the size of the output buffer.

18.2.4. Timers related to serial ports

There are two timers that can be configured for serial port channels. The first of these is available for both Windows and Unix and set through the `-timeout` option to `chan configure`. This sets the interval after which a blocking read operation will time out. The option value is specified in milliseconds. The granularity of the timer value is platform dependent.

The other timer is only applicable to Windows systems. It is set with the `-pollinterval` option and controls the maximum time between polling for file events. **Setting this value lower than the default 10ms Tcl uses to check for all types of events will also increase the frequency of the latter.**

18.2.5. Checking for serial port errors

The read-only `chan configure` option `-lasterror` returns more detailed error information for serial ports than the standard error codes used by Tcl's file I/O commands. Examples include receive buffer overruns, framing errors, etc. See the reference documentation for possible values, their causes and potential remedies.

18.3. Chapter summary

We started with the basics of file input/output in Chapter 9, described interprocess I/O in Chapter 16, delved into advanced operations involving asynchronous I/O, channel transforms and reflected channels in Chapter 17. With the discussion of communications in this chapter, we have now concluded our discussion of Tcl's extensive I/O facilities.

18.4. References

KOC2010

Tcl 8.5 Network Programming, Kocjan, Beltowski, PACKT Publishing, 2010. Contains extensive discussion of building network-aware applications in Tcl including use of a wide variety of network related packages and libraries.

19

Virtual File Systems and Tclkits

The Virtual File System, or VFS, abstraction allows applications to view structured data as a hierarchy of directories and files even though it may not stored as such in a real file system. The data can then be operated on with the standard Tcl channel I/O commands. An example would be a VFS for a remote FTP site that would permit the application to open and perform I/O on a remote file in the same manner as a file on the local system.

Tcl's VFS framework also forms the basis of *tclkits*, which is a technology for deploying entire Tcl applications or packages as a single file without needing any installation steps or additional support files. This chapter describes the use of tclkits as well.

19.1. Using VFS

Although VFS itself is implemented within the Tcl core, accessing it at the script level requires the `tclvfs` extension. It permits both implementation of new VFS types in pure Tcl scripts as well as providing a number of VFS implementations for FTP, WebDAV, ZIP archives and others.

We first demonstrate the use of a VFS through the FTP based VFS available as part of the `tclvfs` extension.

```
% package require vfs::ftp
→ 1.0
```

The next step is to *mount* the VFS at a *mount point* which can be any file system path. By convention, `tclvfs` based packages provide a `Mount` command for this purpose.

```
Mount VFSPATH LOCALPATH
```

Here *VFSPATH* identifies the resource of interest within the VFS and *LOCALPATH* identifies a local file system path where the VFS resource will be made accessible. *LOCALPATH* need not exist and if it does, the file or directory corresponding to the path will not be accessible until the VFS is unmounted. The return value from the `Mount` command is a handle that is later required for unmounting the VFS.

In our FTP example, *VFSPATH* will identify a remote FTP directory.

```
% set mount_handle [vfs::ftp::Mount ftp://ftp.vim.org /ftp-vim]
→ 9
```

We can now treat the remote FTP directory like a local directory modulo some caveats that we list later. For example, we can even change the current directory to that location and list files `glob` or use file system inspection commands.

```
% cd /ftp-vim
% glob *
→ ftp mirror pub site vol
% file size pub/documents/published/books/stevens.netprog.errata.gz
→ 5885
```

We can open channels to files in the VFS and perform I/O on them, **even using channel transforms if desired.**

```
% set fd [open pub/documents/published/books/stevens.netprog.errata.gz rb]
→ rc6
% zlib push gunzip $fd
→ rc6
% gets $fd
→ ------------------------------------------------------
% gets $fd
→ Typos and errors found in "UNIX Network Programming"
% gets $fd
→ (Last updated March 19, 1995)
% close $fd
```

When a VFS is no longer required it should be unmounted. Again by convention each VFS supplies a command named `Unmount` to do the needful.

```
Unmount MOUNTHANDLE LOCALPATH
```

The *MOUNTHANDLE* argument passed to `Unmount` is the return value from the `Mount` command and *LOCALPATH* is the local mount point at which VFS resides. We can unmount our remote FTP directory as follows. Before we do that though, we will switch back to our original directory.

```
% cd ..
% vfs::ftp::Unmount $mount_handle /ftp-vim
→ 1
```

We could also have unmounted the VFS by calling the generic `vfs::unmount` command.

```
vfs::unmount /ftp-vim
```

In that case the mount point suffices to unmount and we do not really require the mount handle.

In general, using `cd` to change the current directory to one located in a VFS is not a good idea. It creates a "split" view of the current directory since the operating system itself is unaware of the existence of the VFS. In any case, changing directories at any time, except possibly during application startup, is a bad idea even outside a VFS since it has process-wide effect.

At this point, we really need to step back and give the Tcl I/O system a big round of applause. Consider what we just did. The "application" code treated a **compressed**, **remote FTP** resource as though it were a plain old local uncompressed file. That is pretty, well, cool!

Before moving on, there are some additional finer points to be noted about VFS.

- A VFS is mounted **process-wide**, meaning all Tcl interpreters within the process see the mount.
- All C code that uses the Tcl file system or channel API's will also see VFS mounts. This means that extensions that use these API's (and not the C runtime routines) will also benefit from being able to treat VFS resources as normal files without any additional coding. So for example, Tk could display images from a VFS exactly as it would from a local file.

By the same token, there are also some caveats,

- A VFS is invisible to the operating system, other processes and any code even within the same process that does not use the Tcl file system API's. For example, you cannot pass an executable stored in a VFS to the `exec` command to run it since the operating system is unaware of the VFS. Note however, that Tcl's `load` command is VFS-aware so you can load a shared library from a VFS. Tcl will copy the shared library to a temporary location on the local file system and pass that path to the operating system loader.

- There are unsurmountable differences between different VFS types that you have to be aware of. For example, not all may support links or may differ in terms of their case-sensitivity (consider a remote FTP VFS on Unix accessed from a Windows client).
- VFS implementations may have some performance-related aspects you need to be aware of. For example, reading even a single byte from a FTP based VFS will cause the entire remote file to be retrieved and loaded into memory.

Even with these caveats, for the most part VFS provides a very generalized way of accessing different resources with the standard Tcl file system and channel commands.

19.1.1. URL mounts

An alternative way of mounting a VFS is based on a URL type. Naturally, this is only possible for virtual file systems based on URL's like FTP or HTTP. The `vfs::urltype` package implements this functionality.

```
% package require vfs::urltype
→ 1.0
```

As before, the `Mount` command is used to mount a specific URL type. It takes a single parameter, the URL type.

```
vfs::urltype::Mount URLTYPE
```

So running the following command

```
% vfs::urltype::Mount ftp
→ Mounted at "ftp://"
```

will result in Tcl treating `ftp://` as an additional volume. We can verify this as below.

```
% file volumes
→ ftp:// C:/ D:/ E:/ F:/
% file split ftp://foo/bar
→ ftp:// foo bar
```

Now **any** FTP based URL can be treated as just another file path. So we could have written our previous example using this method as well.

```
→ set fd [open ftp://ftp.vim.org/pub/documents/published/books/stevens.netprog.errata.gz rb]
  rc8
% zlib push gunzip $fd
→ rc8
% gets $fd
→ ------------------------------------------------------
% gets $fd
→ Typos and errors found in "UNIX Network Programming"
% close $fd
```

When unmounting a `urltype` VFS, simply specify the URL type which serves as the mount handle as well.

```
% vfs::urltype::Unmount ftp
% file volumes
→ C:/ D:/ E:/ F:/
```

19.2. Implementing a VFS

We now turn our attention to implementing a new type of VFS. We will create a VFS that acts as an in-memory file system. Since our purpose is to describe the `tclvfs` interfaces, our virtual file system is very simplistic.

As for channel transforms and reflected channels, implementing a VFS involves writing a handler that implements a set of subcommands for the operations defined by the Tcl file system component. All subcommands take the same first three arguments, shown as *ROOT*, *RELPATH* and *ORIGPATH* in the table below. Here *ROOT* is VFS mount path. In our example in the previous section, this would be `/ftp-vim`. *RELPATH* is the rest of the path (relative to *ROOT*) so that *ROOT/RELPATH* is the full absolute path. *ORIGPATH* is the path as originally specified in the invocation of the Tcl I/O command. This is translated to *ROOT/RELPATH* through normalization.

The subcommands that need to be implemented are shown in Table 19.1.

Table 19.1. VFS driver subcommands

Subcommand	Description
`access` *ROOT RELATIVE ORIGPATH MODE*	Called to check if the specified access mode is compatible with permissions on the given path.
`createdirectory` *ROOT RELATIVE ORIGPATH*	Called to create a directory.
`deletefile` *ROOT RELATIVE ORIGPATH*	Called to delete the specified file.
`fileattributes` *ROOT RELATIVE ORIGPATH* ?*INDEX*? ?*VALUE*?	Called to retrieve file attribute names or set their values.
`matchindirectory` *ROOT RELATIVE ORIGPATH PATTERN TYPES*	Called to retrieve files matching the specified pattern and type.
`open` *ROOT RELATIVE ORIGPATH*	Called to open a channel to the specified file.
`removedirectory` *ROOT RELATIVE ORIGPATH RECURSIVE*	Called to delete the specified directory.
`stat` *ROOT RELATIVE ORIGPATH*	Called to retrieve file information.
`utime` *ROOT RELATIVE ORIGPATH ATIME MTIME*	Called to set the access and modification time of a file.

We will now provide an example implementation of a virtual file system. We will implement the commands shown in the table above and make use of the `vfs::filesystem mount` and `vfs::filesystem unmount` commands for mounting and unmounting our file system.

Our `memfs` package will implement an in-memory virtual file system. Mounting a `memfs` will create a new VFS instance with no content. Any content written to this file system will be erased when the VFS instance is unmounted. An application may mount multiple `memfs` VFS instances, each independent of the others.

We require the `vfs` package which exposes Tcl's VFS features at the script level. We will also need the `tcl::chan::variable` package which we will use to implement channels targeting our VFS.

```
package require vfs
package require tcl::chan::variable
```

We will choose to implement our VFS using namespaces as we did for our virtual channel example and define an ensemble corresponding to the VFS driver subcommands.

```
namespace eval memfs {
    namespace ensemble create -parameters {fs_id} -subcommands {
        access createdirectory deletefile fileattributes
        matchindirectory open removedirectory stat utime
    }
}
```

A minor point is worth noting about our namespace ensemble definition. Since we support multiple instances of our VFS, we need to pass an instance identifier to these subcommand procedures in addition to the arguments passed by the VFS core. The VFS core takes a single command prefix and appends its own arguments to it. Our instance identifier will be part of the command prefix and hence will appear **before** the subcommand. We define the ensemble accordingly with the `-parameters` option to indicate the subcommand position. See Section 12.6 for details.

19.2.1. Signalling VFS errors

Let us first deal with signalling of errors as the VFS subsystem expects implementations to use a specific call for this purpose as opposed to directly raising exceptions using the standard Tcl `error` or `throw` commands. This is to ensure a consistent set of error messages and error codes irrespective of the underlying file system.

Errors within the VFS driver should be signalled by calling the `vfs::filesystem posixerror` command and passing it a numeric code representing a POSIX error. This command will then raise a standard Tcl exception with an appropriately formatted error code. Since numeric codes are hard to remember, we will define a wrapper, `posix_error`, that will also accept the mnemonic equivalent of numeric error codes.

```
proc memfs::posix_error {err} {
    if {![string is integer -strict $err]} {
        set err [::vfs::posixError $err]
    }
    vfs::filesystem posixerror $err
}
```

So for example, we can call

```
posix_error ENOENT
```

to report an error that a file or directory does not exist.

19.2.2. Mounting and unmounting

Following `vfs` package conventions, we will name our command for mounting a `memfs` VFS as `Mount`.

```
proc memfs::Mount {mount_path} {
    set id [init_fs]
    vfs::filesystem mount $mount_path [list [namespace current] $id]
    vfs::RegisterMount $mount_path [list [namespace current]::Unmount $id]
    return $id
}
```

It takes a single argument which is the mount point where our file system will be located. It calls an internal command `init_fs` to create a new file system. The `vfs::filesystem mount` command then mounts the file system at the specified location. It has the general syntax

```
vfs::filesystem mount ?-volume? PATH CMDPREFIX
```

Here *PATH* is the mount point and *CMDPREFIX* is the command prefix that implements the VFS driver. In our example, this is the ensemble command we defined earlier which has the same name as our implementation namespace. Notice we also pass in the `memfs` instance identifier returned by `init_fs` command. This allows our implementation to distinguish between multiple `memfs` instances.

The `-volume` prefix specifies that the file system is also a new volume as seen by the `file volumes` command. We saw an example of such a VFS with the `vfs::urltype::ftp` example earlier. This option must not be specified for file systems, like `memfs` that will mount within an existing native file system path.

The call to `vfs::RegisterMount` is strictly not necessary. However, it allows the application to unmount out VFS by passing the mount point to the generic `vfs::unmount` command instead of having to call our VFS-specific `Unmount` command.

Next we implement the corresponding command for unmounting a `memfs` VFS instance.

```
proc memfs::Unmount {fs_id mount_path} {
    variable file_systems
    if {![info exists file_systems($fs_id)]} {
        return
    }
    vfs::filesystem unmount $mount_path
    unset file_systems($fs_id)
    namespace delete $fs_id
    return
}
```

The main point to note here is the call to `vfs::filesystem unmount`. The rest of the code is specific to our implementation and essentially deletes all data associated with that VFS instance. Instead of directly calling our `Unmount` command, it is recommended applications should call `vfs::unmount` so that it can update its database of mounted file systems.

In any case, applications should not call the `vfs::filesystem unmount` command themselves. That command will remove the file system from Tcl's view but will **not** notify the VFS driver that the file system has been unmounted.

19.2.3. VFS operations

We now move on to implementation of the driver commands called by the Tcl VFS subsystem for operating on a file within the VFS. The first four arguments to all these commands are identical. The first is the `memfs` VFS instance id (which we passed as part of the command prefix), the mount point path, the relative path within the VFS, and the original path as specified in the command that invoked the file operation. For example, if our VFS was mounted at `/tmp/mem` and the current working directory was `/tmp`, then a command like

```
file exists mem/foo.txt
```

would result in the arguments after the `memfs` instance id being `/tmp/mem`, `foo.txt` and `mem/foo.txt` respectively.

19.2.3.1. Checking access: `access`

We start off with implementation of the `access` command. This is used by the VFS subsystem to check if a specific file or directory can be accessed with the specified mode. The mode is passed as an additional argument in the form of a bitmask that indicates the type of desired access. If 0, only existence of the file is to be checked. The low three bits signify execute (least significan bit), write and read access respectively. Our file system does not implement access permissions and so for directories we will allow all access types. For files however, we will disallow execute access since the operating system has no knowledge of our file system and cannot execute files residing in it. (At the Tcl level, `exec` will not work.)

```
proc memfs::access {fs_id root relpath origpath mode} {
    switch -exact -- [node_type $fs_id $relpath] {
        "" { posix_error ENOENT }
        file { if {$mode & 1} { posix_error EACCES } }
        dir { }
    }
    return
}
```

The implementation uses our internal node_type command to check for whether the path is a regular file or directory. If the specified access is allowed the command returns normally with the return value being immaterial. If access is **not** allowed, the command must raise a POSIX error as discussed previously. In our case, we return ENOENT when the path does not exist and EACCES when it does not have the requested execute access permission.

19.2.3.2. Creating directories: createdirectory

Next we implement the createdirectory call which is straightforward. The command is expected to create a new directory at the specified path within our file system if it does already exist. If the path corresponds to an existing regular file, it should raise a POSIX error.

```
proc memfs::createdirectory {fs_id root relpath origpath} {
    node_add_dir $fs_id [node_find $fs_id $relpath dir]
}
```

The implementation uses two internal commands. The first of these, node_find, maps a file path to the location key for the corresponding node in our internal file system structures. The node itself need not exist but if it does, the optional third argument mandates that it must be of the specified type. Thus in the above call, node_find would raise a POSIX error if the node existed and was not a directory. We will see node_find used throughout our implementation.

The other internal command, node_add_dir simply creates an empty directory and associated structures at a specified node location.

19.2.3.3. Deleting directories: removedirectory

The removedirectory command deletes a directory. It takes an additional argument which must be a boolean value. If true then the directory and its contents should be recursively deleted; otherwise the command should raise a POSIX error if the directory is not empty. It is not an error if the directory does not exist.

```
proc memfs::removedirectory {fs_id root relpath origpath recursive} {
    node_del_dir $fs_id [node_find $fs_id $relpath dir] $recursive
}
```

The implementation is pretty much identical to that of createdirectory above except we call node_del_dir which does the hard work of deleting the directory structure and its contents.

19.2.3.4. Creating and opening files: open

The open VFS driver subcommand essentially has to implement the file system level operations of the Tcl open and chan open commands which applications use to create files as well as open them for I/O. The command takes two additional arguments that specify the access mode and permissions in the case of creating a new file. These are equivalent to the ones passed to the Tcl open command.

```
proc memfs::open {fs_id root relpath origpath mode perms} {
    variable file_systems

    set node_key [node_find $fs_id $relpath file]
    set exists [expr {[node_type $fs_id $relpath] ne ""}]
    set truncate 0
    switch -glob -- $mode {
        "" -
        r* {
            if {! $exists} {
                posix_error ENOENT ❶
            }
        }
        a* -
        w* {
```

```
            if {$exists} {
                if {[string index $mode 0] eq "w"} {
                    set truncate 1
                }
            } else {
                node_add_file $fs_id $node_key
            }
        }
        default {
            error "Unsupported mode \"$mode\""
        }
    }
    set chan [node_add_channel $fs_id $node_key $truncate]
    set close_callback [list [namespace current]::node_close_handler  \
                            $fs_id $node_key $chan]
    return [list $chan $close_callback]
}
```

❶ File must exist

The implementation of open is a little longer than others but should be straightforward to follow at this stage. We have already described the `node_find` and `node_type` internal commands. The `mode` parameter, as for the Tcl open, specifies the open mode in the form of `r`, `r+`, `w` etc. and the `switch` statement takes appropriate action depending on the mode. The `node_add_file` command creates a new node of type file in our file system structure.

The return value of the command should be a list of one or two elements. The first element must be the handle to an open channel to use for performing I/O on the file. The second element is optional. If present it should be command prefix to be invoked when the channel is closed. Our internal `node_add_channel` command creates a new channel to a specified file (node). It also adds the channel to the list of channels internally associated with the file since we want to prevent files from being deleted while they have channels open to them. For the same reason, we also need to know when a channel is closed so that we can remove it from this list. Thus we make use of the second optional element in the return value to declare a callback to be invoked on channel close. This callback `node_close_handler` will remove the channel from the channel list and also update the access and modification time stored for the node (file).

19.2.3.5. Deleting files: `deletefile`

The `deletefile` command is almost identical to the `removedirectory` command we implemented above except it deals with removal of files, not directories, and hence has no need for recursion control.

```
proc memfs::deletefile {fs_id root relpath origpath} {
    node_del_file $fs_id [node_find $fs_id $relpath file]
}
```

A design decision we have made is that files with open channels to them cannot be deleted. This follows the Windows model (as opposed to Unix). Our `node_del_file` implementation will report an error via `posix_error` on an attempt to delete a file which is open. Naturally, this decision affects `removedirectory` as well.

19.2.3.6. Setting file timestamps: `utime`

The `utime` command is called to set the last access and modification timestamps for a file or directory. It takes two additional arguments corresponding to the access and modification time respectively. These are specified in terms of the number of seconds since the epoch, January 1, 1970.

```
proc memfs::utime {fs_id root relpath origpath atime mtime} {
    node_set_times $fs_id [node_find $fs_id $relpath] $atime $mtime
}
```

The internal commands in our implementation set these timestamps for various operations. For example, creation of a file will also update the modification time for its parent directory. The `utime` command is specifically called by Tcl in response to the `file atime` and `file mtime` commands.

19.2.3.7. File statistics: `stat`

The `stat` command is returns information about a file or directory. The return value from the command should be a dictionary with the following keys: `dev`, `ino`, `mode`, `nlink`, `uid`, `gid`, `size`, `atime`, `mtime`, `ctime` and `type`. These keys have exactly the same semantics we described for the `file stat` command.

```
proc memfs::stat {fs_id root relpath origpath} {
    return [node_stat $fs_id [node_find $fs_id $relpath]]
}
```

Our node implementation maintains the relevant information internally and our driver API can again just delegate to it.

19.2.3.8. File attributes: `fileattributes`

As discussed in the Files and Basic I/O chapter, files can be associated with certain attributes that are specific to the file system. These attributes are managed with the `file attributes` commmand. VFS drivers need to implement a corresponding `fileattributes` command to allow Tcl to access attributes for files within the VFS. The command is invoked in three forms:

- If no additional arguments (other than the standard ones) are specified, the command should return a list of attribute names supported by the file system. **Every such call must return the same names in the same order.**
- If a single additional argument is specified, it will be an integer index into the list of attribute names. The command should return the value of this attribute.
- If two additional arguments are specified, the first is the integer index into the attribute list as above. The second is the value to assign to the attribute.

Our implementation is shown below.

```
proc memfs::fileattributes {fs_id root relpath origpath args} {
    set attr_names [lsort [node_attr_names]]
    if {[llength $args] == 0} {
        return $attr_names
    }
    set node_key [node_find $fs_id $relpath]
    set attr_name [lindex $attr_names [lindex $args 0]]
    if {[llength $args] == 1} {
        return [node_attr $fs_id [node_find $fs_id $relpath] $attr_name]
    } else {
        return [node_attr $fs_id [node_find $fs_id $relpath] $attr_name [lindex $args 1]]
    }
}
```

As always, it relies on the underlying internal functions to to the actual work. The `node_attr_names` command returns a list of attribute names supported by our VFS. We ensure we always pass them back in the same order by sorting them before returning. We then use the `node_attr` command to get or set the attribute on the node corresponding to the specified path.

Although not shown above, our VFS supports two attributes: `-contenttype` and `-encoding`. Applications can assign any values they want to these attributes as VFS has no interest or control over their semantics. The **intended** use is for applications to store the encoding used for the file content in the `-encoding` attribute and the content type (similar to the `Content-Type` HTTP header) which specifies the format of the content like `text/html`

in the `-contenttype` attribute. Note these are advisory attributes and have to be explicitly set by the application There is no way for the VFS to detect the encoding in use or the type of content stored in the file.

19.2.3.9. Matching files: `matchindirectory`

One final VFS driver command remains to be implemented. The `matchindirectory` command is what Tcl's file system calls to retrieve directory contents. The return value of the command is expected to be a (possibly empty) list containing file names. Commands like `glob` also support matching based on patterns and file types. Correspondingly, the `matchindirectory` command takes two additional arguments that specify a glob pattern and a file type specifier.

If the glob pattern is the empty string, the `relpath` argument is the relative path of a file or directory. The command should then only check if the path exists **and** is of the specified type. If so, it should return a list containing the corresponding **original path** that was specified on the command line, not the relative path. If the path does not exist or is not of the specified type, an empty list is returned.

If the glob pattern is not the empty string, the `relpath` is always a path to an existing directory whose contents are to be matched against the pattern. The command should then return the names that match the pattern and the specified type.

```
proc memfs::matchindirectory {fs_id root relpath origpath pat type} {
    variable file_systems
    if {[string length $pat] == 0} {
        set file_type [node_type $fs_id $relpath]
        if {($file_type eq "dir" && [::vfs::matchDirectories $type]) ||
            ($file_type eq "file" && [::vfs::matchFiles $type])} {
            return [list $origpath]
        } else {
            return {}
        }
    }

    if {[node_type $fs_id $relpath] ne "dir"} {
        return {}
    }

    set node_key [node_find $fs_id $relpath]
    set matches {}
    if {[::vfs::matchDirectories $type]} {
        foreach name [node_subdirs $fs_id $node_key] {
            if {[string match $pat $name]} {
                lappend matches [file join $origpath $name]
            }
        }
    }
    if {[::vfs::matchFiles $type]} {
        foreach name [node_files $fs_id $node_key] {
            if {[string match $pat $name]} {
                lappend matches [file join $origpath $name]
            }
        }
    }
    return $matches
}
```

Our implementations makes use of two internal commands `node_files` and `node_subdirs` that return the names of files and subdirectories under the specified node. These are then matched against the pattern to construct the returned list.

The `type` argument specifies whether the returned list should include files, directories or both. We treat it as opaque and use the utility commands `vfs::matchDirectories` and `vfs::matchFiles` to ascertain whether

entries of a specific type are to be included or not. Note that the type argument can indicate that **both** files and directories are to be included.

The Tcl `glob` command allows the type argument to limit matching files based on other criteria as well such as permissions. These criteria are not exposed for VFS systems.

The `vfs` package provides another utility command `vfs::matchCorrectTypes` that serves as an alternative to `matchDirectories` and `matchFiles`.

```
vfs::matchCorrectTypes TYPES FILELIST ?DIR?
```

The command returns a list of names from *FILELIST* that fulfil the type requirements specified by *TYPES*. If *DIR* is not specified, *FILELIST* must contain absolute paths. If *DIR* is specified, it must be a directory and *FILELIST* should be the list of file and directory names within that directory. Depending on your internal file system implementation, you may find `matchCorrectTypes` more convenient to use.

19.2.3.10. `memfs` internals

Due to space limitations, we will not go into detail regarding our node_* commands that implement our VFS internals. You can download the `memfs.tcl` file from the book's web site to see the implementation.

Time to see if our VFS actually works.

```
% package require memfs
→ 1.0
% memfs::Mount /mem
→ 1
% file mkdir /mem/dir
% set fd [open /mem/dir/foo.txt w]
→ rc24
% puts -nonewline $fd "It lives!"
% set enc [fconfigure $fd -encoding]
→ cp1252
% close $fd
% glob /mem/dir/*
→ C:/mem/dir/foo.txt
% file attribute /mem/dir/foo.txt -contenttype text -encoding $enc
```

It appears the file was created. Let us read it back. We stored the encoding used to write it as a file attribute. So we will configure the channel accordingly while reading.

```
% set fd [open /mem/dir/foo.txt]
→ rc25
% fconfigure $fd -encoding [file attribute /mem/dir/foo.txt -encoding]
% read $fd
→ It lives!
```

An attempt to delete the file should fail as we have the file open. Retrying after closing the file should allow the delete operation to succeed.

```
file delete /mem/dir/foo.txt Ø error deleting "/mem/dir/foo.txt": permission denied
file exists /mem/dir/foo.txt → 1
close $fd                    → (empty)
file delete /mem/dir/foo.txt → (empty)
file exists /mem/dir/foo.txt → 0
```

Finally we verify we can unmount our file system.

```
% vfs::unmount /mem
% file exists /mem
→ 0
```

It's all good! Ship it!

19.3. VFS introspection

The set of current VFS mounts can be retrieved with the `vfs::filesystem info` command. With no arguments, the command returns the list of mount points.

```
% memfs::Mount /mem
→ 1
% vfs::ftp::Mount ftp://ftp.vim.org /ftp-vim
→ 0
% vfs::filesystem info
→ C:/ftp-vim C:/mem
```

If the optional argument is specified, it must be a mount point path. In this case the command returns the VFS driver command prefix that will be invoked to handle requests to that file system.

```
% vfs::filesystem info /mem
→ ::memfs 1
% vfs::filesystem info /ftp-vim
→ vfs::ftp::handler 0 {}
```

19.4. Single file deployment: Tclkit, Starkit, Starpack

Except in the simplest cases, deploying an application or even a large library package involves distribution of

- The Tcl interpreter itself, e.g. `tclsh` or a custom built executable
- The Tcl script files comprising the application
- Any packages, modules and binary extensions required
- Support files such as icons and other resources

The presence of multiple files in a directory hierarchy means deployment cannot be a simple copy and run operation. The files have to combined into some archive or installation format for distribution and then unpacked or installed on the target system. Although this can be an inconvenience, there is another more irksome issue for the user. If multiple unrelated Tcl applications are installed, there is potential for interference between the applications. Each may require different versions of Tcl and libraries, expect different settings in environment variables like `TCLLIBPATH` and so on.

The Tclkit technology is a solution that makes deployment as simple as copying a single file and running it with no additional steps required. Moreover, the application is completely self-contained and will have no interference with other Tcl installations including other Tclkit based applications.

Single file deployment alternatives

There are several alternative solutions for single file deployment similar to that of Tclkit. Here we describe Tclkit because it is probably the most widely used and also the one with which the author is most familiar. However, other solutions may have features not present in Tclkit based applications. For example, Freewrap can optionally encrypt the contents of the deployed file.

19.4.1. Tclkits, starkits, starpacks

A *starkit* (**St**and**a**lone **R**untime) is a way to package an entire directory hierarchy and its contents into a single file. Conceptually starkits are similar to file archival formats such as `tar` or `zip` and in fact there are starkit variations based on those archive formats as well. What sets the starkit apart is some additional internal structure that allows a starkit to be mounted as a VFS.

The internal format of starkit archives was originally based on a database called Metakit. There are now also variations that use other formats like the above-mentioned zip format. For our purposes, the internal formats are immaterial for the most part and we will refer to them collectively as starkits.

The `tclkitsh` and `tclkit` applications are specially enhanced versions of `tclsh` and `wish` respectively that understand the format of Metakit-based starkits. The other formats of starkits have their own corresponding applications. For example, the applications that understand the Vlerq format are `tclkit-cli` and `tclkit-gui`. All these applications, which we will collectively refer to as *tclkits*, mount the starkit file as a VFS.

The tclkit programs can also be used in place of the `tclsh` and `wish` shells. See Section 19.4.3.

Starkits and tclkits together comprise a two-file solution for distributing Tcl applications. So, for example, an application packaged as a starkit `myapp.kit` can be run as

```
tclkitsh myapp.kit ARG ...
```

However, we can go a step further. The tclkit and starkit can be further combined into a **single** executable file, termed a *starpack*. This is a self contained executable holding the Tcl interpreter and libraries as well as as the application code with its supporting files. A starpack `myapp` (`myapp.exe` on Windows) constructed from the `tclkitsh` and `myapp.kit` above can be executed simply as

```
myapp ARG ...
```

19.4.2. Obtaining a tclkit

Tclkits are available from multiple sources on the Internet. We list only some of the more popular ones below. Although originally based the same technology, they have some differences in their internal structure and build systems.

19.4.2.1. Downloading prebuilt tclkits

The easiest way to obtain a prebuilt tclkit is to download the appropriate version for your operating system platform from one of the locations below.

- The KitCreator[1] project
- The Kitgen Build System[2] (KBS) project
- The ActiveState[3] distribution includes tclkit binaries. They are termed *basekits* and available in the `bin` directory of an ActiveTcl installation.

19.4.2.2. Building tclkits

Both KitCreator and KBS also provide build scripts that allow you build your own tclkit executables in case the prebuilt binaries do not support your platform or you want a custom version with a different internal format or

[1] http://tclkits.rkeene.org
[2] https://sourceforge.net/projects/kbskit/files/kbs
[3] http://tcl.activestate.com

additional packages. Both download the required source files from their repositories. Building a tclkit then simply involves invoking the appropriate script, `kitcreator` in the case of KitCreator and `kbs.tcl` in the case of KBS.

KitCreator has two additional features:

- It supports cross-compiling a tclkit for a different target platform than the one on which the build system is running.
- It has an online build system[4] where you can specify through a Web interface one of almost two dozen target platforms and select any additional extensions desired. It will then build a customized tclkit for download.

Both KitCreator and KBS are based on the GNU toolchain. For building on Windows you will need to install MinGW[5] or MinGW-w64[6]. Alternatively, for Microsoft Visual C based builds, you may prefer to instead use the original kitgen[7] system. In addition to the GNU toolchain, this also provides `nmake` based makefiles suitable for Visual C builds. See the `README` file at the toplevel for instructions.

Our examples below assume we are running a downloaded tclkit that we have renamed as `tclkit-cli.exe`.

19.4.3. Using tclkits as Tcl shells

Tclkit binaries can be for the most part used in place of the standard Tcl shells `tclsh` and `wish`. In particular,

- when run without arguments, they will display an interactive prompt and execute any commands entered
- you can pass a Tcl script file to run and additional arguments on the command line just as for the standard shells

These characteristics make it very convenient to use tclkit interpreters on systems where Tcl is not installed. You can copy a single executable and gain full use of a Tcl command shell.

There are however some differences.

- In addition to Tcl script files, tclkit programs will also execute starkits as we explore in a bit. As of Tcl 8.6, the standard shells do not have the requisite VFS drivers built-in and will not recognize the starkit formats.
- Because tclkits are self contained, they do not examine environment variables like the `TCLLIBPATH` environment variable when setting up the `auto_path` variable for locating packages.
- Some tclkit variations do not include a full set of time zone and character encoding data. If this matters to your application, it is simply a matter of downloading or building one that does include this data.

19.4.4. The `sdx` tool

As we described previously, a starkit is a directory tree and its content packaged as a single file. Although a starkit can be constructed using Tcl's base VFS facilities, most commonly the `sdx` tool is used for this purpose. This wraps all the initialization and low level operations required to build a starkit into a set of high level commands callable from a command line. You can download it from several Tcl sites, for example https://chiselapp.com/user/aspect/repository/sdx/index.

Because `sdx` is itself packaged as a starkit, you can only run it using using a tclkit, not with `tclsh`.

The `sdx` tool comes with a built in help system. The general syntax for running the tool is

```
tclkit-cli sdx.kit ?SDX COMMAND? ?COMMAND ARGS?
```

If no arguments are provided, it will print a summary of available commands.

[4] http://kitcreator.rkeene.org/kitcreator
[5] http://www.mingw.org
[6] https://mingw-w64.org
[7] https://github.com/patthoyts/kitgen

```
c:\temp\tclkit-demo> tclkit-cli sdx.kit
→ Specify one of the following commands:
  addtoc eval fetch ftpd httpd httpdist ls lsk md5sum mkinfo mkpack mkshow mksplit mkzipkit
   ↳ qwrap ratarx rexecd starsync sync tgz2kit treetime unwrap update version wrap
  For more information, type:  sdx.kit help ?command?
```

The help command offers more detailed information for each command.

```
c:\temp\tclkit-demo> tclkit-cli sdx.kit help qwrap
→
   Quick-wrap the specified source file into a starkit

          Usage: qwrap file ?name? ?options?

          -runtime   file   Take starkit runtime prefix from file

          Generates a temporary .vfs structure and calls wrap to create
          a starkit for you.  The resulting starkit is placed into file.kit
          (or name.kit if name is specified). If the -runtime option is
          specified a starpack will be created using the specified runtime
          file instead of a starkit.

          Note that file may be a local file, or URL (http or ftp).
```

You will notice the tool comes with wide-ranging functionality, even including a basic FTP and Web server. Our discussion will be limited to the functionality related to the topic at hand — working with starkits and starpacks.

19.4.5. Building a single script starkit: sdx qwrap

Let us start with the simplest possible example — building a starkit from a single script. We will create a starkit containing the ubiquitous Hello World! program. We first create the file containing our script.

```
% write_file hello.tcl {puts "Hello World!"}
```

We now convert this to a starkit using the sdx qwrap command. At the Windows command prompt,

```
c:\temp\tclkit-demo> tclkit-cli sdx.kit qwrap hello.tcl
→ 5 updates applied
```

We see that a starkit hello.kit has been created.

```
c:\temp\tclkit-demo> dir /b *.kit
→ hello.kit
  sdx.kit
```

We can run the created starkit with the tclkit-cli application.

```
c:\temp\tclkit-demo> tclkit-cli hello.kit
→ Hello World!
```

Now, this obviously not very different from running

```
c:\temp\tclkit-demo> tclsh hello.tcl
→ Hello World!
```

But hold on before you thumb your nose at this. We will see next how this can then be used to build a fully self-contained executable.

We will also see later that the starkit is not constrained to contain a single file. An entire directory structure with multiple packages, extensions, and auxiliary files comprising complete application can be included within the starkit.

19.4.6. Building a single script executable: sdx qwrap -runtime

Distributing a starkit implies also distributing a tclkit executable or having the end user obtain it from somewhere. We can do better by combining the tclkit executable and the starkit into a single executable file — a *starpack*.

First, we need to make a copy of our tclkit executable. We will go into the reasons for this when we discuss the structure of tclkits.

```
c:\temp\tclkit-demo> copy tclkit-cli.exe tclkit-cli-runtime
→         1 file(s) copied.
```

We then run the sdx qwrap program again but this time with the -runtime option.

```
c:\temp\tclkit-demo> tclkit-cli sdx.kit qwrap hello.tcl hello.exe -runtime tclkit-cli-runtime
→ 872 updates applied
  4 updates applied
```

And voila, we now have a hello program, our very first starpack!

```
c:\temp\tclkit-demo> dir /b hello.*
→ hello
  hello.kit
  hello.tcl
```

Notice a quirk of the sdx qwrap command. **Even on Windows the executable file is created without an** exe **extension.** So if you are on that platform, it needs to renamed appropriately.

```
c:\temp\tclkit-demo> rename hello hello.exe
```

We now have a fully functional single file executable that comprises our entire application.

```
c:\temp\tclkit-demo> hello
→ Hello World!
```

The convenience of this cannot be overstated. Deployment consists of copying a file to the target system and installation is a no-op. And as we will see as we proceed through the chapter, these benefits are not limited to toy applications implemented in a single file.

19.4.7. Internal structure of a starkit

The sdx utility's lsk command allows us to inspect the internal structure of a starkit.

```
c:\temp\tclkit-demo> tclkit-cli sdx.kit lsk hello.kit
→
  hello.kit:
                            dir  lib/
          80  2017/07/04 11:45:57  main.tcl

  hello.kit/lib:
```

```
                              dir  app-hello/

  hello.kit/lib/app-hello:
          54  2017/07/04 11:45:57  hello.tcl
          79  2017/07/04 11:45:57  pkgIndex.tcl
```

Moreover, the `lsk` can even list the contents of a starkit embedded in a starpack.

```
c:\temp\tclkit-demo> tclkit-cli sdx.kit lsk hello.exe
→
  hello.exe:
        5114  2013/02/16 22:53:30  boot.tcl
          37  2013/02/16 22:53:30  config.tcl
                              dir  lib/
          80  2017/07/04 11:45:57  main.tcl
       57022  2013/02/16 22:53:30  tclkit.ico

  hello.exe/lib:
                              dir  app-hello/
...Additional lines omitted...
```

Notice the contents of the starpack are much larger since it includes not just the `hello` application but also the Tcl runtime. The format of the output should give you a hint that starkits are structured just like file systems and in fact are implemented as a VFS.

We can proceed to examine the internal contents in one of two ways. The first is through the usual VFS and Tcl I/O commands. The second is by extracting the contents of the startkit with the `sdx unwrap` command.

Let us first demonstrate the former, primarily to prove that the starkit is accessible as a VFS. We need to use the `tclkit-cli` application, and not `tclsh` for this as the latter does not by default have the requisite VFS drivers. From within `tclkit-cli` we first load the Metakit VFS driver and then mount the starkit.

```
% package require vfs::mk4
→ 1.10.1
% vfs::mk4::Mount hello.kit /hello
→ mkclvfs1
% glob /hello/*
→ C:/hello/main.tcl C:/hello/lib
```

We see that top level directory of the VFS has two entries, `main.tcl` and `lib`. We can access any file within the VFS with the standard Tcl I/O commands.

The second way to examine the content of a starkit (or starpack) is by extracting its contents with the `sdx unwrap` command.

```
c:\temp\tclkit-demo> tclkit-cli sdx.kit unwrap hello.kit
→ 5 updates applied
```

This will extract the contents of the starkit into a local directory `hello.vfs`.

```
c:\temp\tclkit-demo> dir /s /b hello.vfs
→ C:\temp\tclkit-demo\hello.vfs\lib
  C:\temp\tclkit-demo\hello.vfs\main.tcl
  C:\temp\tclkit-demo\hello.vfs\lib\app-hello
  C:\temp\tclkit-demo\hello.vfs\lib\app-hello\hello.tcl
  C:\temp\tclkit-demo\hello.vfs\lib\app-hello\pkgIndex.tcl
```

We can take a peek at the content of the `main.tcl` file.

```
c:\temp\tclkit-demo> type hello.vfs\main.tcl
→
    package require starkit
    starkit::startup
    package require app-hello
```

We will leave the details about the commands in `main.tcl` for the next section where we build a more complete example of a starkit. For the moment we will comment on some general points points about the tclkit structure.

The root of the VFS contains a file `main.tcl`. This file will be sourced by the tclkit application when the starkit is loaded. The `main.tcl` file above was automatically created by the `sdx qwrap` command. There is no requirement that this file have exactly the contents shown. It could contain any sequence of commands or even an entire application.

The only mandated requirement for starkits is the existence of this `main.tcl` file at the root of the starkit VFS. The rest of the starkit VFS may be structured in any fashion you choose.

The structure of our `hello` starkit is the boilerplate used by `sdx qwrap` for its automatically generated starkits. It has a `lib` directory beneath which it expects all packages to be placed. The `starkit::startup` command in `main.tcl` will add the directories below this to the `auto_path` variable. In the case of `sdx qwrap`, there is no option for including additional packages so this directory contains only a single subdirectory `app-hello`.

This directory contains our `hello.tcl` script **converted to a package form** and loaded with the `package require` command in `main.tcl`. To convert our script to a package, `sdx qwrap` adds the `pkgIndex.tcl` file

```
c:\temp\tclkit-demo> type hello.vfs\lib\app-hello\pkgIndex.tcl
→
    package ifneeded app-hello 1.0 [list source [file join $dir hello.tcl]]
```

and modifies our script to include `package provide` command.

```
c:\temp\tclkit-demo> type hello.vfs\lib\app-hello\hello.tcl
→ package provide app-hello 1.0

  puts "Hello World!"
```

We reiterate at this point that this structure is completely optional. If we were to manually structure the VFS, something we illustrate in the next section, we could have just put our script file in the VFS root directory and directly sourced it instead of converting it to a package. Or we could have included our script within `main.tcl` itself. The structure used by `sdx qwrap` is reflective of the conventions used when a `starkit` is used to deploy multiple packages and more complex applications.

19.4.8. A more complete starkit example: `sdx wrap`

Having seen the creation of a starkit from a single script, let us now create a more complete example. Our demo starkit will be multifunctional: it will include the `sequences` package from Section 13.3.8 as well as the standard `Hello World!` functionality. We foresee great demand for this combination.

Our demo has certain operational requirements:

- The starkit must be usable as a library where it can be loaded in the main application.
- It should also be usable as a standalone application itself when it is passed as the script argument to a tclkit application. In that case it should run the command specified by the user.
- It should be deployable as a single file executable.
- For ease of development, we should be able to run it in normal fashion even when it is not wrapped into a starkit. That way we can edit the files during development and re-test without having to build a starkit after every change.

We will follow the same basic structure as was created by sdx qwrap. At the top level, we have the demo.vfs directory which will be the root of our starkit's virtual file system. The directories and files below this are shown below.

```
c:\demo-dir> ls -R demo.vfs
demo.vfs:
hello.tcl  lib  main.tcl

demo.vfs/lib:
app-demo  sequences

demo.vfs/lib/app-demo:
demo.tcl  pkgIndex.tcl

demo.vfs/lib/sequences:
pkgIndex.tcl  seq_arith.tcl  seq_geom.tcl
```

We will start with the mandatory main.tcl file in the root of the starkit VFS.

```
# main.tcl
namespace eval demo {}
package require starkit
set demo::run_mode [starkit::startup]
set demo::vfs_root [file dirname [file normalize [info script]]]

package require sequences
source [file join $demo::vfs_root hello.tcl]

puts "run_mode: $demo::run_mode"

switch -exact -- $demo::run_mode {
    sourced { }
    unwrapped -
    starkit -
    starpack {
        package require app-demo
    }
    default {
        error "Unknown run mode $demo::run_mode"
    }
}
```

The scripts creates a namespace demo for its own use and then loads the starkit package that implements the required VFS drivers built into a tclkit application. The next command is a call to starkit::startup which has two effects that are directly relevant to us:

- It initializes the auto_path variable used for loading packages to the lib directory in the VFS. You are of course free to further modify auto_path as appropriate for your application. For example, you might have another directory as a sibling of lib that you want to add to the package path.
- It returns a value that indicates how the starkit is being used. We use this to determine whether our starkit should behave as a bundle of packages or a standalone application. The possible return values are shown in Table 19.2.

Specific tclkit variations may return values other than those shown in the table; for example, service indicating the starkit is running as a Windows service. See the documentation for your tclkit for these additions.

Table 19.2. Starkit start-up modes

Mode	Description
unwrapped	Indicates that the main.tcl is **not** part of a starkit and is being read in as a regular Tcl script. As we will see below, it is convenient during development for the application to be in "unwrapped" form as a set of Tcl scripts instead of a monolithic single-file starkit.
sourced	Indicates that the starkit was loaded with the source command either from the application or another starkit. This would generally indicate that the starkit is not itself the main application.
starkit	The starkit is the main file being run by the tclkit application from the command line and as such should provide the application functionality.
starpack	The starkit is bound to the tclkit executable thereby comprising a single-file application.

We will see all these modes in our example scenarios below.

The script then preloads our starkit functionality. It loads the sequences package which is placed under the lib directory and therefore found via auto_path. We chose not to implement hello as a package so the script simply sources it. Note that instead of preloading these, we could have chosen to omit these lines and thereby leaving it up to the application to explicitly load them.

For the purposes of our demonstration, we then print out the mode that the starkit is running in.

Finally, the main script checks the mode. A value of sourced indicates that the starkit is being loaded as a library. Nothing more needs to be done in this case as we have already loaded the contained functionality. All other values of the mode indicate the the starkit should run as an application. It then loads our application code which is implemented, following starkit conventions, as a package.

Our Hello World! functionality is our simple script from the previous section written as a procedure.

```
# hello.tcl
proc hello {} {
    puts "Hello World!"
}
```

Finally, we come to our application code. It is simple enough that we just present it here without any explanation. Our example usage later will clarify working if needed.

```
# demo.tcl
package provide app-demo 1.0
package require sequences

if {[catch {
    switch -exact -- [lindex $argv 0] {
        hello { hello }
        arith { seq::arith_term {*}[lrange $argv 1 end] }
        geom { seq::geom_term {*}[lrange $argv 1 end] }
        default {
            error "Unknown or missing command: must be hello, arith, geom"
        }
    }
} result]} {
    puts stderr $result
    exit 1
} else {
    if {$result ne ""} {
        puts stdout $result
    }
}
```

We are now ready to actually build a starkit from our demo application. The sdx utility's wrap command will create a starkit from a specified directory tree.

```
tclkit-cli sdx.kit wrap NAME ?options?
```

The sdx wrap command takes several options but we only describe basic usage here. It creates a starkit named *NAME* whose contents are created from a directory of the same name but with a .vfs file extension.

```
c:\demo-dir> tclkit-cli sdx.kit wrap demo -interp tclkit-cli
10 updates applied

c:\demo-dir> ls
demo  demo.bat  demo.vfs  sdx.kit  tclkit-cli-runtime  tclkit-cli.exe
```

The command creates two files, the starkit demo and a Windows batch file demo.bat. The latter is simply a Windows batch file that invokes the tclkit application passing it the starkit that was created.

```
@tclkit-cli demo %1 %2 %3 %4 %5 %6 %7 %8 %9
```

On Windows we can then run the starkit through the demo batch file. On Unix systems, the starkit can be directly run because it begins with the header

```
exec tclkit-cli "$0" ${1+"$@"}
```

causing Unix shells to run the starkit by passing it to tclkit-cli.

The purpose of the -interp option when creating the starkit is to specify which tclkit application should be used to run the starkit when the starkit is directly invoked in the Windows or Unix command shell. We specified it as tclkit-cli as that is the tclkit application we are using. If unspecified, it would default to tclkit.

The starkit can be run using **any** tclkit application by explicitly passing it as the command line argument. The use of the -interp option only applies to the case where the starkit is specified as the program name in the shell command line.

We are now ready to try out our application in various modes. First, we will use it as a package bundle in interactive mode.

```
c:\demo-dir> tclkit-cli
% source demo
run_mode: sourced
% hello
Hello World!
% seq::arith_term 2 3 4
11
```

Notice the run mode is printed as sourced.

Next, we will try running as an unwrapped application.

```
c:\demo-dir> tclkit-cli demo.vfs/main.tcl hello
run_mode: unwrapped
Hello World!

c:\demo-dir> tclkit-cli demo.vfs/main.tcl arith 2 3 4
run_mode: unwrapped
11
```

This allows us to edit and modify the files in the `demo.vfs` tree and test it without having to rebuild the application in test mode.

In production, the starkit will be directly invoked as the application either via passing it to the tclkit application,

```
c:\demo-dir> tclkit-cli demo hello
run_mode: starkit
Hello World!
```

or alternatively via the batch file (on Windows) or directly invoking the starkit (on Unix)

```
c:\demo-dir> demo arith 2 3 4
run_mode: starkit
11
```

The final variation for running our application is as a single-file executable. Like `sdx qwrap`, `sdx wrap` takes a `-runtime` option that specifies a tclkit application to use as the runtime for our starkit. The two are then bound together in a single executable.

We will use a slight variation of `sdx wrap` that uses the `-vfs` option.

```
c:\demo-dir> tclkit-cli sdx.kit wrap myapp.exe -vfs demo.vfs -runtime tclkit-cli-runtime
872 updates applied
9 updates applied

c:\demo-dir> ls
demo  demo.bat  demo.vfs  myapp.exe  sdx.kit  tclkit-cli-runtime  tclkit-cli.exe
```

This creates an starpack `myapp.exe`. The `-vfs` option specifies the VFS directory as `demo.vfs` and not `myapp.vfs` which would be the default based on the starpack name.

We now have single file application that can be copied to any Windows system and run without any additional steps.

```
c:\demo-dir> myapp hello
run_mode: starpack
Hello World!

c:\demo-dir> myapp arith 2 3 4
run_mode: starpack
11
```

19.4.9. Considerations for multiplatform starkits

Let us take a moment to talk about issues related to multi-platform support. Starkits which are purely script based and have no binary extensions in their content, can be loaded by any tclkit application on any platform. This is true irrespective of the platform on which the starkit was built. Thus the demo starkit we built on our Windows system would work just as well on OS X or Linux.

Starkits that contain binary extensions are also portable across platforms as long as they follow the appropriate structure for packages that we describe in Section 13.7. The `load` command for loading binary extensions will recognize that the extension is in starkit and copy it out to the local file system from where it can be loaded by the OS loader.

Starpacks are platform-specific executables and, by their very nature, are not portable. You need to build a separate starpack for every platform for which you want to deploy a single-file executable.

Even though starpacks are platform-specific, they do not need to be built on their native platform. All you need to do is specify the appropriate tclkit for the target platform as the value of the `-runtime` option to `sdx wrap`. For example, we could build a x86 Linux-specific version of our `myapp` application by providing a x86 Linux tclkit.

```
c:\demo-dir> tclkit-cli sdx.kit wrap myapp.exe -vfs demo.vfs -runtime \
    tclkit-cli-runtime-linux
```

Here we assume `tclkit-cli-runtime-linux` is a tclkit application that has been built for our target Linux platform.

19.4.10. Starkit mount points

We need to touch upon one issue that you need to be aware of when using starkits and starpacks. As we discussed, starkits are seen by the Tcl I/O system as virtual file systems and as such have to be mounted. The mount point used for the VFS in starkits and starpacks is the local file system path for the starkit or starpack itself.

We can use our `demo` starkit for illustration purposes.

```
c:\demo-dir> tclkit-cli
% set kit [file normalize demo]
C:/demo-dir/demo
% file type $kit
file
```

As expected, `file type` returns the type of our starkit file as `file`. This is **before** we load it. However, **after** loading the starkit, we get a different result.

```
% source $kit
run_mode: sourced
% file type $kit
directory
```

It now show up as a `directory`! This is because the path `C:\demo-dir\demo` is now the mount point for the starkit and the root directory of the VFS.

We can confirm this with the `vfs::filesystem info` command.

```
% vfs::filesystem info
C:/demo-dir/demo C:/demo-dir/tclkit-cli.exe
```

Notice how `C:/demo-dir/demo` is listed as a file system. Furthermore, because tclkit applications themselves are structured as starpacks, our `tclkit-cli` executable file also shows up as a file system which leads to the following quirk.

```
% file type [info nameofexecutable]
directory
```

This is a consequence of the fact that tclkits and starpacks mount their contents as a file system at the mount point corresponding to their executable path. (At the risk of belaboring the point, we need to stress that this only applies to tclkit and starpack executables, not to the standard Tcl shells or applications.)

This implementation quirk is something you need to keep in mind when working with a starkit or starpack. For example, the following command

```
file copy [info nameofexecutable] target
```

will result in very different results in `tclsh` versus `tclkit-cli`. In the former case, `target` will be a copy of the `tclsh` executable. In the latter case, the `tclkit-cli` path is seen as a mounted directory and `target` will contain the entire directory structure contained within the tclkit VFS. You are encouraged to try the command and examine the difference.

19.4.11. Writable starkits

Our discussion so far has only involved read operations on starkits once they are constructed. Depending on the underlying technology used to implement a starkit, it is also possible to modify a starkit by writing to it. Among the three popular starkit technologies, Metakit, Vlerq and zip, only Metakit based starkits support writing.

To create a writable starkit, pass the `-writable` option to `sdx wrap`. For example,

```
c:\temp\demo>tclkit sdx.kit wrap demo.kit -writable
10 updates applied
```

You can then create, delete or write to files within the starkit using standard Tcl I/O commands **provided you are using a tclkit application based on Metakit technology**.

19.5. Chapter summary

In this chapter, we introduced virtual file systems and tclkits. Virtual file systems permit arbitrary structured data to be presented to the application as a local file system. It can then be accessed and worked on with the commonly used file and channel based commands. We saw examples of the utility of this for accessing remote files over FTP and for accessing process information as a file system.

We also saw the use of VFS for the purpose of creating single file Tcl applications. This has great benefits in terms of the ease with which applications can be deployed without the need for installers or additional packaging.

19.6. References

LAND2002

*Beyond TclKit - Starkits, Starpacks and other *stuff*,* Landers, Proceedings of the 2002 Tcl Conference, http://www.digital-smarties.com/Tcl2002/tclkit.pdf

20

Interpreters

A *Tcl interpreter* runs Tcl programs within an execution context that includes a namespace hierarchy, command and variable definitions, and a call stack. When an application starts up, as part of its initialization it creates a new Tcl interpreter through a native call to the Tcl library. Some applications may even create multiple such interpreters from native code, either in the same thread or from different threads. These interpreters all run independently, oblivious to the existence of the others.

All our discussion so far has pertained to running Tcl within a single such interpreter created by the application. In fact, an interpreter may **itself** create additional *child* or *slave* interpreters. Unlike the interpreters created natively by the application, these slave interpreters are **not** independent. They run under the control of the creating interpreter, their *master*, in terms of the commands they may execute, how long they may run for and so on. Each slave may in turn create additional interpreters, resulting in an hierarchy of interpreters.

Multiple interpreters are useful in many situations. For example, a network server implemented in Tcl may use a different interpreter for every client, thereby avoiding even accidental interference between the contexts for each client. Another example is the use of multiple interpreters to implement domain-specific languages as dialects of Tcl. We will see examples of both in this chapter.

A special case of a slave interpreter is a *safe* interpreter which is created with all commands deemed to be dangerous from a security point of view removed. This is useful to allow execution of code from unknown and untrusted sources. An example is the execution of application plug-ins written by the user or third parties in a **safe** interpreter so as to avoid both inadvertent as well as malicious modification of the application code.

This chapter is devoted to the creation and use of multiple interpreters within a Tcl application.

20.1. Interpreter basics

We will start off with a discussion of interpreters are created and destroyed and the hierarchy in which they are arranged.

20.1.1. Creating interpreters: `interp create`

An interpreter is created with the `interp create` command.

```
interp create ?-safe? ?--? ?SLAVEPATH?
```

Specifying the optional `-safe` option results in the creation of a safe interpreter. Safe intepreters are discussed in Section 20.6. The `--` sequence indicates the end of options in case *SLAVEPATH* itself begins with a `-` character.

If *SLAVEPATH* is not specified, the command creates an slave interpreter as a direct child of the current interpreter (i.e. the interpreter invoking the `interp create` command). The name of the slave is automatically generated.

```
set child1 [interp create] → interp0
```

The return value from the command is the name of the newly created interpreter. In addition to creating the interpreter, a new command of the same name is also created in the **current** interpreter in the global namespace. This proxy command can be used to in various ways to evaluate code or otherwise control the slave.

20.1.2. Identifying interpreters

An interpreter is identified through a list of interpreter names that define a path through the interpreter hierarchy relative to the current interpreter. The *SLAVEPATH* argument may be specified to the `interp create` command to create a new interpreter at any level in the hierarchy **below** the current interpreter. The last element of this path is the name of the new interpreter. For example, we can create another direct child of the current interpreter and have it named `level1slave`.

```
set child2 [interp create secondchild] → secondchild
```

Since the list passed as the *SLAVEPATH* argument contains only one element, it becomes the name of the created child interpreter. We can also create interpreters deeper in the hierarchy.

```
set grandchild1 [interp create [list $child1 grandchild]] → interp0 grandchild
set grandchild2 [interp create {secondchild grandchild}]  → secondchild grandchild
```

Both grandchildren have the same name but are distinguished because their paths are different. The new interpreters are created in their **parent** which is then their "master", **not** in the interpreter that invoked the `interp create` command.

Note that all interpreters except the last in the path must already exist.

```
% interp create {nosuchinterp grandchild}
Ø could not find interpreter "nosuchinterp"
```

Interpreter paths are **always** relative to the current interpreter. There is no concept of an "absolute" path and no way to reference an interpreter that is not a descendant of the current interpreter.

As a special case, commands that take an interpreter path as an argument will treat an empty list as referring to the current interpreter.

As is the case when *SLAVEPATH* is unspecified, a new command of the same name as the created interpreter is also created in its master. If a command of that name already exists in the master, it is overwritten. If the name contains one or more namespace separator sequences, the command is created within that namespace which is created if it did not already exist.

```
% interp create ns::ip ❶
→ ns::ip
% interp slaves
→ secondchild ns::ip interp0
% ns::ip eval {set x 1} ❷
→ 1
```

❶ Creates a slave `ns::ip`

❷ The command `ns::ip` is created in the master and can be used to evaluate scripts within the slave `ns::ip`

In such cases, deleting the namespace will also delete the interpreter contained in that namespace.

```
% namespace delete ns
% interp slaves
→ secondchild interp0
```

20.1.3. Inspecting the interpreter hierarchy

An interpreter can retrieve its slaves with the `interp slaves` command.

```
interp slaves ?INTERPPATH?
```

The command returns a list of the names of the interpreters that are direct children of the interpreter identified by *INTERPPATH*. If *INTERPPATH* is not specified, it defaults to the current interpreter.

```
interp slaves                          → secondchild interp0
interp slaves {}                       → secondchild interp0 ❶
interp slaves secondchild              → grandchild
interp slaves {secondchild grandchild} → (empty) ❷
```

❶ Empty path refers to current interpreter
❷ No fourth generation

To check for the existence of a specific descendent with a known path, use the `interp exists` command. This returns 1 if a specified interpreter exists and 0 otherwise.

```
interp exists {secondchild grandchild}                  → 1
interp exists {secondchild grandchild greatgrandchild} → 0
```

20.1.4. Destroying interpreters

An interpreter can destroy one or more of its descendents with the `interp delete` command.

```
interp delete ?INTERPPATH ...?
```

Each *INTERPPATH* argument identifies a descendent of the current interpreter. Deleting an interpreter will also destroy that interpreter's descendents if any. If any specified interpreter does not exist, an error is raised and any interpreters listed after that will not be destroyed.

```
interp slaves             → secondchild interp0
interp delete secondchild → (empty)
interp slaves             → interp0
```

Deleting an interpreter also deletes the command of that name.

20.2. Evaluating scripts in an interpreter

We saw in Chapter 10 the use of the `eval` command to execute scripts. The `interp eval` command is similar except that it allows execution of the script in **any** accessible interpreter by specifying its path.

```
interp eval INTERPPATH ARG ?ARG ...?
```

The command concatenates all *ARG* arguments in the same fashion as the `eval` command and then executes the script in the interpreter identified by *INTERPPATH*. The command

```
interp eval $child1 { set foo 1 } → 1
```

will set the value of the `foo` variable in the specified interpreter.

An alternative means of achieving the same result is to use the `eval` subcommand of the target interpreter's proxy command. The following is equivalent to the above.

```
$child1 eval {set foo 1} → 1
```

In both cases, the result is the result of the script evaluation in the specified interpreter.

Like the `eval` command, the arguments undergo double substitution (see Section 10.1.1). However, in this case, the two rounds of substitution take place in separate interpreters. The first round of substitution happens place in the current interpreter and the second in the specified target interpreter. The following snippet illustrates this.

```
set bar foo              → foo
set foo 2                → 2
$child1 eval set foo 1   → 1
eval set $bar            → 2
$child1 eval set $bar    → 1
```

20.3. Command aliases

A command *alias* maps a command in an interpreter to an implementation in another interpreter. Invoking the command in the first will execute the mapped command in the context of the second.

20.3.1. Defining aliases

An alias is defined with the `interp alias` command.

```
interp alias SRCINTERP SRCCMD TARGETINTERP TARGETCMD ?ARG ...?
```

The command creates a new alias named *SRCCMD* in the interpreter whose path is given by *SRCINTERP*. When *SRCCMD* is invoked within *SRCINTERP*, Tcl will execute *TARGETCMD* in interpreter *TARGETINTERP*. The arguments passed to *TARGETCMD* will consist of the *ARG* arguments, if any, specified in the `interp alias` command, followed by any arguments specified in the invocation of *SRCCMD*.

Another way to create aliases is with the `alias` subcommand of the interpreter's proxy command.

```
INTERPCMD alias SRCCMD TARGETCMD ?ARG ...?
```

This works similarly except that the target interpreter, i.e. the interpreter in which *TARGETCMD* will be run, is the current interpreter. In other words, the above is equivalent to

```
interp alias SRCINTERP SRCCMD {} TARGETCMD ?ARG ...?
```

The result of the command in both forms is an alias token which can later be used to introspect or delete the alias.

The primary motivation for using aliases is to execute commands in controlled fashion on behalf of a "less privileged" interpreter. Section 20.6 describes this in detail; here we stick to a couple of basic examples.

One of the simpler uses of aliases is as a short hand for lazy programmers who hate typing.

```
% interp alias {} substr {} string range
→ substr
% substr "abcd" 0 2
→ abc
%
% interp alias {} xmlify {} string map {< &lt; > &gt; & &amp; \" &quot; ' &apos; }
→ xmlify
% xmlify "2 is < 3"
→ 2 is &lt; 3
```

Note from the above examples that

- An empty path passed as the source or target interpreters refers to the current interpreter.
- You can create an alias within the same interpreter, i.e. where the target and source interpreters are the same.

Of course, you could use procedures for the above as well so here is an example where you do need aliases. Imagine our server application dedicates an interpreter to each client and also has a common logging system shared by all interpreters. We define an interpreter, `logger`, to collect these messages.

```
interp create logger
interp eval logger {
    set ::log_level 1
    proc log {level message} {
        if {$level >= $::log_level} {
            lappend ::messages $message
        }
    }
    proc messages {} {      # Returns collected messages and resets.
        return $::messages[set ::messages ""]
    }
}
```

Then when we create an interpreter for a client, we add to it aliases that map to log messages at each level.

```
interp create client0
interp alias client0 debug logger log 0
interp alias client0 log logger log 1
interp alias client0 err logger log 2
client0 eval {
    debug "This is a debug message."
    log "This is an informational message."
    err "This is an error message."
}
logger eval messages
→ {This is an informational message.} {This is an error message.}
```

20.3.2. Introspecting aliases

The list of aliases defined for an interpreter can be retrieved with the `interp aliases` command or the `aliases` subcommand of its proxy command.

```
interp aliases client0 → err log debug
client0 aliases        → err log debug
```

The target interpreter for an alias can then be determined with the `interp target` command or its proxy form.

```
interp target client0 debug → logger
```

Note that an empty result from the command means the current interpreter is the target.

The specific command prefix in the target interpreter that an alias resolves to can be retrieved with another syntactic form of `interp alias` or its proxy equivalent.

```
interp alias client0 debug → log 0
client0 alias debug        → log 0
```

20.4. Execution context in slaves

When a script is executed via `interp eval`, it is evaluated in the **current** context of the target interpreter, **not** its global context. The following snippet illustrates this.

```
set ip [interp create]
proc get_slave_context {ip} {
    return [$ip eval {namespace current}]
}
$ip alias context_demo get_slave_context $ip
$ip eval {
    namespace eval ns {context_demo}
}
→ ::ns
```

Note how the `namespace current` invocation from `get_slave_context` shows `ns` as the namespace context and not the `::` global context. The `context_demo` alias is invoked in the `ns` namespace context within the slave. This results in the `get_slave_context` procedure being run in the master. When that evaluates `namespace current` in the slave, that command is then invoked within the slave's current context, which is `ns`.

As another example, consider the invocation of an alias from within a procedure.

```
proc get_slave_var {ip} {
    return [$ip eval {set myvar}]
}
$ip alias var_demo get_slave_var $ip
$ip eval {
    set myvar "global"
    proc demo {} {
        set myvar "local"
        var_demo
    }
}
$ip eval demo
→ local
```

Again, note how the value returned is `local` indicating that the `set myvar` was executed in the **current** context of the slave, i.e. within the `demo` procedure context, and not the global context.

20.5. Cancelling script evaluation

The `interp cancel` command allows cancellation of a script running in an interpreter.

```
interp cancel ?-unwind? ?--? INTERP ?MESSAGE?
```

The effect is similar to that of throwing an error from within a script executing in the interpreter identified by the interpreter path *INTERP*. The difference is that

- first, the cancellation can be initiated from outside the script itself, and
- second, unlike error exceptions, the cancellation can be made to be untrappable with `catch` and `try`.

Let us examine the effect of this command by experimenting with the current interpreter itself. First we define three commands that will raise an exception, or call `interp cancel` on the current interpreter with and without the `-unwind` option.

```
proc raise_error {} { error "Error raised!" }
proc cancel {} { interp cancel {} "Interpreter execution cancelled!" }
proc cancel_unwind {} {interp cancel -unwind -- {} "Interpreter execution unwound!"}
```

Then we define a `demo` procedure that will call one of these procedures based on the argument passed. We will add a variable trace to further distinguish the cases.

```
proc demo {procedure} {
    puts "Entering demo"
    set x 1
    trace add variable x unset print_args
    catch $procedure message
    puts "Caught error message: $message"
    puts "Exiting demo"
}
```

Now let us try calling `demo` with each of the procedures we defined earlier.

```
% demo raise_error
→ Entering demo
  Caught error message: Error raised!
  Exiting demo
  Args: x, , unset
```

We do not have much to say for the above case as we have already discussed raising of errors extensively in Section 11.5.

The `interp cancel` command without the `-unwind` option behaves very similarly.

```
% demo cancel
→ Entering demo
  Caught error message: Interpreter execution cancelled!
  Exiting demo
  Args: x, , unset
```

Again, the error is caught and the variable trace fires as the procedure is exited. The real difference with the use of `interp cancel` compared to `error` is that the exception can be initiated from a **different** interpreter than the one in which the script is running.

Our final example uses the `interp cancel` command with the `-unwind` option.

```
% demo cancel_unwind
Ø Entering demo
  Interpreter execution unwound!
```

When the `-unwind` option is specified, the call stack is completely unwound in the target interpreter **without any chance of trap or trace handlers to run** as seen in the example.

Because the `-unwind` option does not give any error trap or trace handlers a chance to run, the common use of these handlers to release resources will be bypassed. Thus the mechanism should be used with extreme care and under limited circumstances.

Note that `interp cancel` aborts the execution of scripts in the target interpreter, the target interpreter itself is **not** deleted.

Our interpreter demonstrated the workings of `interp cancel` using the current interpreter. More commonly it is used in two scenarios:

- cancel scripts in a slave interpreter from within an aliased command. Applying the above discussion to this case should be straightforward.
- cancel scripts running in a interpreter within another thread (see Section 22.6).

20.6. Safe interpreters

> I didn't know I was a slave until I found out I couldn't do the things I wanted.
>
> — Frederick Douglas

There are several application use cases where the application needs to run source code from untrusted sources, the most common example being browsers running Javascript code downloaded from some random web site on your personal desktop. Browsers limit, not eliminate, security vulnerabilities arising from running malicious downloaded code by placing restrictions on the capabilities of the Javascript interpreter itself. This restrictions include access to the local file system and other operating system resources.

Similarly, an application may allow plug-ins from third parties that offer additional functionality. In such cases, the application needs to protect itself, and the user, from being compromised by any untrusted plug-ins that the user might have installed.

In another scenario, the application runs in a privileged mode on behalf of the user but still needs to ensure the user is prevented from accessing unauthorized resources or elevating his privileges.

Yet another example is a distributed system where computationally intensive tasks are assigned to multiple systems. For an additional layer of security, it is good practice to execute such remotely dispensed tasks in restricted envionment.

Tcl's solution for dealing with these situations is the *safe interpreter* which is a Tcl interpreter from which certain commands, which are considered dangerous when invoked from untrusted scripts, have been hidden.

A safe interpreter has the following restrictions placed on it:

- Certain commands like `exec`, open, `socket` etc. that can access system resources or affect the state of a process are hidden so they cannot be directly invoked by scripts running in a safe slave.
- Any slave interpreters created by safe interpreters will also be safe even if the `-safe` option is not specified in their creation.
- The global variable env is not available in safe interpreters as environment variables can contain sensitive information.
- Tcl's default auto-loading facilities described in Section 3.5.1.2 are not implemented.
- Binary extensions cannot be loaded into safe interpreters unless they have a specific entry point different from that used for normal interpreters. When initialized via this entry point, the extension must ensure it does not implement commands that are vulnerable to be misused by malicious scripts.
- Finally, a safe interpreter cannot change the recursion limit on itself or any other interpreter it might create. Recursion limits are discussed in Section 20.8.

A completely restricted safe interpreter can be too limited in functionality to be useful in many circumstances. It can essentially do operations on data and little else. Tcl therefore provides mechanisms for permitting controlled execution of hidden commands in interpreters:

- The aliases mechanism we saw earlier in this chapter which we will discuss further in the context of safe interpreters
- The ability to invoke hidden commands in the safe interpreter under the control of its master

Let us start by looking at how a safe interpreter is created.

20.6.1. Creating a safe interpreter

A safe interpreter is created with the same `interp create` command used to create a normal interpreter but with the added option `-safe` specified.

```
interp create -safe safe0 → safe0
```

We can verify that certain commands are **hidden** and therefore not directly available in the safe interpreter. Attempting to invoke them will result in an error being raised.

```
% safe0 eval {socket www.example.com 80}
Ø invalid command name "socket"
```

See the reference documentation for `interp` for a complete list of the commands that are hidden in safe interpreters by default. Since commands can also be hidden dynamically at runtime, you can also programmatically retrieve the list of currently hidden commands.

Hidden commands are not actually removed from the interpreter and can be made available to the interpreter under the control of its master.

20.6.2. Aliasing in safe interpreters

As we mentioned in the introduction, removing **all** access to resources such as the file system would make safe interpreters usable in very few scenarios. For example, consider the use of safe interpreters in a web server to handle client connections. To be useful for this purpose, the safe interpreters would need to be able to read Web content, write to log files and so forth. At the same time, we would want to prevent it from being able to read files from outside the Web content directories or to overwrite the Web content and so on.

Let us look at the first mechanism available to achieve this goal of providing controlled access to resources — aliases. We already introduced this feature in Section 20.3. Our short illustration below demonstrates how a safe interpreter might write log messages to a file. Since the safe interpreter cannot directly do I/O, the trusted master interpreter must do it on its behalf. This is accomplished by defining a `log_message` command in the **slave** that maps to a command in the **master** that writes to the appropriate log file.

```
set safe_ip [interp create -safe]
set log_chan [open [file join /var/log $safe_ip.log] a]
$safe_ip alias log_message puts $log_chan
```

Note this uses the second form of alias definition. We could have also written it using the first form as

```
interp alias $safe_ip log_message {} puts $log_chan
```

with the same result.

Now scripts running in the slave interpreter can safely log messages to disk while still being restricted in terms of its ability to do I/O.

```
$safe_ip eval {
    log_message "Something noteworthy happened."
}
```

The above example enabled some very specific functionality within the safe interpreter. More often you want the safe interpreter to have a little more generalized capabilities. For example, the safe interpreter serving Web clients has to be allowed to read any file in the Web content directory based on the URL but no other. It may also permit writing of files uploaded by the user but again only to a specific directory. We might thus choose to expose a restricted form of the open command to the safe interpreter. In **pseudocode** form,

```
$safe_ip alias open SafeOpen $slave
proc SafeOpen {slave path access} {
    if {![path_is_allowed $path $access]} {
        error "Access denied!"
    }
    tailcall $slave invokehidden -- open $path $access
}
```

The code maps the open command in the safe slave interpreter to the SafeOpen command in the master passing the slave interpreter command as an additional parameter. When the

```
open index.html r
```

command is invoked in the **slave**, the **master** runs the command

```
SafeOpen interp0 open index.html r
```

The SafeOpen command first checks whether the desired access to the path is permitted (we will elaborate on this aspect in a later section). If not permitted, it raises an error. Otherwise, it opens the specified file in the slave interpreter with the invokehidden command that we describe a little later. The slave can then read and / or write that file as appropriate.

20.6.2.1. Precautions for aliased commands

Care must be exercised when executing code in the master on behalf of an untrusted safe slave.

Commands like SafeOpen that are aliased into safe interpreters must be very careful in how they handle any arguments passed by the slaves. In particular, these arguments must **never** be treated as scripts or parts thereof, for example by passing them to commands like eval, subst, expr etc. Otherwise a malicious script running in the untrusted safe interpreter can trick the master into executing arbitrary code.

Also keep in mind that any exceptions raised in the master interpreter will be propagated back to the slave where they can be caught and error stack examined. Therefore if you want to prevent inadvertent information leakage from the master to the slave, all errors must be caught and passed on to safe interpreter in some sanitized generic form.

20.6.3. Hidden commands

Having looked at the use of aliases with safe interpreters, let us now look at the other mechanism for using safe slaves under the control of a trusted master — hidden commands.

20.6.3.1. Invoking hidden commands

The commands that are hidden in an interpreter are **not** deleted or removed; they just cannot be directly invoked from within the safe interpreter. They can however be invoked by the master through the interp invokehidden command, or equivalently the invokehidden subcommand of the slave interpreter command.

```
interp invokehidden SLAVE ?-global? ?-namespace NAMESPACE? ?--? HIDDEN ?ARG ...?
SLAVE invokehidden ?-global? ?-namespace NAMESPACE? ?--? HIDDEN ?ARG ...?
```

Here the *HIDDEN* argument is the name of the command that is hidden in the slave interpreter and *ARG...* are any arguments to be passed to it. The `-global` and `-namespace` options permit the master to control the namespace context within which the hidden command is executed. By default, the command is executed in the current context of the slave. The `-global` and `-namespace` option force the execution to take place in the global or specified namespace instead.

We saw an example using `invokehidden` in the previous section where the open command was invoked in the slave interpreter to create a channel to a file after the master interpreter checked that the file path was one that the slave was permitted to access.

```
tailcall $slave invokehidden -- open $path $access
```

This results in the open command being run in the **slave** interpreter and returning a channel to the script in the slave.

We could have also written the above `invokehidden` command as

```
$slave invokehidden -- open $path $access
```

The purpose behind using `tailcall` is that in case of errors, the error stack looks cleaner as the `SafeOpen` call will not be present on the stack at all as the `tailcall` replaces it with the open command invocation.

At this point the front benchers in the class will be objecting vociferously thereby waking up the back bench. Since the open command in the safe slave is aliased to `SafeOpen` in the master, won't the `invokehidden` call to open in the slave result in a recursive call back into `SafeOpen` ad infinitum? The answer is that a call to `invokehidden` will always invoke the **original** command that was hidden and not any alias or redefinition. Here is an illustrative example.

```
% safe0 eval { proc demo {} { return "The original demo" } }
% safe0 eval demo
→ The original demo
% safe0 hide demo
% safe0 eval demo ❶
Ø invalid command name "demo"
% safe0 eval { proc demo {} { return "The redefined demo" } } ❷
% safe0 eval demo ❸
→ The redefined demo
% safe0 invokehidden demo ❹
→ The original demo
```

❶ Errors because the command is now hidden
❷ Redefine the procedure
❸ The redefinition is executed.
❹ The original hidden procedure is executed

Notice how invoking demo within the `safe0` interpreter runs the new definition of demo whereas invoking demo from the master via `invokehidden` runs the original hidden procedure. This behaviour ensures that the slave interpreter (remember it is potentially running untrusted code) cannot trick the master into running the wrong code by simply redefinining commands.

Like `interp eval`, `interp invokehidden` also runs code in the slave interpreter. One obvious difference is that `interp eval` runs arbitrary scripts whereas `interp invokehidden` runs a single command which must furthermore be a hidden command. A less obvious but important difference is that while the arguments supplied to `interp eval` undergo two rounds of substitutions, once in the master and once in the slave, arguments to `interp invokehidden` only undergo substitution in the master interpreter. This gives the master precise control over what arguments are passed to the hidden command.

20.6.3.2. Hiding and exposing commands

We noted earlier that certain commands are hidden at the time a safe interpreter is created. However, as our previous example showed, we can hide any command at any time with any of the following equivalent forms.

```
interp hide SLAVE CMDNAME ?HIDDENNAME?
SLAVE hide CMDNAME ?HIDDENNAME?
```

Both forms hide *CMDNAME* in the specified slave interpreter. The command can be invoked by the master via `invokehidden` with the name *HIDDENNAME* which defaults to *CMDNAME* if unspecified.

There is an important restriction that needs to be kept in mind with respect to the hiding of commands. Neither *CMDNAME*, nor *HIDDENNAME* if specified, can contain namespace qualifiers. The commands to be hidden are always looked up in the global namespace of the slave interpreter. This restriction prevents the slave from tricking the master interpreter into hiding the wrong command by executing in a namespace other than the global one.

Hidden commands may be made available to the slave interpreter with the `expose` subcommand.

```
interp expose SLAVE HIDDENNAME ?CMDNAME?
SLAVE expose HIDDENNAME ?CMDNAME?
```

Exposing a command makes it available to be called directly by the slave either under its original name or if *CMDNAME* is specified, under a new name. This technique is an alternative to renaming the original command when redefining it.

```
% safe0 expose demo original_demo
% safe0 eval demo
→ The redefined demo
% safe0 eval original_demo
→ The original demo
```

One final note about hiding and exposing commands is that although our discussion has been focused on safe interpreters, commands can be hidden or exposed in **any** slave interpreter, not just safe ones, though their main use is with the latter.

Here is an example of how the combination of aliases and hidden commands may be put to good use in contexts other than safe interpreters. In previous chapters, we saw several ways of "wrapping" existing commands to modify their behaviour. For example, we might want to log the every outgoing network connection in an application. We could create a new `socket` procedure that overrides the `socket` command and calls it. Here is a simple alternative.

```
interp hide {} socket
proc socket args {
    puts "New connection: $args"
    tailcall interp invokehidden {} socket {*}$args
}
close [socket www.example.com 80]
→ New connection: www.example.com 80
```

Note the current interpreter is invoking a hidden command within itself. This is possible only because the interpreter is not a safe interpreter.

20.6.3.3. Introspecting hidden commands

The list of commands that are currently hidden can be retrieved with the `interp hidden` command or the `hidden` subcommand of the interpreter command.

```
% interp hidden safe0
→ file tcl:file:isdirectory tcl:file:writable tcl:file:type tcl:file:tail tcl:file:readli...
% safe0 hidden
→ file tcl:file:isdirectory tcl:file:writable tcl:file:type tcl:file:tail tcl:file:readli...
```

In case it is not obvious, the above commands are executed from the safe interpreter's master, not from within the safe interpreter. Somewhat surprisingly, the safe interpreter can also find out what commands are hidden from it.

```
% safe0 eval {interp hidden {}}
→ file tcl:file:isdirectory tcl:file:writable tcl:file:type tcl:file:tail tcl:file:readlink
   ↳ tcl:file:executable socket tcl:file:size tcl:file:delete tcl:file:copy open
   ↳ tcl:file:isfile pwd unload tcl:file:rootname tcl:file:owned glob exec tcl:file:attributes
   ↳ tcl:file:dirname encoding tcl:file:normalize fconfigure load source tcl:file:volumes exit
   ↳ tcl:file:stat tcl:file:mtime tcl:file:extension tcl:file:tempfile tcl:file:link
   ↳ tcl:file:mkdir tcl:file:rename tcl:file:readable tcl:file:nativename tcl:file:lstat
   ↳ tcl:file:exists tcl:file:atime cd
```

20.6.4. Trusting safe interpreters

A master can mark a safe interpreter as trusted with the `interp marktrusted` command.

```
interp marktrusted SLAVE
SLAVE marktrusted
```

Marking an interpreter as trusted removes implicit restrictions such as automatically making its children as safe. However, it does **not** immediately expose any hidden commands. They still have to be explicitly made visible in the slave.

```
interp create -safe ip0                    → ip0
interp create {ip0 ip1}                    → ip0 ip1
interp issafe {ip0 ip1}                    → 1 ❶
ip0 marktrusted                            → (empty)
interp create {ip0 ip2}                    → ip0 ip2
interp issafe {ip0 ip2}                    → 0 ❷
ip0 eval {close [open c:/temp/foo.txt w]} Ø invalid command name "open" ❸
```

❶ Any further descendants are automatically safe
❷ After being marked as trusted, descendents are trusted by default
❸ However commands in the interpreter marked trusted are still hidden

20.6.5. Utilities for safe interpreters

As discussed, a completely restricted safe interpreter only has limited use. We therefore described mechanisms that allow execution of potentially dangerous commands within the safe interpreter under the control of the master. Nevertheless, security is a very tricky business and correct use of these mechanisms without introducing security holes when executing untrusted scripts in a safe interpreter is still fraught with potential pitfalls.

As an example, consider the `path_is_allowed` pseudo command we used earlier to check if a particular file was allowed to be accessed by a safe interpreter. An implementation of this command is not just a matter of checking against a database of permitted files and directories. It would have to ensure correct operation for both absolute and relative paths, presence of `..` parent directory tokens, links and so forth. As a matter of principle, the real file paths should also be preferably hidden from the safe interpreter. Without careful coding and review, subtle errors will creep in.

Several libraries provide functionality that reduce the burden of extending safe interpreters. We briefly describe two such libraries and strongly encourage their use instead of rolling your own implementation.

20.6.5.1. Safe Tcl

Safe Tcl is a set of commands in the `safe` namespace that extend safe interpreters in a controlled manner.

Safe Tcl needs to be distinguished from the *safe interpreters* we have discussed above. The former is layered on top of the latter to provide some additional conveniences. The terminology is slightly confusing but follows the Tcl reference documentation.

Safe Tcl defines aliases for the normally hidden commands `source`, `file`, `encoding`, `load` and `exit`. These aliases expose these commands in the slave interpreter but limit their permitted operations as shown in Table 20.1.

Table 20.1. Safe Tcl predefined aliases

Alias	Description
`encoding`	The `encoding` command is normally hidden because in addition to converting strings into various character encodings it allows setting of the `system` encoding. The alias permits all operations except this potentially unsafe change in the `system` encoding.
`exit`	The aliased version deletes the safe slave interpreter in which it is invoked instead of terminating the entire process.
`file`	The alias will only permit subcommands that operate purely on a syntactic basis and do not require access to the file system. The allowed subcommands are `join`, `split`, `dirname`, `extension`, `root`, `tail` and `pathtype`.
`load`	Only allows binary extensions to be loaded under certain restrictions described later.
`source`	Only allows Tcl scripts to be sourced under certain restrictions that are described later.

Creating a Safe Tcl interpreter

An application can make use of the library either by using the `safe::interpCreate` command to create a safe interpreter with the above extensions, or by using the `safe::interpInit` command to enable the extensions in an existing safe interpreter.

```
safe::interpCreate ?SLAVENAME? ?OPTIONS?
safe::interpInit EXISTINGSLAVE ?OPTIONS?
```

The difference with respect to the `interp create -safe` command is that the created interpreters are initialized with the predefined aliases. Let us try out a few commands to illustrate. An interpreter created using `safe::interpCreate` can invoke certain `file` commands that work purely on a syntactic basis. On the other hand, `file` subcommands that access the file system are still not permitted.

```
safe::interpCreate safeA                → safeA
safeA eval {file dirname /var/log}      → /var
safeA eval {file isdirectory /var/log} Ø not allowed to invoke subcommand isdirectory of file
```

In contrast, safe interpreters created using `interp create` cannot invoke the `file` command at all.

```
% interp create -safe safeB
→ safeB
% safeB eval {file dirname /var/log}
Ø invalid command name "file"
```

We can however add the aliases from the Safe Tcl library with `safe::interpInit`.

```
safe::interpInit safeB              → safeB
safeB eval {file dirname /var/log} → /var
```

Deleting a Safe Tcl interpreter

Interpreters created with `safe::interpCreate` or initialized with `safe::interpInit` must be destroyed with the `safe::interpDelete` command, not with `interp delete` as internal structures related to the added safe extensions need to be cleaned up.

```
safe::interpDelete safeB → (empty)
```

The slave interpreter may also commit suicide by calling the `exit` command. Unlike in a normal interpreter, this does not terminate the entire process as it is aliased to the `safe::interpDelete` command.

Safe Tcl interpreter configuration

Both `safe::interpCreate` and `safe::interpInit` accept several options, shown in Table 20.2, that control certain aspects of behaviour. These options can also be set at any later time with the `safe::interpConfigure` command.

Table 20.2. Safe Tcl configuration options

Option	Description
`-accessPath` *DIRECTORIES*	Specifies the list of directories from which the slave interpreter is allowed to source or load files. This defaults to the list of directories that the master uses for auto-loading packages. Any attempt to source or load files from a directory not listed in *DIRECTORIES* will raise an error.
`-deleteHook` *SCRIPT*	Specifies a callback script to be run in the **master** interpreter when the slave interpreter is deleted. This script is run **before** the actual deletion and is passed an additional argument which is the name of the slave interpreter. An empty string for *SCRIPT* removes any existing callback.
`-nested` *BOOLEAN*	If specified as a boolean `true` value, the slave interpreter is permitted to load packages into any interpreters that it creates itself. Default is `false`.
`-statics` *BOOLEAN*	A Tcl application may include binary extensions that are statically linked into the application itself and thus are not associated with any file or directory paths. If this option is `true` (default), the slave interpreter can load such statically linked extensions; if `false`, it cannot.

File paths in Safe Tcl interpreters

Even when a safe interpreter needs to be provided access to (some portion of) the file system, it is still considered good practice to not expose the real file system paths to the untrusted scripts. Safe Tcl helps with this by using virtual tokens to represent the physical directories. You can see this by examining an `auto_path` element in the master and slave interpreters.

```
lindex $auto_path 0              → c:/tcl/lib
safeA eval {lindex $auto_path 0} → $p(:0:)
```

Thus a script running inside the safe interpreter can only see the "virtual" directory tokens and not the real file system paths. When passing paths into the Safe Tcl interpreter, the master needs to translate real paths to the corresponding token. It can use the `safe::interpFindInAccessPath` command for this purpose. For example,

```
safe::interpFindInAccessPath safeA [lindex $auto_path 0] → $p(:30:)
```

The master may also add new paths that are permitted to be accessed by the `source` or `load` commands. The `safe::interpAddToAccessPath` command does the needful.

```
safe::interpAddToAccessPath safeA C:/temp → $p(:265:)
```

The command returns the token that will represent the path in the slave interpreter.

Troubleshooting Safe Tcl interpreters

Debugging scripts in safe interpreters can sometimes be troublesome because certain Tcl features like full stack traces on errors are not made available to avoid information being leaked to the untrusted scripts in the slave. Safe Tcl provides the `safe::setLogCmd` command to help with this problem. It permits specification of a script to be run in the master interpreter when "interesting" events happen in the slave. This script is invoked with an additional message describing the event.

```
% safe::setLogCmd puts
% safe::interpCreate safeB
→ NOTICE for slave safeB : Created
  NOTICE for slave safeB : tcl_libray was not in first in auto_path, moved it to front of...
  NOTICE for slave safeB : Setting accessPath=(c:/tcl/866/x64/lib/tcl8.6 c:/tcl/866/x64/l...
  NOTICE for slave safeB : auto_path in safeB has been set to {$p(:0:)} {$p(:1:)} {$p(:2:...
  safeB
% safeB eval {source foo.tcl}
Ø ERROR for slave safeB : "foo.tcl": not in access_path
  permission denied
```

To turn off logging, pass an empty string to the `safe::setLogCmd` command.

20.6.5.2. The `island` package

The Safe Tcl commands in the previous section only provide control of the directories as related to the `source` and `load` commands. They do not help with general purpose file operations. This functionality is provided by the `island` package available from http://wiki.tcl.tk/44380.

The package provides a single command `island::add` which specifies a directory that is allowed to be accessed by a safe interpreter. Using the package is straightforward and illustrated below.

```
% package require island
→ 0.2
% set ip [interp create -safe]
→ interp0
% $ip eval {close [open c:/temp/demo.txt w]} ❶
→ invalid command name "open"
% island::add $ip c:/temp ❷
→ C:/temp
% $ip eval {close [open c:/temp/demo.txt w]}
```

❶ Fails because open is hidden in a safe interpreter.
❷ Permit `C:/temp` to be accessed by the interpreter.

Note that the `island::add` command may be used multiple times to permit more than one directory to be accessed.

20.7. I/O in slave interps

Channel I/O in slave interpreters merits some special consideration. We have already seen that commands that **create** channels, such as open, `socket` etc., are hidden in safe interpreters (though notice the actual I/O command like `puts`, `gets`, `read` are not) and how channels can be opened in safe slaves through the use of controlled aliases in the master. Here we address another issue which applies to safe and normal interpreters alike. Channels created in an interpreter are not automatically available in other interpreters except for the standard input, output and error channels which are made available to all interpreters.

On Windows platforms, applications like `wish` which are GUI applications and not console-based do not have real standard I/O file descriptors provided by the operating system. In this case, standard I/O is simulated to a pseudo-console window provided by the Tk GUI extension and not implemented as real channels. In this environment, the standard channels are only available to the main Tcl interpreter and not to any slaves.

We now look at how to make channels available in interpreters other than the ones that created them. There are two ways to accomplish this. The `interp share` and `interp transfer` commands both make a channel in an interpreter available in another interpreter. The difference is that with `interp share`, the channel remains available in the original interpreter as well whereas with `interp transfer` it does not.

```
interp share FROMINTERP CHANNEL TOINTERP
interp transfer FROMINTERP CHANNEL TOINTERP
```

Here *FROMINTERP* is the interpreter currently holding the channel *CHANNEL*, and *TOINTERP* is the interpreter where the channel is to be made available. Note that the channel name remains the same in all interpreters where the channel is accessible. In the case of shared channels, the channel seek pointer is also shared across all interpreters so if multiple interpreters perform I/O on the shared channel the data will be interleaved. A shared channel is not closed until all interpreters where it is accessible close the channel.

Here is a simple example illustrating use.

```
% set ip [interp create -safe]
→ interp2
% set fd [file tempfile]
→ file437faa0
% interp share {} $fd $ip
% $ip eval "set fd $fd" ❶
→ file437faa0
% $ip eval {puts $fd "Message from the slave."; close $fd}
% chan seek $fd 0 start
% read $fd
→ Message from the slave.
% close $fd
```

❶ Need to pass in the channel name to the slave

Note how the channel stays open in the parent after the slave closes it and that the parent needs to reset the channel pointer with a seek before reading the file.

20.8. Setting resource limits

Tcl offers a limited form of protection against runaway scripts in interpreters that result in excessive consumption of computing resources. These limits take the form of

- a maximum recursion depth for execution of Tcl scripts
- an upper limit on the number of commands that may be executed within an interpreter
- an absolute time by which an interpreter must finish executing a script

We describe these capabilities in this section.

Besides these explicit limits, a master interpreter may also impose limits using mechanisms already discussed. For example, aliasing `puts` in a slave interpreter can count and limit the number of bytes written to a file by the slave.

20.8.1. The recursion limit

To protect against runaway scripts overflowing the space allocated for the C stack of the process, Tcl places a limit on the depth of the call stack of an interpreter. This limit can be retrieved or set with the `interp recursionlimit` command.

```
interp recursionlimit INTERP ?LIMIT?
SLAVE recursionlimit ?LIMIT?
```

If *LIMIT* is not specified, the command returns the current recursion limit for the specified interpreter (which may be the current one).

```
interp recursionlimit {} → 1000
```

We can test the effect of this limit.

```
% proc recurse {} { incr ::depth ; recurse }
% set depth 0
→ 0
% recurse
Ø too many nested evaluations (infinite loop?)
% set depth
→ 1000
```

As you see, Tcl will raise an error exception when the call stack depth exceeds the recursion limit.

We can also change the recursion limit if we wish. Ususally this change is made for slave interpreters.

```
set ip [interp create]
$ip recursionlimit 10
$ip eval {
    proc recurse {} {incr ::depth; recurse}
    set depth 0
    catch {recurse} message
    return "Error: failed at depth $depth: $message"
}
→ Error: failed at depth 9: too many nested evaluations (infinite loop?)
```

20.8.2. Limiting interpreter lifetimes

Tcl can limit the lifetime of an interpreter either by restricting the number of commands the interpreter can execute or by specifying an absolute time by which the interpreter must have finished execution. Both these limits are retrieved and set with the `interp limit` command.

```
interp limit INTERP command|time ?OPTIONS?
SLAVE limit command|time ?OPTIONS?
```

Depending on whether `command` or `time` is specified, the command pertains to the limit on command execution or the time by which the interpreter must complete.

20.8.2.1. Limiting number of commands executed

We will start with the command limits. If no options are specified, the command will return the current settings for the specified limit type.

```
% interp limit {} command ❶
Ø limits on current interpreter inaccessible
% interp limit $ip command
→ -command {} -granularity 1 -value {}
% $ip limit command
→ -command {} -granularity 1 -value {}
```

❶ Error because an interpreter may not read its own limits

The `-granularity` option controls how often Tcl will check the limit. For example, when set to 1, Tcl will check that the permitted command execution count is not exceeded before **every** command invocation. If set to 5, it will only check on every fifth command.

The `-command` option specifies a callback command that will be invoked in the global context of the interpreter that invoked the `interp limit` command when the command count limit is exceeded in the target interpreter. If an empty string, no callback is registered.

The `-value` option specifies the permitted command count. If empty, no limit is placed on the number of commands that may be executed by an interpreter.

Let us play around with changing the limits on an interpreter. We will define a procedure that will be called when the limit is exceeded. A point to note about the call back procedure is that it can **change the limit to permit the interpreter to continue execution**.

```
proc watchdog {ip max} {
    if {[interp limit $ip command -value] < $max} {
        puts "Raising command count limit further to $max"
        interp limit $ip command -value $max
    }
}
```

Now we create a slave interpreter and set limits on it. An important point to be noted is that the creation of the interpreter itself executes initialization commands in the slave. So when we set limits we may need to take this into account using the `info cmdcount` command to find how many commands have been executed so far.

```
set ip [interp create]
set nexecuted [$ip eval {info cmdcount}]
interp limit $ip command \
    -command [list watchdog $ip [+ $nexecuted 4]] \
    -value [expr {$nexecuted+2}]
```

Now we try executing commands in the slave.

```
$ip eval {info cmdcount} → 315
$ip eval {info cmdcount} → 316
$ip eval {info cmdcount} → Raising command count limit further to 318
  317
$ip eval {info cmdcount} → 318
$ip eval {info cmdcount} Ø command count limit exceeded
$ip eval {info cmdcount} Ø command count limit exceeded
```

A few words of explanation may be in order.

- We retrieved the command count as of interpreter creation (`nexecuted`).
- We want at most 4 more commands to be permitted to be run and this is the number we pass to `watchdog` as the second argument.
- However, for illustrative purposes we initially set the interpreter command count limit to be only 2 more, not 4.
- Now when we execute commands in the slave, the first two execute with not much fanfare. The third one however causes our initial limit to be exceeded and Tcl runs our `watchdog` procedure.
- The `watchdog` procedure **increases** the limit since it has not reached the maximum it was passed.
- After `watchdog` returns, Tcl checks the limit again. Since the **new** limit is greater than the current command count, it permits execution to continue for this command and the next.
- When the new limit is also exceeded, `watchdog` is called again. However, since the limit is now the maximum specified by the `max` argument, it is not increased further.
- Tcl therefore aborts execution of any further commands with an error message.

Note that the interpreter is **not** destroyed when the limit is exceeded. Only errors are generated. The master interpreter could choose to remove or extend the limits and continue to use the interpreter.

```
% $ip limit command -value {} ❶
% $ip eval {info cmdcount}
→ 320
% $ip eval {info cmdcount} ❷
→ 321
```

❶ Remove the limit
❷ Happy days are here again

20.8.2.2. Limiting interpreter duration

An alternative way to control the lifetime of an interpreter is by specifying a time by which the interpreter must finish execution. We can retrieve the current settings for an interpreter as for command count limits except that we specify `time` as an argument to `interp limit` instead of `command`.

```
% interp limit $ip time
→ -command {} -granularity 10 -milliseconds {} -seconds {}
```

The `-command` and `-granularity` settings have the same meaning as was discussed in the previous section. The former specifies a callback and the latter controls how frequently limits are checked.

The `-seconds` value is the absolute time, expressed as the number of seconds after the epoch (see Section 8.1), beyond which the interpreter must not be allowed to execute.

The `-milliseconds` value is an additional interval (specified in milliseconds) beyond the time specified by the `-seconds` value. This is of use in cases where units of seconds is not fine grained enough for the application's purpose.

Since the general working of time limits is similar to command limits in the previous section (including the ability to change limits in the callback), we only present a simple example.

```
% set now [clock seconds]
→ 1499148960
% $ip limit time -seconds [expr {$now + 2}] -granularity 1
% after 1000 ❶
% $ip eval {set foo 1} ❷
→ 1
% after 1000
```

```
% $ip eval {set foo 2} ❸
Ø time limit exceeded
```

❶ Hang around for a second
❷ Succeeds because time limit not crossed
❸ Fails because time limit exceeded

The attempt to set the value of `foo` to 2 fails because the specified absolute time for the interpreter to complete has been exceeded.

We set `-granularity` to 1 above for illustrative purposes as otherwise we would have had to execute more commands to demonstrate the effect. The cost of checking of absolute time limits can be significant so in general the granularity should not be lowered for time limits.

20.9. Using multiple interpreters

There are many ways applications might make use of multiple interpreters. We illustrate two common ones here.

20.9.1. A safe network server

A very common and obvious use of multiple interpreters is to handle client connections in a server by associating each client with a private slave interpreter. This has two benefits

- The client contexts are isolated from each other "for free". All context for a client is contained within a slave and no special care need be taken to ensure there is no leakage of data or other state between clients.
- Running safe slaves adds another layer of safety for the server. At the same time, depending on authentication and established authorization roles, different slaves may be configured with different privileges in terms of what they can execute and access.

Here is a variation of the simple network server we saw in Section 18.1.4.2. The difference here is that while that server simply reversed the line sent by the client, this server will execute it as an expression, essentially acting as a remote calculator. This is obviously dangerous since the `expr` command can evaluate arbitrary Tcl scripts. We will therefore have to do the evaluation within a safe interpreter.

As in our original network server, we start off by defining the command that will invoked in response to every connection request. For each new connection we create a safe interpreter that will be dedicated to that connection. Furthermore, we place limits on the number of commands that may be executed and the maximum time the interpreter is allowed to live. Then as before we make the accepted client channel non-blocking and attach a read handler passing it our slave interpreter.

```
proc on_accept {so client_ip client_port} {
    set slave [interp create -safe]
    set cmdlimit [$slave eval {info cmdcount}] ❶
    $slave limit command -value [incr cmdlimit 3]
    $slave limit time -seconds [expr {[clock seconds] + 60}]

    chan configure $so -buffering line -encoding utf-8 -blocking 0 -translation crlf
    chan event $so readable [list on_read $slave $so $client_port]
}
```

❶ Get number of commands executed so far in slave

Notice how we set the command limit. The slave internally evaluates commands as part of its internal initialization. So the limit we set has to be some increment to the number of commands already executed by the slave.

Our read handler reads each line from the client and evaluates it as an expression in the slave. For ease of illustration we do not bother dealing with expressions across multiple lines.

```
proc on_read {slave so client_port} {
    set n [gets $so line]
    if {$n >= 0} {
        set status [catch {$slave eval [list expr $line]} result]
        puts $so $result
        if {$status &&
            [lindex $::errorCode 0] eq "TCL" &&
            [lindex $::errorCode 1] eq "LIMIT"} {
            close $so
            interp delete $slave
        }
    } elseif {[chan eof $so]} {
        close $so
        interp delete $slave
    }
}
```

Make a note of the error handling. If the slave evaluation results in an error because of limits being exceeded, we close down the connection. Other errors like invalid input are simply reported back to the client.

Now we start up the server on our local address.

```
set listener [socket -server on_accept -myaddr 127.0.0.1 10042]
vwait forever
```

There is a design choice we made in our implementation to have the master interpreter handle the I/O and invoke the slave only to evaluate the expression. An alternative would have been to hand off the channel to the slave and have it handle the I/O itself. Our choice was based on simpler clean up of the channel when the slave interpreter crossed its limits.

Let us try out our server by connecting to it with telnet.

```
C:\temp\book> start tclsh safe_server.tcl
C:\temp\book> telnet 127.0.0.1 10042
Trying 127.0.0.1...
Connected to 127.0.0.1.
2+2
4
2 ** 4
16
[exit]
invalid command name "exit"
5+5
command count limit exceeded
Connection closed by foreign host.
```

As expected, because of the use of the safe slave, the client is not allowed to execute `exit` which would cause the entire server to exit. And once the client has used up its quota of commands, it is kicked out.

You can see that the use of multiple safe interpreters did not add much complexity to the code.

20.9.2. Implementing domain specific languages

Our second example illustrating the use of slave interpreters involves their use of implementing domain specific languages. A *domain specific language* (DSL) is a programming language that is targeted towards a specific

problem domain. The characteristics that distinguish DSL's from general purpose languages like Tcl arise from the fact that users of the language are often not programmers by profession. Rather, they are experts in their respective domains. Consequently, DSL's are generally simple in syntax and programming constructs, often closer in appearance to natural languages. At the same time, their functions are very high level and directly reflect the terminology and abstractions of the target domain.

Since DSL's are not flexible or powerful enough for writing entire applications, they are themselves implemented in the general purpose language, the *hosting* language, used to write the application. An application may support several DSL's at the same time. For example, a database front-end application may include a DSL for queries, a DSL for report generation and a DSL for screen design.

Some well known and commonly used DSL's include:

- Cascading style sheets (CSS) for web pages. The original motivation for these was to enable graphics designers, **not programmers**, to control the appearance of dynamically generated web pages through a declarative style without having to put their grubby little fingers into the beautiful page generation code.
- The syntax used by most `make` utilities is a DSL that expresses targets, dependencies, build rules etc.
- Even regular expressions syntax is a DSL that allows compact specification of how patterns are to be matched.
- Configuration files such as those used by Puppet or good old Windows `.INI` files can also be viewed as simple DSL's.

Internal and external DSL's

DSL's may be classified as *internal* or *external*. Internal DSL's are syntactically based on the host language and follow a similar structure. External DSL's on the other hand may follow a completely different syntax from the host language, often because the latter's structure is deemed too foreign from the perspective of the domain experts. Imagine for example if an accountant were asked to write auditing rules using C++ syntax[1]! The disadvantage of external DSL's is that you now have to write a parser for the DSL syntax which, although not hard for simple DSL's, does involve a non-trivial amount of work.

Happily, with Tcl we can get the benefits of a natural syntax without having to write an external DSL:

- Tcl's syntax as a list of words closely resembles natural language. For example, `deduct federal 10%` as a Tcl command is easily understood as English.
- There is no need for declarations, typing and other programming artifacts that an accountant may not be familiar with.
- The DSL can be executed using a separate Tcl interpreter with appropriate safety boundaries.
- Separate interpreters also allows unneeded elements of the language to be easily disabled **if so desired**. This is important because the DSL as seen by the domain expert should be as simple as possible.

For demonstration purposes, we will implement a little internal DSL that an accountant might use to generate paychecks. The following program is a sample that an accountant expects to be run against every employee record in a database to generate paychecks.

```
deduct insurance 100
deduct federal 10% when salary between 20000 and 30000
deduct federal 20% when salary above 30000
deduct state 5% when federal above 2500 ❶
generate paycheck
generate paystub
```

❶ Yes, silly I know but that's taxes for you

Our little language should be understandable to an accountant right away. Most hosting languages would need to implement the above as external DSL's. Not so in Tcl becauase **it is standard Tcl syntax**.

[1] It's hard enough for programmers!

We start with a couple of utility procedures to check whether a word is included in a permitted list and to ensure a word is an integer. Nothing to see here.

```
proc check_word {word allowed} {
    if {$word ni $allowed} {
        error "Invalid word '$word'. Must be one of [join $allowed ,]"
    }
}
proc check_number {word} {
    if {![string is integer -strict $word]} {
        error "'$word' is not an integer"
    }
}
```

We will assume employee data is passed to us in a dictionary. We will need a procedure to map the keywords in our DSL to keys in the employee dictionary. Once again, not much to see here either.

```
proc word_to_var {word} {
    dict get {
        salary  Salary
        federal FederalTax
        state   StateTax
        insurance Insurance
    } $word
}
```

As we come to the meat of the implementation, we have a decision to make. The obvious design is to map the commands in the DSL to commands in the safe interpreter that are aliased in the master. At that point, we need to make choice — should the DSL be interpreted or compiled to a Tcl script. In the first case, our alias implementations would be called for every employee record. They would then take the appropriate actions to deduct taxes, print the check and so on. We will go with the second alternative and compile the DSL to a Tcl script. This Tcl script will then be directly evaluated for each employee record. The advantage is that the DSL program need not be repeatedly parsed for every employee record. This method is slightly more complex that the interpreted DSL implementation but since you have already seen metaprogramming in Section 10.8, it should not be hard to follow.

We create a class to store some context — the compiled Tcl script for the DSL code.

```
oo::class create Paymaster {
    variable Script
}
```

Our paymaster needs to be told the rules for generating a paycheck. These rules are provided by our accountant using our DSL. We will pass these in to our paymaster through its constructor.

```
oo::define Paymaster constructor {dslscript} {
    interp create dslengine -safe
    dslengine eval {namespace delete ::} ❶
    foreach alias {deduct generate} {
        dslengine alias $alias [self] $alias
    }
    dslengine eval $dslscript
    interp create calculator -safe
    calculator alias puts puts
}
```

❶ Deleting the global namespace will remove all commands from the slave interpreter

Our constructor needs some explanation. We create not one, but two, slave interpreters. The first one, `dslengine`, is fundamental to our design because it is the one that will parse the DSL script. We remove **all** commands from it and then add the aliases `deduct` and `generate` corresponding to the DSL language commands. These then will be the only commands accessible in that slave.

The second interpreter is not strictly necessary. Once `dslengine` has generated the Tcl script, we could run that in the master interpreter. Its presence is to add another layer of security. We do not trust our accountant. He may be annoyed about his annual raise and create a rule that "deducts" `[exec reformat drive]` instead of a number. Although our aliases will validate arguments, there is a chance that something will slip through. Running the generated script in another safe interpreter protects against oversights. Note we cannot use our `dslengine` for this because it does not have any commands other than the DSL-specific ones.

Next we define our destructor which has nothing to do but release the slaves.

```
oo::define Paymaster destructor {
    catch { interp delete dslengine }
    catch { interp delete calculator }
}
```

Time now to define our aliases that implement the DSL. Only two are needed for our simple language. We will start with the more complex one, `deduct`. Remember our design involves **transpiling** the DSL into Tcl so these methods need to generate fragments of Tcl that reflect the DSL statements. These fragments are concatenated to construct the transpiled Tcl script.

We have already seen techniques for constructing scripts in Section 10.7.1.2. We follow that method here by defining a script template that reflects the logic of the script fragment implementing the `deduct` DSL statement. We then construct the script fragments for each place holder in the template and then replace them at the end with `string map`. We will not go into further detail here as that is not the point of this exercise.

```
oo::define Paymaster method deduct {what deduction args} {
    check_word $what {federal state insurance}
    set category [word_to_var $what]
    if {[regexp {^(\d+)(%)?$} $deduction -> amount percent] == 0} {
        error "Invalid deduction amount $deduction"
    }
    set template {
        if {%CONDITION%} {
            if {%USE_PERCENT%} {
                set deduction [expr {int(($Salary * %AMOUNT%)/100)}] ❶
            } else {
                set deduction %AMOUNT% ❷
            }
            incr %CATEGORY% $deduction ❸
            incr NetAmount -$deduction ❹
        }
    }

    if {[llength $args] == 0} {
        set condition true ❺
    } else {
        if {[llength $args] < 4} {
            error "Incomplete line"
        }
        set args [lassign $args cond_keyword cond_var cond_cmp number]
        check_word $cond_keyword {when if}
        set cond_var [word_to_var $cond_var]
        check_number $number
        switch -exact -- $cond_cmp {
            over - above {
```

```
                set condition "\[set $cond_var\] > $number"
            }
            under - below {
                set condition "\[set $cond_var\] < $number"
            }
            within - between {
                if {[llength $args] != 2 ||
                        [lindex $args 0] ne "and"} {
                    error "Invalid \"$cond_cmp\" arguments"
                }
                set upper [lindex $args 1]
                set condition "\[set $cond_var\] > $number && \[set $cond_var\] < $upper"
            }
        }
    }

    append Script [string map [list \
                                %CATEGORY% $category \
                                %CONDITION% $condition \
                                %AMOUNT% $amount \
                                %USE_PERCENT% [expr {$percent ne ""}]] \
                        $template]
}
```

❶ Deduction is a percentage
❷ Deduction is an absolute value
❸ Allocate deduction to a category...
❹ ...and remove net amount accordingly
❺ No condition for the deduction

The other DSL statement is `generate`. The generation of the script fragment here is much simpler.

```
oo::define Paymaster method generate {what} {
    switch -exact -- $what {
        paycheck {
            append Script {puts "Pay to $Name the amount of $NetAmount"} \n
        }
        paystub {
            append Script {
                puts "Paystub: $Name Salary=$Salary Net=$NetAmount"
                puts "         Fed=$FederalTax State=$StateTax Insurance=$Insurance"
            } \n
        }
        default { error "No means of generating \"$what\"" }
    }
}
```

Now we define the method that will be called by the application to pay an employee. The employee record is passed in the form of a dictionary with the employee's name and salary. It then executes the generated Tcl script within the scope of the dictionary in our `calculator` slave.

```
oo::define Paymaster {
    method pay {emp} {
        set emp [dict merge {
            FederalTax 0
            StateTax 0
            Insurance 0
        } $emp]
        dict set emp NetAmount [dict get $emp Salary]
```

```
        calculator eval [list set emp $emp]
        calculator eval [list dict with emp $Script]
    }
}
```

Our final method is just for the purposes of inspecting the Tcl script we generated. Just so we can see what havoc we have wrought.

```
oo::define Paymaster method script {} {
    return $Script
}
```

We are ready to try our DSL. Let us create our paymaster, passing in the rules for payment. Normally these would be read in from some resource edited by the accountant.

```
Paymaster create paymaster {
    deduct insurance 100
    deduct federal 10% when salary between 20000 and 30000
    deduct federal 20% when salary above 30000
    deduct state 5% when federal above 2500 ❶
    generate paycheck
    generate paystub
}
→ ::paymaster
```

Just for kicks let us see what our generated script looks like.

```
% paymaster script
→
          if {true} {
              if {0} {
  set deduction [expr {int(($Salary * 100)/100)}]; # Deduction is a percentage
              } else {
...Additional lines omitted...
```

Not pretty, but then again neither are we. At least it seems functional so we can try it out.

```
% paymaster pay {Name Guido Salary 25000}
→ Pay to Guido the amount of 22400
  Paystub: Guido Salary=25000 Net=22400
           Fed=2500 State=0 Insurance=100
% paymaster pay {Name Fredo Salary 35000}
→ Pay to Fredo the amount of 26150
  Paystub: Fredo Salary=35000 Net=26150
           Fed=7000 State=1750 Insurance=100
```

Extending this DSL is fairly easy. You just need to add another aliased method for the new statement, for example `reimburse`.

Let us reiterate what we have accomplished. In essence, we have implemented a domain specific language using syntax that would be natural to a non-programmer. Moreover we have done this without needing to write a parser and in just about a page of code.

And if you are not sufficiently impressed, think about how you would make our DSL **localized** so a French accountant could use the terms he is comfortable with. It would only take a few changes with the help of message catalogs (see Section 4.15).

That's Tcl; very few languages are as capable.

20.10. Chapter summary

This chapter described the use of multiple Tcl interpreters within a single application. Multiple interpreters bring two primary capabilities to the language:

- encapsulation of state into an independent computing unit
- provision for safe and controlled environments in which to run untrusted components

We also demonstrated the use of multiple interpreters for two common use cases — network servers and a DSL implementation — that benefit from these capabilities.

21

Coroutines

Programs are composed out of computational tasks which may themselves be recursively decomposed further into subtasks. There may be dependencies between these tasks where a task may depend on the result of one or more other tasks before it can proceed with its own computation. *Concurrency* refers to the ability of two or more such tasks to progress at the "same time". When we say "same time", we do not necessarily mean that both computations are running at any particular instant. The computations may in fact do that, running in parallel on separate processors, or they may be interleaved on a single processor so that each gets some share of the processor time. In either case, they all progress without any particular task having to be completed first.

Parallelism on the other hand refers to tasks being able to execute **simultaneously** on separate processors so that at any instant they are actually all changing state in some fashion. By definition then, parallelism implies availability of multiple processors each of which is running a separate task.

For a short introduction to the two concepts and their semantic difference, see Rob Pike's conference talk.

We have already seen an example of concurrency in Tcl in our discussion of network server implementations. The service provided by a network server to each client can be thought of as a computation task and thus actively servicing multiple clients at the same time through the event loop and asynchronous I/O mechanisms can be thought of as one form of concurrent processing.

We will now look at another form of concurrency using *coroutines*. Like the event loop and asynchronous I/O mechanisms, this allows for concurrent tasks **but not parallel** ones. In the next chapter we will examine yet another alternative — *threads* — which allows tasks to progress not just concurrently, but in parallel as well.

At some level, even a procedure can be viewed as a computational task. It is called with some set of input values, performs a computation based on those values and produces a result which may include side effects. Coroutines are similar to procedures in terms of how they are invoked but with one important difference: they run on their own **separate** call stack that maintains the "state" of the computation. Unlike procedures, which lose their **implicit** internal state (local variables etc.) once they return, coroutines can return values (we call this *yielding*) **while maintaining their internal state**. When invoked again, they can then resume from where they left off.

We will start off by describing the mechanics of creating and using coroutines. That will allow us to then present their motivation, benefits and working in detail through examples in subsequent sections.

21.1. Creating coroutines: `coroutine`

A coroutine is created with the `coroutine` command.

```
coroutine CORO CMD ?ARG ...?
```

This creates a new coroutine context called *CORO* and an associated Tcl command of the same name. The coroutine context consists primarily of a call stack that is used for the execution of all code run in that coroutine's context. The coroutine begins with execution of the command *CMD* which is passed any additional arguments

provided. This command and any further commands it may call will all run off the newly created coroutine call stack. We will describe the call stacks further in a later section.

Although the coroutine runs with a separate call stack, it still runs in the same interpreter context with the same procedure, global and namespace definitions.

The return value of the coroutine commmand is the value returned by *CMD* either when it completes normally, or a value it returns while suspending itself with the `yield` command. Since we have not discussed the latter yet, here is a trivial example of a coroutine that immediately completes normally.

```
proc adder args { return [tcl::mathop::+ {*}$args ] }
coroutine mycoro adder 1 2 3
→ 6
```

This command will create a new coroutine named `mycoro` with its own stack and execute the `adder` procedure within that coroutine context. Our simple procedure computes the sum and immediately returns. The coroutine, along with its associated context, are deleted on the procedure completion. The procedure return value is returned as the result of the `coroutine` command.

All in all, this is just an inefficient way of calling `adder` directly. There is not much point in a coroutine with an independent stack without a means to suspend its execution while preserving the stack context. That's where we will go next.

21.2. Suspending and resuming coroutines

Procedures run to completion returning a result when they are done. As we saw above, coroutines may also behave the same way but their distinguishing ability is being able to suspend their own execution while returning a result to the caller. While suspended, their call stack and context is preserved so that when next invoked they continue to execute from the point of suspension.

This magic is accomplished with the `yield` and `yieldto` commands.

21.2.1. Yielding to the caller: `yield`

Just like the `return` command in the case of procedures, the `yield` command invoked from a coroutine passes control back to the caller. The difference is that the coroutine call stack context is preserved.

```
yield ?RESULT?
```

The command may only be invoked from within a coroutine context, i.e. from the command passed to the `coroutine` command as its *CMD* argument or from any other command invoked from it to any depth. It preserves the state of the current coroutine context and reverts control back to the coroutine's caller returning *RESULT*, which defaults to the empty string, as the result of the coroutine invocation. While the caller continues execution, the coroutine is suspended inside the `yield` command. The coroutine may be called again at any time by its name and passed a single argument at which point it resumes from its suspended state inside of the `yield` command within the coroutine context. The argument passed to the coroutine command is returned as the result of the `yield` command.

From the caller's perspective, the whole sequence of invoking a coroutine looks similar to the following:

```
set result [coroutine CORO CMD ?ARG ...?]
set result [CORO ?ARG?]
set result [CORO ?ARG?]
```

The first line above creates and invokes the coroutine. When the coroutine yields, we can invoke it repeatedly using its name. On each such invocation, the corouting resumes its computation from the point of its suspension.

As always, an example will make it all, ahem, crystal clear.

```
coroutine mycoro eval {
    set j [yield 1]
    yield [incr j]
    return 0
}
→ 1
```

The above command creates a coroutine context called mycoro and a command of the same name. This coroutine begins executing the eval command, **running on a newly allocated stack**. When the first yield command is executed, the coroutine execution is suspended and the passed value 1 is returned to the caller of the coroutine command. This is what we see as the result above.

Since the coroutine **yielded** back to the caller and did not complete via an implicit or explicit return, it remains suspended with its call stack preserved and can be reinvoked with an optional argument.

```
mycoro 10 → 11
```

This passes control back to the mycoro coroutine which resumes execution by returning from the yield command. The result of yield, 10, is assigned to variable j. The coroutine execution continues with the increment of j which is then passed to the second call to yield resulting in mycoro returning 11 above.

We invoke the coroutine one more time.

```
mycoro → 0
```

Here we did not bother to pass the optional argument to mycoro and yield therefore would return an empty string as its result. When the coroutine resumes, the next command is a return, not a yield. The returned value 0 is the result of the mycoro call but having completed execution, the coroutine context and call stack are freed and the corresponding command removed. An attempt to call the coroutine will result in an error.

```
% mycoro
Ø invalid command name "mycoro"
```

There are a couple of other points about yielding from coroutines that should be noted. The first is that though our example yields directly from within a simple eval, a coroutine can yield from any level in nested procedure calls. Our example could also be mutated as follows:

```
proc demo1 {} {
    set j [yield 1]
    demo2 [incr j]
}
proc demo2 {val} {
    yield $val
}
proc demo {} {
    demo1
    return 0
}
```

We can then define the coroutine as below and run it in the same manner as before.

```
coroutine mycoro demo → 1
mycoro 10             → 11
```

You might expect yielding from a nested procedure call to work and wonder why we spell it out explicitly. This is only because some popular languages that support coroutines only allow yielding from the top level of the coroutine context.

The other point to note is that when a coroutine suspends with the `yield` command, it may only be called with at most one argument. Thus a call like

```
mycoro arg1 arg2
```

will fail. This is not a limitation because (a) multiple arguments can be wrapped and passed as a list, and (b) this capability can be easily added with a wrapper around the `yieldto` command that we will see in the next section.

21.2.2. Yielding after initialization

You will often see the main body of a coroutine contain what appears to be an extraneous `yield` before the main section. For example, we want the coroutine `naturals` to return successive integers starting with 1. We may write it in the following straighforward fashion.

```
proc natural_loop {} {
    while 1 {
        yield [incr i]
    }
}
coroutine naturals natural_loop
→ 1
```

However, the very first `yield` executed in a coroutine is the value of the `coroutine` command itself as seen above. So the first natural number is returned as the result of `coroutine` call and the first `naturals` invocation returns 2.

```
naturals → 2
```

This is a little awkard and has to be treated specially. Therefore, you will often find coroutines structured such that there is a special `yield` after the coroutine does any internal initialization required. The purpose of this `yield` is simply to return to the caller. The coroutine can then be invoked by its own name for its core functionality.

Our rewritten example would look like

```
proc natural_loop {} {
    set i 0 ❶
    yield ❷
    while 1 {
        yield [incr i] ❸
    }
}
coroutine naturals natural_loop
naturals
→ 1
```

❶❶ Initialization. Superfluous for our example.
❷ Returns to caller
❸ Core functionality

That gives us the desired behaviour. You will find this pattern more often than not in all but the simplest coroutines.

21.2.3. Yielding to arbitrary commands: yieldto

In Chapter 10, we described the tailcall command which returns from a procedure. However, unlike the return command, tailcall does not pass control back to the caller but instead invokes a target command replacing the call frame of the containing procedure invocation with a call to the target command. The result seen by the original caller of the procedure is then the result of the target of the tailcall.

The yieldto command is analogous to the tailcall command. Like yield, it suspends the coroutine in whose context it is invoked but instead of returning control to the caller, it invokes a target command using the same call frame that was used for the coroutine invocation. As with tailcall, the result of the original coroutine invocation as seen by the caller is the result of the target command invoked via yieldto.

```
yieldto CMD ?ARG ...?
```

Here *CMD* and optional *ARG* arguments comprise the target command to which control is to be passed. Note that *CMD* **may itself be another coroutine**, presenting a way of implementing a "co-operative multitasking" architecture.

Like yield, yieldto can only be invoked from within a coroutine context and results in that coroutine being suspended. It differs from yield in the following respects:

- Whereas yield always returns control to the caller of the coroutine, yieldto will transfer control to the specified target command.
- When a coroutine suspended using yield is resumed, it can be passed at most one argument. On the other hand, a coroutine suspended with yieldto can be passed multiple arguments on resumption. The return value from yieldto when the coroutine is resumed is the list of arguments that were passed to the coroutine invocation.

The first difference is illustrated by the following example.

```
% coroutine mycoro eval {
    yield
    yield abc ❶
    yieldto string length abc ❷
}
% mycoro
→ abc
% mycoro
→ 3
```

❶ Caller sees abc as the return value
❷ Caller sees result of string length as return value

The second difference with respect to yield allows a simple way to construct a yieldm that will behave like a yield in that it returns control to the caller but still allows the coroutine to be resumed with multiple arguments.

```
proc yieldm {{val {}}} {
    yieldto return -level 0 $val
}
```

(You may wish to revisit Section 11.3 for a refresher on the return command's -level option.) The implementation of yieldm uses yieldto to allow the coroutine to be resumed with multiple arguments while passing control to the return command to return the specified value to the caller.

As an example of use, consider implementing a coroutine that accumulates the sum of values passed to it and returns the accumulated total on each call.

```
proc accumulate {} {
    set sum 0
    while {1} {
        incr sum [+ {*}[yieldm $sum]]
    }
}
coroutine accumulator accumulate
→ 0
```

The use of `yieldm` in lieu of `yield` permits the multiple arguments to be passed to the coroutine in an invocation.

```
accumulator 5     → 5
accumulator 1 2 3 → 11
```

The above illustrated one feature of `yieldto`: the ability to accept multiple arguments on resumption of the coroutine. We will find applications of the other feature, handing off control to arbitrary commands, as we move along in this chapter.

21.3. Checking coroutine context: `info coroutine`

The `info coroutine` command returns the name of the current coroutine context or an empty string if called from outside a coroutine context.

```
% info coroutine ❶
% coroutine demo eval {
    yield
    yield [info coroutine]
    yield [info coroutine]
}
% demo
→ ::demo
% rename demo demo2
% demo2 ❷
→ ::demo2
```

❶ Empty string returned because not within a coroutine context
❷ Renaming a coroutine will be reflected in `info coroutine` as well

The command is generally used in library routines that may change their behaviour based on whether they are running within a coroutine context or not. For example, the `coroutine::auto` package in Tcllib [1] provides a set of I/O commands that use this to hide differences between coroutine and non-coroutine contexts. It implements a version of `read`, `gets` etc. that will behave like the standard `read` outside of a coroutine context. Within coroutines however, it will only block the calling coroutine while yielding to permit other coroutines and non-coroutine scripts to run.

21.4. Coroutine termination

A coroutine and its associated command will continue to exist until it runs to normal completion, raises an uncaught exception, or until the command is deleted with the `rename` command. We will look at exceptions in Section 21.5 so here we will consider just the other two methods.

The first of these is simple — when the command used to create the coroutine completes, the coroutine is also terminated.

[1] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

```
% coroutine demo eval {yield 0; return}
→ 0
% demo
% demo ❶
Ø invalid command name "demo"
```

❶ Error because the coroutine has completed.

There is a variation of this when a coroutine wants to terminate from within a nested procedure call. This is shown below.

```
% proc exit_coroutine {} {
    return -level [info level]
}
% proc demo_proc {} {
    puts "In demo"
    yield
    exit_coroutine
    puts "Exiting demo"
}
% coroutine demo demo_proc
→ In demo
% demo
% demo
Ø invalid command name "demo"
```

Notice that the `Exiting demo` message is never printed and moreover, the second call to demo fails as the coroutine no longer exists. See Section 11.3 if you need a refresher on the `-level` option of the `return` command.

An alternate method of terminating a coroutine is to `rename` the corresponding command to the empty string. The coroutine may even commit suicide by renaming itself within its own context. In such a case, the coroutine will continue to run until it yields at which point it will be deleted. The following example illustrates this.

```
% coroutine mycoro eval {
    yield
    rename mycoro "" ❶
    puts "Goodbye cruel world!" ❷
    yield ❸
    puts "Exiting coroutine"
}
% mycoro
→ Goodbye cruel world!
% mycoro ❹
Ø invalid command name "mycoro"
```

❶ Deletes the coroutine command.
❷ Coroutine continues to run until the next `yield`
❸ Coroutine destroyed after yielding and cannot be called again.
❹ We never see the `Exiting coroutine` message

There is one important difference in the two methods described above. Use of the `return` command can be caught within the coroutine context to prevent termination of the coroutine. This is not possible with the `rename` method which will terminate the coroutine with extreme prejudice after the next yield. Your choice will depend on what's appropriate for your application scenario.

21.4.1. Releasing resources on termination

The fact that termination via the `rename` command will not run any exception handlers has ramifications with regards to resource management. Let us look at the following simple example which returns a line from a file on each invocation.

```
coroutine getline {*}[lambda path {
    set chan [open $path]
    try {
        while {[gets $chan line] >= 0} {
            yield $line
        }
    } finally {
        close $chan
    }
}]
```

The code takes care to ensure that the opened channel is closed on both normal and exceptional completions. However, consider what happens if the coroutine has yielded and the caller does a

```
rename getline ""
```

The `rename` would literally make the coroutine just go Poof! — "disappear" without a trace. In particular the `finally` handler would never run leaving the file open resulting in channels being leaked.

A robust design warrants that we should guard against this possibility and variable traces provide a way. We can write the coroutine / procedure as follows:

```
coroutine getline {*}[lambda path {
    set chan [open $path]
    trace add variable DUMMY unset [list close $chan]
    while {[gets $chan line] >= 0} {
        yield $line
    }
}]
```

Now when the DUMMY variable goes out of scope, whether it is by the coroutine completing normally or with an exception or **even via the coroutine being deleted as described above**, our trace will trigger and close the file.

Note that we did not need to even have a value assigned to DUMMY (see Section 10.6.1.1).

21.5. Exception handling in coroutines

Any exceptions that are not caught within a coroutine are propagated to the top level of the coroutine context and passed back to the caller of the coroutine. Moreover, the **coroutine** context and associated command are deleted so that the coroutine cannot be invoked again.

We can verify this behaviour by passing our `accumulator` coroutine a value that is not a number.

```
% accumulator notanumber
Ø can't use non-numeric string as operand of "+"
% accumulator 3 ❶
Ø invalid command name "accumulator"
```

❶ Fails because the above error deletes the coroutine

In many cases, it is desirable to not terminate the coroutine but simply notify the caller of the error while preserving the coroutine context so it can be called again. We can accomplish this by combining `yieldto`

with one of the exception raising commands. We demonstrated this technique by modifying our `accumulator` implementation as shown below.

```
proc accumulate {} {
    set sum 0
    while {1} {
        if {[catch {
            set part_sum [+ {*}[yieldm $sum]]
        } result ropts]} {
            set part_sum [+ {*}[yieldto return -options $ropts $result]]
        }
        incr sum $part_sum
    }
}
coroutine accumulator accumulate
→ 0
```

Our modified implementation catches any exceptions raised in the sum operation so that they are not propagated to the top level of the coroutine which is therefore not terminated. At the same time, we need to pass the exception with its original error stack to the caller. We already saw how we can do this for the normal procedural case in Section 11.6.1. Here we cannot use that technique directly because that would again terminate the coroutine. Instead we use `yieldto` to pass the buck. That will suspend the coroutine and run the `return` target command in the context in which the coroutine was invoked. Thus we achieve the goal of returning the exception while preserving the coroutine.

Now when an error occurs, an exception is raised to the caller as before but the coroutine is not terminated and can be invoked again.

```
% accumulator 1 2 3
→ 6
% accumulator notanumber
Ø can't use non-numeric string as operand of "+"
% set errorInfo ❶
→ can't use non-numeric string as operand of "+"
      while executing
  "+ {*}[yieldm $sum]"
      invoked from within
  "accumulator notanumber"
% accumulator 4
→ 10
```

❶ Notice the error stack of the error is preserved as promised

21.6. Variable scopes in coroutines

As we have stated earlier, every new coroutine begins life with its own spanking new stack. The execution however begins in the global context. This is important to keep in mind as the following example shows.

Suppose we had written our `naturals` coroutine example from an earlier section as follows:

```
coroutine naturals eval {
    yield
    while {1} {
        yield [incr i]
    }
}
```

This seems to work as well as the previous version and does not need a separate procedure definition.

```
naturals → 1
naturals → 2
```

The problem is that the code actually modifies the variable `i` in the **global** scope.

```
set i    → 2
naturals → 3
set i    → 3
```

This is undesirable as global variables should be avoided as far as possible. For example, the above implementation would not permit two independent natural number generators to co-exist. Their use of the `i` global would cause interference between themselves.

For this reason, it is advisable to have the top level of a coroutine to be invoked as a named or anonymous procedure in all but the simplest cases. Any use of variables then is within the scope of that procedure. We follow this advice in all but the simplest of our examples.

21.6.1. Private variables

The fact that a coroutine runs on its own stack can be taken advantage of to implement "private" variables that are accessible only from procedures running in that coroutine's context. This is illustrated by the example below from Tcler's Wiki[2]. This defines a procedure `corovars`:

```
proc corovars {args} {
    foreach var $args {
        lappend vars $var $var
    }
    tailcall upvar #1 {*}$vars
}
```

The procedure is simply a wrapper to link local variables in the caller to the local variables of the same name at level 1 in the coroutine stack. It can be used as a "declaration" similar to how the `global` or `variable` commands are used to declare variables in the global and namespace scopes except that it declares variables in the "coroutine" scope.

Here is a demonstration of its use.

```
proc init {} {
    corovars x y
    set x 100
    set y 200
}
proc getx {} {
    corovars x
    yield $x
}
proc demo_helper {} {
    init
    yield
    getx
}
coroutine demo demo_helper
demo
→ 100
```

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

Notice how the "coroutine-private" variable x is referenced from two different procedures running in the coroutine context while remaining inaccessible outside.

21.7. Coroutines, uplevel and upvar

A point to be noted about coroutines vis-a-vis procedure calls is that because coroutines run off an independent stack, the uplevel and upvar commands do not work the same way in the two cases. A procedure may use these commands to run code or access variables in the caller's context. This is not possible with coroutines. We can see this simply by checking the `info level` command.

```
% coroutine mycoro while 1 {yield [info level]}
→ 0
% proc demo {} {puts "demo: [info level]"; puts "coro: [mycoro]"}
% demo
→ demo: 1
  coro: 0
```

Notice the coroutine is running in the global context and has no access to the stack of the caller demo. Of course, all nested procedure calls within a coroutine context can use upvar and uplevel as normal within that context.

There is however a way to emulate the functionality of upvar and uplevel with some explicit programming. The idea is to use `yieldto` to execute a script that does the needful in the caller's context and then calls back into the coroutine. The technique, accompanied with code and an explanation, is described in TIP #396: Symmetric Coroutines, Multiple Args, and yieldto.

21.8. Coroutines and multiple interpreters

Coroutines are limited to the interpreter in which they are defined. For example, the following generates an error.

```
% set ip [interp create]
→ interp0
% coroutine mycoro $ip eval {yield 1}
Ø yield can only be called in a coroutine
```

The error occurs because the coroutine context is defined within the master interpreter but the attempt to yield is made from the slave. If your situation calls for running coroutines within a slave, the coroutine must be defined in that slave. Then you can access it using `interp eval`.

```
$ip eval {
    coroutine mycoro try {yield 1; yield 2; yield 3}
}
$ip eval mycoro
→ 2
```

Optionally, you can create an alias for some syntactic sugar.

```
% interp alias {} mycoro $ip mycoro
→ mycoro
% mycoro
→ 3
```

21.9. Code injection

When a coroutine is resumed, it continues execution from the statement following the `yield` or `yieldto` command that had suspended the coroutine. The `inject` command offers a means of inserting code into the

coroutine at the point of resumption so that the next time the coroutine is executed, it resumes with the inserted code instead of with the statements following the `yield`.

The syntax of the command is

```
tcl::unsupported::inject CORO CMD ?ARG ...?
```

The first thing to note is that the command lies in the `tcl::unsupported` namespace. This means it is still experimental and may change syntax, behaviour or even be removed in future releases. It would not be wise to rely on it in production code despite Tcl's history of stability and compatibility. We describe the command here because it can be useful for debugging and troubleshooting.

The command arranges for *CMD* to be executed, within the context of the coroutine *CORO*, with any provided *ARG* arguments the next time *CORO* is invoked.

Let us define the following simple coroutine.

```
coroutine slogan apply {{} {
    while {1} {yield ; puts "Tastes great!"}
}}
```

Every time we invoke it, it returns from `yield` and prints our slogan and yields again.

```
% slogan
→ Tastes great!
```

Now we inject other code to run when the coroutine is invoked.

```
% tcl::unsupported::inject slogan puts "Less filling!"
```

Note that our code is not run yet. It will be executed when the coroutine is invoked next.

```
% slogan
→ Less filling!
  Tastes great!
```

As you see above, when the coroutine is resumed, it first runs our injected code, printing `Less filling!`. The injection is a one-time deal, not permanent so if you call the coroutine again, it will be back to the old behaviour.

```
% slogan
→ Tastes great!
```

Note that the coroutine can run valid commands in the coroutine context. For example, it could print and yield right away before the original code runs.

```
% tcl::unsupported::inject slogan try {puts "Less filling!" ; yield}
% slogan
→ Less filling!
```

Or even terminate the coroutine by executing a `return`.

```
% tcl::unsupported::inject slogan return
% slogan
```

The `return` causes the coroutine to complete on the following call we made above. So a succeeding attempt to run it again will meet with an error.

```
% slogan
→ invalid command name "slogan"
```

The primary purpose for `inject` should be for inspecting the state of a coroutine that has yielded. For example, a coroutine might have multiple yield points. To figure out where it has yielded and other state information, you can inject a command to dump the stack and yield. Then invoke the coroutine to collect the information.

As you can imagine, code injection can modify coroutine behaviour in powerful ways but its use should be limited to debugging and troubleshooting until it is moved out of the `unsupported` namespace.

21.10. Using coroutines

Having discussed the mechanics of working with coroutines, we now explore the motivation behind them and their use in some common software patterns. A key point to remember is that coroutines do **not** add any new capabilities to the language. What they do is enable simpler program structure that is easier to write and understand. We will attempt to justify that claim in the next few sections.

21.10.1. Explicit and implicit state

To understand how coroutines can simplify programming, let us first expound a little about the state encapsulated by a computation. This state can be thought of as being a combination of *explicit* and *implicit* states. The former is named such because the program has to explicitly maintain the state as a collection of global, namespace and object variables. The implicit state on the other hand is automatically kept (well, kind of) in the form of the call stack as execution proceeds through a sequence of procedure calls. At any point of time, the explicit and implicit states together tell us "where we are in the computation".

Let us illustrate the difference between the two with an example. Consider implementing a task which given a list will return the next element in the list on each subsequent call. When the end of the list has been reached, it should wrap back to return elements from the beginning of the list.

Our first implementation will use objects. We could use namespaces as well but that would involve a little more bookkeeping code to deal with the case where we want to use multiple such instances.

```
oo::class create Lgen {
    variable List
    variable Index
    constructor {l} {
        set List $l ❶
        set Index -1
    }
    method next {} {
        incr Index
        if {$Index >= [llength $List]} {
            set Index 0
        }
        return [lindex $List $Index]
    }
}
Lgen create onotes {do re mi}
→ ::onotes
```

❶ We will not bother with error conditions like empty lists

The state of computation is maintained through the member variable `List` and `Index`. Not exactly rocket science because the state information is very simple. We can verify that our implementation meets our requirements.

```
onotes next → do
onotes next → re
```

In contrast, here is a coroutine based implementation that does the same thing through a coroutine.

```
proc list_iterator {lst} {
    yield
    while {1} {
        foreach elem $lst {
            yield $elem
        }
    }
}
coroutine conotes list_iterator {do re me}
```

Just to prove that it works, we try it out.

```
conotes → do
conotes → re
```

The point of our little exercise was to contrast explicit and implicit state. The object based implementation stores explicit state, in the form of the object member variables `List` and `Index`, to keep track of the value to be returned on the next call. On the other hand, in the coroutine-based implementation, the state information is implicitly contained in the `while` and `foreach` loops. This makes the intent of the code more immediately obvious compared to the object-based implementation. This is true even in our trivial example but the difference can be even more pronounced in real world programming tasks with more complex program states.

21.10.2. Generators

One of the simplest uses for coroutines is for implementing *generators*. A generator is a command that returns successive elements from a collection on each invocation. Our `naturals` coroutine and the example in the previous section were essentially generators. In the former case, the "collection" did not actually exist in concrete form (it is infinite after all) but its elements were generated on demand. In the latter case, the generator returned elements of an existing simple (repeated) list.

Here we illustrate a couple of slightly more complex examples of generators, again with the purpose of demonstrating how coroutines simplify their implementation.

Consider a collection of integers stored as a nested list of **arbitrary** depth.

```
% set nested {0 {1 {2 3}} {4 5}}
→ 0 {1 {2 3}} {4 5}
```

Writing a recursive routine to invoke a command on each element of such a tree-like structure is very straightforward.

```
proc iterate {l cmd} {
    set n [llength $l]
    if {$n > 1} {
        foreach e $l {
            iterate $e $cmd
        }
    } elseif {[llength $l] == 1} {
        {*}$cmd [lindex $l 0]
    }
}
```

We can then apply some arbitrary operation to each element, for example output its square.

```
% iterate $nested [lambda i {puts [expr {$i*$i}]}]
→ 0
  1
  4
...Additional lines omitted...
```

However, this is not what we are after. What we actually want is an iterator command that would return the successive integers from this collection on each call.

Generators versus callbacks

The callback model of iteration where a script or command is invoked for every element in a collection is simple and adequate for many uses. However, generators are more flexible in some regards, for instance when the elements are not processed immediately. An example might be if each element was a token to be passed to a client on connecting. Clearly, this would not be suitable for a callback based implementation. Other cases where callbacks are not suitable is when processing of the element depends on significant local state which cannot be passed in the callback although `uplevel` and `upvar` can help some in that regard.

Converting recursive code to an iterator command is normally not possible because by its very nature the recursion encodes the state of the computation and will be lost when the iterator returns. On the other hand, writing a non-recursive implementation that explicitly keeps track of state involves quite a bit more bookkeeping.

Coroutines come to the rescue. Because they can return values while preserving execution state, we can easily wrap a coroutine around the above to construct an iterator command.

```
proc iterator_wrapper {collection} {
    yield
    iterate $collection yield
}
coroutine iterator iterator_wrapper $nested
```

We can now retrieve one element at a time.

```
iterator → 0
iterator → 1
iterator → 2
```

To better appreciate the simplicity of the above iterator implementation, the reader might want to write one without using coroutines.

Many algorithms, such as tree traversal above, are most clearly expressed in recursive form. In many cases, coroutines can encapsulate these recursive algorithms in a manner that allows them to be used in an iterative manner. Our example was a simple illustration of this.

As another more real world example assume we want an iterator that will return each file or directory within a directory tree. We saw the `fileutil::find` command in Chapter 9 that will return a list of files within a directory.

```
% print_list [fileutil::find c:/temp]
→ c:/temp/fromDir
  c:/temp/newDir
  ...Additional lines omitted...
```

However, we do not want this entire list of files at one shot for many reasons. It may consume too much memory, we may only be interested in the first few entries and so on. We would therefore like an iterator that will return a single entry at a time. Now it so happens that the `fileutil::find` command takes a second optional argument that is the name of a filter command. This command is invoked for each file or directory. If the command returns a boolean true value, the file is included in the returned list. We will misappropriate this feature for our own use.

```
proc yield_one {fname} {
    yield [file join [pwd] $fname]
    return 0
}
proc file_iterator {dir} {
    yield
    fileutil::find $dir yield_one
}
```

We have cunningly fooled `fileutil::find` to yield as part of its filter functionality. Let us try it out.

```
% coroutine nextfile file_iterator c:/temp
% nextfile
→ C:/temp/fromDir
% nextfile
→ C:/temp/newDir
% while {[set next [nextfile]] ne ""} { puts $next }
→ C:/temp/fromDir/fileA.txt
  C:/temp/fromDir/subDir
  C:/temp/newDir/fileA.txt
  ...Additional lines omitted...
```

Once again, the benefits of coroutines are best understood by implementing the above example without their use. **The ability to return values from a computation while preserving its implicit state greatly simplifies programming.**

21.10.2.1. The `generator` package

Generators are commonly used in programming and have essentially the same structure in most instances and need to support the same set of high-level operations such as map and fold, something we have not discussed in our earlier examples. Thus it makes sense to implement a framework that provides the common functionality required for generators. The `generator` package in Tcllib[3] does exactly that.

The package covers a lot of functionality with many commands and is well documented so we will only introduce basic usage here. You are encouraged to read its documentation to get a feel for all the ways it can be used.

```
package require generator
→ 0.1
```

All commands in the package are implemented via the `generator` ensemble command.

Our first example is to recreate the `naturals` command we implemented earlier, this time using the `generator` package. The `generator define` command is used to define the generator implementation.

```
generator define naturals {} {
    while 1 { generator yield [incr i] }
}
→ ::naturals
```

[3] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

This looks similar to our earlier implementation of `naturals` but there are a couple of notable differences. The obvious one is that the implementation uses `generator yield`, and not `yield` to yield values. Less obvious is that invoking `naturals` will not return the next natural number. Rather, it returns a *generator instance* which can then be used to iterate through the sequence.

```
% set nseq [naturals]
→ ::generator::generator1
```

The `generator next` command returns one or more elements from the generator instance.

```
generator next $nseq first               → (empty)
generator next $nseq second third fourth → (empty)
puts "$first, $second, $third, $fourth"  → 1, 2, 3, 4
```

Of course, we may create additional generator instances. These will all be independent.

```
set nseq2 [naturals]      → ::generator::generator2
generator next $nseq2 val → (empty)
set val                   → 1
```

The generator instances must be released with `generator destroy` when no longer required.

```
generator destroy $nseq $nseq2 → (empty)
```

So far, it does not appear there is a whole lot we gain from using the package as opposed to raw coroutines. The real benefits come from the additional commands in package that implement operations like iteration, folding, mapping and filters.

Iteration

The `generator foreach` command is like Tcl's `foreach` except that it iterates over the elements returned by a generator. We can iterate over the sequence of natural numbers using a very natural (pun unintended) syntax.

```
generator foreach n [naturals] {
    puts $n
    if {$n >= 2} {
        break
    }
}
→ 1
  2
```

On completion of the loop, the generator instance (returned by `naturals` above) used for the loop is automatically destroyed so we do not have clean up.

We have to `break` out of the above loop because `naturals` produces an infinite sequence. A finite sequence on the other hand will terminate the loop naturally. The next example defines such a finite generator, all integers between a specified range. We then iterate over it as above.

```
generator define range {low high} {
    for {set i $low} {$i <= $high} {incr i} { generator yield $i }
}
generator foreach n [range 4 6] { puts $n }
→ 4
  5
  6
```

Filtering

The `generator filter` command creates a new generator containing elements of an existing generator that satisfy a given predicate. Here we apply odd as the predicate condition on a range and iterate over the resulting generator.

```
proc odd {n} { tcl::mathop::& $n 1 }
generator foreach n [generator filter odd [range 2 8]] {
    puts $n
}
→ 3
  5
  7
```

Reducing

Another common operation is implemented by the `generator reduce` command. This combines an initial value supplied by the caller with each successive element by applying a specified operator.

```
generator reduce ::tcl::mathop::+ 0 [range 1 4] → 10
generator reduce ::tcl::mathop::* 1 [range 1 4] → 24
```

Map

The last operation we will mention is `generator map` which creates a new generator by applying a function to an existing generator.

```
set powers_of_2 [generator map {::tcl::mathop::** 2} [naturals]]
generator next $powers_of_2 i j k
puts "$i, $j, $k"
→ 2, 4, 8
```

These examples only provide a flavor of the `generator` package. The most important takeaways from these examples are:

- The elements of generated sequences are not actually instantiated until they are actually needed. This not only allows them to represent infinite sequences but also results in greatly reduced memory requirements for long finite ones.
- Despite the above, transforms and other operators can be applied to the generators as they would be for scalar values. The last example is an illustration of this.

21.10.3. Emulating objects

Coroutines can be used to implement simple object based frameworks. Here is the implementation of a simple calculator object.

```
proc plus {args} {
    corovars acc
    set acc [tcl::mathop::+ $acc {*}$args]
}
proc mul {args} {
    corovars acc
    set acc [tcl::mathop::* $acc {*}$args]
}
proc calculator {} {
    set acc 0
    set args [yieldm]
```

```
    while 1 {
        if {[llength $args] == 0} {
            set args [yieldm $acc]
        } else {
            switch -exact [lindex $args 0] {
                mul  -
                plus {
                    {*}$args
                    set args [yieldm]
                }
                default {
                    set args [yieldto error "Unknown method [lindex $args 0]"]
                }
            }
        }
    }
}
```

Our implementation could be simpler in the case of our example but we wanted to illustrate the use of private variables using the `corovars` procedure defined earlier to serve as member variables. It is even possible to extend this to support prototype-based inheritance with the help of the `yieldto` command.

Proof that our calculator works as designed:

```
% coroutine calc calculator
% calc plus 1 2
% calc mul 5
% calc div 2
Ø Unknown method div
% calc
→ 15
```

Of course our little object system is missing pretty much all the features that Tcl's native OO system provides. But there are situations where a simple object type interface is useful in conjunction with uses of coroutines, for example, the multitasking system we describe in Section 21.10.6.

21.10.4. Producers, consumers, transformers and filters

The producer-consumer pattern is a common pattern in software where the former generates values out of thin air, figuratively speaking, and the latter consumes them for its own internal purposes. Related abstractions are transformers and filters. Both act as consumers, by retrieving values from a producer, as well as producers, by passing those values to a consumer. The difference between the two is that transformers pass on **all** values after potentially modifying them whereas filters may pass on only a subset of them. This has an effect on how they are implemented.

Below we discuss implementation of all these roles in the context of coroutines.

Producers

A coroutine behaves like a producer by returning values to the caller with the `yield` command. Because it is a producer, it does not take any input and in effect ignores the result of `yield` command itself.

We have already seen simple producers, such as the `naturals` example earlier, but it being a legal requirement to include a Fibonacci sequence generator in every discourse on programming, below is a producer that generates such a sequence.

```
proc fibonacci_generate {} {
    yield
    set prev 0
    set fib 1
    while 1 {
        yield $fib ❶
        lassign [list $fib [incr fib $prev]] prev fib
    }
}
coroutine fib_producer fibonacci_generate
```

❶ Note result of `yield` thrown away

Consumers

A consumer is a parasite, ingesting values without giving anything back in return. Thus a coroutine playing the role of a consumer does not pass any arguments to `yield` but processes its result. Our simple consumer simply prints what's passed to it.

```
coroutine print_consumer while 1 {puts [yield]} ❶
```

❶ Note no arguments passed to `yield`

Transformers

Transformers have both input and output and therefore pass arguments to `yield` as do producers, as well as use its return values as do consumers. We can write a filter that squares values before passing them on.

```
 coroutine squarer {*}[lambda {} {
     set val [yield]
     while 1 {
         set val [yield [expr {$val*$val}]]
     }
}]
```

We can now hook up the producer and consumer with an intermediate transformer following the pattern

```
CONSUMER [TRANSFORMER [PRODUCER]]
```

In our example then,

```
print_consumer [squarer [fib_producer]] → 1
print_consumer [squarer [fib_producer]] → 1
print_consumer [squarer [fib_producer]] → 4
```

You can of course extend this chain by adding on additional transformers.

Filters

Filters are a generalized form of transformers in that they remove the restriction of a one-to-one correspondence between input and output values. Most commonly they pass on only a subset of input values (possibly modified) but could potentially insert additional values as well. This generalization means the form we showed above cannot be used for filters. For instance, imagine we had a filter that would only pass through even numbers. We cannot print a list of even fibonacci numbers as

```
print_consumer [evens [fib_producer]]
```

The problem is that if the generated number is not even, the filter cannot pass it to the consumer and nor can it call the producer to retrieve the next number as it is passed values, not the producer command itself. We have to therefore write filters in a slightly different form where we pass the producer command to the filter when it is created.

Our implementation of the a producer-filter-consumer pipeline that prints even fibonacci numbers would look like

```
proc filter_proc {producer} {
    yield
    while 1 {
        set number [$producer]
        if {[expr {$number & 1}] == 0} {
            yield $number
        }
    }
}
```

We can then invoke the pipeline as

```
coroutine fib_producer2 fibonacci_generate    → (empty) ❶
coroutine even_fibs filter_proc fib_producer2 → (empty)
print_consumer [even_fibs]                    → 2
print_consumer [even_fibs]                    → 8
```

❶ Fresh copy of our generator

An alternative to the above would be to have the consumer also be driven by the filter as shown below. As a variation, here we also **insert** additional output elements by preceding each the fibonacci number by its position in the **full** Fibonacci sequence.

```
coroutine fib_producer3 fibonacci_generate ❶
coroutine even_fibs2 {*}[lambda {producer consumer} {
    yield
    while 1 {
        incr seq
        set number [$producer]
        if {[expr {$number & 1}] == 0} {
            yieldto $consumer "Position: $seq"
            yieldto $consumer "Number:   $number"
        }
    }
}] fib_producer3 print_consumer
```

❶ Fresh copy of our generator

Trying it out,

```
even_fibs2 → Position: 3
even_fibs2 → Number:   2
even_fibs2 → Position: 6
even_fibs2 → Number:   8
```

In this last example, we use `yieldto` to call the consumer. As an exercise, consider how this is different from directly calling the consumer coroutine instead of via `yieldto`.

21.10.4.1. Coroutines and the event loop

Coroutines are invoked just like any other command so there is no great magic involved in calling them via the event loop. However, interaction with the event loop comes up in some common situations, like I/O and multi-tasking, that we describe in later sections so let us go through a simpler example first.

We would like to print the space taken up by each directory in a directory tree. Given that this might take some time on a large file system, we do not want to block the rest of the application in the meanwhile. One way to do this is through a coroutine that reschedules itself through the event loop after doing a block of work. We first define the worker procedure to recurse through the directory tree.

```
proc print_dir_size {dir} {
    set total 0
    foreach fn [glob -nocomplain [file join $dir *]] {
        incr total [file size $fn]
        if {[file isdirectory $fn]} {
            incr total [print_dir_size $fn]
        }
    }
    puts "$dir: $total"
    after idle after 0 [info coroutine]
    yield
    return $total
}
```

The script is very simple, in part because it ignores errors and special files such as links. After processing each directory, it reschedules itself using the idiom described in Chapter 15. It then gives up control through `yield`, allowing other parts of the application to run. Note the call to `info coroutine` which returns the name of the current coroutine. The event loop will then call back into the coroutine allowing it to resume.

Now if you run this as a coroutine in `wish` or `tclsh` with the event loop running, you will see the output being printed **without the rest of the application being blocked**.

```
% coroutine print_windir print_dir_size c:/windows/system32
→ c:/windows/system32/0409: 0
  c:/windows/system32/AdvancedInstallers: 3252576
  c:/windows/system32/AppLocker: 0
  c:/windows/system32/appraiser: 2532564
  c:/windows/system32/ar-SA/Licenses/OEM/Core: 0
  ...Additional lines omitted...
```

It is possible to rewrite the above without the use of coroutines but the resulting solutions are significantly more complex as in every round trip through the event loop you have to explicitly keep track of the current depth in the directory tree and the node within that depth.

We might have given the impression from our examples so far that coroutines are primarily useful in conjunction with recursive algorithms. That is not so. Coroutines are useful any time significant state information is retained in the call stack. Recursive algorithms are just a special category of these and their simple nature makes them particularly suitable for illustrative purposes.

21.10.5. Emulating blocking calls: `coroutine::util`

As you might have noticed from earlier chapters, particularly those dealing with I/O, code that uses synchronous blocking calls is both simpler to write and easier to read than its asynchronous counterpart. The drawback of course is that the entire application is blocked if a channel is not ready. Let us look at how coroutines might allow us the benefit of a synchronous style without its drawbacks.

Consider a simple application that acts as a network proxy connected to two remote end points. Its sole purpose in life is to copy data received on a connection to the other connection and vice versa. In real life, such a proxy might have a wide variety of uses such as firewalling, load balancing, protocol translation etc.

The core function of the proxy is very simple and ignoring peripheral matters such as connection management, error handling etc., it can be implemented by the following procedure.

```
proc proxy {from to} {
    while {[gets $from data] >= 0} {
        puts $from $data
    }
}
```

The procedure copies data from an input channel to an output channel. (There is an implicit assumption here that the two parties are using a text-based protocol.) Our application then is **conceptually** just a pair of calls, one for each direction of the connection.

```
proxy $chan_a $chan_b
proxy $chan_b $chan_a
```

This of course does not work. There are two problems that need to be solved:

- The I/O calls to read data should not block the application.
- The second call needs to run concurrently with the first whereas as written above it will not be invoked until the first completes.

Our coroutine-based version solves both of the above but needs the `coroutine` package from Tcllib[4]. This package includes several utility procedures that are useful when running in a coroutine context. Once again, this requires the Tcl event loop to be running.

```
package require coroutine → 1.1.3 ❶
```

❶ Required for coroutine::util::gets

We can then write our implementation as follows.

```
proc proxy {from to} {
    while {[coroutine::util gets $from data] >= 0} {
        puts $to $data
    }
}
coroutine atob proxy $chan_a $chan_b
coroutine btoa proxy $chan_b $chan_a
```

Before we get into the explanation of the code, note the similarity of structure with the (non-working) synchronous version we showed earlier. Within the `proxy` procedure we have replaced the `gets` call with a call to `coroutine::util gets`. This takes care of our first problem, blocking. Then instead of calling `proxy` directly, we call it through a coroutine. This takes care of the second problem, concurrency.

The key to the implementation is the `coroutine::util gets` command. This command has the same API as the standard Tcl `gets` command but is intended to be called from a coroutine context so that instead of blocking as `gets` does when no data is available, it invokes `yield` to allow other code to execute while keeping the caller, `proxy` in our case, suspended. It uses the event loop behind the scenes similar to what we described earlier to get

[4] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

called back when data is available. It is well commented and self-explanatory so we will not repeat it here. You can see it in the `coroutine` module of Tcllib or in the online repository[5].

In addition to `gets`, the `coroutine` package includes emulation of several other blocking calls, not all of which deal with I/O. For example, here is a short fragment that keeps writing a message to the console at regular intervals that uses an emulation of the blocking mode of the Tcl `after` command.

```
coroutine print_loop while 1 {
    coroutine::util after 1000; puts "Still going..."
}
```

The Tcl `after` command with a single argument will block until the specified interval has elapsed. The `coroutine::util after` command will block from the coroutine caller's perspective but **will yield under the covers allowing other parts of the application to run**.

21.10.6. Co-operative multitasking

We now move on to our final illustration of coroutine use — a framework for co-operative multitasking or *green threads*. Unlike preemptive multitasking which uses native threads at the operating system level, green threads are implemented purely at the user level and require each thread of computation to voluntarily give up the CPU to allow other threads to run. Hence the term *co-operative* multitasking. Tcl also supports native threads as discussed in detail in Chapter 22. We will compare and contrast the two in Section 22.15.

Actors versus Communicating Sequential Processes (CSP)

There are two commonly used models of concurrency — *actors* and *communicating sequential processes (CSP)*. The primary difference between the two is that the former uses asynchronous message passing as the communication mechanism. Messages are sent using a "fire and forget" policy where the sender does not wait for the message to be delivered and processed by the receiver. In the CSP model, messages are sent through synchronous *channels* (not to be confused with Tcl I/O channels) where the sender is blocked from proceeding until the receiver accepts the message. Both models can be implemented with Tcl coroutines as evinced by the fiber[6] and ycl coro relay[7] packages which implement a form of the actor model, and the csp[8] package which implements CSP.

An example actor implementation

For our illustration of coroutines for co-operative multitasking we will construct a package, `task`, that implements the actor model. Our concurrent units of computation are called tasks. They are implemented as coroutines and voluntarily give up control at appropriate times by yielding to a dispatcher which then gives other tasks a chance to run. Tasks may be completely independent of each other or may interact through *messages*.

Our package is rudimentary and is meant to highlight key considerations involved in implementing a co-operative multitasking framework in Tcl. Amongst other things, a production level package will need to be both more featureful and more robust with better error handling and recovery, resource limits on message queues, a more sophisticated scheduling algorithm to prevent starvations and so on.

We start off by importing the `coroutine` package from Tcllib[9] and defining our package namespace.

```
package require coroutine
namespace eval task {
    variable tasks {}
}
```

[5] http://core.tcl.tk/tcllib/dir?ci=tip&name=modules/coroutine
[6] https://sourceforge.net/projects/tclfiber/
[7] http://wiki.tcl.tk/44051
[8] https://github.com/securitykiss-com/csp/
[9] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

Creating a new task

The `task::task` command creates a new task in which the provided script will be evaluated.

```
proc task::task {script} {
    variable next_id
    variable tasks
    variable tasks_added

    set task_id [namespace current]::task#[incr next_id]
    dict set tasks $task_id State INIT
    dict set tasks $task_id Messages {}
    set tasks_added 1
    coroutine $task_id apply {script {
        yield
        try $script finally { task::cleanup [task::myself] }
    }} $script
    wakeup_dispatcher
    return $task_id
}
```

The `tasks` variable is a dictionary of active tasks keyed by task identifiers which are dynamically generated using the `next_id` counter. The task identifiers also serve as the name of the coroutine hosting the task. To create a new task, the command initializes the state of the task to `INIT` with an empty `Messages` that will hold any messages sent to it. It then creates a coroutine context in which to run the supplied script. As a general principle, we prefer to run the script within an anonymous procedure so as to avoid potentially polluting the global namespace.

The coroutine immediately yields on entry rather than executing the task script. The reason for this is that we want all execution to be controlled by the dispatcher. Moreover, we do not want the current task to be preempted by the new one unless it voluntarily gives up control explicitly.

Just before returning, we wake up the task dispatcher in case it was previously suspended.

Task identity

A script often needs to know the task under which it is running. The `myself` command returns the id of the current task. If called from outside a task, it raises an exception.

```
proc task::myself   {} {
    variable tasks
    set me [info coroutine]
    if {[dict exists $tasks $me]} {
        return $me
    }
    error "Not running in a task."
}
```

Task cleanup

Any structures allocated by our package need to be cleaned up when the task exits. The `cleanup` command is more or less self explanatory.

```
proc task::cleanup {task_id} {
    variable tasks
    if {[dict exists $tasks $task_id]} {
        catch {rename $task_id ""}
        dict unset tasks $task_id
    }
}
```

Note that this does not release any resources allocated by the task itself, for example open channels. That is the responsibility of the task script.

Sending messages

Now we start coming to the meat of the package, implementation of messages and dispatching of tasks. Our package itself does not care about the content of a message. Its interpretation is left entirely to the receiving task. Sending a message involves placing it on the task's message queue and waking up our task dispatcher in case it is not running. By design, we choose to ignore messages sent to non-existing tasks.

```
proc task::send {task_id message} {
    variable tasks
    if {[dict exists $tasks $task_id]} {
        dict with tasks $task_id {
            lappend Messages $message
        }
        wakeup_dispatcher
    }
    return
}

proc task::wakeup_dispatcher {} {
    variable dispatcher_alarm
    set dispatcher_alarm 1
}
```

We will explain `wakeup_dispatcher` later when we describe the dispatcher implementation.

Note that as stated earlier message passing is asynchronous in our framework. The sender does not wait for the receiver to accept or process the message.

The message queue

It is worth digressing a little bit at this point to elaborate on a design choice we made to use an explicit message queue as opposed to alternative implementations that you might find being used elsewhere.

The first of these dispenses with queueing of messages altogether. Sending a message to a task is implemented by calling the corresponding coroutine, either directly or via `yieldto`, passing the message as an argument. You might think of this as a *push* model where messages are pushed to the task whereas ours is a *pull* model where messages are explicitly read by the task with the `receive` command described later. There are two problems with the push method in a general purpose package like ours:

- The task (coroutine) might have yielded for reasons other than for the purpose of receiving messages. For example, the task may use calls such as `coroutine::util gets` or `coroutine::util after` that we saw in previous sections. In such cases, it expects to be resumed from the `gets` I/O handler or from the `after` timer implementation. Being resumed with a message argument at that point instead of the result of a `gets` would completely confuse the poor coroutine.
- Directly pushing messages to coroutines can lead to issues of reentrancy that can be difficult to diagnose and resolve.

The other implementation you might see is the use of Tcl's event queue for passing messages instead of the explicit queue in our design. This also is essentially a *push* model where it is the event dispatcher that is pushing messages to the tasks. This does not have the reentrancy problems but suffers from the first issue noted above.

Maintaining an explicit message queue where tasks pull messages avoids these issues. It also has the added benefit of allowing more sophisticated capabilities such as message priorities, pattern matching etc.

Receiving messages

The complementary command to `send` is `recv` which returns the next message from the tasks message queue.

```
proc task::recv {} {
    variable tasks
    set me [myself]
    while {1} {
        set msgs [dict get $tasks $me Messages]
        if {[llength $msgs] != 0} {
            dict set tasks $me Messages [lassign $msgs msg]
            return $msg
        } else {
            dict set tasks $me State RECEIVE
            suspend
        }
    }
}
```

The implementation is again more or less obvious. If the message queue is not empty, the first message is removed and passed back to the task. Otherwise, the task suspends itself thereby allowing other tasks to run.

Note that in our simple implementation, if a task continuously receives messages, it will not allow other tasks to run. In production quality software, a limit on the number of messages processed before suspension would be well advised. It could be up to the task to do this or our package could be enhanced to keep track of the number of messages received since the last suspension and suspend the thread even if the message queue is not empty.

Relinquishing the CPU

There may be tasks that run independently without sending or receiving messages. Being good citizens however, they may wish to relinquish the CPU to other tasks. The `relinquish` command lets them do that.

```
proc task::relinquish {} {
    variable tasks
    dict set tasks [myself] State RELINQUISHED
    suspend
}
```

We will see the purpose of the state `RELINQUISHED` later.

Suspending a task is nothing more than a simple `yield`.

```
proc task::suspend {} {
    yield
}
```

The task dispatcher

We finally come to the heart of our package, the dispatcher. The role of the dispatcher is to keep looping through the list of tasks looking for any that have messages pending in their receive queue and invoking them. There are several finer points that need discussion but first we show the code.

Our dispatcher is itself is a coroutine whose innards are implemented by the `dispatch_loop` procedure. This will loop continuously initializing some state variables and then running an inner loop that iterates over the **current** tasks invoking those that are ready to run. There is a subtle point here in that tasks may be added or removed by any of the current tasks. We will not schedule any new tasks to run until the current tasks have been invoked first. This is a matter of policy; you may choose otherwise.

A task is considered ready to run if

- it is a new task indicated by the `INIT` state.
- it has voluntarily given up the CPU and is awaiting its next turn as indicated by the `RELINQUISHED` state.
- it is waiting for messages as indicated by its `RECEIVE` state **and** there are messages pending in its queue.

```
proc task::dispatch_loop {} {
    variable tasks
    variable dispatcher_alarm
    variable tasks_added

    while {1} {
        set woken 0
        set tasks_added 0
        foreach task_id [dict keys $tasks] {
            if {! [dict exists $tasks $task_id]} continue
            dict with tasks $task_id {} ❶
            if {($State in {INIT RELINQUISHED}) ||
                ($State eq "RECEIVE" && [llength $Messages])} {
                incr woken
                dict set tasks $task_id State RUNNING
                if {[catch { $task_id } mesg]} {
                    puts stderr "$task_id ended with error $mesg"
                    cleanup $task_id
                }
            }
        }
        if {$woken == 0 && $tasks_added == 0} {
            coroutine::util vwait [namespace current]::dispatcher_alarm
        }
    }
}
```

❶ Copy dictionary keys to local variables

We need to make an important point about the task's state — why we mark a task as INIT, RELINQUISHED, RECEIVE or RUNNING. This gets back to our earlier discussion of the push versus pull model. A task in the RUNNING state may actually have yielded outside of our framework through a call such as `coroutine::util gets`. We must not invoke such tasks from our dispatcher for the same reasons discussed there. Maintaining state information allows us to eliminate these RUNNING tasks from consideration. The RECEIVE and RELINQUISHED states are distinguished by the former indicating the task should only be awoken **only** if it has messages waiting while the latter has no such requirement.

Invocation of a task is done by transferring control to its hosting coroutine. In case of any exceptions, we delete the task from the task database.

Another option for transferring control to the task coroutine would have been to use `yieldto` instead of the direct call. You might find this used in some implementations. We prefer to call the task coroutine directly because our dispatcher would then regain control when the coroutine does a `yield`. This is important for some edge cases where the task yields but not using one of our package commands. For example, a task that sits in a loop using `coroutine::util gets` or the like would relinquish control but not to the dispatcher.

The other detail we need to take care of is with regards to the actions to take when none of the tasks are in a ready condition. Clearly we do not want the dispatcher to just spin endlessly without permitting other code to run. We would like it to suspend itself until there is a reason for it to run. We have a couple of options here. We can pass control to Tcl's event loop after scheduling ourselves to run after some short period of time. Instead of this polling mechanism, we choose the other option: we wait via the `coroutine::util vwait` call for a variable, `dispatcher_alarm` to be set. Correspondingly, the `wakeup_dispatcher` procedure, called on a `send` to any task, sets this variable. If the dispatcher is already active, it has no effect; otherwise, it releases the dispatcher from its suspended state.

Next we have to consider when the dispatcher should suspend itself via the `vwait`. Two conditions should be met for this:

- No tasks should be ready to run. We track this with the woken variable which is reset at the top of every dispatcher iteration and incremented every time a task is run. If this is 0 at the end of a complete loop through the task list, we know no tasks are ready.
- No new tasks must have been created since we initiated the iteration through the tasks. Again we use a variable, `tasks_added`, to keep track of this. The variable is reset at the top of the iteration and set when a new task is created within the `task` procedure.

The dispatcher suspends itself when both the above conditions are satisfied.

Starting the dispatcher

We have defined our dispatcher loop procedure but need to provide a means for the application to create it. So we do that now.

```
proc task::start_dispatcher {} {
    coroutine dispatcher dispatch_loop
}
```

Our package is now ready for use.

```
package provide task 1.0
```

We can try out our little package with some simple tasks. The first will simply print any message that is sent to it.

```
set logger [task::task {
    while 1 { puts "Received message: [task::recv]" }
}]
```

The second will send messages to the logger at regular intervals.

```
proc delay100 {} {coroutine::util after 100}
proc now {} {clock format [clock seconds] -format %T}
task::task "while 1 {task::send $logger \[now\]; delay100}"
```

Our final task is a variation of our previous example that printed directory sizes. This version sends the output to the logger task instead of printing it directly and is also considerate to relinquish the CPU.

```
proc dir_size_task {logger dir} {
    set total 0
    foreach fn [glob -nocomplain [file join $dir *]] {
        if {![catch {file size $fn} size]} {
            incr total $size
        }
        if {[file isdirectory $fn]} {
            if {![catch {dir_size_task $logger $fn} size]} {
                incr total $size
            }
        }
    }
    task::send $logger "$dir: $total"
    task::relinquish
    return $total
}
task::task "dir_size_task $logger c:/windows/system32"
```

We must not forget to start our task dispactcher.

```
task::start_dispatcher
```

Running the above commands in the `wish` shell or `tclsh` **with the event loop active** will result in the console printing the timestamps intermixed with the directory size data.

In our example, we started the dispatcher after creating the task. This is of course not mandated. Tasks can be created after the dispatcher is running as well.

21.11. Chapter summary

In this chapter, we covered one of the mechanisms for doing concurrent computation in Tcl — coroutines, a lightweight and flexible mechanism adaptable to a wide variety of situations, some of which we also described. In the next chapter, we will look at an operating system facility for concurrency — threads, which provides access to parallel computation on multiple processors.

21.12. References

PIKE2013

Concurrency Is Not Parallelism, Rob Pike, https://vimeo.com/49718712

SOF2008

TIP #328: Coroutines, Sofer, Madden, http://tip.tcl.tk/328. The Tcl Implementation Proposal describing coroutines.

KBK2012

TIP #396: Symmetric Coroutines, Multiple Args, and yieldto, Kevin Kenny, http://tip.tcl.tk/396. The Tcl Implementation Proposal describing enhancements to coroutines.

22

Threads

In the previous chapter, we described the use of coroutines for concurrent computation. We now describe a more heavyweight, but also more powerful, alternative that uses operating system threads. We will contrast the two later in Section 22.15.

By default, Tcl applications run within a single operating system thread. Generally speaking, threads are used in software for one of several reasons:

- In languages with no or limited support for event-driven I/O, threads are used for performing I/O without blocking other computational tasks.
- Since each operating system thread runs on its own stack, threads maintain their own implicit context and state. This allows simpler coding of some programs where multiple tasks have to run concurrently and share results.
- Some application programming interfaces, database drivers for instance, may only provide a blocking mode of operation. In this case as well, threads may be used for these operations while allowing other computational tasks to proceed.
- Threads are also used for performance reasons in compute-intensive applications as each thread may run on its own processor in multiprocessor configurations.

When it comes to Tcl, the first two motivations above are not very compelling. The use of threads specifically for I/O is obviated by Tcl's first-class native support for asynchronous and event-driven I/O. Similarly, as discussed in Chapter 21, Tcl's coroutines also run on their own private stacks making the use of operating system threads purely for this purpose unnecessary.

Thus the use of threads in Tcl is driven primarily by the last two factors — a desire to utilize multiple processors within a program and for non-blocking operations with programming interfaces that only support a synchronous mode.

22.1. Enabling thread support: the Thread package

Use of threads within a Tcl interpreter requires the Tcl libraries to have been built with thread support. This has been the default on Windows platforms for many years. On other platforms, thread support requires the `--enable-threads` build option to have been specified to the `configure` script at the time of building the Tcl libraries. This is the default only in very recent Tcl releases for non-Windows platforms. Even then, some Tcl applications explicitly disable thread support for one of two reasons — non-threaded builds are slightly faster, and threaded builds do not work correctly in some cases involving binary extensions that invoke the `fork` system call.

You can check if the Tcl interpreter has thread support enabled by either inspecting the `threaded` element of the `tcl_platform` array or with the `tcl::pkgconfig` call.

```
puts $tcl_platform(threaded) → 1
tcl::pkgconfig get threaded  → 1
```

In addition to the core Tcl libraries being multithread-enabled, working with threads at the script level also requires the Thread package to be loaded.

```
package require Thread → 2.8.0
```

The commands implemented by the package are placed in four namespaces shown in Table 22.1 based on their functionality.

Table 22.1. Thread package namespaces

Namespace	Description
thread	Base commands for working with threads
tpool	Implements a pool of worker threads
tsv	Commands for sharing data between threads
ttrace	Enables sharing of interpreter state across threads

In addition to the above, other third-party packages are available that supplement or enhance the functionality of the Thread package. We however do not describe them in this book.

22.2. Threading model

Before going into the specifics of individual commands, a few words regarding the threading model used by Tcl are in order.

Unless otherwise noted, the term "thread" will henceforth refer to any thread running a Tcl interpreter. A process may contain other threads as well, even those created in the background by the Tcl libraries for specific tasks like socket I/O on Windows. These are not germane to our discussion as they are not visible at the script level.

When a Tcl application starts up, there is only one thread executing Tcl code. We refer to this as the *main thread* and it has a special role as we will see in a bit. This thread executes Tcl code in a Tcl interpreter which may create threads each of which runs a Tcl interpreter. These may in turn create additional threads ad infinitum.

Threads and interpreters

We discussed the use of multiple Tcl interpreters at length in Chapter 20. The difference here is that the interpreter in a new thread is **not** a slave of the interpreter that created the thread. Each thread runs an independent "top-level" interpreter. Naturally, these interpreters may in turn create slave interpreters running in the same thread. As always, each interpreter, whether a top-level interpreter in a thread or a slave to one, is responsible for its own initialization such as loading any packages it needs, creating the proper namespaces and so on.

The Tcl threading model thus allows for multiple threads where each thread is (potentially) running multiple interpreters. **However each interpreter runs within the thread that created it.** There is no sharing of interpreter contexts between threads. Normal Tcl code therefore does not need to worry about having to synchronize access to its variables and data structures.

Thread run modes

A Tcl thread generally runs in one of two modes. It may execute a script and terminate, or it may sit in the event loop waiting for commands from other threads and executing them until it is asked to exit. Threads may be started and exit at any time with one important condition: **when the main thread terminates, the entire process exits.** It is therefore important for the main thread to monitor the status of the additional threads before it exits.

Thread reference counting

Some threads may act as worker threads wherein they run scripts sent to them by other threads. These threads have no defined exit point and have no way of knowing when their services are no longer required. Moreover, their "client" threads do not necessarily know who else might be requiring the services of that worker thread

either. To help with this situation, a reference count is maintained for each thread. The worker thread can choose to exit when this reference count reaches 0. Conversely, threads that have an interest in that thread being available can increment its reference count and only decrement it when the worker thread is no longer of interest.

Thread interaction

The design center for threads in Tcl is one where thread interaction is not expected to be fine grained. They are expected to run independently for the most part with occasional interaction and exchange of data. Thus unlike many programming languages, Tcl does not permit **normal** variables to be shared across threads (and in fact even across interpreters within a single thread). Threads interact by sending each other messages in the form of scripts which are executed by a target thread with the result being optionally returned to the source thread. This model simplifies multithread programming as the potential for race conditions, deadlocks and the like is greatly reduced.

One point to keep in mind is that the inter-thread communication "messages" are always received by the top-level interpreter of each thread. It can then be programmed to dispatch to any slave interpreters if so desired.

Shared properties

While threads run independently, there are some properties that shared. This is actually not specific to Tcl but is true for processes in general on most modern operating systems.

- The current working directory, as returned by the pwd command, is common to all threads. Changing it with the cd command in any thread changes it for the entire process.
- The environment variables, available in the env global array of the interpreter, is also shared amongst all threads. Changing a value in one thread is reflected across all threads in the process.

22.3. Creating threads: thread::create

Having summarized the threading model used by Tcl, we are now ready to describe the actual nitty-gritty of thread commands. Threads are created with the thread::create command.

```
thread::create ?-joinable? ?-preserved? ?SCRIPT?
```

The -joinable and -preserved options are related to managing thread lifetimes and are described in later sections. If the optional *SCRIPT* argument is not specified, the created thread will wait in a loop for messages to arrive. Otherwise, it will execute *SCRIPT* in its top-level interpreter and terminate when the script completes. Of course, as we will see *SCRIPT* itself may contain commands that activate the message communication loop. Note that the result of the script execution is not made available unless the script itself takes some action to return it as a message.

The command returns a thread id for the created thread that can be used for various purposes such as sending messages and synchronization.

Here is a short example that creates a thread to retrieve a URL in the background and save it to a file.

```
thread::create {
    package require http
    package require fileutil
    set tok [http::geturl http://www.example.com]
    fileutil::writeFile example.html [http::data $tok]
    http::cleanup $tok
}
→ tid00000000000014B4
```

The script loads any packages that it needs as it starts in a new interpreter in the created thread. It uses the blocking form of the http::geturl command but because it is running in a separate thread, our current thread can proceed with other computations. Once the script completes, the created thread will terminate.

Our simple example is missing many pieces. Since the threads run independently and asynchronously, the main thread has no way to know when the created thread has completed its task. There is also no mechanism for reporting errors and exceptions. We will come to these topics in a little bit.

22.4. Interthread communication

Threads communicate with each other through messages that are actually scripts executed in the receiving thread.

22.4.1. Waiting for messages: `thread::wait`

For a thread to receive a message, it must be running its event loop. A thread that is created without the *SCRIPT* argument specified to the `thread::create` command runs its event loop by default and is ready to receive messages. Alternatively, the script passed to the thread may use the `thread::wait` command to enter its event loop and process messages. The following commands are thus equivalent.

```
thread::create
thread::create {
    thread::wait
}
```

The latter form is generally used when there is some initialization required before messages are received. The pattern looks like

```
thread::create {
    ...load packages...
    ...other initialization...
    thread::wait
    ...cleanup...
}
```

The `thread::wait` command is similar to the `vwait` command in that it enters the thread's event loop dispatching events. The difference is that while `vwait` stays in the loop until a specified variable is set, `thread::wait` stays in the loop until the thread's reference count drops to 0 signalling it is time for the thread to exit. We discuss this in further detail in Section 22.2.

Do **not** use `vwait` in a thread in lieu of `thread::wait` for the purposes of entering the event loop at the top level. The former is ignorant with respect to thread reference counts and therefore does not return when the count drops to 0. The thread will then not exit as expected.

Once the thread's event loop is running, events are handled just as we described in Chapter 15. Messages sent from other threads are just a special form of event. The associated event handler is the message itself which is a script to be executed in the context of the receiving thread's top-level interpreter.

22.4.1.1. Limiting message queue size: `thread::configure -eventmark`

The number of outstanding messages permitted on a thread's receive queue can be controlled through the configuration option `-eventmark` set with the `thread::configure` command.

```
thread::configure TID -eventmark ?QLIMIT?
```

If the *QLIMIT* argument is not specified, the command returns the current value of the option. If *QLIMIT* is 0, no limit is placed on the number of messages in the queue. Otherwise, it specifies the maximum number of messages allowed in the queue. When this limit is reached, even the asynchronous form of the `thread::send` command will block until the receiving thread processes enough messages for the number of queued messages to drop below the limit.

22.4.2. Sending messages: thread::send

On the sender's side, the thread::send command is used to send messages to a target thread.

```
thread::send ?-async? ?-head? TID SCRIPT ?VARNAME?
```

Here *TID* is the thread id of the target thread as returned by the thread::create command. The command appends, or prepends if the -head option is specified, the supplied *SCRIPT* argument to the target's event queue.

The command may operate either synchronously or asynchronously. If the -async option is not present, the command works in synchronous mode and waits for *SCRIPT* to be executed. If the *VARNAME* argument is not provided, the command returns the result of evaluation of *SCRIPT* in the target thread. If *VARNAME* is provided, the result of the command is the the **return code** from the script evaluation. For example, the command will return 0 on a successful evaluation of the script, 1 if an exception was raised and so on. The **result** of the script is stored in the variable *VARNAME*.

If the -async option is specified, the command returns immediately (with one exception described below) with an empty result. If the *VARNAME* argument is provided, the result will be placed in the variable of that name when the script completes. The sending thread can track completion of the script by either placing a trace on the variable or waiting on it with vwait. In case of errors or other exceptions this variable is still updated with the result, for example the error message. However, as the return code is not available, there is no way to distinguish this from a normal completion without somehow encoding this information in the result value.

Even with the -async option specified, there is one situation where the thread::send command may block. If there is an upper limit on the message queue size for the receiving thread as described in Section 22.4.1.1, the thread::send will block until the queue shrinks sufficiently for the new message to be placed on it.

Handling of erors during evaluation of the script is discussed in detail in Section 22.7.

You need to be careful to avoid deadlocks when using the synchronous form of thread::send. Thread A may send a script to Thread B. If as part of the evaluation of the script in Thread B, it executes a send back to Thread A, deadlock will result. Thread A cannot respond to this send because it is still blocked awaiting the result of the first send. That never completes because Thread B is still waiting for the result of the second send. This situation is not specific to Tcl or the Thread package. The same will apply to all synchronous communication mechanisms in any language or platform.

22.4.3. Broadcasting messages: thread::broadcast

Whereas the thread::send command sends a message to a specific thread, the thread::broadcast message sends a message to all existing threads that have been created with the thread::create command.

```
thread::broadcast SCRIPT
```

The command always works asynchronously and has no means of returning the response from each thread unless explicitly done via a thread::send from within the supplied script. Notice that the sending thread is not included amongst the recipients.

```
% thread::create
→ tid00000000000004F0
% thread::create
→ tid0000000000000660
% thread::broadcast [format {
    thread::send -async %s "puts {Ping from [thread::id]}"
} [thread::id]]
→ Ping from tid00000000000004F0
  Ping from tid0000000000000660
```

As an aside, the above example illustrates use of `format` to create the code fragment to be sent to the remote threads. This makes it a little less awkward to construct the script where part of the substitution happens in the current thread and part in the receiving thread. See Section 10.7.1.2.

22.5. Thread lifetime management

The issue of tracking and managing thread lifetimes comes up in a couple of scenarios:

- It is sometimes important to know when a thread has completed execution. For example, the main thread may fire off several threads to execute various tasks. It must make sure it does not exit before those threads as completion of the main thread causes the entire process to exit.
- Conversely, a thread may need to know when it itself may exit. For example, a server thread may service requests from multiple independent client threads. It needs some way of knowing when its services are no longer required so it can exit and not take up unnecessary resources. Moreover, the client threads may not be even aware of each other so any solution must not presume any coordination between them.

Although both these issues can be solved using explicit low-level synchronization primitives we discuss later, the scenarios are common enough that the package provides simpler built-in solutions for both.

22.5.1. Waiting for thread completion: `thread::join`

The `thread::join` command can be used to wait for the completion of a thread.

```
thread::join TID
```

Here *TID* must be the thread id of a *joinable* thread, one that is created using the `-join` option of the `thread::create` command. An attempt to join a thread that is not joinable will generate an error.

The command blocks and returns only when the specified thread has exited. The result of the command is the exit code for the thread. No events are processed in the thread calling `thread::join` until the command returns. At that point the thread being waited on has exited and the thread id *TID* no longer valid.

Any thread, not necessarily the parent thread, may use `thread::join` to wait for the completion of another thread. However, only one thread may issue a `thread::join` for a particular thread. Further attempts to wait on that same thread will raise an error.

Every joinable thread must be waited on by some `thread::join` command. Otherwise, when it completes it will stay around as a "zombie" taking up system resources.

To wait for the completion of multiple threads, you need to serially wait on each in turn. To expand a little on an earlier example, we can start two threads to download URL's and wait for their completion as below.

```
set fetch_script {
    package require http
    package require fileutil
    set tok [http::geturl http://www.example.com/%1$s]
    fileutil::writeFile %1$s.html [http::data $tok]
    http::cleanup $tok
}
set tid1 [thread::create -joinable [format $fetch_script page1]]
set tid2 [thread::create -joinable [format $fetch_script page2]]
thread::join $tid1
thread::join $tid2
puts "[file exists page1.html] [file exists page2.html]"
→ 1 1
```

An alternative to using thread::join to ensure a thread has exited is to use the thread::names command discussed in Section 22.9. This returns a list of threads created with thread::create that are still running. The command can be invoked periodically and the returned list checked for the thread(s) of interest. This method may be more suitable when you do not want the calling thread to block until the target thread exits.

22.5.2. Thread reference counting: thread::preserve, thread::release

We now move on to the next topic related to thread lifetimes — how a thread knows when to exit its event loop and terminate. More specifically, how does the thread::wait command which sits in the event loop processing messages know that it should stop looking for more events and return to the caller.

The answer lies in an internal reference count that is maintained for each thread. When the value of this reference drops to 0 or a negative number, the thread::wait loop is terminated regardless of any additional messages that may be pending in the queue. Note that the thread itself is not terminated. What happens next will depend on the commands following the thread::wait call. In the normal case, the thread should do clean up and exit.

The thread reference count can be manipulated with the thread::preserve and thread::release commands which increment and decrement the reference count respectively.

```
thread::preserve ?TID?
thread::release ?TID?
```

The commands operate on the reference count of the thread identified by *TID* which defaults to the current thread if unspecified. Both commands return the reference count value after it has been modified.

The reference documentation advises that the thread should not be referenced if the return value from thread::release is 0 or negative. This is misleading. You should not reference the released thread after a thread::release from the thread doing the release irrespective of the return value. This is because other threads holding a reference to the released thread might have released it in the meantime.

The simplest demonstration of thread management is creation of a single thread which will implicitly run its thread::wait loop.

```
set tid [thread::create] → tid0000000000001804
```

The new thread is created with a reference count of 0 by default. We can send this thread some compute-intensive tasks.

```
thread::send $tid {expr 1+1} → 2
thread::send $tid {expr 2*2} → 4
```

Once we are done with the thread, we call thread::release on it. This decrements its reference count causing the thread::wait loop to return allowing the thread to exit.

```
thread::release $tid → 0
```

In a slightly more complex scenario, multiple threads might be using this "server" thread we created in which case the application has to ensure that the thread does not disappear until all client threads are done with it. To ensure this, each client thread should call thread::preserve on the server thread and match that with a thread::release once they are done with it. This is true of the creating thread as well if it will also make use of the server thread. Alternatively, it can use the -preserved option on thread creation. As we noted earlier, new threads are created with a default reference count of 0. However, if the -preserved option is provided to thread::create, the new thread is created with a reference count of 1. The creating thread must then execute a corresponding thread::release at the appropriate time.

22.6. Canceling script execution: thread::cancel

The thread::cancel command cancels the evaluation of a script in a thread.

```
thread::cancel ?-unwind? TID ?RESULT?
```

The command cancels the evaluation of the script running in the target thread identified by *TID* by raising an exception. If the *RESULT* argument is provided, it is used as the result or error message of the exception. Otherwise a default error message is used. If the -unwind option is specified, the exception cannot be caught by the target thread and propagates all the way to the top level of the thread. Note that the thread is **not** terminated.

The following console sample session should clarify the operation. It creates a thread and defines a procedure demo within it that prints two lines separated by a delay. The delay is simply to permit us to type a cancel command in time.

```
% set tid [thread::create]
→ tid0000000000002EA0
% thread::send $tid {proc demo {} {puts foo; after 5000; puts bar}}
```

We then ask the thread to execute the procedure and immediately issue a cancel command.

```
% thread::send -async $tid demo
→ foo

% thread::cancel $tid
% Error from thread tid0000000000002EA0
→ eval canceled
      while executing
  "after 5000"
      (procedure "demo" line 1)
      invoked from within
  ...Additional lines omitted...
```

As seen, bar is never printed because the evaluation of the script was aborted in the meantime. However, the thread is still active and can process further messages. We can try it again but this time without cancellation.

```
% thread::send -async $tid demo
→ foo
  bar
```

22.7. Error handling in threads: thread::errorproc

The manner in which errors and exceptions are handled in threads depends on whether the thread is executing a script passed as an argument to thread::create or is running scripts via the thread::wait loop. Furthermore, the latter case is further distinguished by whether the script execution is synchronous or not.

The simpler case is that of a thread executing a script passed through the thread::create command. Any errors or exceptions will result in the thread being terminated.

The above does not apply if the script invokes the thread::wait command and the exception is thrown in a script executed through its event loop. The behaviour for that case discussed later in this section.

By default, Tcl writes the error stack from the exception to the standard error channel. This behaviour can be changed by calling the thread::errorproc command.

```
thread::errorproc ?ERRORPROC?
```

If provided an argument, this command registers it as a procedure to be called to report the error. Tcl will append two additional arguments to the call: the id of the thread that generated the exception, and the error stack. An example procedure would be:

```
proc thread_error_handler {tid error_stack} {
    puts -nonewline stdout "[clock format [clock seconds]]: "
    puts "Thread $tid died. Error stack:\n$error_stack"
}
```

We can then register it and check out the effects.

```
% thread::errorproc thread_error_handler
% thread::create {error "Something horrible happened"}
→ tid0000000000000C60
  Sun Feb 19 22:56:34 IST 2017: Thread tid0000000000000C60 died. Error stack:
  Something horrible happened
      while executing
  "error "Something horrible happened""
```

The registered error handler is run in the interpreter that registered it irrespective of the thread creator. Moreover, that interpreter must have an active event loop running.

If thread::errorproc is called without any arguments, it returns the currently registered error handler.

```
% thread::errorproc
→ thread_error_handler
```

To reset the error handler to the default, call thread::errorproc with an empty string as the argument.

The other case to consider is that of exceptions thrown by scripts run by the thread::wait event loop, i.e. one sent by the thread::send command. Unlike for scripts passed as arguments to the thread::create command, exceptions in scripts run within thread::wait do not cause the thread to die by default. These exceptions are reported as described below but the thread continues to run the the event loop processing messages. The thread::configure command's -unwindonerror option may be used to change this behaviour.

```
thread::configure TID -unwindonerror ?UNWIND?
```

If *UNWIND* is 1, exceptions during execution of scripts from thread::wait will cause the thread to exit. The default value is 0. If the *UNWIND* argument is not given, the command returns the current value of the option.

Reporting of exceptions in scripts run with thread::send depends on whether the -async option was specified with the command. If present, exceptions are reported as described earlier for scripts passed to thread::create.

If the -async handler was not specified, the thread::send command executes synchronously. If the *VARNAME* argument is provided, the command behaves similar to the catch command. The result of the command is the return code and the script result is placed in the variable *VARNAME*.

```
% set tid [thread::create]
→ tid0000000000000FF4
% thread::send $tid {expr 1+1} result ❶
→ 0
% set result
```

```
→ 2
% thread::send $tid {error "An error!"} result ❷
→ 1
% set result
→ An error!
```

❶ Normal completion
❷ Error exception

If *VARNAME* was not provided, then a synchronous send results in exceptions being directly reflected as for any other command.

```
% thread::send $tid {error "An error!"}
Ø An error!
% set errorInfo
→ An error!
      while executing
  "error "An error!""
      invoked from within
  "thread::send $tid {error "An error!"}"
```

22.8. Threads and I/O channels: thread::transfer

As we described in Chapter 20, I/O channels are specific to an interpreter and not shared. The same then naturally applies to interpreters running in different threads. With the exception of standard I/O channels, a channel created in one thread is not available in other threads and cannot be shared. It can however be transferred from one thread to another with the thread::transfer command.

```
thread::transfer TID CHAN
```

The channel *CHAN* is made accessible to the **main** interpreter of the thread identified by *TID* and no longer available to the current interpreter. The channel name remains the same in the thread to which it is transferred. However, this name has to be explicitly provided to the target thread as otherwise it has no means to know how to reference the channel. The following trivial example illustrates the sequence.

We open a channel to a temporary file in our current interpreter and write to it.

```
set chan [file tempfile tempname]
puts $chan "File created by [thread::id]"
chan names
→ file413a920 stdin stdout rc28 stderr
```

Notice that chan names lists the channel amongst the open channels in this interpreter.

We then create a new thread and transfer the channel to it. We see that chan names no longer shows the channel as being available in the current interpreter. It does show up in the list of channels in the new thread.

```
set tid [thread::create]
thread::transfer $tid $chan
chan names
thread::send $tid {chan names}
→ file413a920 stderr stdout stdin
```

Although the channel is now available in the new thread, its name is not known to the interpreter in the thread. We have to inform it through some means. In our example, we simply set the global chan in the new thread to the channel name.

```
thread::send $tid [list set chan $chan]
→ file413a920
```

Finally we have the new thread write to the channel and close it.

```
thread::send $tid {
    puts $chan "Message from [thread::id]"
    close $chan
}
thread::release $tid
→ 0
```

As confirmation, we can read back the temporary file.

```
print_file $tempname
→ File created by tid0000000000002FE8
  Message from tid000000000000081C
```

There is one issue you have to be wary of when transferring channels. The `thread::transfer` command will block until the target thread has internally completed acceptance of the channel. This leads to the potential for deadlocks as illustrated by the following fragment.

```
thread::send $tid { set chan [file tempfile] }
thread::send $tid [list thread::transfer [thread::id] \$chan]
```

The thread executing the above code has the target thread open a temporary file. It then **synchronously** asks it to transfer the open channel back. However, because the `thread::send` command is used in synchronous mode, the first thread is blocked and cannot complete acceptance of the transfer. This results in the second thread being blocked as well since the `thread::transfer` command will not return until the channel transfer is complete. Deadlock results.

The general lesson in the above is to always be careful when using synchronous calls between threads. Again, this is nothing specific to Tcl or its Thread package.

An alternative to `thread::transfer` for moving channels between threads is the pair of commands `thread::detach` and `thread::attach`. The former removes the channel from the calling interpreter. The channel then stays in an "unowned" state until an interpreter (generally, not necessarily) in another thread calls `thread::attach` to claim ownership of the channel. This splitting of `thread::transfer` functionality is useful when the thread to which the channel should be transferred is not known a priori. This situation is common when using a "worker thread" model of the kind we will describe in Section 22.12.

Note that the same care for avoiding deadlocks must be taken with `thread::detach` / `thread::attach` as was described above for `thread::transfer`.

Transferring socket channels

There is one minor quirk to keep in mind regarding transfer of server sockets. As described in Chapter 18, a server socket is created through an accept callback on a listening socket. One common design in multithreaded servers is to hand off the socket to a separate thread for the actual data transfer [1]. For internal implementation reasons however, a server socket cannot be transferred to another thread from within the accept callback. In other words, the following accept callback will not work.

[1] This design is far less common in Tcl applications thanks to its strong support for async event-driven I/O

```
proc accept {chan remote_addr port} {
    set tid [thread::create]
    thread::transfer $tid $chan
}
```

The workaround is to reschedule the transfer of the channel through the event loop as shown below.

```
proc accept {chan remote_addr port} {
    after 0 [list transfer_socket $chan $remote_addr $port]
}
proc transfer_socket {chan remote_addr port} {
    set tid [thread::create]
    thread::transfer $tid $chan
}
```

This quirk originates from additional references held internally on the channel during the processing of the accept callback. These references prevent the channel from being transferred. Once the callback returns, these references are released allowing the transfer to take place in the rescheduled procedure.

22.9. Introspecting threads: thread::id, names, exists

All commands in the Thread package for introspecting threads only deal with threads created with the thread::create command. Threads that have been created by other means, such as at C level, are not considered by these commands.

The thread::id command returns the id of the current thread.

```
thread::id → tid0000000000002FE8
```

The thread::names command returns the list of currently executing threads.

```
% thread::names
→ tid0000000000000FF4 tid0000000000002FE8
```

You can check for the existence of a thread with the thread::exists command.

```
set tid [thread::create -joinable] → tid00000000000020E8
thread::exists $tid                → 1
thread::release $tid               → 0
thread::join $tid                  → 0
thread::exists $tid                → 0
```

22.10. Thread-shared variables

Variables in a Tcl interpreter are always contained within that interpreter. They are not directly accessible from other threads or even other interpreters within the same thread. Data is shared between threads by passing **values** through messages. There are times however where it is beneficial to have a shared **location** where data is stored and accessed from multiple threads:

- Situations where the same exact values for data items must be seen from multiple threads is greatly simplified if the data is stored in a single location and not passed around.
- When large amounts of data is being shared, passing it around by value entails memory copies with a significant cost in performance.

For these situations, the Thread package provides the *thread-shared variables*, or simply shared variables, and a set of commands for manipulating them under the tsv namespace.

Thread-shared variables have the following characteristics:

- They are not "real" Tcl variables in that they cannot be referenced using $ or the `set` command, they are not traceable and so on. They can only be manipulated through the `tsv` commands implemented by the `Thread` package.
- Shared variables exist outside of any interpreter or thread in that they continue to exist irrespective of the creating thread or any other thread terminating. When no longer required they have to be explicitly unset so as to free up memory resources.
- A shared variable is not a scalar but a collection of values indexed by a key similar to Tcl arrays and dictionaries.
- Access to these variables is always protected under a lock so threads do not need to explicitly synchronize when manipulating these variables. Note however that each variable has an independent lock so if a transaction involves multiple shared variables, additional locking may be required.

Commands for manipulating shared variables can be categorized as follows:

- Scalar operations such as setting, incrementing etc. on an individual element of a shared variable
- List operations similar to `lappend`, `lsearch` etc. on an individual element of a shared variable
- Array commands, similar to the standard Tcl `array` command, that operate on the entire shared variable, not just individual elements.
- Commands that treat the shared variable as a set of *keyed lists*, similar in utility to Tcl dictionaries
- Miscellaneous commands for introspection and utility functions

22.10.1. Scalar operations on shared variables

The commands `tsv::set`, `tsv::unset`, `tsv::incr`, `tsv::append` and `tsv::exists` work very similar to their standard Tcl counterparts `set`, `unset`, `incr`, `append` and `info exists` except that they operate on individual elements of a thread-shared variable rather than a Tcl variable.

```
% set tid [thread::create]
→ tid00000000000001A18
% tsv::exists mytsv myelem ❶
→ 0
% tsv::set mytsv myelem "A value" ❷
→ A value
% tsv::exists mytsv myelem
→ 1
% thread::send $tid {
    tsv::append mytsv myelem " in" " shared storage" ❸
}
→ A value in shared storage
% tsv::set mytsv myelem ❹
→ A value in shared storage
% tsv::set mytsv myinteger 0
→ 0
% tsv::incr mytsv myinteger 5 ❺
→ 5
```

❶ Check existence of `myelem` in the `mytsv` shared variable
❷ Set the value of the `myelem` element in the shared variable `mytsv`
❸ Append a string to the element from another thread
❹ Retrieve its value
❺ Increment the value of the `myinteger` element of the shared variable `mytsv`

There are additional commands that have no direct counterparts in the core Tcl command set. These are required because of the potential for race conditions between threads, something which is not an issue for normal variables. For example, consider the following code fragment:

```
if {[tsv::exists mytsv myelem]} {
    puts "Value is [tsv::set mytsv myelem]"
}
→ Value is A value in shared storage
```

The above seems to work but there is a hidden race condition. Although individual `tsv` commands operate under a lock protecting access to the shared variable, other threads may still change the variable between command invocations. For example, in the above code fragment, some other thread could unset the shared variable at some point between the `tsv::exists` check and the `tsv::set` invocation resulting in an exception.

The commands, `tsv::get`, `tsv::move` and `tsv::pop`, are provided to deal with common cases of this type where multiple base operations have to be performed atomically. The `tsv::get` command is meant for exactly the case described above.

```
tsv::get TSVAR ELEM ?VARNAME?
```

The command **atomically** checks for the existence of the element *ELEM* in the shared variable *TSVAR* and returns its value if it exists. If the *VARNAME* argument is provided, the command returns 1 if the element exists and 0 otherwise. The element value is stored in the variable *VARNAME*. Our previous (incorrect) code fragment can then be written as

```
if {[tsv::get mytsv myelem val]} {
    puts "Value is $val"
}
→ Value is A value in shared storage
```

If the *VARNAME* argument is not specified, the command returns the value of the shared variable element if it exists and raises an error otherwise.

The `tsv::move` command renames an element.

```
tsv::move mytsv myinteger yourinteger → (empty)
tsv::exists mytsv myinteger           → 0
tsv::set mytsv yourinteger            → 5
```

The `tsv::pop` command retrieves the value of an element while simultaneously removing it from the shared variable.

```
tsv::pop mytsv yourinteger    → 5
tsv::exists mytsv yourinteger → 0
```

22.10.2. List operations

The next set of commands also operate on individual elements of a shared variable but in this case the value stored in the element is treated as a list. Again, the majority of these commands parallel the standard Tcl commands of the same name for operating on lists. These are `tsv::lappend`, `tsv::linsert`, `tsv::lreplace`, `tsv::llength`, `tsv::lindex`, `tsv::lrange`, `tsv::lset` and `tsv::lsearch`. While some list modification commands operate on **variables** and others on list **values**, all `tsv` list modification commands operate on shared variable elements, not values (the latter would not make sense).

```
tsv::lappend mytsv mylist C D E                    → C D E ❶
thread::send $tid {
    tsv::linsert mytsv mylist 0 A B                ❷
    tsv::lreplace mytsv mylist end-1 end X [list Y Z] ❸
    tsv::lset mytsv mylist end 1 P                 ❹
}                                                  → A B C X {Y P}
tsv::lindex mytsv mylist 2                         → C ❺
tsv::lrange mytsv mylist 1 3                       → B C X ❻
tsv::lsearch mytsv mylist -exact Y                 → -1 ❼
```

❶ Create a list
❷ Insert list elements from another thread
❸ Replace elements
❹ Set nested list element
❺ Retrieve a single list element
❻ Retrieve a range of list elements
❼ Search for exact match. Other options are `-glob` and `-regexp`

Just as in the case of scalar operations, shared variable lists also have commands for common multi-step sequences that need to be atomic. The `tsv::lpop` command is similar to `tsv::lindex` but in addition to returning the value at the specified index position, it also removes it from the shared variable element.

```
tsv::get mytsv mylist    → A B C X {Y P}
tsv::lpop mytsv mylist 2 → C
tsv::get mytsv mylist    → A B X {Y P}
tsv::lpop mytsv mylist   → A ❶
tsv::get mytsv mylist    → B X {Y P}
```

❶ If index is not specified, it defaults to 0.

The `tsv::lpush` command does the reverse operation.

```
tsv::lpush mytsv mylist J 3 → (empty)
tsv::get mytsv mylist       → B X {Y P} J
tsv::lpush mytsv mylist K   → (empty) ❶
tsv::get mytsv mylist       → K B X {Y P} J
```

❶ If index is not specified, it defaults to 0.

The `tsv::lpush` command always returns the empty string as its result.

22.10.3. Array operations

We have seen `tsv` commands that parallel the standard Tcl scalar and list operations. As you might expect, a similar set of commands exists that treat shared variables in a fashion similar to Tcl's array variables. Unlike the `tsv` commands previously discussed which operated on individual elements of a shared variable, these commands treat the shared variable itself as an array.

The commands related to `tsv` arrays are all implemented as the `tsv::array` ensemble. The ensemble subcommands `tsv::array set`, `tsv::array get`, `tsv::array names` and `tsv::array size` have the same function as their array command counterparts.

```
tsv::array set myarr {AB 1 BC 2 AC 3} → (empty) ❶
tsv::array get myarr                  → AC 3 BC 2 AB 1 ❷
tsv::array get myarr a*               → (empty) ❸
tsv::array size myarr                 → 3 ❹
```

❶ Create, set or modify elements
❷ Get all elements of an array
❸ Get all elements matching a pattern
❹ Get number of elements in the shared variable

The above commands operate on the shared variable as an array but it is still a shared variable and therefore can be accessed with the previously discussed commands. For example,

```
tsv::get myarr AB       → 1 ❶
tsv::set myarr XY 4     → 4 ❷
tsv::lappend myarr BC 5 → 2 5 ❸
tsv::array names myarr  → AC XY BC AB ❹
```

❶ Get the value of array key AB
❷ Creating a new element in the shared variable is equivalent to creating an element of the array
❸ List commands on an array element
❹ All element names. Notice it includes XY

Notice the symmetry between shared variables and Tcl arrays. A shared variable is analogous to a Tcl array where elements are indexed by a key. Just as elements of a Tcl array can be operated on by list and string commands, so can elements of a shared variable with the equivalent `tsv` commands.

There is one feature of thread shared variables that we do not discuss here. Shared variable arrays can be bound to persistent disk storage so that their contents are stored in a database. As of this writing the package supports the `gdbm` and `ldbm` databases. Depending on the platform, this feature may or may not be included. You can use the `tsv::handlers` command to list the available database backends. To use the feature, refer to `tsv::bind` in the `Thread` package documentation.

22.10.4. Keyed lists

One final basic Tcl structured data type whose analogue we have not described are dictionaries. Shared variables do not have any commands that work on dictionaries. However, they do have *keyed lists* which serve a similar purpose.

A keyed list is a list each element of which is itself a list with two elements, the first being the "key" and the second being the corresponding "value". For example, the keyed list

```
{Name {{First Sherlock} {Last Holmes}}} {Address {221B Baker Street}}
```

has two keys `Name` and `Address`. Moreover, the value associated with `Name` is itself a nested key list with keys `First` and `Last`.

The `tsv` commands related to keyed lists operate on the assumption that the element of the shared variable is a keyed list. The `tsv::keylset` command sets the values of elements in a keyed list.

```
tsv::keylset detectives holmes Name {{First Sherlock} {Last Holmes}}
tsv::keylset detectives holmes Address {221B Baker Street}
tsv::keylset detectives poirot Name {{First Hercule} {Last Poirot}} \
    Address {Whitehaven Mansions}
```

This creates elements holmes and poirot in the thread shared variable detectives.

We can retrieve elements as usual with tsv::get.

```
% tsv::get detectives holmes
→ {Name {{First Sherlock} {Last Holmes}}} {Address {221B Baker Street}}
```

To operate on individual fields, we use the tsv keyed list commands. The tsv::keylset command we used for creation can also be used to modify or add new keys. Field values are retrieved with tsv::keylget.

```
tsv::keylget TSVAR ELEM KEY ?RETVAR?
```

If the *RETVAR* argument is not specified, the command returns the value associated with *KEY* in the *ELEM* element of the shared variable *TSVAR*. Nested keys are accessed using special syntax where each key level is separated by a period.

```
% tsv::keylget detectives holmes Address
→ 221B Baker Street
% tsv::keylget detectives holmes Name
→ {First Sherlock} {Last Holmes}
% tsv::keylget detectives holmes Name.First ❶
→ Sherlock
```

❶ Nested field

The command will raise an error if the key does not exist.

When the *RETVAR* argument is provided, the command returns 1 if the key exists in the keyed list and 0 otherwise. In the former case the corresponding value is placed in the variable *RETVAR*.

```
tsv::keylget detectives mason Address addr  → 0
tsv::keylget detectives poirot Address addr → 1
set addr                                    → Whitehaven Mansions
```

Deleting a key is accomplished with the tsv::keyldel command.

```
tsv::keyldel detectives poirot Address → (empty)
tsv::get detectives poirot             → {Name {{First Hercule} {Last Poirot}}}
```

The tsv::keylkeys command retrieves the list of keys.

```
tsv::keylkeys detectives holmes      → Name Address
tsv::keylkeys detectives holmes Name → First Last ❶
```

❶ Retrieve nested keys

22.10.5. Introspection and utility commands: thread::names, object, lock

The list of thread shared variables can be enumerated with the tsv::names command.

```
tsv::names      → myarr detectives mytsv ❶
tsv::names det* → detectives ❷
```

❶ All shared variables
❷ Shared variables matching a pattern

All shared variables should be deleted when no longer needed. The `tsv::names` command can be useful for cleaning up at an appropriate time.

```
foreach tsv [tsv::names my*] {
    tsv::unset $tsv
}
```

The `tsv::object` command is a convenience for working with a specific element in a shared variable. It creates an object command bound to a shared variable element with its methods redirected to the standard `tsv` commands. For example,

```
% set sherlock [tsv::object detectives holmes]
→ ::0000000004573C78
% $sherlock keylset Sidekick Watson
% $sherlock get
→ {Name {{First Sherlock} {Last Holmes}}} {Address {221B Baker Street}} {Sidekick Watson}
```

The created command is automatically deleted when the associated element is unset.

The `tsv::lock` command evaluates multiple commands while holding the internal lock associated with a shared variable.

```
tsv::lock TSVAR ARG ?ARG ...?
```

The command obtains the internal lock to the shared variable *TSV* creating the variable if necessary. It then evaluates the script formed through the concatenation of the *ARG* arguments and then releases the internal lock.

The `tsv::pop` command described earlier could be implemented as the following procedure.

```
proc pop {tsv elem} {
    tsv::lock $tsv {
        set val [tsv::get $tsv $elem]
        tsv::unset $tsv $elem
    }
    return $val
}
```

The lock ensures retrieval of value followed by unsetting of the element as an atomic operation.

22.11. Synchronization and locking

Threading models in many languages entail sharing of data between threads making the use of synchronization primitives commonplace. On the other hand, threads in Tcl are generally written as message-passing constructs with limited need for these primitives. Nevertheless, there are situations where synchronized access is required for commonly shared resources which may be internal, such as the shared variables we described in Section 22.10, or external, such as a set of files. The `Thread` package therefore provides mutexes and condition variables for such synchronization purposes.

This section assumes you are familiar with locking and synchronization in multithreaded programs. We will not explain what mutexes and condition variables are and how they might be used. Our discussion is restricted to a description of the synchronization facilities commands available in Tcl.

22.11.1. Mutexes: thread::mutex, thread::rwmutex

The Thread package implements three types of mutexes that can be used to ensure mutual exclusion when accessing shared resources.

- An exclusive mutex can be locked at most once. A second attempt to lock it will block the thread attempting the lock, even if it is the same thread that is holding the lock. The second thread will be unblocked when the thread holding the lock releases it. Note that a second attempt to lock by the thread already holding the lock will result in a deadlock. Operations on exclusive mutexes are implemented by the thread::mutex command ensemble.
- A recursive mutex only allows at most one thread to lock it but that thread can recursively lock it any number of times. Attempts by another thread to lock the mutex will result in that thread being blocked until the first thread releases **all** the locks it is holding on the mutex. Operations on recursive mutexes are also implemented by the thread::mutex command ensemble.
- A reader-writer mutex differs in that it supports two modes of locking — reader and writer. Multiple threads may simultaneously hold reader locks on the mutex. However at most one thread may hold a writer lock. Morever, a reader lock cannot be obtained while a writer lock is held and vice versa. Threads holding reader locks are allowed by contract to read the shared resource being protected but operations that modify the resource require a writer lock to be obtained. Operations on read-write mutexes are implemented by the thread::rwmutex command ensemble.

The create subcommand of both ensembles creates a mutex of the appropriate type. A recursive mutex requires the -recursive option to be specified to the thread::mutex command.

```
set exclusive_mutex [thread::mutex create]            → mid0
set recursive_mutex [thread::mutex create -recursive] → rid1
set rw_mutex [thread::rwmutex create]                 → wid2
```

Exclusive and recursive mutexes are locked with the thread::mutex lock command.

```
thread::mutex lock $exclusive_mutex → (empty)
thread::mutex lock $recursive_mutex → (empty)
```

The locks are released with thread::mutex unlock.

```
thread::mutex unlock $exclusive_mutex → (empty)
thread::mutex unlock $recursive_mutex → (empty)
```

However for reader-writer mutexes, two separate commands, thread::rwmutex rlock and thread::rwmutex wlock are required corresponding to reader and writer locks.

```
thread::rwmutex rlock $rw_mutex  → (empty) ❶
thread::rwmutex unlock $rw_mutex → (empty)
thread::rwmutex wlock $rw_mutex  → (empty) ❷
thread::rwmutex unlock $rw_mutex → (empty)
```

❶ Obtain a reader lock
❷ Obtain a writer lock

All mutexes should be destroyed when no longer required with thread::mutex destroy or thread::rwmutex destroy depending on the type of mutex.

```
thread::mutex destroy $exclusive_mutex → (empty)
thread::mutex destroy $recursive_mutex → (empty)
thread::rwmutex destroy $rw_mutex      → (empty)
```

You must ensure there are no outstanding locks on a mutex before destroying it. Otherwise, undefined behaviour including process crashes may result depending on the platform.

Here is a trivial example of the use of mutexes. As we stated earlier, standard channels are available in all threads. If we want to ensure output from multiple threads is not intermixed, we can use a mutex for the purpose.

```
% set io_mutex [thread::mutex create]
→ mid3
```

Then threads have to **by convention** lock the mutex before doing I/O.

```
% thread::mutex lock $io_mutex
% puts "Line one"
→ Line one
% puts "Line two"
→ Line two
% thread::mutex unlock $io_mutex
```

There is one issue with the above pattern when more complex code is involved. If an exception is raised in the script, the mutex will stay locked. Safer practice is to enclose the block within a `try` or `catch` command which will release the lock even in error cases. This situation is common enough that the `Thread` package provides the `thread::eval` command as a convenience.

```
thread::eval ?-lock MUTEX? ARG ?ARG ...?
```

The command locks the specified mutex, or an internal global mutex if the `-lock` option is not provided. The command then executes the script formed from concatenating the *ARG* arguments and returns its result taking care to unlock the mutex even under exceptional returns.

Our example above could then be written with better error handling as

```
thread::eval -lock $io_mutex {
    puts "Line one"
    puts "Line two"
}
```

For an equally superficial illustration of reader-writer mutexes, assume we are keeping bank account information in shared variables `current` and `savings` whose elements are keyed by an account number. To ensure consistent data, we have to ensure we do not read in the middle of a transaction that modifies the data. However, there is no reason to disallow multiple readers so we can use a reader-writer lock. Our desire for a reader-writer lock precludes us from using `thread::eval`.

```
proc bank::init {} {
    variable acct_mutex [thread::rwmutex create]
    ...create and initialize account data...
}

proc transfer {from_type from_acct to_type to_acct amount} {
    variable acct_mutex
    thread::rwmutex wlock $acct_mutex
    try {
        tsv::incr $from_type $from_acct -$amount
        tsv::incr $to_type $to_acct $amount
    } finally {
        thread::rwmutex unlock $acct_mutex
```

```
    }
}

proc balance {acct_type acct} {
    variable acct_mutex
    thread::rwmutex rlock $acct_mutex
    try {
        set balance [tsv::get $acct_type $acct]
    } finally {
        thread::rwmutex unlock $acct_mutex
    }
    return $balance
}
```

22.11.2. Condition variables: thread::cond

Condition variables are used in conjunction with exclusive mutexes as a race-free mechanism for a thread to block until some predicate is true. The condition variable is used for notification while the mutex is used to ensure that the data underlying the predicate is not modified by another thread while being checked.

The general usage pattern is as follows. The initialization code

1. Creates an **exclusive** mutex
2. Creates a condition variable

Each waiting thread (there may be multiple of these) runs the following sequence of steps:

1. Lock the mutex. Since the mutex is held, the predicate can be safely checked in the next step without fear of interference from other threads.
2. Check the predicate. If the predicate evaluates to true, go to step 5. Otherwise go to the next step.
3. Block on the condition variable. Blocking on the condition variable also releases the held mutex lock allowing other threads to modify the protected predicate data.
4. When the blocking call returns, go back to Step 2. See the explanation below as to why. Note that the mutex is held when the blocking call returns.
5. Do the expected work. Note the mutex is held at this point.
6. Unlock the mutex.

Steps 3 and 4 warrant some additional explanation. The blocking call on the condition variable also internally releases the mutex. This allows another thread to modify the protected data possibly changing the predicate to evaluate to true. If so the thread will signal the condition variable (see below) thereby waking up the thread waiting on the condition variable. Before returning in Step 3, the woken thread also relocks the mutex thereby ensuring the predicate is again protected. **However between the time the thread was signalled and the time the mutex was relocked, the predicate might have changed again.** This can happen for any number of reasons. For example, in a worker thread model, multiple threads may wait on a condition variable in which all of them are woken up on the signal. The first one to succesfully grab the mutex may modify the predicate data, for example by removing a work item from a queue. When other worker threads lock the mutex, the predicate is then no longer true. For this reason, after a thread wakes up it needs to recheck the predicate as indicated by Step 4.

The sequence for the signalling threads is simpler.

1. Lock the mutex.
2. Modify the protected data.
3. If the predicate evaluates to true, signal the condition variable to wake up the waiting thread(s).
4. Unlock the mutex.

We will illustrate the use of condition variables by extending our previous examples to implement a worker thread model for fetching URLs. Our main thread will queue URL requests to a shared queue. A set of worker threads will pick up entries from the queue, fetch the URL and write the content to a file.

First we initialize the data — the URL queue, the condition variable and mutex — that will be shared between the threads. We have already described mutex creation but make a note that this must be an exclusive mutex. The condition variables is created with the `thread::cond create` command. The returned handles are stored in shared variables since they will need to be available to the worker threads as well.

```
tsv::set shared_data url {}       → (empty)
set mutex [thread::mutex create]  → mid4
tsv::set shared_data mutex $mutex → mid4
set cond [thread::cond create]    → cid5
tsv::set shared_data cond $cond   → cid5
```

Next we define the script that each worker thread will run.

```
set worker_script {
    package require http
    package require fileutil
    proc url_to_file {url} {
        return [file join \
                    [fileutil::tempdir] \
                    [file rootname [file tail $url]]_content.html]
    }
    set mutex [tsv::get shared_data mutex]
    set cond  [tsv::get shared_data cond]
    while {1} {
        thread::mutex lock $mutex
        while {[tsv::llength shared_data urls] == 0} {
            thread::cond wait $cond $mutex
        }
        set url [tsv::lpop shared_data urls]
        thread::mutex unlock $mutex
        set tok [http::geturl $url]
        fileutil::writeFile [url_to_file $url] [http::data $tok]
        http::cleanup $tok
    }
}
```

The script initializes the thread by loading the omnipresent `http` package and retrieving the mutex and condition variable handles from shared memory. It then sits in an infinite loop waiting for work to be queued. Within each iteration of the loop, it follows the general pattern we described earlier. The predicate in this case is for the URL queue to have at least one entry in it. If the queue is empty, it waits on the condition variable with the `thread::cond wait` command.

```
thread::cond wait COND MUTEX ?TIMEOUT?
```

This command will unlock the specified mutex and then suspend execution until the condition variable *COND* is signalled or the optional timeout occurs. The timeout value *TIMEOUT* is specified in milliseconds. If unspecified or 0, the command will only return when the condition variable is signalled.

The timeout value has some caveats associated with it. It controls how long the command will internally wait for the condition to be signalled. However, even after being awoken, the `wait` command semantics call for the mutex to be re-acquired. There is no control over how long this might take if there are many threads competing for the mutex.

Before returning to the caller, the `wait` command re-locks the mutex. Our worker thread is then free to check the status of the URL queue without suffering race conditions from other threads. If the queue is empty (some other worker beat us to it), the code loops back into the conditional wait. If the queue is not empty, the thread removes one entry from it. It then unlocks the mutex to let other threads process additional entries if any, and proceeds to download the dequeued URL. The whole sequence is then repeated.

The work dispatcher is even simpler following our aforementioned pattern. We first start up a suitable number of worker threads, say 4, to run the worker thread script we showed earlier.

```
for {set i 0} {$i < 4} {incr i} {
    lappend workers [thread::create $worker_script]
}
```

We can then queue up URL's to be fetched in the background at any time with a sequence of calls similar to the following.

```
thread::mutex lock $mutex
tsv::lpush shared_data urls http://www.example.com/page.html end
thread::cond notify $cond
thread::mutex unlock $mutex
```

The only new command in this sequence is `thread::cond notify`. This command signals the specified condition variable causing all threads waiting on it to resume execution. The first thread to run and acquire the mutex will then dequeue and download the URL.

One final point about condition variables. They should be freed when no longer required by calling `thread::cond destroy`.

```
thread::cond destroy COND
```

As with mutexes, care must be taken that the condition variable is not in use when the command is invoked.

22.12. Thread pools

In the previous section we used a minimal, incomplete and custom implementation of the commonly used worker thread model. The `Thread` package includes a generalized version of this functionality through its *thread pools*. A thread pool consists of a work queue where *jobs* can be posted and an associated set of worker threads that pull them off the queue and execute them. An application may create any number of such thread pools.

Each thread pool has a variable number of threads that can be configured to run in it with minimum and maximum limits. The pool starts out with the minimum number when it is created. If the number of active or queued jobs exceeds the current number of threads, new threads are created to handle the additional jobs until the maximum thread limit is reached. Beyond that point jobs remain queued until they are removed by some thread that has completed its previous job. If a thread finds there are no jobs remaining to be processed, it goes into an idle state. If it stays in that state for some period of time without getting a new job, it will exit provided the current number of threads is greater than the minimum limit.

All commands related to thread pools are contained in the `tpool` namespace.

22.12.1. Creating a thread pool: `tpool::create`

A thread pool is created with `tpool::create`.

```
tpool::create ?OPTIONS?
```

The command returns the id of the thread pool which can be used to post jobs to the work queue for the pool. The options shown in Table 22.2 may be specified to control various associated configuration parameters.

Table 22.2. Thread pool options

Option	Description
`-exitcmd` *SCRIPT*	Specifies a script to be run just before the thread exits. A thread will exit if it has been idle for some period of time.
`-idletime` *SECONDS*	Specifies the number of seconds that a thread has to remain idle before it exits as described above.
`-initcmd` *SCRIPT*	Specifies a script to be run in the worker thread when it first starts up. This can be used to load packages, initialize data and so on. If the script raises an error in a worker thread, the exception is reflected back into `tpool::create` or `tpool::post` depending on which call initiated the creation of the thread.
`-maxworkers` *COUNT*	The maximum number of worker threads that should be maintained in the thread pool. The default is 4,
`-minworkers` *COUNT*	The minimum number of worker threads that should be maintained in the thread pool. The default is 0 so that the pool starts with 0 threads until the first job is queued.

Any number of thread pools may be created. The command `tpool::names` returns the list of currently existing thread pools.

Let us create a thread pool version of the example in the last section with one difference. It returns the content of the URL back to the main thread instead of writing it to a file. This provides a more complete example as it shows communication in both directions and some rudimentary error handling.

First we define the initialization script each worker thread needs to run. This just loads the `http` package and defines a procedure to retrieve URL's.

```
set init_script {
    package require http
    proc fetch_url {url} {
        set tok [http::geturl $url]
        try {
            switch -exact -- [http::status $tok] {
                ok { return [http::data $tok] }
                eof { error "Server closed connection." }
                error { error [http::error $tok] }
                default { error "Unknown error retrieving $url" }
            }
        } finally {
            http::cleanup $tok
        }
    }
}
```

Creating the thread pool is straightforward. We will use the defaults for all options except `-initcmd` since we need to pass in the above script.

```
set tpool [tpool::create -initcmd $init_script] → tpool00000000041564D0
tpool::names                                    → tpool00000000041564D0 ❶
```

❶ Lists all thread pools

Just as for threads, we will mark the thread pool as "in-use". This is discussed further in Section 22.12.7.

```
tpool::preserve $tpool → 1
```

22.12.2. Posting jobs to a thread pool: tpool::post, tpool::get

Any thread can post a *job* to a thread pool with the tpool::post command. A job is nothing but a Tcl script to executed by a worker thread just as we referred to scripts sent to specific threads as messages. The difference in terminology is conceptual.

```
tpool::post ?-detached? ?-nowait? TPOOL SCRIPT
```

The result of the command is a *job identifier* unless the -detached option, described below, is specified. The command initiates evaluation of *SCRIPT* in a worker thread in the following manner:

- If an idle thread is available in the thread pool, the *SCRIPT* is passed to it for execution and the command returns.
- If the -nowait option is specified, the command places the job on the work queue and returns.
- If no threads are idle and the maximum thread count limit has not been reached for the thread pool, a new thread is created. The tpool::post command will then wait in the event loop of the current thread processing events until the new thread is initialized. It will then pass the script to the new thread and return.
- If no threads are idle and the thread limit is reached the command will wait for a thread to become idle. In this case as well, while waiting the current thread's event loop will continue processing events.

The -detached option provides a "fire and forget" capability. If specified, the command creates a detached job which cannot be canceled or waited on, and whose result cannot be obtained. In this case the command returns an empty string.

Make note of one difference between using the thread::send command to send messages to a specific thread and the tpool::post command. The script passed in the tpool::post command may be processed by **any** thread in the pool. Therefore do not rely on context to be maintained between two tpool::post calls. Thus the following sequence

```
tpool::post $tpool {set x 1}
tpool::post $tpool {puts $x}
```

is likely to fail with an undefined variable error since the second script may not be executed by the same thread as the first.

Let us fetch two URL with our previously created thread pool. The second URL is invalid just so we can demonstrate error handling in fetching results.

```
set job [tpool::post $tpool {fetch_url http://www.example.com}]    → 1
set bad_job [tpool::post $tpool {fetch_url xyz://www.example.com}] → 2
```

Now that the jobs have been posted, we need to know when they complete so we can retrieve the results.

22.12.3. Waiting for job completion: tpool::wait

The tpool::wait command is used to wait for the completion of one or more posted jobs.

```
tpool::wait TPOOL JOBLIST ?VARNAME?
```

Here *JOBLIST* is a list of job identifiers as returned by the tpool::post command. The command enters the current thread's event loop processing events until at least one of the jobs in *JOBLIST* has completed. It then returns the list of completed job identifiers (there may be more than one). If the optional *VARNAME* argument is provided, the command stores the identifier for jobs still pending into the variable of that name.

We can wait for our previously posted jobs to complete with the following loop.

```
set jobs [list $job $bad_job]
while {[llength $jobs]} {
    tpool::wait $tpool $jobs jobs
}
```

22.12.4. Retrieving a job result: tpool::get

Once a job is completed, its result can be retrieved with the tpool::get command.

```
tpool::get TPOOL JOBID
```

The result of the job is the result of the script evaluation in the worker thread. We can get the content of our URL in our example.

```
% tpool::get $tpool $job
→ <!doctype html>
  <html>
  <head>
      <title>Example Domain</title>

...Additional lines omitted...
```

If the script threw an error, the tpool::get command will also throw an error with the same errorCode and errorInfo values set by the original error. The command will also throw an error on an attempt to retrieve the result of a job that has not completed yet.

We provided one URL above that was invalid. Accordingly, an attempt to retrieve the result will raise an error.

```
% tpool::get $tpool $bad_job
Ø Unsupported URL type "xyz"
% set errorInfo
→ Unsupported URL type "xyz"
      while executing
  "http::geturl $url"
      (procedure "fetch_url" line 2)
      invoked from within
...Additional lines omitted...
```

Note how the error message and stack are those raised by the worker thread.

22.12.5. Canceling a job: tpool::cancel

Jobs on the work queue that are still pending **and not yet assigned to worker threads** can be cancelled with the tpool::cancel command.

```
tpool::cancel TPOOL JOBLIST ?VARNAME?
```

Here *JOBLIST* is the list of identifiers for the jobs to be cancelled. The command returns the list of identifiers for the cancelled jobs. If the *VARNAME* argument is provided, the list of jobs that could not be cancelled is stored in a variable of that name.

22.12.6. Suspending thread pools: tpool::suspend, tpool::resume

A thread pool can be suspended at any time by calling the tpool::suspend command.

```
tpool::suspend TPOOL
```

Suspending a thread pool will suspend execution of all threads in the pool. However, a caller may still post jobs to the pool's work queue.

The suspension may be rescinded with the tpool::resume command.

```
tpool::resume TPOOL
```

The command will then cause the worker threads to resume execution. Idle workers will pick up any additional jobs placed on the work queue while the thread pool was suspended. However, no new threads are created to handle additional pending jobs **even if the maximum thread count limit has not been reached**.

22.12.7. Thread pool lifetimes: tpool::preserve, tpool::release

Just as for threads, thread pool lifetimes need to be managed. The reasons are much the same. There may be multiple users of a thread pool and they do not want the pool to be disappearing from under them while in use. The solution provided is also much the same. A reference count is associated with each thread pool. The reference count is incremented with tpool::preserve and decremented with tpool::release. The pool is freed when the reference count drops to 0. Beyond that point, attempts to post a job to the pool will result in a failure. However, this does not necessarily mean the threads in the pool have exited. They will do so on their own schedule.

Since we done with our URL fetching thread pool, we can inform the package accordingly.

```
tpool::release $tpool → 0
```

22.13. Distributing interpreter state: the Ttrace package

A common need when working with multiple threads is to ensure that they are all running in the same runtime "environment" in terms of namespaces, procedure definitions and the like. The Ttrace package partially satisfies this need. It allows for procedure and namespace definitions to be replicated across all threads.

Because this is a separate package and not part of Thread, we have to load it separately.

```
package require Ttrace → 2.8.0
```

Although the package contains many commands, we only describe one, ttrace::eval, which encapsulates the other commands into a simple-to-use interface. The other commands allow for finer grain control but we do not describe them here.

```
ttrace::eval ARG ?ARG ...?
```

The ttrace::eval command evaluates the script formed from concatenation of it arguments. Any changes in procedure or namespace definitions are then propagated to all threads.

The functionality is most easily illustrated with an example. Let us start up a thread. All threads making use of this feature must load the package as well.

```
set tid [thread::create {
    package require Ttrace
    thread::wait
}]
→ tid0000000000002730
```

Obviously this thread will not have the namespace `ns` or the `ping` procedure defined.

```
% set tid [thread::create]
→ tid00000000000002AAC
% thread::send $tid {namespace exists ns}
→ 0
% thread::send $tid ping
Ø invalid command name "ping"
```

Now we define both the namespace and the procedure within the **current** thread but do so within a `ttrace::eval`.

```
ttrace::eval {
    namespace eval ns {}
    proc ping {} {return "Ping from [thread::id]"}
}
```

We can then verify that the namespace and procedure have been replicated in the current thread **and** in the other thread.

```
namespace exists ns                     → 1
ping                                    → Ping from tid00000000000002FE8
thread::send $tid {namespace exists ns} → 1
thread::send $tid ping                  → Ping from tid00000000000002AAC
```

If a new thread is created, that will also automatically have these definitions.

```
% set tid2 [thread::create {
    package require Ttrace
    thread::wait
}]
→ tid000000000000174C
% thread::send $tid2 {namespace exists ns}
→ 1
% thread::send $tid2 ping
→ Ping from tid000000000000174C
```

You can see how the `Ttrace` package simplifies keeping threads in sync in terms of the namespace and procedure definitions. However, there are some important limitations to keep in mind. In particular, data and TclOO classes and objects are not replicated.

22.14. Using extensions in threads

Generally speaking, packages that are purely script-based do not require any special consideration for use in threads. However, use of binary extensions in a Tcl application that uses multiple threads takes some care. These fall into three categories:

- Extensions that are written to be thread-safe and can be safely loaded and used in multiple threads.
- Extensions that are **not** thread-safe but can be safely loaded and used in a **single** thread within a multithreaded applications. If other threads want to make use of the extension functionality, they need to do so by sending messages to the thread in which the extension is loaded. The Tk extension is one such example.

- Extensions that not thread-safe and should not be used in a threaded Tcl application.

Refer to the documentation for each extension to determine its thread safety characteristics.

22.15. Comparing coroutines and threads

Having worked through both coroutines and threads, you can see that at some level they serve a similar purpose in enabling computational tasks to proceed in concurrent fashion. We now summarise their differences to help you choose the one appropriate for the problem you are trying to solve.

The primary differences between coroutines and threads arise from the fact that threads are operating system level constructs whereas coroutines are implemented purely in user mode.

- Multiple threads can be assigned to multiple processors leading to increased performance. Coroutines run within a single thread and therefore are limited to the processor on which the thread is running. They derive no benefit from the presence of multiple processors. Of course, there is no reason not to run coroutines within multiple threads but that is a different kettle of fish as coroutines can only communicate within the interpreter where they are defined and would have to use the thread mechanisms for anything else.
- Blocking operations in a thread do not block other threads. In contrast, a blocking operation in a coroutine will block other coroutines as well. This is often a primary consideration in the decision to move to a thread-based architecture. Certain operations, for example accessing some database implementations, are only implemented in blocking form by the database drivers. In such cases, moving those operations to a separate thread ensures other parts of an application continue to run while a long database operation is in progress.
- Coroutines are defined within a single interpreter and share its namespaces, commands, channels etc. There is limited isolation between coroutines. Threads on the other hand enclose independent interpreters. There is no sharing of any kind except through the threading commands discussed earlier. This means errors in one thread can be isolated to that thread.
- Threads are relatively expensive to create and consume significant system resources like memory while coroutines are cheap in that respect. Inter-thread communication is also slower as it forces operating system context switches.
- Partly because they are cheap and share code and data, coroutines can be used for implementing generators and the like. Threads are not suitable for such purposes.

Having contrasted the two, keep in mind that coroutines and threads are not mutually exclusive. It is common and perfectly reasonable to use both in an application architecture.

The article Modeling a Queuing System compares and contrasts multiple implementations of a queueing system model that include threads and coroutines in addition to other mechanisms.

22.16. Chapter summary

In this chapter, we covered the use of operating system threading capabilities from within Tcl. Multithreading permits the application to take advantage of multiple processors in the system. However, its use must be carefully considered as it adds significant complexity to an application. In some languages, multithreading is required for concurrent I/O; however, this is not the case in Tcl where the asynchronous I/O model through the event loop is not only adequate but often more efficient. Similarly, in cases where the purpose of multithreading is to simplify programming of concurrent independent tasks while writing in a "sequential" style, coroutines can fulfil the need at a cheaper cost.

Nevertheless there are instances where only threads meet the desired needs and Tcl provides for that possibility. In conjunction with Tcl's event loop and coroutines, practically any software architecture geared towards multitasking and concurrent computation can be implemented in Tcl.

22.17. References

MARK2015

Modeling a Queuing System, Arjen Markus, http://www.magicsplat.com/articles/queueing.html. Compares thread and coroutine-based implementations of a queueing system model.

23

Database Connectivity with TDBC

The *Tcl Database Connectivity* (TDBC) extension provides a Tcl API for accessing SQL databases. Because this API is independent of the underlying database system, most[1] of the code accessing databases in an application can run unchanged between different database implementations.

A bit of history

TDBC 1.0, authored by Kevin B. Kenny, was released with Tcl 8.6. Prior to its release, the lack of a standard Tcl database access API lead to a number of extensions with different interfaces including

- *tclodbc* for connecting to databases using ODBC
- *DIO* which is part of Apache Rivet
- *nstcl* and *nsdbi*, both derived from AOL Server web server
- Various database-specific extensions like *oratcl* for Oracle and *pgtcl* for PostgreSQL.

With the advent of TDBC, applications can now rely on a standard interface to databases from Tcl.

TDBC is broken up into two layers:

- The upper layer, which is what we cover here, is the interface used by applications to access the database.
- The lower layer consists of different drivers that implement access to specific databases. At the time of writing, the TDBC distribution includes drivers for MySQL, PostgreSQL, Sqlite3 and any database accessible via an ODBC interface. The TDBC documentation also defines an interface that allows new drivers to be written for other database implementations.

Due to space limitations, this book only covers the first of these — the application interface to databases.

23.1. Installing TDBC

The TDBC extension is included in the standard Tcl 8.6 source distributions as well as all binary distributions. However, individual database driver components may have to be installed separately. Most binary distributions of Tcl include drivers for SQLite and Windows has native support for ODBC. In other cases, the drivers, usually implemented as shared libraries, are available from the database vendor.

23.2. Loading TDBC

TDBC is loaded with the standard `package require` Tcl command. The specific package to be loaded depends on which database driver is desired. The packages included in the core distribution are shown in Table 23.1.

[1] Some code will be necessarily specific to databases because of differing capabilities and quirks in database implementations.

Table 23.1. Core TDBC driver packages

Package	Database
tdbc::sqlite3	Sqlite3
tdbc::postgres	PostgreSQL
tdbc::mysql	MySQL
tdbc::odbc	Any database that is provides an ODBC interface

In addition, open source TDBC drivers are also available such as ones for JDBC[2], CUBRID[3] and MonetDB[4].

Naturally, when dealing with multiple database implementations in an application, more than one of these packages may be loaded if desired.

For our code examples, we will make use of the Sqlite3 database and thus load the corresponding package

```
% package require tdbc::sqlite3
→ 1.0.4
```

23.3. Concepts

TDBC follows the same general pattern as other database access API's and involves the following steps:

1. First a *connection*[5] has to be established to the database (and database provider) of interest. In addition to identifying the database this may also include authorization credentials and other options. All subsequent interactions for the database are done through this connection object and its surrogates.
2. Next a SQL *statement* is *prepared* using the connection object and then executed with the results returned as a *result set*.
3. The *result set* is iterated over to operate on the returned data.
4. The statement and result sets are freed so as to not use up resources.
5. Steps 2-4 are repeated as needed.
6. When all done, the database connection is closed.

TDBC encapsulates all the above abstractions as the TclOO classes `tdbc::connection`, `tdbc::statement` and `tdbc::resultset`.

23.4. Connecting to databases

A database connection is represented by an object of the appropriate `tdbc::`*`DRIVER`*`::connection` class. To connect to a database, you create an object of this class specifying the database of interest. The manner in which the database is identified depends on the database driver in use.

23.4.1. Connecting to SQLite: `tdbc::sqlite3::connection`

The `tdbc::sqlite3` package must be loaded to access SQLite databases.

```
package require tdbc::sqlite3
```

A connection to a SQLite database is created by passing the path to the sqlite3 database file to the `tdbc::sqlite3::connection` command. For example, to open a SQLite database in the current directory,

[2] https://github.com/ray2501/TDBCJDBC
[3] https://github.com/ray2501/tclcubrid
[4] https://sites.google.com/site/tclmonetdb/
[5] Not to be confused with a *network* connection.

```
tdbc::sqlite3::connection create db my-database.sqlite3
```

This will create the object db representing the database connection to the `my-database.sqlite3` database. As we see in a moment, we can operate on the database by invoking methods on this object.

For our sample code, we will create and make use of an in-memory SQLite database. The special token `:memory:` results in the database being created purely in memory with no disk store. It will be erased once closed which suffices for the sample code purposes.

```
% set dbconn [tdbc::sqlite3::connection new :memory:]
→ ::oo::Obj601
```

Note here we use the TclOO method new, as opposed to `create`, which generates the connection object name for us.

23.4.2. Connecting via ODBC: tdbc::odbc::connection

Open Database Connectivity (ODBC) is an industry standard API for accessing databases. Database implementations that support this interface can be accessed through the `tdbc::odbc` package.

```
package require tdbc::odbc
```

Windows comes with ODBC support built-in but to use ODBC on Unix or Linux, you may need to install an ODBC package such as unixODBC[6] or iODBC[7].

To connect to an ODBC database, pass its connection string to the `tdbc::odbc::connection` command. This takes the form of a series of attribute name and value pairs that specify the connection characteristics. For example, (assuming we are on Windows)

```
tdbc::odbc::connection create db "Driver={SQL
 Server};Server=localhost;Trusted_Connection=Yes;Database=YourDatabaseName;"
```

will return a connection object for the SQL Server database `YourDatabaseName` on the local system. Notice additional attributes can be specified in the connection string. For instance, the `Trusted_Connection=Yes` attribute and value specify that the credentials of the Windows account under which the application is running are to be used for authorization.

Depending on the system and the database, you can define a *data source name* (DSN) that stores the data used to construct a connection string. Then you can simply specify the DSN to connect to the database. So the above call would then become

```
tdbc::odbc::connection create db "DSN=YourDSN"
```

You can use the ODBC utilities in TDBC to define DSN's if the underlying ODBC implementation supports the ODBC Installer API. Alternatively, on Windows systems, you can use the ODBC applet in the Control Panel to define and configure DSN's. On Unix/Linux, unixODBC and iODBC both provide GUI and command line means of defining DSN's.

[6] http://www.unixodbc.org
[7] http://www.iodbc.org

Connection strings are ODBC driver specific and sometimes difficult to get right. The Connection Strings Reference web site is a useful resource to understand and construct these.

In addition to the common options (see Table 23.4) supported by all TDBC drivers, some ODBC environments support the -parent option which results in a prompt for user credentials if required. See the tdbc::odbc [8] reference for details on its use.

As described in Section 23.11, the package also implements some utility commands for interacting with the system ODBC manager.

23.4.3. Connecting to MySQL: tdbc::mysql::connection

Connecting to MySQL requires the tdbc::mysql package.

```
package require tdbc::mysql
```

The package differs from SQLite and ODBC in that the tdbc::mysql::connection command for establishing a database connection identifies the database to be connected through a set of options as opposed to a file name or connection string. These options are shown in Table 23.2.

Table 23.2. MySQL connection options

Option	Description
-host *HOSTNAME*	The name of the system on which the database server is running. Defaults to the local system.
-port *PORTNUMBER*	The TCP/IP port number on which the server is listening for connections.
-socket *PATH*	Connects to the Unix socket or named pipe specified by *PATH*.
-user *USERNAME*	The name of the user name to be used to access the database. Defaults to the current user id of the process.
-password *PASSWORD*	The password to be presented to the server. By default no password is presented.
-database *DATABASE*	The name of the database to default to if no database is specified in a query. Defaults to the default database for the user specified by the -user option.
-db *DATABASE*	Same as the -database option.
-interactive *BOOL*	If specified as true, sets the default timeout to be that for an interactive user; otherwise, the default timeout is set as for batch users.
-ssl_ca *PATH*	Specifies the **file** containing the list of trusted certificate authorities permitted for an SSL connection.
-ssl_capath *PATH*	Specifies the **directory** containing the files containing certificates for trusted authorities permitted for an SSL connection.
-ssl_cert *PATH*	Specifies the file containing the client certificate.
-ssl_key *PATH*	Specifies the file containing the client private key.
-ssl_cipher *CIPHERLIST*	Specifies the permissible ciphers to use for an SSL connection. *CIPHERLIST* is a list of cipher names separated by colons.

[8] http://www.tcl.tk/man/tcl8.6/TdbcodbcCmd/tdbc_odbc.htm

23.4.4. Connecting to PostgreSQL: tdbc::postgres::connection

Connecting to a PostgreSQL database is more or less identical to what was described previously for MySQL. The tdbc::postgres package is loaded

```
package require tdbc::postgres
```

and the connection is made via tdbc::postgres::connection using the options shown in Table 23.3.

Table 23.3. PostgreSQL connection options

Option	Description
-host *HOSTNAME*	The name of the system on which the database server is running.
-hostaddr *IPADDR*	The IP address of the system on which the database server is running. Takes precedence over the -host option.
-port *PORTNUMBER*	The TCP/IP port number on which the server is listening for connections.
-user *USERNAME*	The name of the user name to be used to access the database. Defaults to the current user id of the process.
-password *PASSWORD*	The password to be presented to the server. By default no password is presented.
-pw *PASSWORD*	Same as the -password option.
-database *DATABASE*	The name of the database to default to if no database is specified in a query. Defaults to the default database for the user specified by the -user option.
-db *DATABASE*	Same as the -database option.
-options *OPTS*	Specifies additional command line options to send to the server.
-sslmode disable|allow|prefer|require	A value of disable mandates a non-SSL connection to the server, require mandates an SSL connection, allow prioritizes a non-SSL over SSL and prefer (default) prioritizes SSL over non-SSL.
-service *SVCNAME*	Specifies that additional connection parameters are to be picked up from the entry corresponding to *SVCNAME* in the pg_service.conf file.

23.4.5. Common connection options

All TDBC drivers understand the common set of options shown in Table 23.4.

Table 23.4. Connection options common to all TDBC drivers

Option	Description
-encoding *NAME*	Name of the character encoding to be used on the connection. *NAME* should be one of the names returned by the Tcl encoding command. It is generally not necessary to specify this but be aware that drivers differ in their handling of this option.
-isolation *ISOLATION*	Specifies the transaction isolation level needed for transactions on the database. *ISOLATION* must be one of readuncommitted, readcommitted, repeatableread, or serializable. See the tdbc::connection[9] reference for details.
-timeout MILLISECS	Specifies the interval after which an operation should time out with an error. The default value of 0 indicates no timeout. The operations to which the

[9] http://www.tcl.tk/man/tcl8.6/TdbcCmd/tdbc_connection.htm

Option	Description
	timeout is applicable differs between the various drivers and databases. Refer to the appropriate reference pages for each driver.
-readonly BOOLEAN	If specified as true or 1, the connection will not modify the database.

23.4.6. Configuring connections: *DBCONN* configure

The values of the options that can be specified at the time a tdbc::connection object is created can be retrieved via its configure method. The same method can also be used to modify the values of some options.

So to retrieve the configuration of our in-memory sample database.

```
% $dbconn configure
→ -encoding utf-8 -isolation serializable -readonly 0 -timeout 0
```

We can also pass one or more configuration options to be modified.

```
% $dbconn configure -timeout 1000
```

23.4.7. Releasing connection resources: *DBCONN* close

When no longer required, connections must be closed by invoking the close method on the connection object.

```
db close
```

This will also close and release resources related to tdbc::statement and tdbc::resultset objects created through the connection.

23.5. Executing SQL

Executing a SQL statement involves first *preparing* the statement and then running it one or more times with different parameter values.

23.5.1. Preparing a statement: *DBCONN* prepare

The first step in executing SQL is to create a tdbc::statement object via the prepare method of a tdbc::connection.

```
DBCONN prepare SQL
```

Here *SQL* is the SQL statement to be executed. The following creates a table in our sample database.

```
set stmt [$dbconn prepare {
    CREATE TABLE Accounts
    (Name text,
     AcctNo text NOT NULL PRIMARY KEY,
     Balance double)
}]
→ ::oo::Obj601::Stmt::1
```

We can then use the created tdbc::statement object to run the SQL script against the database.

23.5.2. Executing a prepared statement: *STMT* execute

Once a tdbc::statement is created, its execute method in invoked invoked to run the corresponding SQL.

```
% set res [$stmt execute]
→ ::oo::Obj602::ResultSet::1
```

The `execute` method returns a `tdbc::resultset` object which we will examine later. For now, we free up both objects by invoking their `close` method. Like `tdbc::connection` objects, `tdbc::statement` and `tdbc::resultset` objects should also be freed when no longer required.

```
% $stmt close
```

Note that closing the `$stmt` also closes any contained `resultset` objects so we do not need to explicitly close `$res` here. Similarly, `tdbc::statement` objects that are not closed explicitly will be closed when the owning `tdbc::connection` object is closed. However, for the sake of saving resources, it is generally a good idea to explicitly release them when no longer needed. Since we have more we want to do with the connection and are not closing it, we explicitly close `$stmt`.

Insertions and queries follow a similar pattern.

```
% set stmt [$dbconn prepare {INSERT INTO Accounts (Name, AcctNo, Balance) VALUES ('Tom', \
    'A001', 100.00)}]
→ ::oo::Obj601::Stmt::2
% $stmt execute
→ ::oo::Obj604::ResultSet::1
% $stmt close ❶
```

❶ Will also close the result set returned by `execute`

This multi-step sequence of prepare and execute can be a little tedious and TDBC provides some methods that act as wrappers and make it more convenient. We will discuss these and their pros and cons a little later.

23.5.3. Bound variables

The above example hard-coded the values that were to be inserted into the table. Naturally, that is not a viable option when values are not known apriori at the time a program is written.

TDBC allows for Tcl variable values to be passed into the SQL statement by binding names within the SQL that begin with `:` by their corresponding values. These values may either come from Tcl variables of the same name or from a dictionary passed in as an argument.

```
% set stmt [$dbconn prepare {
    INSERT INTO Accounts (Name, AcctNo, Balance) VALUES (:name, :acctno, :balance)
}]
→ ::oo::Obj601::Stmt::3
```

Here the bound variables are `name`, `acctno` and `balance`. In the script below, the values for these will be sourced from the Tcl variables of the same name.

```
foreach {name acctno balance} {
    Dick  A002 200.00
    Harry A003 300.00
} {
    $stmt execute
}
```

Alternatively, we can pass in a dictionary to the `execute` command. The values will be picked up from the keys of the same name in the dictionary.

```
% $stmt execute {acctno A004 name Moe balance 100.00} ❶
→ ::oo::Obj606::ResultSet::3
```

❶ Order of elements does not matter

Note from the sequence above that a prepared statement can be executed multiple times with different values.

23.5.3.1. Bound variable configuration: *STMT* paramtype

Most databases drivers automatically figure out the type and direction (input, output, or both) of bound variables. A few require the application to provide this information. The paramtype method is provided for this purpose.

```
STMT paramtype NAME ?DIRECTION? TYPE ?PRECISION? ?SCALE?
```

Here *NAME* is the name of the bound variable. The *DIRECTION* argument specifies whether the bound variable is used to pass input (in), receive output (out) or both (inout). The *TYPE*, *PRECISION* and *SCALE* arguments correspond to the type, precision and scale column attributes shown in Table 23.5.

Although this is not required for SQLite, which figures out the information on its own, we could configure the balance variable in our statement as follows:

```
% $stmt paramtype balance in double
```

Conversely, the params method of the tdbc::statement object returns the configuration of the bound variables.

```
% print_dict [$stmt params]
→ acctno    = type Tcl_Obj precision 0 scale 0 nullable 1 direction in
  balance   = type Tcl_Obj precision 0 scale 0 nullable 1 direction in
  name      = type Tcl_Obj precision 0 scale 0 nullable 1 direction in
```

The result of the method is a nested dictionary keyed by the names of the bound variables. Each subdictionary contains details about the corresponding bound variable. The keys of this subdictionary are the same that were described for columns in Table 23.5 with an additional key, direction which may have the value in, out or inout.

23.5.4. Closing prepared statements: *STMT* close

Once a tdbc::statement has outlived its utility, resources associated with it should be freed via its close method.

```
% $stmt close
```

The result sets are also automatically freed when the associated tdbc::statement object or containing tdbc::connection object are closed.

23.5.5. Direct evaluation (MySQL only): *DBCONN* evaldirect

For cases where MySQL does not support data management language statements via the prepare call, the evaldirect method can be used to directly pass SQL code for execution in MySQL without going through a prepare call first.

```
DBCONN evaldirect SQLSTMT
```

This method is only supported by the MySQL TDBC driver and should only be used in those cases where MySQL does not support the statement via the prepare method.

23.6. Retrieving data from result sets

Any data returned from executing SQL is stored in a result set wrapped as a `tdbc::resultset` object.

```
% set stmt [$dbconn prepare {SELECT Name, Balance from Accounts}]
→ ::oo::Obj601::Stmt::4
% set res [$stmt execute]
→ ::oo::Obj610::ResultSet::1
```

23.6.1. Introspecting result sets: *RESULTSET* `columns`

The result set is a table whose column names can be retrieved with the `tdbc::resultset` object's `columns` method.

```
$res columns → Name Balance
```

The `rowcount` method returns the number of rows in the result set table.

```
$res rowcount → 1
```

23.6.2. Retrieving result set rows: *RESULTSET* `nextlist|nextdict|nextrow`

The data itself can be retrieved using one of several methods. The most basic of these are the `nextlist`, `nextdict` and `nextrow` methods.

```
RESULTSET nextlist VAR
RESULTSET nextdict VAR
RESULTSET nextrow ?-as lists|dicts? VAR
```

All three methods return 0 if there are no more rows in the result set. Otherwise they return 1 and store the next row from the result set into the variable named *VAR*. The difference between them is the format in which the row is stored in the variable:

- `nextlist` stores the row as a list in the same order as returned by the `columns` method.
- `nextdict` stores the row as a dictionary whose keys are the column names of the result set.
- `nextrow` stores the row either as a list or a dictionary depending on the value of the `-as` option (which defaults to `dicts`).

```
% $res nextlist val
→ 1
% puts $val
→ Tom 100.0
% while {[$res nextdict val]} {
    puts $val
}
→ Name Dick Balance 200.0
  Name Harry Balance 300.0
  Name Moe Balance 100.0
```

23.6.3. Multiple result sets: *RESULTSET* `nextresults`

Some databases support a single SQL statement returning multiple result sets, each of which may have a different column structure. The presence of additional result sets may be detected by calling the `nextresults` method. This

method must be called after the nextlist or nextdict command returns 0 indicating there are no more rows in the current result set.

```
$res nextresults → 0
```

The method returns 0, as in our example, if there are no more result sets. A return value of 1 indicates there are more result sets. The application can access them in the same manner as described above while noting that the columns may differ between result sets.

23.6.4. Releasing result sets: *RESULTSET* close

Once the data of interest within a result set is retrieved, associated resources should be released by calling the close method on the tdbc::resultset object.

```
% $res close
```

The result sets are also automatically freed when the associated tdbc::statement object or containing tdbc::connection object are closed.

23.6.5. Convenience wrappers for retrieval

As we have seen above, database operations involve calling the prepare and execute methods and then freeing the tdbc::statement and tdbc::resultset objects. To ensure proper cleanup, the sequence has to be wrapped in try or catch blocks. So in pseudocode the code looks roughly like this:

```
set stmt [$dbconn prepare SQL STATEMENT]
try {
    set res [$stmt execute]
    try {
        loop using [$res nextdict] or [$res nextlist]
    } finally {
        $res close
    }
} finally {
    $stmt close
}
```

TDBC provides two convenience methods, allrows and foreach, that take care of all the boilerplate in the above and are supported by all the major TDBC classes, tdbc::connection, tdbc::statement and tdbc::resultset,

23.6.5.1. Retrieving a complete result set: allrows

The allrows method encapsulates the above pseudocode where the loop processing consists of simply collecting all results returned by nextdict or nextlist into a single list.

The method is implemented by tdbc::connection, tdbc::statement and tdbc::resultset.

```
RESULTSET allrows ?-as lists|dicts? ?-columnsvariable COLVAR?
STATEMENT allrows ?-as lists|dicts? ?-columnsvariable COLVAR? ?--? ?DICT?
DBCONN    allrows ?-as lists|dicts? ?-columnsvariable COLVAR? ?--? SQL ?DICT?
```

- In the case of tdbc::resultset, allrows simply iterates over the result set collecting the output of nextlist or nextdict methods.
- In the case of tdbc::statement, allrows executes the statement and then collects rows from the returned result set as described in the previous case. If the optional *DICT* argument is provided, it contains the bound variables else their values are taken from Tcl variables in the caller's context. See Section 23.5.3.

- In the case of `tdbc::connection`, `allrows` prepares the SQL passed as the *SQL* argument, then executes the returned statement as described in the previous case. The *DICT* argument has the same purpose as above.

The `-as` option controls whether each element of the returned list is itself a list containing the column values for a row or a dictionary keyed by column name (default). If the `-columnsvariable` option is specified, the column names for the result set are stored in the variable *COLVAR* in the caller's context.

In all cases, the `allrows` method takes care to free up objects and resources appropriately even in case of errors.

Below we illustrate a simple query using the different methods, first without using `allrows`.

```
% set query_values [dict create amount 200]
→ amount 200
% set stmt [$dbconn prepare {
    SELECT Name FROM Accounts WHERE Balance >= :amount
}]
→ ::oo::Obj601::Stmt::5
% set rows {}
% try {
    set res [$stmt execute $query_values]
    try {
        while {[$res nextdict row]} {
            lappend rows $row
        }
    } finally {
        $res close
    }
} finally {
    $stmt close
}
% print_list $rows
→ Name Dick
  Name Harry
```

Now the same code but using the `allrows` method of the `tdbc::resultset` object.

```
% set stmt [$dbconn prepare {
    SELECT Name FROM Accounts WHERE Balance >= :amount
}]
→ ::oo::Obj601::Stmt::6
% set rows {}
% try {
    set res [$stmt execute $query_values]
    try {
        set rows [$res allrows] ❶
    } finally {
        $res close
    }
} finally {
    $stmt close
}
→ {Name Dick} {Name Harry}
% print_list $rows
→ Name Dick
  Name Harry
```

❶ Replaces the inner loop in the previous example

Notice this returns the rows as dictionaries by default.

Above, we have only saved writing the innermost loop in the code. We can go another step further and use the `allrows` method of the `tdbc::statement` object. Note the difference from the `allrows` method of the `tdbc::resultset` in that here we need to pass in the values to be used for querying to the `allrows` method.

```
% set rows {}
% set stmt [$dbconn prepare {
    SELECT Name FROM Accounts WHERE Balance >= :amount
}]
→ ::oo::Obj601::Stmt::7
% try {
    set rows [$stmt allrows $query_values] ❶
} finally {
    $stmt close
}
→ {Name Dick} {Name Harry}
% print_list $rows
→ Name Dick
  Name Harry
```

❶ We do not have to explicitly deal with result sets

Finally, we go the whole hog and invoke `allrows` on the database connection itself. Obviously, in this case we have to tell it the SQL we want to run in addition to passing in the query values.

```
% set rows [$dbconn allrows {
    SELECT Name FROM Accounts WHERE Balance >= :amount
} $query_values] ❶
→ {Name Dick} {Name Harry}
% print_list $rows
→ Name Dick
  Name Harry
```

❶ We do not have to deal with statements

Given this last illustration is so much shorter than the previous examples, why would one pick any of the others? The Tcl Database Connectivity paper provides some hints:

- With very large databases and result sets, `allrows` may be unworkable because of the infeasibility of collecting all rows in memory before processing.
- Explicitly dealing with result sets allows for fine-grained control of the iteration, for example terminating the iteration based on some complex rules outside of SQL's capabilities.

The use of the `-as` and `-columnsvariable` is shown below.

```
% set rows [$dbconn allrows -as lists -columnsvariable cols {
    SELECT Name,Balance FROM Accounts WHERE Balance >= :amount
} $query_values]
→ {Dick 200.0} {Harry 300.0}
% print_list $rows
→ Dick 200.0
  Harry 300.0
% puts $cols
→ Name Balance
```

23.6.5.2. Iterating over result sets: `foreach`

The `foreach` method has a purpose very similar to `allrows` except that instead simply collecting results into a list, it executes a given script for every row in the result set. Like `allrows`, it is implemented by the `tdbc::connection`, `tdbc::statement` and `tdbc::resultset` classes.

```
RESULTSET allrows ?-as lists|dicts? ?-columnsvariable COLVAR? VAR SCRIPT
STATEMENT allrows ?-as lists|dicts? ?-columnsvariable COLVAR? ?--? VAR ?DICT? VAR SCRIPT
DBCONN    allrows ?-as lists|dicts? ?-columnsvariable COLVAR? ?--? SQL ?DICT? SCRIPT
```

The method will iterate over all rows in the result set evaluating *SCRIPT* after assigning the value of the row to the variable *VAR*. The options `-as` and `-columnsvariable`, as well as other arguments, have the same semantics as described for allrows in the previous section.

Because of its similarity to `allrows`, we do not discuss the method in detail but only illustrate it as invoked on a `tdbc::connection` object.

```
% $dbconn foreach row {
    SELECT Name FROM Accounts WHERE Balance >= :amount
} $query_values {
    puts $row
}
→ Name Dick
  Name Harry
```

Like `allrows`, `foreach` also takes care of all intermediate bookkeeping in terms allocating and release objects.

23.7. Database transactions

There are a couple of ways an application may make use of transactions. We describe both below.

23.7.1. Using the `transaction` method

The first is making use of the `transaction` method of `tdbc::connection`.

```
DBCONN transaction SCRIPT
```

This begins a transaction on the connection and evaluates the passed script. If the script completes with a return code of `ok`, `return`, `break` or `continue`, the transaction is committed. For other return codes, including errors, the transaction is rolled back and the error is rethrown.

Use of the method is illustrated by the simplistic example below to transfer funds from one account to another.

```
% set transfer { from "Tom" to "Dick" amount 50 }
→  from "Tom" to "Dick" amount 50
% $dbconn transaction {
    $dbconn allrows -as lists -- {
        UPDATE Accounts
        SET Balance = Balance - :amount
        WHERE Name=:from
    } $transfer ❶

    puts "Within transaction: [$dbconn allrows -as lists -- {
            SELECT Name, Balance FROM ACCOUNTS WHERE Name=:from
    } $transfer]" ❷

    error "Pretend something went wrong"
```

```
    $dbconn allrows -as lists -- {
        UPDATE Accounts
        SET Balance = Balance + :amount
        WHERE Name=:to
    } $transfer ❸
}
Ø Within transaction: {Tom 50.0}
  Pretend something went wrong
% $dbconn allrows -as lists -- {
    SELECT Name, Balance FROM Accounts WHERE Name=:from
} $transfer ❹
→ {Tom 100.0}
```

❶ Deduct from Tom's balance
❷ Verify balance updated within transaction
❸ Add to Dick's balance
❹ Verify Tom's balance restored to original

Notice that Tom's balance is restored as the transaction was aborted by an error exception.

23.7.2. Using begintransaction, commit and rollback

In cases where the sequence of operations in a transaction cannot be neatly wrapped in a script that can be passed to the transaction method, an application can explicitly manage the transaction itself by calling the begintransaction method of a tdbc::connection object.

```
DBCONN begintransaction
```

Then at some later point, it can call the commit or rollback methods to either complete or abort the transaction.

```
DBCONN commit
DBCONN rollback
```

We can rewrite the previous example as below.

```
% set transfer { from "Tom" to "Dick" amount 50 }
→  from "Tom" to "Dick" amount 50
%
% $dbconn begintransaction ❶
% $dbconn allrows -as lists -- {
    UPDATE Accounts
    SET Balance = Balance - :amount
    WHERE Name=:from
} $transfer
%
% puts "Within transaction: [$dbconn allrows -as lists -- {
          SELECT Name, Balance FROM ACCOUNTS WHERE Name=:from
} $transfer]"
→ Within transaction: {Tom 50.0}
%
% if {[catch {
    error "Pretend something went wrong"
    $dbconn allrows -as lists -- {
        UPDATE Accounts
        SET Balance = Balance + :amount
        WHERE Name=:to
    } $transfer
}]} {
```

```
    $dbconn rollback ❷
} else {
    $dbconn commit ❸
}
% $dbconn allrows -as lists -- {
    SELECT Name, Balance FROM Accounts WHERE Name=:from
} $transfer
→ {Tom 100.0}
```

❶ Begin the transaction
❷ On error, rollback the transaction
❸ On success, commit the transaction

23.8. Handling NULL values

Noting that the empty string "" is not the same as the SQL NULL value, there is no way to represent NULL in Tcl where everything is a string. In some applications, the distinction is not important and the empty string can be used interchangeably with NULL. In cases where the distinction is important, the dictionary-based interface to TDBC methods should be used as illustrated here.

Writing NULL values

To write NULL to a table column, pass a dictionary containing the bound variable values for the columns. A NULL will be stored in any column for which a corresponding key is not present in the dictionary.

```
% $dbconn allrows {
    INSERT INTO Accounts (Name, AcctNo, Balance) VALUES (:name, :acctno, :balance)
} {name Curly acctno C007}
```

Similarly, to retrieve data containing NULL, use one of the forms that returns rows as dictionaries. If a value is NULL, the returned dictionary for the row will not contain the corresponding key.

```
% $dbconn allrows {SELECT Name, Balance, AcctNo FROM Accounts WHERE Name='Curly'}
→ {Name Curly AcctNo C007}
```

Note that the key `Balance` is missing. You could have also used the `tdbc::resultset::nextdict` method for the same purpose.

Note however the result when list format is used.

```
% $dbconn allrows -as lists {SELECT Name, Balance, AcctNo FROM Accounts WHERE Name='Curly'}
→ {Curly {} C007}
```

In this case there is no way to distinguish whether the stored value in the database was actually "" or NULL.

23.9. Stored procedures: *DBCONN* `preparecall`

Stored procedures can be invoked with the `preparecall` method of a `tdbc::connection` object.

```
DBCONN preparecall CALL
```

The syntax of the stored procedure call is

```
?RESULTVAR =? STOREDPROCNAME (? arg, ...?)
```

Like the `prepare` method, this also returns a `tdbc::statement` object which can then be used as described earlier.

23.10. Introspection

All TDBC classes allow for introspection and inspection of the meta-information associated with databases.

23.10.1. Enumerating objects: *DBCONN* `statements`

TDBC keeps track of the objects that are still open within a database connection. The `statements` and `resultsets` methods retrieve the names of any existing `tdbc::statement` and `tdbc::resultset` objects within the `tdbc::connection`.

```
% $dbconn statements
→ ::oo::Obj601::Stmt::4
% $dbconn resultsets
```

Clearly we forgot to release some objects. This is actually useful for a final cleanup, for example on a per request basis to a web server that leaves the database connection open.

23.10.2. Introspecting tables: *DBCONN* `tables`

We can introspect the tables within a database with the `tables` method of a `tdbc::connection` object.

```
DBCONN tables ?SQLPAT?
```

If the *SQLPAT* argument is not provided, the commands returns information about all tables in the database. Otherwise, only tables whose name matches *SQLPAT* are included in the result. Note that *SQLPAT* should be in SQL pattern syntax.

```
% $dbconn tables
→ accounts {type table name accounts tbl_name Accounts rootpage 2 sql {CREATE TABLE Accounts
      (Name text,
       AcctNo text NOT NULL PRIMARY KEY,
       Balance double)}}
% $dbconn tables A% ❶
→ accounts {type table name accounts tbl_name Accounts rootpage 2 sql {CREATE TABLE Accounts
      (Name text,
       AcctNo text NOT NULL PRIMARY KEY,
       Balance double)}}
```

❶ % is a wildcard in SQL pattern syntax

The command returns a nested dictionary with the first level keys being the table names. The second level keys and values are dependent on the specific database driver. Refer to the reference documentation for the driver for details.

23.10.3. Introspecting columns: *DBCONN* `columns`

Similarly, the `columns` method retrieves column information for one or more columns in a table.

```
DBCONN columns TABLE ?SQLPAT?
```

If the *SQLPAT* argument is not provided, the commands returns information about all columns in the table *TABLE*. Otherwise, only columns with names matching *SQLPAT* are included in the result.

```
% print_dict [$dbconn columns Accounts]
→ acctno    = cid 1 name acctno type text notnull 1 pk 1 precision 0 scale 0 nullable 0
  balance   = cid 2 name balance type double notnull 0 pk 0 precision 0 scale 0 nullable 1
  name      = cid 0 name name type text notnull 0 pk 0 precision 0 scale 0 nullable 1
% print_dict [$dbconn columns Accounts Bal%]
→ balance   = cid 2 name balance type double notnull 0 pk 0 precision 0 scale 0 nullable 1
```

The command result is a nested dictionary keyed by the column name. The second level dictionary for each column provides details about the column and contains the keys shown in Table 23.5.

Table 23.5. Column description keys

Key	Description
type	Data type of the column
precision	Column precision in bits, decimal digits or width in characters, depending on the column type
scale	Scale of the column, i.e. number of digits after the radix point
nullable	Has the value 1 if the column can contain SQL NULL values and 0 otherwise

Additional keys may be present in the second level dictionaries depending on the database driver in use. Refer to the TDBC documentation for the driver for details.

23.10.4. Introspecting keys: *DBCONN* primarykeys|foreignkeys

Information about the primary keys defined for a table can be obtained with the primarykeys method of a tdbc::connection object.

```
DBCONN primarykeys TABLE
```

The method returns a list of descriptors for the primary keys defined for the table. Each descriptor is a dictionary with at least the key columnName containing the name of a column that is a member of the primary key. The descriptor may contain other database-dependent keys.

```
% $dbconn primarykeys Accounts
→ {ordinalPosition 2 columnName AcctNo}
```

In a similar vein, the foreignkeys method retrieves information about foreign key relationships for the specified table.

```
DBCONN foreignkeys ?-primary TABLE? ?-foreign TABLE?
```

If the -foreign option is specified, only the keys appearing in that table are included in the result. If the -primary option is specified, only the keys that refer to that table are included. It is recommended that one or both options be specified as otherwise the returned results depend on the database driver in use.

The method returns a list of descriptors of foreign key relationships. Each descriptor is a dictionary with the keys shown in Table 23.6. Depending on the database in use, the descriptor may contain additional keys apart from the ones shown in the table.

Table 23.6. Foreign key dictionary

Key	Description
`foreignTable`	The table containing the foreign key.
`foreignColumn`	The column containing the foreign key.
`primaryTable`	The table being referenced by the foreign key.
`primaryColumn`	The column being referenced by the foreign key.

23.11. ODBC utilities

The `tdbc::odbc` package provides some utility commands related to interacting with the ODBC manager. Some of these depend on the system ODBC manager's support of the ODBC Installer API.

The `tdbc::odbc::drivers` command enumerates the installed ODBC drivers on the system.

```
% package require tdbc::odbc
→ 1.0.4
% print_dict [tdbc::odbc::drivers]
→ SQL Server = APILevel=2 ConnectFunctions=YYY CPTimeout=60 DriverODBCVer=03.50 FileUsage=0
   ↳ SQLLevel=1 UsageCount=1
```

The `tdbc::odbc::datasources` command enumerates the configured ODBC data sources on the system. The command accepts the `-user` and `-system` options to limit the returned list to those configured for the current user and system respectively.

```
% print_dict [tdbc::odbc::datasources]
→ Excel Files              = Microsoft Excel Driver (*.xls, *.xlsx, *.xlsm, *.xlsb)
  MS Access Database       = Microsoft Access Driver (*.mdb, *.accdb)
  Visio Database Samples   = Microsoft Access Driver (*.mdb, *.accdb)
% print_dict [tdbc::odbc::datasources -user]
→ Excel Files              = Microsoft Excel Driver (*.xls, *.xlsx, *.xlsm, *.xlsb)
  MS Access Database       = Microsoft Access Driver (*.mdb, *.accdb)
  Visio Database Samples   = Microsoft Access Driver (*.mdb, *.accdb)
```

Finally, the ODBC driver provides an ensemble command, `tdbc::odbc::datasource` for management of ODBC data sources. It takes the one of the forms

```
tdbc::odbc::datasource SUBCMD DRIVER ?KEYWORD=VALUE?
```

Here *DRIVER* identifies the ODBC driver being targeted. The possible values for *SUBCMD* are shown in Table 23.7.

Table 23.7. tdbc::odbc::datasource commands

Command	Description
`add`	Adds a new user data source.
`add_system`	Adds a new system data source.
`configure`	Configures a user data source.
`configure_system`	Configures a system data source.
`remove`	Removes a user data source.
`remove_system`	Removes a system data source.

For all the above, the data source that is the target of the command is specified by a DSN entry in the list of keywords supplied to the command. See the reference documentation for examples.

23.12. Chapter summary

Databases comprise an important component of many software applications. Different database implementations provide different driver API's and there exist many Tcl extensions that provide programmatic access to specific databases.

TDBC is a means to access these different implementations through a uniform object-oriented API. In this chapter, we covered how you can generically use TDBC to

- establish connections
- prepare and execute statements
- execute transactions
- retrieve results

and also covered some specifics pertaining to the SQLite, MySQL, PostgreSQL and ODBC drivers.

23.13. References

TIPTDBC
Tcl Database Connectivity (TDBC), Kevin B. Kenny et al, Tcl Improvement Proposal #350, http://www.tcl.tk/cgi-bin/tct/tip/350

KBKPAPER
Tcl Database Connectivity, Kevin B. Kenny. Tcl Conference Proceedings, 2008. The original paper describing TDBC. Available from http://www.tclcommunityassociation.org/wub/proceedings/Proceedings-2008/proceedings/tdbc/tcl2k8-kenny-withfonts.pdf

TDBCREF
TDBC reference pages, http://www.tcl.tk/man/tcl8.6/TdbcCmd/contents.htm

WWWCONNSTR
Connection Strings Reference, https://www.connectionstrings.com/

24

Testing and Performance

A very large part of every software development process is (or should be!) ensuring quality — making sure that the software works as expected and at a satisfactory performance level. This chapter describes some of the tools available within Tcl for meeting these goals in Tcl applications.

24.1. Testing

> Quality is not an act, it is a habit.
>
> — Aristotle

If developing software in Tcl, the interactive nature of Tcl streamlines the development of unit tests. As you code procedures and commands, you can, and **must**, interactively call them from a Tcl shell to verify they work correctly with various inputs. On failures, you can fix the code and reload the changes into the shell with the `source` command for further testing. This mode of development should, as the great philosopher stated, become a habit.

These interactive "off-the-cuff" tests can then be easily formalized, expanded and translated to a form suitable for inclusion in an automated suite. A comprehensive test suite is crucial to shipping robust software. The direct benefits are not just a better customer experience and reduced support costs but also freedom to refactor, optimize and enhance the software while maintaining a high level of confidence that working functionality is not broken by the changes.

Even when the software under test is written in a different language, by its very nature Tcl is very well suited as a language for building test automation suites and is widely used for that purpose. Several commercial test automation products as well as the open source DejaGNU[1] framework are based on Tcl.

Here we will only describe the `tcltest` package which comes with the Tcl source distribution and can be used for testing your own packages as well as applications.

24.1.1. The `tcltest` package

We will describe the `tcltest` package first as that is included in the Tcl core and used in the testing of not only Tcl itself but many other packages and extensions. The package is included in many but not all binary distributions. You can check if it is installed by attempting to load it.

```
package require tcltest → 2.4.0
```

If you get an error, your binary distribution probably does not include it. In that case, you will need to download a source distribution and extract the `tcltest` directory into a directory included in your Tcl installation's `auto_path` variable.

A `tcltest`-based test script consists primarily of a series of `tcltest::test` commands.

```
test NAME DESCRIPTION ?OPTION ...?
```

[1] http://www.gnu.org/software/dejagnu/

The *NAME* argument is a label given to the test for identification purposes. It is recommended to use names with some structure so as to be able select a subset of tests using glob wildcard matching. The *DESCRIPTION* argument is strictly for human consumption but clear descriptions greatly help in formulating a complete test suite. The rest of the arguments are specified as options and we will describe these as we go through our example.

We will demonstrate the use of the package with a simple example, a script containing some tests for Tcl's `incr` command. Test scripts using the package are simply Tcl scripts so you can use any Tcl commands, load other packages and so on. By convention, test scripts are given the `.test` extension so we name our test script `incr.test`.

The test script should start off by loading the `tcltest` package. You may also load any other packages required for testing, define supporting procedures and so on. For our simple example, we do not need anything else.

```
package require tcltest
```

A test case is defined with the `tcltest::test` command. For convenience we import this command to save us some typing.

```
namespace import tcltest::test
```

Let us write a test for the basic use of `incr`.

```
test incr-1.0 {
    Verify default increment of 1, the new value is returned and stored
} -setup {
    set i 1
} -body {
    list [incr i] $i
} -cleanup {
    unset i
} -result {2 2}
```

The first argument labels the test case and the second describes what we are testing. The `-setup` option specifies a script that does any required set up before the actual test can be run. In our case, it just initializes a variable to a known value. The `-body` option specifies the actual test itself. For our test case, it is meant to check the result of the command as well the new value of the variable. Many times the `-setup` script is combined with the `-body` script but we do not recommend this as it obfuscates exactly what is being tested. The `-cleanup` option specifies a script to run to remove any side-effects of the test case. Finally, the `-result` option specifies the value that is expected as the result of executing the script provided as the `-body` argument.

The options may be specified in any oder and not all need be specified though obviously it does not make sense to have a test case without a `-body` option. The `-result` option will default to the empty string. In test cases as simple as the one above you will generally see neither `-setup` nor `-cleanup` specified.

It is important for a test suite not only to verify correct behaviour on valid input, but to also verify that the command handles bad input correctly. So we will now define a test case to ensure invalid input is correctly rejected by the command.

```
test incr-err-1.0 {
    Verify non-integer variable generate an error.
} -setup {
    set i notaninteger
} -body {
    incr i
} -cleanup {
    unset i
} -returnCodes error -result "expected integer.*notaninteger" -match regexp
```

There are two additional options we see here. The `-returnCodes` option specifies the expected return code from evaluation of the `-body` script. By default, as in our previous test case, it is expected to be ok indicating a normal script completion. In this case, we expect an error exception and accordingly specify that as the `-returnCodes` value. The `-result` option now holds the expected error message. Since error messages may change slightly between builds, we do not want to specify an exact message value. We therefore use the `-match` option to indicate that the `-result` option value should be treated as a regular expression to be matched with the `-body` script result. You may also specify glob matching or even define your own matching criteria.

Having compiled a comprehensive set of tests, we end our test script with a call to clean up.

```
::tcltest::cleanupTests
```

The `tcltest::cleanupTests` command does two things. It cleans up any temporary files and other changes made by the test harness. It also prints out summary statistics from the test run.

Here is our entire test script.

```
# incr.test
package require tcltest

namespace import tcltest::test

test incr-1.0 {
    Verify default increment of 1, the new value is returned and stored
} -setup {
    set i 1
} -body {
    list [incr i] $i
} -cleanup {
    unset i
} -result {2 2}

test incr-err-1.0 {
    Verify non-integer variable generate an error.
} -setup {
    set i notaninteger
} -body {
    incr i
} -cleanup {
    unset i
} -returnCodes error -result "expected integer.*notaninteger" -match regexp

::tcltest::cleanupTests
```

Let us try running this test script from the Windows or Unix shell.

```
C:\temp>tclsh incr.test
incr.test:      Total   2       Passed  2       Skipped 0       Failed  0
```

In case of any failures, `tcltest` will print details about the failing test cases.

We can also choose to run a subset of the tests by matching test labels and ask for more verbose output.

```
C:\temp>tclsh incr.test -match *err* -verbose p
++++ incr-err-1.0 PASSED
incr.test:      Total   2       Passed  1       Skipped 1       Failed  0
```

Notice only the error test case was run and the other one skipped. There is also a `-skip` option that is complementary to `-match` and specifies tests that should be skipped.

There are a number of other features that we will not describe here. These include, among others

- ability to set up test constraints so that tests are only run if certain conditions are satisfied, for example limiting certain tests to specific platforms
- checking expected output on standard channels
- utility commands for creation and clean up of temporary files and directories.

Hai Vu's blog[2] contains series of posts related to both basic and advanced use of the `tcltest` package. You may find it a more accessible tutorial for the `tcltest` package than the reference documentation.

24.1.2. Testing interactive applications

Testing of interactive applications poses some unique challenges in terms of simulating user actions and verifying the output which may be in the form of terminal output, a web page or a native user interface. The `tcltest` package is not suitable for this purpose so we list some packages below that can be used to supplement `tcltest` with the required functionality. Note that these written in Tcl, but with the exception of `TkTest`, are not limited to only testing Tcl applications.

The Expect extension

The Expect[3] extension, is widely used not just for testing but also for automation of systems administration tasks. It is primarily used on Unix systems; there is a Windows version also available but has some stability issues.

Expect is targeted toward applications that interact with the terminal. It derives its power through its ability through Tcl commands for managing pseudo-terminals, emulating user keystrokes, capturing program output and responding accordingly. It is not even limited to the local system as it can be used to drive telnet, ssh or similar terminal based network applications. This makes it suitable even for testing embedded devices, networking equipment etc. as long as they expose a telnet or serial line type of interface.

Expect itself does not provide any test framework and its use for testing interactive applications is through integration with `tcltest` or other frameworks like Caius discussed below.

The Caius test framework

The Caius[4] framework is an alternative to `tcltest` where Tests are written using Incr Tcl, an object-oriented extension of Tcl.

It has three distinguishing features:

- Integration with Expect for testing interactive applications
- Ability to interface to the Selenium Webdriver API for testing Web pages
- Support for continuous integration tools like Jenkins and report generation

Although `tcltest` and Caius are both test frameworks, it is possible to run them in integrated fashion where one drives test cases written using the other.

The TkTest package

The TkTest[5] package is meant for testing GUI applications written in Tcl with the Tk extension. The package works by recording events and application state while you manually use the application. The recorded events can then be replayed and compared to the saved application state snapshots for regression testing purposes.

[2] https://wuhrr.wordpress.com/category/programming/tcl-programming/
[3] http://expect.sf.net
[4] http://caiusproject.com
[5] http://www.cwflynt.com/tktest/

24.1.3. Source code checkers

As a dynamic interpreted language, Tcl does not have a "front-end" compiler. A procedure body for example is compiled the first time it is invoked. Thus syntactic errors may not be detected until runtime. A couple of tools are available to help detect such errors through source code analysis. We will not describe them here but only mention them for your reference.

Nagelfar[6] performs syntactic analysis on Tcl source code to detect not only syntactic errors but some common programming errors such as uninitialized variables. It can also be extended with plugins for additional checks, call analysis etc.

Frink[7] is primarily a tool for formatting and pretty-printing Tcl but also includes some features for syntax checking and programming errors.

24.2. Improving performance

> If you optimize everything, you will always be unhappy.
>
> — Donald Knuth

Given that, improving performance involves multiple steps:

- profiling the application to determine which parts of the application are the "hotspots" that need attention
- measuring execution time for implementations of alternative algorithms for these hotspots
- **if needed**, examining the code for possible "microoptimizations" that can nevertheless significantly speed up the program under certain circumstances.

The next few sections go into these topics.

24.2.1. Profiling scripts

Writing a basic profiler for Tcl scripts is easy because of Tcl's malleability and introspective capabilities. For example, procedures can be redefined or wrapped to collect timing information on entry and exit. An alternative method is to use the command and execution tracing facility described in Section 10.6. You will find several such profilers referenced in Tcler's Wiki[8]. Here we will describe the `profiler` package included in Tcllib[9].

Note however that although writing the profiler is easy, interpreting the data needs to be done with care for many reasons. The statistics relating to call counts are generally accurate but timing information is less so, particularly for profilers measuring at the source line level. The profiler itself has significant impact on speed of execution. It is best to not rely on profiler output for fine grained measurements and use the output for **relative** comparison of where time is being spent.

When profiling an application, the `profiler` package must be loaded and the `profiler::init` command called **before** any of the procedures to be profiled are defined.

```
package require profiler → 0.3
profiler::init           → (empty)
```

Let us now write some sample code that we will use for demonstration purposes. The procedures do not do anything but pretend they are doing useful work by napping.

```
proc fiddle {} { after 10 ; twiddle }
proc twiddle {} { after 20 }
proc diddle {} { after 30 }
```

[6] http://nagelfar.sourceforge.net/
[7] http://catless.ncl.ac.uk/Programs/Frink/
[8] http://wiki.tcl.tk
[9] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

```
proc main {} {
    for {set i 0} {$i < 10} {incr i} {
        fiddle
        diddle
    }
}
```

Now we call `main` and then print the collected information.

```
% main
% profiler::print ::fiddle
→ Profiling information for ::fiddle
  ============================================================
              Total calls:  10
       Caller distribution:
    ::main:  10
             Compile time:  42296
            Total runtime:  463968
          Average runtime:  46396
            Runtime StDev:  1442
           Runtime cov(%):  3.1
    Total descendant time:  312627
  Average descendant time:  31262
  Descendants:
    ::twiddle:  10
```

We have only printed the profiler data for `fiddle`. If we had not provided an argument to `profiler::print` it would have printed it for all procedures.

The `profiler::print` command require the **fully qualified** command name to be provided.

Most of the information is self-explanatory. The one item that needs some explanation is the `Compile time` line. The first time a procedure is invoked, Tcl compiles its body into bytecode. Thus the first invocation of a procedure is often significantly longer than subsequent invocation. This difference does not show up in our example because our artificial delay swamps any additional compilation time required.

An alternative to `profiler::print` is the `profiler::dump` command which returns the same information in a dictionary format. The package offers utility commands to suspend, resume and reset profiling specific procedures or all of them.

24.2.2. Timing scripts

Having pinpointed what parts of the application need to be worked on, we can try different algorithms to improve performance, say using a skip list versus a binary search tree. We then need a means for measuring execution time for each alternative. Tcl provides the `time` command for this purpose.

```
time SCRIPT ?COUNT?
```

The command evaluates *SCRIPT* a *COUNT* number (default once) of times and prints the average execution time for one evaluation. It is important to pick a sufficiently large *COUNT* value, not just to smooth variations, but because the first execution of a script will often result in the script being compiled to bytecode resulting in an additional cost. We have more to say about this later but first let us just go through an example.

We want to generate a list of the first *N* natural numbers. We write two procedures that use a `for` and a `while` respectively. (These are not two different algorithms, but never mind that.)

```
proc rangefor n {
    for {set i 1} {$i <= $n} {incr i} {
        lappend l $i
    }
    return $l
}

proc rangewhile n {
    set i 0
    while {$i < $n} {
        lappend l [incr i]
    }
    return $l
}
```

We then use `time` to measure their execution time.

```
time {rangefor 10000} 100   → 5522.35 microseconds per iteration
time {rangewhile 10000} 100 → 5103.37 microseconds per iteration
```

Let us get back to the question of what value to use for *COUNT*. A value that is too small will be skewed by the compilation time. Too large a value will take longer to measure. The Tcler's Wiki[10] page on the time[11] command suggests a minimum 2-second run with at least 500 iterations and provides a procedure for automating this.

```
proc measure args {
    set it [expr {int(ceil(10000./[lindex [time $args] 0]))}]
    set it [expr {int(ceil(2000000./[lindex [time $args $it] 0]))}]
    while {$it < 500} {set it 500}
    lindex [time $args $it] 0
}
```

We can then measure execution time as follows:

```
measure rangefor 10000 → 5611.71
```

As an aside, our example was for pedagogic purposes demonstrating the use of the `time` command. In general, it does not make sense to do microoptimization at this level as continued Tcl bytecode compiler enhancements may change the timings of specific commands.

You will sometimes find `time` being used as an iterative control structure for repeating a loop. For example, we could construct our integer range list with the following procedure.

```
proc rangetime n {
    time {lappend l [incr i]} $n
    return $l
}
measure rangetime 10000
→ 13020.888
```

Here the `time` is used in `rangetime` simply to repeat a script evaluation, not to actually measure time. As you can see, it is slower than the others but for interactive use in a shell it is less typing and convenient.

[10] http://wiki.tcl.tk
[11] http://wiki.tcl.tk/734

24.2.3. Performance hints

The top three factors that affect application performance are algorithms, algorithms and algorithms. Nevertheless, the speed of low level operations can come into play as well so we provide some hints here that you might keep in mind when writing Tcl.

Directly modify variables

Most Tcl commands accept values as arguments and return results as values. Commands like append, `lappend`, `lset` and `incr` on the other hand operate on variables. In cases where the result of a command is assigned back to the variable, these should be preferred. For example, prefer append to string interpolation

```
append var "foo"
set var "${var}foo"
```

or `incr` to `expr`

```
incr x $y
set x [expr {$x+$y}]
```

Similarly, wherever possible `lappend` and `lset` should be preferred to other list commands like `linsert`, `lreplace`. A simple timing test illustrates the difference:

```
set l [lrepeat 10000 X]; llength $l  → 10000
time {set l [lreplace $l 0 0 Y]} 100 → 262.42 microseconds per iteration
time {lset l 0 X} 100                → 0.96 microseconds per iteration
```

As you can see, the difference can be very large although this is an extreme case.

The reason for the difference in performance is Tcl's reference counting mechanism described in Section 10.10.1.2. In the case of `lreplace` for example, a copy of the list storage has to be made for modification since it is has multiple references and therefore cannot be modified. In the case of `lset`, since we are modifying the variable itself, the associated value can be modified in place saving on memory and copying costs.

Explicitly drop references

Consider deleting the last element of a list stored in a variable. There is no command that operates directly on a variable so we are forced to use `lreplace`. Again this has the potential to be slow with large lists. We can time how long it takes us to drop the last element of a list.

```
set l [lrepeat 10000 X]; llength $l     → 10000
time {set l [lreplace $l end end]} 1000 → 84.186 microseconds per iteration
```

Here is an alternative implementation that is much faster. The explanation follows.

```
set l [lrepeat 10000 X]; llength $l                → 10000
time {set l [lreplace $l[set l ""] end end]} 1000 → 0.925 microseconds per iteration
```

As you can see, that is again a very large difference. Although, it may not be obvious, the reason for the speed-up is the same as we described for modification of variables. When Tcl executes the statement

```
lreplace $l end end
```

the `lreplace` command has to return a new list contains the same elements as that stored in `l` except for the last element. It cannot modify the list directly because the variable `l` is still referencing it and the command semantics do not allow the variable to be modified. Thus, the cost of an additional memory allocation and copying of elements is incurred. Our second implementation fixes this.

```
lreplace $l[set l ""] end end
```

Now when Tcl parses the command arguments for `lreplace`, the first argument is still `$l` since the appended empty string does not change it (in fact the compiler will optimize it away). But now, it assigns an empty string to `l` which means when `lreplace` runs the variable `l` no longer points to the original list storage. Because the list storage is not referenced from anywhere else, the command is free to modify it in place by simply decrementing its element count. No memory allocation or copying of elements in needed.

We could thus write a complement to `lappend`, a `lpop` that removes elements from end of a list stored in a variable.

```
proc lpop {lvar} {
    upvar $lvar l
    lreplace $l[set l ""] end end
}
```

Though obviously not universally applicable, this trick is not limited to `lreplace` but can be used with other commands that modify large lists or strings. Its effectiveness depends on the specific operation. For example, if we were deleting the **first** element of the list instead of the **last** above, you would not see much benefit as only the memory allocation would be saved; the elements would still have to be copied to the previous slot.

It is also confusing to programmers unfamiliar with the idiom so selective usage with suitable comments is advised when put into use.

Avoid string generation

In Section 10.10.1.1 we described how Tcl stores data internally in different formats depending on its type. Its string representation is only generated when required. This happens either when some string operation is applied including I/O operations like printing to the console. This string representation incurs an unnecessary cost in memory and speed so should be avoided where possible. Examine the code below

```
% set l [list a b c] ; tcl::unsupported::representation $l
→ value is a list with a refcount of 3, object pointer at 0000000004BE7DC0, internal
   ↳ representation 0000000004B485C0:0000000000000000, no string representation
% if {$l eq ""} {
    puts "List is empty"
}
% tcl::unsupported::representation $l
→ value is a list with a refcount of 2, object pointer at 0000000004BE7DC0, internal
   ↳ representation 0000000004B485C0:0000000000000000, string representation "a b c"
```

The list starts out without a string representation but string operation

```
$l eq ""
```

causes a string representation to be generated. The internal type-specific representation is still a list as shown (which is a good thing) but nevertheless the generation of a string representation can slow down and consume memory unnecessarily in the case of large lists. The correct way to check for an empty list would be

```
if {[llength $l] == 0} ...
```

In short, for logical correctness as well as performance, stick to list operations for lists, dictionary operations for dictionaries etc.

Avoid shimmering

A similar (except worse!) situation is when in addition to a string representation being generated, the internal type-specific representation is also lost. For example,

```
% if {[string length $l] == 0} {
    puts "List is empty"
}
% tcl::unsupported::representation $l
→ value is a string with a refcount of 2, object pointer at 0000000004BE7DC0, internal
  ↳ representation 0000000044B77F0:0000000000000000, string representation "a b c"
```

Here even list internal representation is lost when the `string length` command is called. Any further list operations will then require the list representation to be recreated — a double whammy.

Again, most such cases are logical inconsistencies as much as performance issues.

Brace your expressions

Any expressions passed to commands like `expr`, `if` etc. should always be enclosed in braces. This important for reasons other than performance as we discussed in Section 7.2.2 but here we will only focus on the latter. Braced expressions are significantly faster to evaluate. This is true even for the simplest expressions.

```
set a 1                 → 1
time {expr $a} 10000    → 0.873 microseconds per iteration
time {expr {$a}} 10000 → 0.5083 microseconds per iteration
```

The reason for this is that braced expressions can be compiled by `expr` internally because the content inside the braces are not subject to substitution at the Tcl level. This is not true of unbraced expressions.

Use `tcl::mathop` commands for numerical lists

Do not forget the presence of the mathematical operator commands in `tcl::mathop`. Adding or multiplying a large list of numbers is much faster with these commands than with a loop using `expr` or `incr`.

```
set l {1 2 3}          → 1 2 3
tcl::mathop::+ {*}$l → 6
```

Use procedures instead of scripts

Procedure bodies are always compiled before execution unlike scripts. So for example when dynamically constructing code that will be called frequently, you are often better off constructing a procedure and calling that rather than constructing a script and running it with `eval`.

Use `try` instead of `eval`

Evaluating a script with `try` is faster than with `eval` (though still not as fast as a procedure body). This is true even for the single argument form of `eval`. Thus you should use `try` to evaluate scripts even if you have no need for the exception handler or finalization clauses. This is actually just a historical artifact and this difference may go away in a future release.

In general, dynamically created scripts are always slower as the compiler can do fewer optimizations.

Use local variables

Access to local variables is much faster than for any other type of variable. Therefore, cache namespace and global variables in locals in tight loops or other performance sensitive areas.

A special case of this relates to variables accessed in `uplevel` scripts. These run much faster if the caller precreates the variable. The example below from http://wiki.tcl.tk illustrates this.

```
set l [lrepeat 100000]
proc p l {
    uplevel 1 [list foreach v $l {}]
}
proc q l {
    p $l
}
proc r l {
    set v {}
    p $l
}
```

The procedures q and r are identical except that the latter initializes the variable v used in the uplevel call. This reserves a slot in the local variable table making access to it significantly faster as shown below.

```
% time {q $l} 10
→ 76058.7 microseconds per iteration
% time {r $l} 10
→ 66597.2 microseconds per iteration
```

Use string operations instead of patterns and regular expressions

Commands that operate on plain strings are usually faster than those that work with glob patterns or regular expressions and should be preferred where possible. For example, use string map in lieu of regsub for replacement of constant strings. Note however that some special cases are highly optimized in the regular expression engine so always "measure before cutting".

Minimize I/O calls

Tcl's I/O system is extremely flexible but that comes at some cost in performance. Consequently it makes sense to minimize the number of times the I/O boundary is crossed. One way to do this is to increase the size of the I/O buffer via the -buffersize option to chan configure and read data in larger chunks. At one extreme, unless the file is very large it is faster to read it all in one shot and process it as a string. For example, rather than use gets in a loop, use the idiom

```
set chan [open $path]
foreach line [split [read $chan [file size $path]] \n] {
    ...process line...
}
```

We reiterate that many of the optimizations mentioned above may reduce the clarity of the code and their use should be carefully considered and limited to the really performance sensitive areas based on actual measurements.

24.3. Chapter summary

Testing and performance tuning are often the focus only in the last few stages of software delivery. Tcl's interactive nature however encourages integration of these tasks right from the beginning of the development process. In this chapter we described the tools and packages that Tcl provides for the purpose.

Appendix A. Libraries and Extensions

The programming libraries available for a language are as important as the language itself. Here we list a tiny subset of those available for Tcl — the ones that are in the author's opinion commonly used or otherwise worthy of attention. There are several sites that have a more complete listing; the author's personal favorite is the Great Unified Tcl/Tk Extension Repository or GUTTER[1] [2] which lists and categorises libraries and extensions.

A.1. GUI toolkits

Package	Description
`Tk`	The most well known Tcl extension is of course the Tk graphical toolkit which is so closely associated with Tcl as to be referred collectively as Tcl/Tk. Tk's cross-platform portability, widget set and ease of use has made it the de facto graphical toolkit of choice not just for Tcl but also for other languages like Python, Ruby and Perl. Tk is generally included in all binary distribution of Tcl and available in source form from the same location[3] as Tcl. Most existing books on Tcl include Tk in their coverage and an excellent online book that includes the latest features is available from http://www.tkdocs.com.
`gnocl`	This is an alternative GUI toolkit based on GTK+/Gnome and comes with its own extensive set of widgets. This is however not widely used in the Tcl world and primarily meant for Unix systems. Available from http://www.gnocl.org.

A.2. Internet protocols

Package	Description
`tls`	Implements the SSL/TLS security protocols. Available from https://core.tcl.tk/tcltls/home.
`http`	Implements the client-side HTTP protocol. Available as part of the core Tcl distribution.
`rl_http`	An alternative implementation of the client-side HTTP protocol from RubyLane. Available from https://github.com/RubyLane/rl_http.
`Tnm`	Part of the Scotty network management suite, this package includes support for SNMP, ICMP and DNS protocols. Available from https://github.com/flightaware/scotty.
`tclcurl`	A wrapper around the well-known Curl[4] multiprotocol client library which supports a very wide range of Internet protocols including HTTP, FTP, Gopher, IMAP, LDAP, POP3, SMTP, Telnet and many others. Available from https://github.com/jdc8/tclcurl.
`WS::Server`, `WS::Client`	Server and client implementations of Web Services using Web Services over SOAP. Also supports JSON responses via REST. Available from http://core.tcl.tk/tclws/home.
Tcllib[5]	Contains modules for implementations of several network protocols like FTP, SMTP, NNTP and NTP. In some cases, the server side protocol is also supported.

[1] http://core.tcl.tk/jenglish/gutter/
[2] Yes, Tcl'ers often have a warped sense of humor.
[3] https://sourceforge.net/projects/tcl/files/Tcl/
[4] https://curl.haxx.se/
[5] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

A.3. Web servers and frameworks

Package	Description
`tclhttpd`	A pure Tcl web server with a Tcl based templating system. Available from https://sourceforge.net/projects/tclhttpd.
Rivet	An Apache module for creating web applications. Available from https://tcl.apache.org/rivet.
Naviserver	Web server with dynamic pages in Tcl, and high end features like database connection pooling and multithreading. Available from https://sourceforge.net/projects/naviserver.
Wub	The web server that hosts Tcler's Wiki[6]. Available from https://code.google.com/archive/p/wub/.
Woof!	A Web framework similar to Rails for Ruby. Web server agnostic and can be used with Apache, IIS, Lighttpd etc.
Tclssg	This is a static site generator using Markdown for content and tools for website management. Available from https://github.com/tclssg/tclssg.

A.4. Numeric computing

Package	Description
Tcllib[7]	Tcllib[8] includes modules for several types of numeric computation such as numerical integration, solving ODE's, combinatorics, fourier transforms, linear algebra, statistics, complex numbers, geometrical computations, and more.
mathemaTcl	The mathemaTcl project provides package wrappers for a variety of C and Fortran libraries for numerical computation. Home page at http://chiselapp.com/user/arjenmarkus/repository/mathemaTcl/index.
`vectcl`	The VecTcl extension is geared towards processing of numerical arrays with support for vectors, matrices and tensors. It also supports complex numbers. The extension is written in C for high performance and also offers a specialized syntax for numerical computation. Available from http://auriocus.github.io/VecTcl.
`nap`	The Tcl-nap (n-dimensional array processor) is another C based extension that implements efficient commands for processing n-dimensional arrays. It includes support for HDF and netCDF file storage formats. Available from http://tcl-nap.sourceforge.net.
`mpexpr`	This extension offers the ability to calculate with arbitrary precision. Unlike Tcl's native capability which has support for integers of unlimited size, `mpexpr` works with floating point numbers as well. Available from https://sourceforge.net/projects/mpexpr/files.

A.5. Database access

We described the generic core package for database access, TDBC, in Chapter 23. There are however also additional extensions that either target non-SQL databases, or are customized for specific database implementations. There are too many to enumerate here and there is no basis for which ones are "important" so we will just point you to their listing[9] in the GUTTER catalog.

[6] http://wiki.tcl.tk
[7] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html
[8] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html
[9] http://core.tcl.tk/jenglish/gutter/#cat-database

A.6. XML processing

Package	Description
tdom	The most widely used Tcl package for XML processing is tDOM. This extension is a very fast engine for parsing and generating XML with excellent XPath and XSLT support. You can also use the package to parse HTML. Available from http://core.tcl.tk/tdom.
tclxml	Another option for processing XML comes from the TclXML project. However, this is not under active development although it is stable enough for production use. It's biggest advantage over tDOM is that it has an optional implementation in pure Tcl that does not require any binary extensions. Available from https://tclxml.sourceforge.net.

A.7. Integration with other languages

Package	Description
ffidl	Allows direct calling of functions implemented in shared libraries from Tcl scripts as long as the functions use the C calling convention. Available from https://github.com/prs-de/ffidl.
tcc4tcl	This extension is a full C compiler that can compile and call C code embedded in a Tcl script at runtime. Note however that unlike most Tcl extensions, it is covered under the more restrictive LGPL license. Available from http://chiselapp.com/user/rkeene/repository/tcc4tcl/index
java	The Tcl Blend extension implements the java package which offers the ability to access and execute Java code from Tcl as well embed a Tcl interpreter into a Java application. Available from http://tcljava.sourceforge.net.
garuda	The Garuda extension integrates Tcl with the .Net platform. It allows access to libraries written in any .Net based language, such as C#, VB.Net etc. Available from http://eagle.to.
duktape	Implements bindings to the Duktape Javascript interpreter library. It can be used to run Javascript code from within Tcl. Available from https://github.com/dbohdan/tcl-duktape.
libtclpy	Supports calling Python code from Tcl and vice versa. Available from https://github.com/aidanhs/libtclpy.

A.8. Image processing

Package	Description
Img	The Img package adds support for a large number of image formats. Its primary use is in conjunction with the Tk graphical extension. Available from https://sourceforge.net/projects/tkimg/.
crimp	The CRIMP extension implements a set of commands for manipulation of raster images. Available from http://chiselapp.com/user/andreas_kupries/repository/crimp/home.
tclmagick	A wrapper for the GraphicsMagick and ImageMagick image processing libraries. Available from http://tclmagick.sourceforge.net/.
tclgd	A wrapper for the libGD graphics drawing library with support for JPEG, PNG, GIF and other popular formats. Available from https://flightaware.github.io/tcl.gd/.
pdf4tcl	A pure Tcl package for generating PDF files. Available from http://pdf4tcl.sourceforge.net.
tclhpdf	A wrapper around the Haru PDF library for generating PDF files. Available from http://reddog.s35.xrea.com/wiki/tclhpdf.html.

`tclMuPdf`	A wrapper for the MuPDF framework for rendering PDF or extracting data from a PDF file. See http://wiki.tcl.tk/48296.

A.9. Platform-specific extensions

A.9.1. Windows extensions

Package	Description
`registry`	Provides commands for reading and writing the Windows registry. The package is part of the core Tcl distribution.
`dde`	Implements the DDE protocol used for interprocess communication. The protocol itself is however obsolete and should not be used in new applications. This package is also part of the core Tcl distribution,
`twapi`	The Tcl Windows API package implements a large portion of the Windows API. It allows Windows services, as well as COM clients and servers, to be written as pure Tcl scripts. It provides access to the Windows event log, WMI, administrative API's, crypto services, systems services, desktop integration and more. Available from https://twapi.sf.net.
`cawt`	Includes modules for integrating with Microsoft Office applications and file formats. Available from http://www.posoft.de/html/extCawt.html.

A.9.2. Unix extensions

Package	Description
`expect`	The Expectextension, which we briefly mentioned in Chapter 24, is one of the most popular Tcl extensions. It is most commonly used in Unix environments for automation of tasks that require user interaction. Available from http://expect.sf.net.
`tuapi`	The Tcl Unix API package wraps several system calls. Currently this is only available on Linux. Available from https://chiselapp.com/user/rkeene/repository/tuapi/index.
`tclx`	The Extended Tcl (TclX) package also provides additional Posix interfaces to files, system services, signals etc. Available from https://sourceforge.net/projects/tclx.

A.9.3. Android extensions

In the case of the Android platform, the AndroWish[10] distribution includes a full set of built-in Android-specific commands, `borg`, `sdltk`, `rfcomm`, `usbserial` for interacting with the Android operating system.

[10] http://www.androwish.org

Appendix B. Utility scripts

We use some simple utility scripts in this book for purposes of pretty-printing etc. These are listed here. Some of these require the `fileutil` package from Tcllib[1] to be loaded.

```
package require fileutil
```

The `print_args` utility simply prints the arguments passed to it, separated by commas.

```
proc print_args {args} {
    puts "Args: [join $args {, }]"
}
```

The `print_list` utility prints each element of a list on a new line. The `print_sorted` utility is similar but prints them in sorted order.

```
proc print_list {l} {
    puts [join $l \n]
}

proc print_sorted {l} {
    print_list [lsort -dictionary $l]
}
```

The `print_dict` utility, transcribed from the Tcllib[2] debug module prints a formatted dictionary.

```
proc print_dict {dict args} {
    if {[llength $args] == 0} {
        set names [lsort -dict [dict keys $dict]]
    } else {
        set names {}
        foreach pattern $args {

            lappend names {*}[lsort -dict [dict keys $dict $pattern]]
        }
    }
    set maxl 0
    foreach name $names {
        if {[string length $name] > $maxl} {
            set maxl [string length $name]
        }
    }
    set maxl [expr {$maxl + 2}]
    set lines {}
    foreach name $names {
        set nameString [format %s $name]
        lappend lines [format "%-*s = %s"  $maxl $nameString  [dict get $dict $name]]
    }
    puts [join $lines \n]
}
```

The `print_array` prints the contents of an array or a subset thereof.

[1] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html
[2] http://core.tcl.tk/tcllib/doc/trunk/embedded/index.html

```
proc print_array {args} {
    uplevel 1 parray $args
}
```

The `print_file` utility dumps the contents of a file while `write_file` writes specified contents to a file.

```
proc print_file {path} {
    fileutil::cat $path
}

proc write_file {path content} {
    fileutil::writeFile $path $content
}
```

The `wait` utility enters the event loop for the specified amount of time.

```
proc wait {ms} {
    after $ms [list set ::_wait_flag 1]
    vwait ::_wait_flag
}
```

The `lambda` command is syntactic sugar for defining an anonymous procedure.

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

The `bin2hex` command pretty prints binary data in hex.

```
proc bin2hex {args} {
    regexp -inline -all .. [binary encode hex [join $args ""]]
}
```