P A R T II

Advanced Tcl

Part II describes advanced programming techniques that support sophisticated applications. The Tcl interfaces remain simple, so you can quickly construct powerful applications.

Chapter 10 describes `eval`, which lets you create Tcl programs on the fly. There are tricks with using `eval` correctly, and a few rules of thumb to make your life easier.

Chapter 11 describes regular expressions. This is the most powerful string processing facility in Tcl. This chapter includes a cookbook of useful regular expressions.

Chapter 12 describes the library and package facility used to organize your code into reusable modules.

Chapter 13 describes introspection and debugging. Introspection provides information about the state of the Tcl interpreter.

Chapter 14 describes namespaces that partition the global scope for variables and procedures. Namespaces help you structure large Tcl applications.

Chapter 15 describes the features that support Internationalization, including Unicode, other character set encodings, and message catalogs.

Chapter 16 describes event-driven I/O programming. This lets you run process pipelines in the background. It is also very useful with network socket programming, which is the topic of Chapter 17.

Chapter 18 describes TclHttpd, a Web server built entirely in Tcl. You can build applications on top of TclHttpd, or integrate the server into existing applications to give them a web interface. TclHttpd also supports regular Web sites.

Chapter 19 describes Safe-Tcl and using multiple Tcl interpreters. You can create multiple Tcl interpreters for your application. If an interpreter is safe, then you can grant it restricted functionality. This is ideal for supporting network applets that are downloaded from untrusted sites.

CHAPTER 10

Quoting Issues and Eval

This chapter describes explicit calls to the interpreter with the `eval` command. An extra round of substitutions is performed that results in some useful effects. The chapter describes the quoting problems with `eval` and the ways to avoid them. The `uplevel` command evaluates commands in a different scope. The `subst` command does substitutions but no command invocation.

Dynamic evaluation makes Tcl flexible and powerful, but it can be tricky to use properly. The basic idea is that you create a string and then use the `eval` command to interpret that string as a command or a series of commands. Creating program code on the fly is easy with an interpreted language like Tcl, and very hard, if not impossible, with a statically compiled language like C++ or Java. There are several ways that dynamic code evaluation is used in Tcl:

- In some cases, a simple procedure isn't quite good enough, and you need to glue together a command from a few different pieces and then execute the result using `eval`. This often occurs with *wrappers*, which provide a thin layer of functionality over existing commands.
- *Callbacks* are script fragments that are saved and evaluated later in response to some event. Examples include the commands associated with Tk buttons, `fileevent` I/O handlers, and `after` timer handlers. Callbacks are a flexible way to link different parts of an application together.
- You can add new control structures to Tcl using the `uplevel` command. For example, you can write a function that applies a command to each line in a file or each node in a tree.
- You can have a mixture of code and data, and just process the code part with the `subst` command. For example, this is useful in HTML templates described in Chapter 18. There are also some powerful combinations of `subst` and `regsub` described in Chapter 11.

Constructing Code with the `list` Command

It can be tricky to assemble a command so that it is evaluated properly by `eval`. The same difficulties apply to commands like `after`, `uplevel`, and the Tk `send` command, all of which have similar properties to `eval`, except that the command evaluation occurs later or in a different context. Constructing commands dynamically is a source of many problems. The worst part is that you can write code that works sometimes but not others, which can be very confusing.

Use `list` when constructing commands.

The root of the quoting problems is the internal use of `concat` by `eval` and similar commands to concatenate their arguments into one command string. The `concat` can lose some important list structure so that arguments are not passed through as you expect. The general strategy to avoid these problems is to use `list` and `lappend` to explicitly form the command callback as a single, well-structured list.

The `eval` Command

The `eval` command results in another call to the Tcl interpreter. If you construct a command dynamically, you must use `eval` to interpret it. For example, suppose we want to construct the following command now but execute it later:

```
puts stdout "Hello, World!"
```

In this case, it is sufficient to do the following:

```
set cmd {puts stdout "Hello, World!"}
=> puts stdout "Hello, World!"
# sometime later...
eval $cmd
=> Hello, World!
```

In this case, the value of `cmd` is passed to Tcl. All the standard grouping and substitution are done again on the value, which is a `puts` command.

However, suppose that part of the command is stored in a variable, but that variable will not be defined at the time `eval` is used. We can artificially create this situation like this:

```
set string "Hello, World!"
set cmd {puts stdout $string}
=> puts stdout $string
unset string
eval $cmd
=> can't read "string": no such variable
```

In this case, the command contains `$string`. When this is processed by `eval`, the interpreter looks for the current value of `string`, which is undefined. This example is contrived, but the same problem occurs if `string` is a local variable, and `cmd` will be evaluated later in the global scope.

A common mistake is to use double quotes to group the command. That will

let $string be substituted now. However, this works only if string has a simple value, but it fails if the value of string contains spaces or other Tcl special characters:

```
set cmd "puts stdout $string"
=> puts stdout Hello, World!
eval $cmd
=> bad argument "World!": should be "nonewline"
```

The problem is that we have lost some important structure. The identity of $string as a single argument gets lost in the second round of parsing by eval. The solution to this problem is to construct the command using list, as shown in the following example:

Example 10–1 Using list to construct commands.

```
set string "Hello, World!"
set cmd [list puts stdout $string]
=> puts stdout {Hello, World!}
unset string
eval $cmd
=> Hello, World!
```

The trick is that list has formed a list containing three elements: puts, stdout, and the value of string. The substitution of $string occurs before list is called, and list takes care of grouping that value for us. In contrast, using double quotes is equivalent to:

```
set cmd [concat puts stdout $string]
```

Double quotes lose list structure.

The problem here is that concat does not preserve list structure. The main lesson is that you should use list to construct commands if they contain variable values or command results that must be substituted now. If you use double quotes, the values are substituted but you lose proper command structure. If you use curly braces, then values are not substituted until later, which may not be in the right context.

Commands That Concatenate Their Arguments

The uplevel, after and send commands concatenate their arguments into a command and execute it later in a different context. The uplevel command is described on page 130, after is described on page 218, and send is described on page 560. Whenever I discover such a command, I put it on my danger list and make sure I explicitly form a single command argument with list instead of letting the command concat items for me. Get in the habit now:

```
after 100 [list doCmd $param1 $param2]
send $interp [list doCmd $param1 $param2];# Safe!
```

The danger here is that concat and list can result in the same thing, so

you can be led down the rosy garden path only to get errors later when values change. The two previous examples always work. The next two work only if `param1` and `param2` have values that are single list elements:

```
after 100 doCmd $param1 $param2
send $interp doCmd $param1 $param2;# Unsafe!
```

If you use other Tcl extensions that provide `eval`-like functionality, carefully check their documentation to see whether they contain commands that `concat` their arguments into a command. For example, Tcl-DP, which provides a network version of `send`, `dp_send`, also uses `concat`.

Commands That Use Callbacks

The general strategy of passing out a command or script to call later is a flexible way to assemble different parts of an application, and it is widely used by Tcl commands. Examples include commands that are called when users click on Tk buttons, commands that are called when I/O channels have data ready, or commands that are called when clients connect to network servers. It is also easy to write your own procedures or C extensions that accept scripts and call them later in response to some event.

These other callback situations may not appear to have the "`concat` problem" because they take a single script argument. However, as soon as you use double quotes to group that argument, you have created the `concat` problem all over again. So, all the caveats about using `list` to construct these commands still apply.

Command Prefix Callbacks

There is a variation on command callbacks called a *command prefix*. In this case, the command is given additional arguments when it is invoked. In other words, you provide only part of the command, the command prefix, and the module that invokes the callback adds additional arguments before using `eval` to invoke the command.

For example, when you create a network server, you supply a procedure that is called when a client makes a connection. That procedure is called with three additional arguments that indicate the client's socket, IP address, and port number. This is described in more detail on page 227. The tricky thing is that you can define your callback procedure to take four (or more) arguments. In this case you specify some of the parameters when you define the callback, and then the socket subsystem specifies the remaining arguments when it makes the callback. The following command creates the server side of a socket:

```
set virtualhost www.beedub.com
socket -server [list Accept $virtualhost] 8080
```

However, you define the `Accept` procedure like this:

```
proc Accept {myname sock ipaddr port} { ... }
```

The `myname` parameter is set when you construct the command prefix. The

remaining parameters are set when the callback is invoked. The use of list in this example is not strictly necessary because "we know" that virtualhost will always be a single list element. However, using list is just a good habit when forming callbacks, so I always write the code this way.

There are many other examples of callback arguments that are really command prefixes. Some of these include the scrolling callbacks between Tk scrollbars and their widgets, the command aliases used with Safe Tcl, the sorting functions in lsort, and the completion callback used with fcopy. Example 13–6 on page 181 shows how to use eval to make callbacks from Tcl procedures.

Constructing Procedures Dynamically

The previous examples have all focused on creating single commands by using list operations. Suppose you want to create a whole procedure dynamically. Unfortunately, this can be particularly awkward because a procedure body is not a simple list. Instead, it is a sequence of commands that are each lists, but they are separated by newlines or semicolons. In turn, some of those commands may be loops and if commands that have their own command bodies. To further compound the problem, you typically have two kinds of variables in the procedure body: some that are to be used as values when constructing the body, and some that are to be used later when executing the procedure. The result can be very messy.

The main trick to this problem is to use either format or regsub to process a template for your dynamically generated procedure. If you use format, then you can put %s into your templates where you want to insert values. You may find the positional notation of the format string (e.g., %1$s and %2$s) useful if you need to repeat a value in several places within your procedure body. The following example is a procedure that generates a new version of other procedures. The new version includes code that counts the number of times the procedure was called and measures the time it takes to run:

Example 10–2 Generating procedures dynamically with a template.

```
proc TraceGen {procName} {
    rename $procName $procName-orig
    set arglist {}
    foreach arg [info args $procName-orig] {
        append arglist "\$$arg "
    }
    proc $procName [info args $procName-orig] [format {
        global _trace_count _trace_msec
        incr _trace_count(%1$s)
        incr _trace_msec(%1$s) [lindex [time {
            set result [%1$s-orig %2$s]
        } 1] 0]
        return $result
    } $procName $arglist]
}
```

Suppose that we have a trivial procedure `foo`:

```
proc foo {x y} {
     return [expr $x * $y]
}
```

If you run `TraceGen` on it and look at the results, you see this:

```
TraceGen foo
info body foo
=>
     global _trace_count _trace_msec
     incr _trace_count(foo)
     incr _trace_msec(foo) [lindex [time {
        set result [foo-orig $x $y]
     } 1] 0]
     return $result
```

Exploiting the `concat` inside `eval`

The previous section warns about the danger of concatenation when forming commands. However, there are times when concatenation is done for good reason. This section illustrates cases where the `concat` done by `eval` is useful in assembling a command by concatenating multiple lists into one list. A `concat` is done internally by `eval` when it gets more than one argument:

```
eval list1 list2 list3 ...
```

The effect of `concat` is to join all the lists into one list; a new level of list structure is *not* added. This is useful if the lists are fragments of a command. It is common to use this form of `eval` with the `args` construct in procedures. Use the `args` parameter to pass optional arguments through to another command. Invoke the other command with `eval`, and the values in `$args` get concatenated onto the command properly. The special args parameter is illustrated in Example 7–2 on page 82.

Using `eval` in a Wrapper Procedure.

Here, we illustrate the use of `eval` and `$args` with a simple Tk example. In Tk, the `button` command creates a button in the user interface. The `button` command can take many arguments, and commonly you simply specify the text of the button and the Tcl command that is executed when the user clicks on the button:

```
button .foo -text Foo -command foo
```

After a button is created, it is made visible by packing it into the display. The `pack` command can also take many arguments to control screen placement. Here, we just specify a side and let the packer take care of the rest of the details:

```
pack .foo -side left
```

Even though there are only two Tcl commands to create a user interface button, we will write a procedure that replaces the two commands with one. Our first version might be:

```
proc PackedButton {name txt cmd} {
    button $name -text $txt -command $cmd
    pack $name -side left
}
```

This is not a very flexible procedure. The main problem is that it hides the full power of the Tk button command, which can really take about 20 widget configuration options, such as -background, -cursor, -relief, and more. They are listed on page 391. For example, you can easily make a red button like this:

```
button .foo -text Foo -command foo -background red
```

A better version of PackedButton uses args to pass through extra configuration options to the button command. The args parameter is a list of all the extra arguments passed to the Tcl procedure. My first attempt to use $args looked like this, but it was not correct:

```
proc PackedButton {name txt cmd args} {
    button $name -text $txt -command $cmd $args
    pack $name -side left
}
PackedButton .foo "Hello, World!" {exit} -background red
=> unknown option "-background red"
```

The problem is that $args is a list value, and button gets the whole list as a single argument. Instead, button needs to get the elements of $args as individual arguments.

Use eval with $args

In this case, you can use eval because it concatenates its arguments to form a single list before evaluation. The single list is, by definition, the same as a single Tcl command, so the button command parses correctly. Here we give eval two lists, which it joins into one command:

```
eval {button $name -text $txt -command $cmd} $args
```

The use of the braces in this command is discussed in more detail below. We also generalize our procedure to take some options to the pack command. This argument, pack, must be a list of packing options. The final version of PackedButton is shown in Example 10–3:

Example 10–3 Using eval with $args.

```
# PackedButton creates and packs a button.
proc PackedButton {path txt cmd {pack {-side right}} args} {
    eval {button $path -text $txt -command $cmd} $args
    eval {pack $path} $pack
}
```

In PackedButton, both pack and args are list-valued parameters that are used as parts of a command. The internal concat done by eval is perfect for this situation. The simplest call to PackedButton is:

```
PackedButton .new "New" { New }
```

The quotes and curly braces are redundant in this case but are retained to convey some type information. The quotes imply a string label, and the braces imply a command. The pack argument takes on its default value, and the args variable is an empty list. The two commands executed by PackedButton are:

```
button .new -text New -command New
pack .new -side right
```

PackedButton creates a horizontal stack of buttons by default. The packing can be controlled with a packing specification:

```
PackedButton .save "Save" { Save $file } {-side left}
```

The two commands executed by PackedButton are:

```
button .new -text Save -command { Save $file }
pack .new -side left
```

The remaining arguments, if any, are passed through to the button command. This lets the caller fine-tune some of the button attributes:

```
PackedButton .quit Quit { Exit } {-side left -padx 5} \
    -background red
```

The two commands executed by PackedButton are:

```
button .quit -text Quit -command { Exit } -background red
pack .quit -side left -padx 5
```

You can see a difference between the pack and args argument in the call to PackedButton. You need to group the packing options explicitly into a single argument. The args parameter is automatically made into a list of all remaining arguments. In fact, if you group the extra button parameters, it will be a mistake:

```
PackedButton .quit Quit { Exit } {-side left -padx 5} \
    {-background red}
=> unknown option "-background red"
```

Correct Quoting with eval

What about the peculiar placement of braces in PackedButton?

```
eval {button $path -text $txt -command $cmd} $args
```

By using braces, we control the number of times different parts of the command are seen by the Tcl evaluator. Without any braces, everything goes through two rounds of substitution. The braces prevent one of those rounds. In the above command, only $args is substituted twice. Before eval is called, the $args is replaced with its list value. Then, eval is invoked, and it concatenates its two list arguments into one list, which is now a properly formed command. The second round of substitutions done by eval replaces the txt and cmd values.

Do not use double quotes with eval.

You may be tempted to use double quotes instead of curly braces in your uses of eval. *Don't give in!* Using double quotes is, mostly likely, wrong. Suppose the first eval command is written like this:

```
eval "button $path -text $txt -command $cmd $args"
```

Incidentally, the previous is equivalent to:

```
eval button $path -text $txt -command $cmd $args
```

These versions happen to work with the following call because txt and cmd have one-word values with no special characters in them:

```
PackedButton .quit Quit { Exit }
```

The button command that is ultimately evaluated is:

```
button .quit -text Quit -command { Exit }
```

In the next call, an error is raised:

```
PackedButton .save "Save As" [list Save $file]
=> unknown option "As"
```

This is because the button command is this:

```
button .save -text Save As -command Save /a/b/c
```

But it should look like this instead:

```
button .save -text {Save As} -command {Save /a/b/c}
```

The problem is that the structure of the button command is now wrong. The value of txt and cmd are substituted first, before eval is even called, and then the whole command is parsed again. The worst part is that sometimes using double quotes works, and sometimes it fails. The success of using double quotes depends on the value of the parameters. When those values contain spaces or special characters, the command gets parsed incorrectly.

Braces: the one true way to group arguments to eval.

To repeat, the safe construct is:

```
eval {button $path -text $txt -command $cmd} $args
```

The following variations are also correct. The first uses list to do quoting automatically, and the others use backslashes or braces to prevent the extra round of substitutions:

```
eval [list button $path -text $txt -command $cmd] $args
eval button \$path -text \$txt -command \$cmd $args
eval button {$path} -text {$txt} -command {$cmd} $args
```

Finally, here is one more *incorrect* approach that tries to quote by hand:

```
eval "button {$path} -text {$txt} -command {$cmd} $args"
```

The problem above is that quoting is not always done with curly braces. If a value contains an unmatched curly brace, Tcl would have used backslashes to quote it, and the above command would raise an error:

```
set blob "foo\{bar space"
=> foo{bar space
eval "puts {$blob}"
=> missing close brace
eval puts {$blob}
=> foo{bar space
```

The uplevel Command

The uplevel command is similar to eval, except that it evaluates a command in a different scope than the current procedure. It is useful for defining new control structures entirely in Tcl. The syntax for uplevel is:

```
uplevel ?level? command ?list1 list2 ...?
```

As with upvar, the *level* parameter is optional and defaults to 1, which means to execute the command in the scope of the calling procedure. The other common use of level is #0, which means to evaluate the command in the global scope. You can count up farther than one (e.g., 2 or 3), or count down from the global level (e.g., #1 or #2), but these cases rarely make sense.

When you specify the *command* argument, you must be aware of any substitutions that might be performed by the Tcl interpreter before uplevel is called. If you are entering the command directly, protect it with curly braces so that substitutions occur in the other scope. The following affects the variable x in the caller's scope:

```
uplevel {set x [expr $x + 1]}
```

However, the following will use the value of x in the current scope to define the value of x in the calling scope, which is probably not what was intended:

```
uplevel "set x [expr $x + 1]"
```

If you are constructing the command dynamically, again use list. This fragment is used later in Example 10–4:

```
uplevel [list foreach $args $valueList {break}]
```

It is common to have the command in a variable. This is the case when the command has been passed into your new control flow procedure as an argument. In this case, you should evaluate the command one level up. Put the level in explicitly to avoid cases where $cmd looks like a number!

```
uplevel 1 $cmd
```

Another common scenario is reading commands from users as part of an application. In this case, you should evaluate the command at the global scope. Example 16–2 on page 220 illustrates this use of uplevel:

```
uplevel #0 $cmd
```

If you are assembling a command from a few different lists, such as the args parameter, then you can use concat to form the command:

```
uplevel [concat $cmd $args]
```

The lists in $cmd and $args are concatenated into a single list, which is a valid Tcl command. Like eval, uplevel uses concat internally if it is given extra arguments, so you can leave out the explicit use of concat. The following commands are equivalent:

```
uplevel [concat $cmd $args]
uplevel "$cmd $args"
uplevel $cmd $args
```

Example 10–4 shows list assignment using the foreach trick described on Page 75. List assignment is useful if a command returns several values in a list.

The lassign procedure assigns the list elements to several variables. The lassign procedure hides the foreach trick, but it must use the uplevel command so that the loop variables get assigned in the correct scope. The list command is used to construct the foreach command that is executed in the caller's scope. This is necessary so that $variables and $values get substituted before the command is evaluated in the other scope.

Example 10–4 lassign: list assignment with foreach.

```
# Assign a set of variables from a list of values.
# If there are more values than variables, they are returned.
# If there are fewer values than variables,
# the variables get the empty string.

proc lassign {valueList args} {
    if {[llength $args] == 0} {
        error "wrong # args: lassign list varname ?varname..?"
    }
    if {[llength $valueList] == 0} {
        # Ensure one trip through the foreach loop
        set valueList [list {}]
    }
    uplevel 1 [list foreach $args $valueList {break}]
    return [lrange $valueList [llength $args] end]
}
```

Example 10–5 illustrates a new control structure with the File_Process procedure that applies a callback to each line in a file. The call to uplevel allows the callback to be concatenated with the line to form the command. The list command is used to quote any special characters in line, so it appears as a single argument to the command.

Example 10–5 The File_Process procedure applies a command to each line of a file.

```
proc File_Process {file callback} {
    set in [open $file]
    while {[gets $file line] >= 0} {
        uplevel 1 $callback [list $line]
    }
    close $in
}
```

What is the difference between these two commands?

```
uplevel 1 [list $callback $line]
uplevel 1 $callback [list $line]
```

The first form limits callback to be the name of the command, while the second form allows callback to be a command prefix. Once again, what is the bug with this version?

```
uplevel 1 $callback $line
```

The arbitrary value of `$line` is concatenated to the `callback` command, and it is likely to be a malformed command when executed.

The subst Command

The `subst` command is useful when you have a mixture of Tcl commands, Tcl variable references, and plain old data. The `subst` command looks through the data for square brackets, dollar signs, and backslashes, and it does substitutions on those. It leaves the rest of the data alone:

```
set a "foo bar"
subst {a=$a date=[exec date]}
=> a=foo bar date=Thu Dec 15 10:13:48 PST 1994
```

The `subst` command does not honor the quoting effect of curly braces. It does substitutions regardless of braces:

```
subst {a=$a date={[exec date]}}
=> a=foo bar date={Thu Dec 15 10:15:31 PST 1994}
```

You can use backslashes to prevent variable and command substitution.

```
subst {a=\$a date=\[exec date]}
=> a=$a date=[exec date]
```

You can use other backslash substitutions like `\uXXXX` to get Unicode characters, `\n` to get newlines, or `\`-newline to hide newlines.

The `subst` command takes flags that limit the substitutions it will perform. The flags are `-nobackslashes`, `-nocommands`, or `-novariables`. You can specify one or more of these flags before the string that needs to be substituted:

```
subst -novariables {a=$a date=[exec date]}
=> a=$a date=Thu Dec 15 10:15:31 PST 1994
```

String Processing with subst

The `subst` command can be used with the `regsub` command to do efficient, two-step string processing. In the first step, `regsub` is used to rewrite an input string into data with embedded Tcl commands. In the second step, `subst` or `eval` replaces the Tcl commands with their result. By artfully mapping the data into Tcl commands, you can dynamically construct a Tcl script that processes the data. The processing is efficient because the Tcl parser and the regular expression processor have been highly tuned. Chapter 11 has several examples that use this technique.

CHAPTER 11

Regular Expressions

This chapter describes regular expression pattern matching and string processing based on regular expression substitutions. These features provide the most powerful string processing facilities in Tcl. Tcl commands described are: `regexp` and `regsub`.

Regular expressions are a formal way to describe string patterns. They provide a powerful and compact way to specify patterns in your data. Even better, there is a very efficient implementation of the regular expression mechanism due to Henry Spencer. If your script does much string processing, it is worth the effort to learn about the `regexp` command. Your Tcl scripts will be compact and efficient. This chapter uses many examples to show you the features of regular expressions.

Regular expression substitution is a mechanism that lets you rewrite a string based on regular expression matching. The `regsub` command is another powerful tool, and this chapter includes several examples that do a lot of work in just a few Tcl commands. Stephen Uhler has shown me several ways to transform input data into a Tcl script with `regsub` and then use `subst` or `eval` to process the data. The idea takes a moment to get used to, but it provides a very efficient way to process strings.

Tcl 8.1 added a new regular expression implementation that supports Unicode and *advanced regular expressions* (ARE). This implementation adds more syntax and escapes that makes it easier to write patterns, once you learn the new features! If you know Perl, then you are already familiar with these features. The Tcl advanced regular expressions are almost identical to the Perl 5 regular expressions. The new features include a few very minor incompatibilities with the regular expressions implemented in earlier versions of Tcl 8.0, but these rarely occur in practice. The new regular expression package supports Unicode, of course, so you can write patterns to match Japanese or Hindu documents!

When to Use Regular Expressions

Regular expressions can seem overly complex at first. They introduce their own syntax and their own rules, and you may be tempted to use simpler commands like `string first`, `string range`, or `string match` to process your strings. However, often a single regular expression command can replace a sequence of several `string` commands. Any time you can replace several Tcl commands with one, you get a performance improvement. Furthermore, the regular expression matcher is implemented in optimized C code, so pattern matching is fast.

The regular expression matcher does more than test for a match. It also tells you what part of your input string matches the pattern. This is useful for picking data out of a large input string. In fact, you can capture several pieces of data in just one match by using subexpressions. The `regexp` Tcl command makes this easy by assigning the matching data to Tcl variables. If you find yourself using `string first` and `string range` to pick out data, remember that `regexp` can do it in one step instead.

The regular expression matcher is structured so that patterns are first compiled into an form that is efficient to match. If you use the same pattern frequently, then the expensive compilation phase is done only once, and all your matching uses the efficient form. These details are completely hidden by the Tcl interface. If you use a pattern twice, Tcl will nearly always be able to retrieve the compiled form of the pattern. As you can see, the regular expression matcher is optimized for lots of heavy-duty string processing.

Avoiding a Common Problem

Group your patterns with curly braces.

One of the stumbling blocks with regular expressions is that they use some of the same special characters as Tcl. Any pattern that contains brackets, dollar signs, or spaces must be quoted when used in a Tcl command. In many cases you can group the regular expression with curly braces, so Tcl pays no attention to it. However, when using Tcl 8.0 (or earlier) you may need Tcl to do backslash substitutions on part of the pattern, and then you need to worry about quoting the special characters in the regular expression.

Advanced regular expressions eliminate this problem because backslash substitution is now done by the regular expression engine. Previously, to get `\n` to mean the newline character (or `\t` for tab) you had to let Tcl do the substitution. With Tcl 8.1, `\n` and `\t` inside a regular expression mean newline and tab. In fact, there are now about 20 backslash escapes you can use in patterns. Now more than ever, remember to group your patterns with curly braces to avoid conflicts between Tcl and the regular expression engine.

The patterns in the first sections of this Chapter ignore this problem. The sample expressions in Table 11–7 on page 151 are quoted for use within Tcl scripts. Most are quoted simply by putting the whole pattern in braces, but some are shown without braces for comparison.

Regular Expression Syntax

This section describes the basics of regular expression patterns, which are found in all versions of Tcl. There are occasional references to features added by advanced regular expressions, but they are covered in more detail starting on page 138. There is enough syntax in regular expressions that there are five tables that summarize all the options. These tables appear together starting at page 145.

A regular expression is a sequence of the following items:

- A literal character.
- A matching character, character set, or character class.
- A repetition quantifier.
- An alternation clause.
- A subpattern grouped with parentheses.

Matching Characters

Most characters simply match themselves. The following pattern matches an `a` followed by a `b`:

```
ab
```

The general wild-card character is the period, "`.`". It matches any single character. The following pattern matches an `a` followed by any character:

```
a.
```

Remember that matches can occur anywhere within a string; a pattern does not have to match the whole string. You can change that by using anchors, which are described on page 137.

Character Sets

The matching character can be restricted to a set of characters with the `[`*xyz*`]` syntax. Any of the characters between the two brackets is allowed to match. For example, the following matches either `Hello` or `hello`:

```
[Hh]ello
```

The matching set can be specified as a range over the character set with the `[`*x*`-`*y*`]` syntax. The following matches any digit:

```
[0-9]
```

There is also the ability to specify the complement of a set. That is, the matching character can be anything except what is in the set. This is achieved with the `[^`*xyz*`]` syntax. Ranges and complements can be combined. The following matches anything except the uppercase and lowercase letters:

```
[^a-zA-Z]
```

Using special characters in character sets.

If you want a `]` in your character set, put it immediately after the initial

opening bracket. You do not need to do anything special to include [in your character set. The following matches any square brackets or curly braces:

```
[][{}]
```

Most regular expression syntax characters are no longer special inside character sets. This means you do not need to backslash anything inside a bracketed character set except for backslash itself. The following pattern matches several of the syntax characters used in regular expressions:

```
[][+*?()|\\]
```

Advanced regular expressions add names and backslash escapes as shorthand for common sets of characters like white space, alpha, alphanumeric, and more. These are described on page 139 and listed in Table 11–3 on page 146.

Quantifiers

Repetition is specified with *, for zero or more, +, for one or more, and ?, for zero or one. These *quantifiers* apply to the previous item, which is either a matching character, a character set, or a subpattern grouped with parentheses. The following matches a string that contains b followed by zero or more a's:

```
ba*
```

You can group part of the pattern with parentheses and then apply a quantifier to that part of the pattern. The following matches a string that has one or more sequences of ab:

```
(ab)+
```

The pattern that matches anything, even the empty string, is:

```
.*
```

These quantifiers have a *greedy* matching behavior: They match as many characters as possible. Advanced regular expressions add nongreedy matching, which is described on page 140. For example, a pattern to match a single line might look like this:

```
.*\n
```

However, as a greedy match, this will match all the lines in the input, ending with the last newline in the input string. The following pattern matches up through the first newline.

```
[^\n]*\n
```

We will shorten this pattern even further on page 140 by using nongreedy quantifiers. There are also special newline sensitive modes you can turn on with some options described on page 143.

Alternation

Alternation lets you test more than one pattern at the same time. The matching engine is designed to be able to test multiple patterns in parallel, so alternation is efficient. Alternation is specified with |, the pipe symbol. Another way to match either Hello or hello is:

```
hello|Hello
```

You can also write this pattern as:

```
(h|H)ello
```

or as:

```
[hH]ello
```


Anchoring a Match

By default a pattern does not have to match the whole string. There can be unmatched characters before and after the match. You can anchor the match to the beginning of the string by starting the pattern with ^, or to the end of the string by ending the pattern with $. You can force the pattern to match the whole string by using both. All strings that begin with spaces or tabs are matched with:

```
^[ \t]+
```

If you have many text lines in your input, you may be tempted to think of ^ as meaning "beginning of line" instead of "beginning of string." By default, the ^ and $ anchors are relative to the whole input, and embedded newlines are ignored. Advanced regular expressions support options that make the ^ and $ anchors line-oriented. They also add the \A and \Z anchors that always match the beginning and end of the string, respectively.

Backslash Quoting

Use the backslash character to turn off these special characters :

```
. * ? + [ ] ( ) ^ $ | \
```

For example, to match the plus character, you will need:

```
\+
```

Remember that this quoting is not necessary inside a bracketed expression (i.e., a character set definition.) For example, to match either plus or question mark, either of these patterns will work:

```
(\+|\?)
[+?]
```

To match a single backslash, you need two. You must do this everywhere, even inside a bracketed expression. Or you can use \B, which was added as part of advanced regular expressions. Both of these match a single backslash:

```
\\
\B
```

Unknown backslash sequences are an error.

Versions of Tcl before 8.1 ignored unknown backslash sequences in regular expressions. For example, \= was just =, and \w was just w. Even \n was just n, which was probably frustrating to many beginners trying to get a newline into their pattern. Advanced regular expressions add backslash sequences for tab, newline, character classes, and more. This is a convenient improvement, but in rare cases it may change the semantics of a pattern. Usually these cases are

where an unneeded backslash suddenly takes on meaning, or causes an error because it is unknown.

Matching Precedence

If a pattern can match several parts of a string, the matcher takes the match that occurs earliest in the input string. Then, if there is more than one match from that same point because of alternation in the pattern, the matcher takes the longest possible match. The rule of thumb is: *first, then longest*. This rule gets changed by nongreedy quantifiers that prefer a shorter match.

Watch out for *, which means zero or more, because zero of anything is pretty easy to match. Suppose your pattern is:

```
[a-z]*
```

This pattern will match against `123abc`, but not how you expect. Instead of matching on the letters in the string, the pattern will match on the zero-length substring at the very beginning of the input string! This behavior can be seen by using the `-indices` option of the `regexp` command described on page 148. This option tells you the location of the matching string instead of the value of the matching string.

Capturing Subpatterns

Use parentheses to capture a subpattern. The string that matches the pattern within parentheses is remembered in a matching variable, which is a Tcl variable that gets assigned the string that matches the pattern. Using parentheses to capture subpatterns is very useful. Suppose we want to get everything between the `<td>` and `</td>` tags in some HTML. You can use this pattern:

```
<td>([^<]*)</td>
```

The matching variable gets assigned the part of the input string that matches the pattern inside the parentheses. You can capture many subpatterns in one match, which makes it a very efficient way to pick apart your data. Matching variables are explained in more detail on page 148 in the context of the `regexp` command.

Sometimes you need to introduce parentheses but you do not care about the match that occurs inside them. The pattern is slightly more efficient if the matcher does not need to remember the match. Advanced regular expressions add noncapturing parentheses with this syntax:

```
(?:pattern)
```

Advanced Regular Expressions

The syntax added by advanced regular expressions is mostly just shorthand notation for constructs you can make with the basic syntax already described. There are also some new features that add additional power: nongreedy quantifi-

ers, back references, look-ahead patterns, and named character classes. If you are just starting out with regular expressions, you can ignore most of this section, except for the one about backslash sequences. Once you master the basics, of if you are already familiar with regular expressions in Tcl (or the UNIX *vi* editor or *grep* utility), then you may be interested in the new features of advanced regular expressions.

Compatibility with Patterns in Tcl 8.0

Advanced regular expressions add syntax in an upward compatible way. Old patterns continue to work with the new matcher, but advanced regular expressions will raise errors if given to old versions of Tcl. For example, the question mark is used in many of the new constructs, and it is artfully placed in locations that would not be legal in older versions of regular expressions. The added syntax is summarized in Table 11–2 on page 145.

If you have unbraced patterns from older code, they are very likely to be correct in Tcl 8.1 and later versions. For example, the following pattern picks out everything up to the next newline. The pattern is unbraced, so Tcl substitutes the newline character for each occurrence of `\n`. The square brackets are quoted so that Tcl does not think they delimit a nested command:

```
regexp "(\[^\n\]+)\n" $input
```

The above command behaves identically when using advanced regular expressions, although you can now also write it like this:

```
regexp {([^\n]+)\n} $input
```

The curly braces hide the brackets from the Tcl parser, so they do not need to be escaped with backslash. This saves us two characters and looks a bit cleaner.

Backslash Escape Sequences

The most significant change in advanced regular expression syntax is backslash substitutions. In Tcl 8.0 and earlier, a backslash is only used to turn off special characters such as: `. + * ? [ ]`. Otherwise it was ignored. For example, `\n` was simply `n` to the Tcl 8.0 regular expression engine. This was a source of confusion, and it meant you could not always quote patterns in braces to hide their special characters from Tcl's parser. In advanced regular expressions, `\n` now means the newline character to the regular expression engine, so you should never need to let Tcl do backslash processing.

Again, *always group your pattern with curly braces* to avoid confusion.

Advanced regular expressions add a lot of new backslash sequences. They are listed in Table 11–4 on page 146. Some of the more useful ones include `\s`, which matches space-like characters, `\w`, which matches letters, digit, and the underscore, `\y`, which matches the beginning or end of a word, and `\B`, which matches a backslash.

Character Classes

Character classes are names for sets of characters. The named character class syntax is valid only inside a bracketed character set. The syntax is

```
[:identifier:]
```

For example, `alpha` is the name for the set of uppercase and lowercase letters. The following two patterns are *almost* the same:

```
[A-Za-z]
[[:alpha:]]
```

The difference is that the alpha character class also includes accented characters like è. If you match data that contains nonASCII characters, the named character classes are more general than trying to name the characters explicitly.

There are also backslash sequences that are shorthand for some of the named character classes. The following patterns to match digits are equivalent:

```
[0-9]
[[:digit:]]
\d
```

The following patterns match space-like characters including backspace, form feed, newline, carriage return, tag, and vertical tab:

```
[ \b\f\n\r\t\v]
[:space:]
\s
```

The named character classes and the associated backslash sequence are listed in Table 11–3 on page 146.

You can use character classes in combination with other characters or character classes inside a character set definition. The following patterns match letters, digits, and underscore:

```
[[:digit:][:alpha:]_]
[\d[:alpha:]_]
[[:alnum:]_]
\w
```

Note that `\d`, `\s` and `\w` can be used either inside or outside character sets. When used outside a bracketed expression, they form their own character set. There are also `\D`, `\S`, and `\W`, which are the complement of `\d`, `\s`, and `\w`. These escapes (i.e., `\D` for not-a-digit) cannot be used inside a bracketed character set.

There are two special character classes, `[[:<:]` and `[[:>:]]`, that match the beginning and end of a word, respectively. A word is defined as one or more characters that match `\w`.

Nongreedy Quantifiers

The *, +, and ? characters are *quantifiers* that specify repetition. By default these match as many characters as possible, which is called *greedy* matching. A *nongreedy* match will match as few characters as possible. You can specify non-

greedy matching by putting a question mark after these quantifiers. Consider the pattern to match "one or more of not-a-newline followed by a newline." The not-a-newline must be explicit with the greedy quantifier, as in:

```
[^\n]+\n
```

Otherwise, if the pattern were just

```
.+\n
```

then the "." could well match newlines, so the pattern would greedily consume everything until the very last newline in the input. A nongreedy match would be satisfied with the very first newline instead:

```
.+?\n
```

By using the nongreedy quantifier we've cut the pattern from eight characters to five. Another example that is shorter with a nongreedy quantifier is the HTML example from page 138. The following pattern also matches everything between <td> and </td>:

```
<td>(.*?)</td>
```

Even ? can be made nongreedy, ??, which means it prefers to match zero instead of one. This only makes sense inside the context of a larger pattern. Send me e-mail if you have a compelling example for it!

Bound Quantifiers

The {m,n} syntax is a quantifier that means match at least m and at most n of the previous matching item. There are two variations on this syntax. A simple {m} means match exactly m of the previous matching item. A {m,} means match m or more of the previous matching item. All of these can be made nongreedy by adding a ? after them.

Back References

A back reference is a feature you cannot easily get with basic regular expressions. A back reference matches the value of a subpattern captured with parentheses. If you have several sets of parentheses you can refer back to different captured expressions with \1, \2, and so on. You count by left parentheses to determine the reference.

For example, suppose you want to match a quoted string, where you can use either single or double quotes. You need to use an alternation of two patterns to match strings that are enclosed in double quotes or in single quotes:

```
("[^"]*"|'[^']*')
```

With a back reference, \1, the pattern becomes simpler:

```
('|").*?\1
```

The first set of parenthesis matches the leading quote, and then the \1 refers back to that particular quote character. The nongreedy quantifier ensures that the pattern matches up to the first occurrence of the matching quote.

Look-ahead

Look-ahead patterns are subexpressions that are matched but do not consume any of the input. They act like constraints on the rest of the pattern, and they typically occur at the end of your pattern. A positive look-ahead causes the pattern to match if it also matches. A negative look-ahead causes the pattern to match if it would not match. These constraints make more sense in the context of matching variables and in regular expression substitutions done with the `regsub` command. For example, the following pattern matches a filename that begins with `A` and ends with `.txt`

```
^A.*\.txt$
```

The next version of the pattern adds parentheses to group the file name suffix.

```
^A.*(\.txt)$
```

The parentheses are not strictly necessary, but they are introduced so that we can compare the pattern to one that uses look-ahead. A version of the pattern that uses look-ahead looks like this:

```
^A.*(?=\.txt)$
```

The pattern with the look-ahead constraint matches only the part of the filename before the `.txt`, but only if the `.txt` is present. In other words, the `.txt` is not consumed by the match. This is visible in the value of the matching variables used with the `regexp` command. It would also affect the substitutions done in the `regsub` command.

There is negative look-ahead too. The following pattern matches a filename that begins with `A` and does not end with `.txt`.

```
^A.*(?!\.txt)$
```

Writing this pattern without negative look-ahead is awkward.

Character Codes

The `\`*nn* and `\`*mmm* syntax, where *n* and *m* are digits, can also mean an 8-bit character code corresponding to the octal value *nn* or *mmm*. This has priority over a back reference. However, I just wouldn't use this notation for character codes. Instead, use the Unicode escape sequence, `\u`*nnnn*, which specifies a 16-bit value. The `\x`*nn* sequence also specifies an 8-bit character code. Unfortunately, the `\x` escape consumes all hex digits after it (not just two!) and then truncates the hexadecimal value down to 8 bits. This misfeature of `\x` is not considered a bug and will probably not change even in future versions of Tcl.

The `\U`*yyyyyyyy* syntax is reserved for 32-bit Unicode, but I don't expect to see that implemented anytime soon.

Collating Elements

Collating elements are characters or long names for characters that you can use inside character sets. Currently, Tcl only has some long names for various

ASCII punctuation characters. Potentially, it could support names for every Unicode character, but it doesn't because the mapping tables would be huge. This section will briefly mention the syntax so that you can understand it if you see it. But its usefulness is still limited.

Within a bracketed expression, the following syntax is used to specify a collating element:

```
[.identifier.]
```

The identifier can be a character or a long name. The supported long names can be found in the `generic/regc_locale.c` file in the Tcl source code distribution. A few examples are shown below:

```
[.c.]
[.#.]
[.number-sign.]
```


Equivalence Classes

An equivalence class is all characters that sort to the same position. This is another feature that has limited usefulness in the current version of Tcl. In Tcl, characters sort by their Unicode character value, so there are no equivalence classes that contain more than one character! However, you could imagine a character class for 'o', 'ò', and other accented versions of the letter o. The syntax for equivalence classes within bracketed expressions is:

```
[=char=]
```

where *char* is any one of the characters in the character class. This syntax is valid only inside a character class definition.

Newline Sensitive Matching

By default, the newline character is just an ordinary character to the matching engine. You can make the newline character special with two options: `lineanchor` and `linestop`. You can set these options with flags to the `regexp` and `regsub` Tcl commands, or you can use the embedded options described later in Table 11–5 on page 147.

The `lineanchor` option makes the `^` and `$` anchors work relative to newlines. The `^` matches immediately after a newline, and `$` matches immediately before a newline. These anchors continue to match the very beginning and end of the input, too. With or without the `lineanchor` option, you can use `\A` and `\Z` to match the beginning and end of the string.

The `linestop` option prevents `.` (i.e., period) and character sets that begin with `^` from matching a newline character. In other words, unless you explicitly include `\n` in your pattern, it will not match across newlines.

Embedded Options

You can start a pattern with embedded options to turn on or off case sensitivity, newline sensitivity, and expanded syntax, which is explained in the next section. You can also switch from advanced regular expressions to a literal string, or to older forms of regular expressions. The syntax is a leading:

```
(?chars)
```

where *chars* is any number of option characters. The option characters are listed in Table 11–5 on page 147.

Expanded Syntax

Expanded syntax lets you include comments and extra white space in your patterns. This can greatly improve the readability of complex patterns. Expanded syntax is turned on with a `regexp` command option or an embedded option.

Comments start with a # and run until the end of line. Extra white space and comments can occur anywhere except inside bracketed expressions (i.e., character sets) or within multicharacter syntax elements like `(?=`. When you are in expanded mode, you can turn off the comment character or include an explicit space by preceding them with a backslash. Example 11–1 shows a pattern to match URLs. The leading `(?x)` turns on expanded syntax. The whole pattern is grouped in curly braces to hide it from Tcl. This example is considered again in more detail in Example 11–3 on page 150:

Example 11–1 Expanded regular expressions allow comments.

```
regexp {(?x)        # A pattern to match URLS
       ([^:]+):     # The protocol before the initial colon
       //([^:/]+)   # The server name
       (:([0-9]+))? # The optional port number
       (/.*)        # The trailing pathname
} $input
```

Syntax Summary

Table 11–1 summarizes the syntax of regular expressions available in all versions of Tcl:

Table 11–1 Basic regular expression syntax.

`.`	Matches any character.
`*`	Matches zero or more instances of the previous pattern item.
`+`	Matches one or more instances of the previous pattern item.
`?`	Matches zero or one instances of the previous pattern item.
`( )`	Groups a subpattern. The repetition and alternation operators apply to the preceding subpattern.
`|`	Alternation.
`[ ]`	Delimit a set of characters. Ranges are specified as [*x-y*]. If the first character in the set is ^, then there is a match if the remaining characters in the set are *not* present.
`^`	Anchor the pattern to the beginning of the string. Only when first.
`$`	Anchor the pattern to the end of the string. Only when last.

Advanced regular expressions, which were introduced in Tcl 8.1, add more syntax that is summarized in Table 11–2:

Table 11–2 Additional advanced regular expression syntax.

{*m*}	Matches *m* instances of the previous pattern item.
{*m*}?	Matches *m* instances of the previous pattern item. Nongreedy.
{m,}	Matches *m* or more instances of the previous pattern item.
{m,}?	Matches *m* or more instances of the previous pattern item. Nongreedy.
{m,n}	Matches *m* through *n* instances of the previous pattern item.
{m,n}?	Matches *m* through *n* instances of the previous pattern item. Nongreedy.
*?	Matches zero or more instances of the previous pattern item. Nongreedy.
+?	Matches one or more instances of the previous pattern item. Nongreedy.
??	Matches zero or one instances of the previous pattern item. Nongreedy.
(?:*re*)	Groups a subpattern, *re*, but does not capture the result.
(?=*re*)	Positive look-ahead. Matches the point where *re* begins.
(?!*re*)	Negative look-ahead. Matches the point where *re* does not begin.
(?*abc*)	Embedded options, where *abc* is any number of option letters listed in Table 11–5.

Table 11–2 Additional advanced regular expression syntax. (Continued)

`\`*c*	One of many backslash escapes listed in Table 11–4.
`[: :]`	Delimits a character class within a bracketed expression. See Table 11–3.
`[. .]`	Delimits a collating element within a bracketed expression.
`[= =]`	Delimits an equivalence class within a bracketed expression.

Table 11–3 lists the named character classes defined in advanced regular expressions and their associated backslash sequences, if any. Character class names are valid inside bracketed character sets with the `[:`*class*`:]` syntax.

Table 11–3 Character classes.

`alnum`	Upper and lower case letters and digits.
`alpha`	Upper and lower case letters.
`blank`	Space and tab.
`cntrl`	Control characters: `\u0001` through `\u001F`.
`digit`	The digits zero through nine. Also `\d`.
`graph`	Printing characters that are not in `cntrl` or `space`.
`lower`	Lowercase letters.
`print`	The same as `alnum`.
`punct`	Punctuation characters.
`space`	Space, newline, carriage return, tab, vertical tab, form feed. Also `\s`.
`upper`	Uppercase letters.
`xdigit`	Hexadecimal digits: zero through nine, a-f, A-F.

Table 11–4 lists backslash sequences supported in Tcl 8.1.

Table 11–4 Backslash escapes in regular expressions.

`\a`	Alert, or "bell", character.
`\A`	Matches only at the beginning of the string.
`\b`	Backspace character, `\u0008`.
`\B`	Synonym for backslash.
`\c`*X*	Control-*X*.
`\d`	Digits. Same as `[[:digit:]]`
`\D`	Not a digit. Same as `[^[:digit:]]`

Table 11–4 Backslash escapes in regular expressions. (Continued)

`\e`	Escape character, `\u001B`.
`\f`	Form feed, `\u000C`.
`\m`	Matches the beginning of a word.
`\M`	Matches the end of a word.
`\n`	Newline, `\u000A`.
`\r`	Carriage return, `\u000D`.
`\s`	Space. Same as `[[:space:]]`
`\S`	Not a space. Same as `[^[:space:]]`
`\t`	Horizontal tab, `\u0009`.
`\u`*XXXX*	A 16-bit Unicode character code.
`\v`	Vertical tab, `\u000B`.
`\w`	Letters, digit, and underscore. Same as `[[:alnum:]_]`
`\W`	Not a letter, digit, or underscore. Same as `[^[:alnum:]_]`
`\x`*hh*	An 8-bit hexadecimal character code. Consumes all hex digits after `\x`.
`\y`	Matches the beginning or end of a word.
`\Y`	Matches a point that is not the beginning or end of a word.
`\Z`	Matches the end of the string.
`\0`	NULL, `\u0000`
`\`*x*	Where *x* is a digit, this is a back-reference.
`\`*xy*	Where *x* and *y* are digits, either a decimal back-reference, or an 8-bit octal character code.
`\`*xyz*	Where *x*, *y* and *z* are digits, either a decimal back-reference or an 8-bit octal character code.

Table 11–5 lists the embedded option characters used with the (`?`*abc*) syntax.

Table 11–5 Embedded option characters used with the (?*x*) syntax.

`b`	The rest of the pattern is a basic regular expression (a la *vi* or *grep*).
`c`	Case sensitive matching. This is the default.
`e`	The rest of the pattern is an extended regular expression (*a la* Tcl 8.0).
`i`	Case insensitive matching.
`m`	Synonym for the `n` option.

Table 11–5 Embedded option characters used with the (?*x*) syntax. (Continued)

n	Newline sensitive matching . Both `lineanchor` and `linestop` mode.
p	Partial newline sensitive matching. Only `linestop` mode.
q	The rest of the pattern is a literal string.
s	No newline sensitivity. This is the default.
t	Tight syntax; no embedded comments. This is the default.
w	Inverse partial newline-sensitive matching. Only `lineanchor` mode.
x	Expanded syntax with embedded white space and comments.

The `regexp` Command

The `regexp` command provides direct access to the regular expression matcher. Not only does it tell you whether a string matches a pattern, it can also extract one or more matching substrings. The return value is 1 if some part of the string matches the pattern; it is 0 otherwise. Its syntax is:

```
regexp ?flags? pattern string ?match sub1 sub2...?
```

The *`flags`* are described in Table 11–6:

Table 11–6 Options to the regexp command.

`-nocase`	Lowercase characters in *`pattern`* can match either lowercase or uppercase letters in *`string`*.
`-indices`	The match variables each contain a pair of numbers that are in indices delimiting the match within *`string`*. Otherwise, the matching string itself is copied into the match variables.
`-expanded`	The pattern uses the expanded syntax discussed on page 144.
`-line`	The same as specifying both `-lineanchor` and `-linestop`.
`-lineanchor`	Change the behavior of ^ and $ so they are line-oriented as discussed on page 143.
`-linestop`	Change matching so that `.` and character classes do not match newlines as discussed on page 143.
`-about`	Useful for debugging. It returns information about the pattern instead of trying to match it against the input.
`--`	Signals the end of the options. You must use this if your pattern begins with `-`.

The *`pattern`* argument is a regular expression as described earlier. If *`string`* matches *`pattern`*, then the results of the match are stored in the variables named in the command. These match variable arguments are optional. If present, *`match`* is set to be the part of the string that matched the pattern. The

remaining variables are set to be the substrings of *string* that matched the corresponding subpatterns in *pattern*. The correspondence is based on the order of left parentheses in the pattern to avoid ambiguities that can arise from nested subpatterns.

Example 11–2 uses regexp to pick the hostname out of the DISPLAY environment variable, which has the form:

```
hostname:display.screen
```

Example 11–2 Using regular expressions to parse a string.

```
set env(DISPLAY) sage:0.1
regexp {([^:]*):} $env(DISPLAY) match host
=> 1
set match
=> sage:
set host
=> sage
```

The pattern involves a complementary set, [^:], to match anything except a colon. It uses repetition, *, to repeat that zero or more times. It groups that part into a subexpression with parentheses. The literal colon ensures that the DISPLAY value matches the format we expect. The part of the string that matches the complete pattern is stored into the match variable. The part that matches the subpattern is stored into host. The whole pattern has been grouped with braces to quote the square brackets. Without braces it would be:

```
regexp (\[^:\]*): $env(DISPLAY) match host
```

With advanced regular expressions the nongreedy quantifier *? can replace the complementary set:

```
regexp (.*?): $env(DISPLAY) match host
```

This is quite a powerful statement, and it is efficient. If we had only had the string command to work with, we would have needed to resort to the following, which takes roughly twice as long to interpret:

```
set i [string first : $env(DISPLAY)]
if {$i >= 0} {
    set host [string range $env(DISPLAY) 0 [expr $i-1]]
}
```

A Pattern to Match URLs

Example 11–3 demonstrates a pattern with several subpatterns that extract the different parts of a URL. There are lots of subpatterns, and you can determine which match variable is associated with which subpattern by counting the left parenthesis. The pattern will be discussed in more detail after the example:

Example 11–3 A pattern to match URLs.

```
set url http://www.beedub.com:80/index.html
regexp {([^:]+)://([^:/]+)(:([0-9]+))?(/.*)} $url \
   match protocol x serverport path
=> 1
set match
=> http://www.beedub.com:80/index.html
set protocol
=> http
set server
=> www.beedub.com
set x
=> :80
set port
=> 80
set path
=> /index.html
```

Let's look at the pattern one piece at a time. The first part looks for the protocol, which is separated by a colon from the rest of the URL. The first part of the pattern is one or more characters that are not a colon, followed by a colon. This matches the `http:` part of the URL:

```
[^:]+:
```

Using nongreedy `+?` quantifier, you could also write that as:

```
.+?:
```

The next part of the pattern looks for the server name, which comes after two slashes. The server name is followed either by a colon and a port number, or by a slash. The pattern uses a complementary set that specifies one or more characters that are *not* a colon or a slash. This matches the `//www.beedub.com` part of the URL:

```
//[^:/]+
```

The port number is optional, so a subpattern is delimited with parentheses and followed by a question mark. An additional set of parentheses are added to capture the port number without the leading colon. This matches the `:80` part of the URL:

```
(:([0-9]+))?
```

The last part of the pattern is everything else, starting with a slash. This matches the `/index.html` part of the URL:

```
/.*
```

Use subpatterns to parse strings.

To make this pattern really useful, we delimit several subpatterns with parentheses:

```
([^:]+)://([^:/]+)(:([0-9]+))?(/.*)
```

These parentheses do not change the way the pattern matches. Only the optional port number really needs the parentheses in this example. However, the `regexp` command gives us access to the strings that match these subpatterns. In

one step regexp can test for a valid URL and divide it into the protocol part, the server, the port, and the trailing path.

The parentheses around the port number include the : before the digits. We've used a dummy variable that gets the : and the port number, and another match variable that just gets the port number. By using noncapturing parentheses in advanced regular expressions, we can eliminate the unused match variable. We can also replace both complementary character sets with a nongreedy .+? match. Example 11–4 shows this variation:

Example 11–4 An advanced regular expression to match URLs.

```
set url http://www.beedub.com:80/book/
regexp {(.+?)://(.+?)(?::([0-9]+))?(/.*)} $url \
   match protocol server port path
=> 1
set match
=> http://www.beedub.com:80/book/
set protocol
=> http
set server
=> www.beedub.com
set port
=> 80
set path
=> /book/
```

Sample Regular Expressions

The table in this section lists regular expressions as you would use them in Tcl commands. Most are quoted with curly braces to turn off the special meaning of square brackets and dollar signs. Other patterns are grouped with double quotes and use backslash quoting because the patterns include backslash sequences like \n and \t. In Tcl 8.0 and earlier, these must be substituted by Tcl before the regexp command is called. In these cases, the equivalent advanced regular expression is also shown.

Table 11–7 Sample regular expressions.

{^[yY]}	Begins with y or Y, as in a Yes answer.
{^(yes|YES|Yes)$}	Exactly "yes", "Yes", or "YES".
"^\[^ \t:\]+:"	Begins with colon-delimited field that has no spaces or tabs.
{^\S+:}	Same as above, using \S for "not space".
"^\[\t]*$"	A string of all spaces or tabs.
{(?n)^\s*$}	A blank line using newline sensitive mode.

Table 11–7 Sample regular expressions. (Continued)

`"(\n|^)\[^\n\]*(\n|$)"`	A blank line, the hard way.
`{^[A-Za-z]+$}`	Only letters.
`{^[[:alpha:]]+$}`	Only letters, the Unicode way.
`{[A-Za-z0-9_]+}`	Letters, digits, and the underscore.
`{\w+}`	Letters, digits, and the underscore using `\w`.
`{[][${}\\]}`	The set of Tcl special characters:] [$ { } \
`"\[^\n\]*\n"`	Everything up to a newline.
`{.*?\n}`	Everything up to a newline using nongreedy `*?`
`{\.}`	A period.
`{[][$^?+*()|\\]}`	The set of regular expression special characters:] [$ ^ ? + * ( ) | \
`<H1>(.*?)</H1>`	An `H1` HTML tag. The subpattern matches the string between the tags.
`<!--.*?-->`	HTML comments.
`{[0-9a-hA-H][0-9a-hA-H]}`	2 hex digits.
`{[[:xdigit:]]{2}}`	2 hex digits, using advanced regular expressions.
`{\d{1,3}}`	1 to 3 digits, using advanced regular expressions.

The `regsub` Command

The `regsub` command does string substitution based on pattern matching. It is very useful for processing your data. It can perform simple tasks like replacing sequences of spaces and tabs with a single space. It can perform complex data transforms, too, as described in the next section. Its syntax is:

```
regsub ?switches? pattern string subspec varname
```

The `regsub` command returns the number of matches and replacements, or 0 if there was no match. `regsub` copies *`string`* to *`varname`*, replacing occurrences of *`pattern`* with the substitution specified by *`subspec`*. If the pattern does not match, then *`string`* is copied to *`varname`* without modification. The optional switches include:

- `-all`, which means to replace all occurrences of the pattern. Otherwise, only the first occurrence is replaced.
- The `-nocase`, `-expanded`, `-line`, `-linestop`, and `-lineanchor` switches are the same as in the `regexp` command. They are described on page 148.
- The `--` switch separates the pattern from the switches, which is necessary if your pattern begins with a `-`.

The replacement pattern, *subspec*, can contain literal characters as well as the following special sequences:

- & is replaced with the string that matched the pattern.
- *x* , where *x* is a number, is replaced with the string that matched the corresponding subpattern in *pattern*. The correspondence is based on the order of left parentheses in the pattern specification.

The following replaces a user's home directory with a ~:

```
regsub ^$env(HOME)/ $pathname ~/ newpath
```

The following constructs a C compile command line given a filename:

```
set file tclIO.c
regsub {([^\.]*)\.c$} $file {cc -c & -o \1.o} ccCmd
```

The matching pattern captures everything before the trailing .c in the file name. The & is replaced with the complete match, tclIO.c, and \1 is replaced with tclIO, which matches the pattern between the parentheses. The value assigned to ccCmd is:

```
cc -c tclIO.c -o tclIO.o
```

We could execute that with:

```
eval exec $ccCmd
```

The following replaces sequences of multiple space characters with a single space:

```
regsub -all {\s+} $string " " string
```

It is perfectly safe to specify the same variable as the input value and the result. Even if there is no match on the pattern, the input string is copied into the output variable.

The regsub command can count things for us. The following command counts the newlines in some text. In this case the substitution is not important:

```
set numLines [regsub -all \n $text {} ignore]
```

Transforming Data to Program with regsub

One of the most powerful combinations of Tcl commands is regsub and subst. This section describes a few examples that use regsub to transform data into Tcl commands, and then use subst to replace those commands with a new version of the data. This technique is very efficient because it relies on two subsystems that are written in highly optimized C code: the regular expression engine and the Tcl parser. These examples are primarily written by Stephen Uhler.

URL Decoding

When a URL is transmitted over the network, it is encoded by replacing special characters with a %*xx* sequence, where *xx* is the hexadecimal code for the character. In addition, spaces are replaced with a plus (+). It would be tedious

and very inefficient to scan a URL one character at a time with Tcl statements to undo this encoding. It would be more efficient to do this with a custom C program, but still very tedious. Instead, a combination of `regsub` and `subst` can efficiently decode the URL in just a few Tcl commands.

Replacing the + with spaces requires quoting the + because it is the one-or-more special character in regular expressions:

```
regsub -all {\+} $url { } url
```

The *%xx* are replaced with a `format` command that will generate the right character:

```
regsub -all {%([0-9a-hA-H][0-9a-hA-H])} $url \
    {[format %c 0x\1]} url
```

The `%c` directive to `format` tells it to generate the character from a character code number. We force a hexadecimal interpretation with a leading `0x`. Advanced regular expressions let us write the "2 hex digits" pattern a bit more cleanly:

```
regsub -all {%([[:xdigit:]]{2})} $url \
    {[format %c 0x\1]} url
```

The resulting string is passed to `subst` to get the `format` commands substituted:

```
set url [subst $url]
```

For example, if the input is `%7ewelch`, the result of the `regsub` is:

```
[format %c 0x7e]welch
```

And then `subst` generates:

```
~welch
```

Example 11–5 encapsulates this trick in the `Url_Decode` procedure.

Example 11–5 The `Url_Decode` procedure.

```
proc Url_Decode {url} {
   regsub -all {\+} $url { } url
   regsub -all {%([:xdigit:]]{2})} $url \
      {[format %c 0x\1]} url
   return [subst $url]
}
```

CGI Argument Parsing

Example 11–6 builds upon `Url_Decode` to decode the inputs to a CGI program that processes data from an HTML form. Each form element is identified by a name, and the value is URL encoded. All the names and encoded values are passed to the CGI program in the following format:

```
name1=value1&name2=value2&name3=value3
```

Example 11–6 shows `Cgi_List` and `Cgi_Query`. `Cgi_Query` receives the form data from the standard input or the `QUERY_STRING` environment variable,

depending on whether the form data is transmitted with a POST or GET request. These HTTP operations are described in detail in Chapter 17. Cgi_List uses split to get back a list of names and values, and then it decodes them with Url_Decode. It returns a Tcl-friendly name, value list that you can either iterate through with a foreach command, or assign to an array with array set:

Example 11–6 The Cgi_Parse and Cgi_Value procedures.

```
proc Cgi_List {} {
    set query [Cgi_Query]
    regsub -all {\+} $query { } query
    set result {}
    foreach {x} [split $query &=] {
        lappend result [Url_Decode $x]
    }
    return $result
}
proc Cgi_Query {} {
    global env
    if {![info exists env(QUERY_STRING)] ||
            [string length $env(QUERY_STRING)] == 0} {
        if {[info exists env(CONTENT_LENGTH)] &&
                [string length $env(CONTENT_LENGTH)] != 0} {
            set query [read stdin $env(CONTENT_LENGTH)]
        } else {
            gets stdin query
        }
        set env(QUERY_STRING) $query
        set env(CONTENT_LENGTH) 0
    }
    return $env(QUERY_STRING)
}
```

An HTML form can have several form elements with the same name, and this can result in more than one value for each name. If you blindly use array set to map the results of Cgi_List into an array, you will lose the repeated values. Example 11–7 shows Cgi_Parse and Cgi_Value that store the query data in a global cgi array. Cgi_Parse adds list structure whenever it finds a repeated form value. The global cgilist array keeps a record of how many times a form value is repeated. The Cgi_Value procedure returns elements of the global cgi array, or the empty string if the requested value is not present.

Example 11–7 Cgi_Parse and Cgi_Value store query data in the cgi array.

```
proc Cgi_Parse {} {
    global cgi cgilist
    catch {unset cgi cgilist}
    set query [Cgi_Query]
    regsub -all {\+} $query { } query
    foreach {name value} [split $query &=] {
        set name [CgiDecode $name]
```

```
        if {[info exists cgilist($name)] &&
               ($cgilist($name) == 1)} {
            # Add second value and create list structure
            set cgi($name) [list $cgi($name) \
               [Url_Decode $value]]
        } elseif {[info exists cgi($name)]} {
            # Add additional list elements
            lappend cgi($name) [CgiDecode $value]
        } else {
            # Add first value without list structure
            set cgi($name) [CgiDecode $value]
            set cgilist($name) 0    ;# May need to listify
        }
        incr cgilist($name)
    }
    return [array names cgi]
}
proc Cgi_Value {key} {
    global cgi
    if {[info exists cgi($key)]} {
        return $cgi($key)
    } else {
        return {}
    }
}
proc Cgi_Length {key} {
    global cgilist
    if {[info exist cgilist($key)]} {
        return $cgilist($key)
    } else {
        return 0
    }
}
```

Decoding HTML Entities

The next example is a decoder for HTML *entities*. In HTML, special characters are encoded as entities. If you want a literal < or > in your document, you encode them as the entities `<` and `>`, respectively, to avoid conflict with the <*tag*> syntax used in HTML. HTML syntax is briefly described in Chapter 3 on page 32. Characters with codes above 127 such as copyright © and egrave è are also encoded. There are named entities, such as `<` for < and `&egrave;` for è. You can also use decimal-valued entities such as `&#169;` for ©. Finally, the trailing semicolon is optional, so `&lt` or `<` can both be used to encode <.

The entity decoder is similar to `Url_Decode`. In this case, however, we need to be more careful with `subst`. The text passed to the decoder could contain special characters like a square bracket or dollar sign. With `Url_Decode` we can rely on those special characters being encoded as, for example, `%24`. Entity encoding is different (do not ask me why URLs and HTML have different encoding standards), and dollar signs and square brackets are not necessarily encoded. This

requires an additional pass to quote these characters. This regsub puts a backslash in front of all the brackets, dollar signs, and backslashes.

```
regsub -all {[][$\\]} $text {\\&} new
```

The decimal encoding (e.g., ©) is also more awkward than the hexadecimal encoding used in URLs. We cannot force a decimal interpretation of a number in Tcl. In particular, if the entity has a leading zero (e.g.,
) then Tcl interprets the value (e.g., 010) as octal. The scan command is used to do a decimal interpretation. It scans into a temporary variable, and set is used to get that value:

```
regsub -all {&#([0-9][0-9]?[0-9]?);?}  $new \
    {[format %c [scan \1 %d tmp; set tmp]]} new
```

With advanced regular expressions, this could be written as follows using bound quantifiers to specify one to three digits:

```
regsub -all {&#(\d{1,3});?}  $new \
    {[format %c [scan \1 %d tmp;set tmp]]} new
```

The named entities are converted with an array that maps from the entity names to the special character. The only detail is that unknown entity names (e.g., &foobar;) are not converted. This mapping is done inside HtmlMapEntity, which guards against invalid entities.

```
regsub -all {&([a-zA-Z]+)(;?)} $new \
    {[HtmlMapEntity \1 \\\2 ]} new
```

If the input text contained:

```
[x &lt; y]
```

then the regsub would transform this into:

```
\[x [HtmlMapEntity lt \; ] y\]
```

Finally, subst will result in:

```
[x < y]
```

Example 11–8 Html_DecodeEntity.

```
proc Html_DecodeEntity {text} {
   if {![regexp & $text]} {return $text}
   regsub -all {[][$\\]} $text {\\&} new
   regsub -all {&#([0-9][0-9]?[0-9]?);?}  $new {\
      [format %c [scan \1 %d tmp;set tmp]]} new
   regsub -all {&([a-zA-Z]+)(;?)} $new \
      {[HtmlMapEntity \1 \\\2 ]} new
   return [subst $new]
}
proc HtmlMapEntity {text {semi {}}} {
   global htmlEntityMap
   if {[info exist htmlEntityMap($text)]} {
      return $htmlEntityMap($text)
   } else {
      return $text$semi
   }
}
```

```
# Some of the htmlEntityMap
array set htmlEntityMap {
    lt <  gt >  amp &
    aring \xe5  atilde \xe3
    copy  \xa9  ecirc  \xea  egrave \xe8
}
```

A Simple HTML Parser

The following example is the brainchild of Stephen Uhler. It uses `regsub` to transform HTML into a Tcl script. When it is evaluated the script calls a procedure to handle each tag in an HTML document. This provides a general framework for processing HTML. Different callback procedures can be applied to the tags to achieve different effects. For example, the `html_library-0.3` package on the CD-ROM uses `Html_Parse` to display HTML in a Tk text widget.

Example 11–9 `Html_Parse`.

```
proc Html_Parse {html cmd {start {}}} {

    # Map braces and backslashes into HTML entities
    regsub -all \{ $html {\&ob;} html
    regsub -all \} $html {\&cb;} html
    regsub -all {\\} $html {\&bsl;} html

    # This pattern matches the parts of an HTML tag
    set s" \t\r\n"   ;# white space
    set exp <(/?)(\[^$s>]+)\[$s]*(\[^>]*)>

    # This generates a call to cmd with HTML tag parts
    # \1 is the leading /, if any
    # \2 is the HTML tag name
    # \3 is the parameters to the tag, if any
    # The curly braces at either end group of all the text
    # after the HTML tag, which becomes the last arg to $cmd.
    set sub "\}\n$cmd {\\2} {\\1} {\\3} \{"
    regsub -all $exp $html $sub html

    # This balances the curly braces,
    # and calls $cmd with $start as a pseudo-tag
    # at the beginning and end of the script.
    eval "$cmd {$start} {} {} {$html}"
    eval "$cmd {$start} / {} {}"
}
```

The main `regsub` pattern can be written more simply with advanced regular expressions:

```
set exp {<(/?)(\S+?)\s*(.*?)>}
```

An example will help visualize the transformation. Given this HTML:

```
<Title>My Home Page</Title>
<Body bgcolor=white text=black>
<H1>My Home</H1>
This is my <b>home</b> page.
```

and a call to Html_Parse that looks like this:

```
Html_Parse $html {Render .text} hmstart
```

then the generated program is this:

```
Render .text {hmstart} {} {} {}
Render .text {Title} {} {} {My Home Page}
Render .text {Title} {/} {} {
}
Render .text {Body} {} {bgcolor=white text=black} {
}
Render .text {H1} {} {} {My Home}
Render .text {H1} {/} {} {
This is my }
Render .text {b} {} {} {home}
Render .text {b} {/} {} { page.
}
Render .text {hmstart} / {} {}
```

One overall point to make about this example is the difference between using eval and subst with the generated script. The decoders shown in Examples 11–5 and 11–8 use subst to selectively replace encoded characters while ignoring the rest of the text. In Html_Parse we must process all the text. The main trick is to replace the matching text (e.g., the HTML tag) with some Tcl code that ends in an open curly brace and starts with a close curly brace. This effectively groups all the unmatched text.

When eval is used this way you must do something with any braces and backslashes in the unmatched text. Otherwise, the resulting script does not parse correctly. In this case, these special characters are encoded as HTML entities. We can afford to do this because the cmd that is called must deal with encoded entities already. It is not possible to quote these special characters with backslashes because all this text is inside curly braces, so no backslash substitution is performed. If you try that the backslashes will be seen by the cmd callback.

Finally, I must admit that I am always surprised that this works:

```
eval "$cmd {$start} {} {} {$html}"
```

I always forget that $start and $html are substituted in spite of the braces. This is because double quotes are being used to group the argument, so the quoting effect of braces is turned off. Try this:

```
set x hmstart
set y "foo {$x} bar"
=> foo {hmstart} bar
```

Stripping HTML Comments

The `Html_Parse` procedure does not correctly handle HTML comments. The problem is that the syntax for HTML commands allows tags inside comments, so there can be > characters inside the comment. HTML comments are also used to hide Javascript inside pages, which can also contain >. We can fix this with a pass that eliminates the comments.

The comment syntax is this:

```
<!-- HTML comment, could contain <markup> -->
```

Using nongreedy quantifiers, we can strip comments with a single `regsub`:

```
regsub -all <!--.*?--> $html {} html
```

Using only greedy quantifiers, it is awkward to match the closing `-->` without getting stuck on embedded > characters, or without matching too much and going all the way to the end of the last comment. Time for another trick:

```
regsub -all --> $html \x81 html
```

This replaces all the end comment sequences with a single character that is not allowed in HTML. Now you can delete the comments like this:

```
regsub -all "<!--\[^\x81\]*\x81" $html {} html
```

Other Commands That Use Regular Expressions

Several Tcl commands use regular expressions.

- `lsearch` takes a `-regexp` flag so that you can search for list items that match a regular expression. The `lsearch` command is described on page 64.
- `switch` takes a `-regexp` flag, so you can branch based on a regular expression match instead of an exact match or a `string match` style match. The `switch` command is described on page 71.
- The Tk text widget can search its contents based on a regular expression match. Searching in the text widget is described on page 461.
- The *Expect* Tcl extension can match the output of a program with regular expressions. *Expect* is the subject of its own book, *Exploring Expect* (O'Reilly, 1995) by Don Libes.

CHAPTER 12

Script Libraries and Packages

Collections of Tcl commands are kept in libraries and organized into packages. Tcl automatically loads libraries as an application uses their commands. Tcl commands discussed are: `package`, `pkg_mkIndex`, `auto_mkindex`, `unknown`, and `tcl_findLibrary`.

Libraries group useful sets of Tcl procedures so that they can be used by multiple applications. For example, you could use any of the code examples that come with this book by creating a script library and then directing your application to check in that library for missing procedures. One way to structure a large application is to have a short main script and a library of support scripts. The advantage of this approach is that not all the Tcl code needs to be loaded to start the application. Applications start up quickly, and as new features are accessed, the code that implements them is loaded automatically.

The Tcl package facility supports version numbers and has a *provide*/*require* model of use. Typically, each file in a library provides one package with a particular version number. Packages also work with shared object libraries that implement Tcl commands in compiled code, which are described in Chapter 44. A package can be provided by a combination of script files and object files. Applications specify which packages they require and the libraries are loaded automatically. The package facility is an alternative to the auto loading scheme used in earlier versions of Tcl. You can use either mechanism, and this chapter describes them both.

If you create a package you may wish to use the namespace facility to avoid conflicts between procedures and global variables used in different packages. Namespaces are the topic of Chapter 14. Before Tcl 8.0 you had to use your own conventions to avoid conflicts. This chapter explains a simple coding convention for large Tcl programs. I use this convention in *exmh*, a mail user interface that

has grown from about 2,000 to over 35,000 lines of Tcl code. A majority of the code has been contributed by the *exmh* user community. Such growth might not have been possible without coding conventions.

Locating Packages: The auto_path Variable

The package facility assumes that Tcl libraries are kept in well-known directories. The list of well-known directories is kept in the auto_path Tcl variable. This is initialized by *tclsh* and *wish* to include the Tcl script library directory, the Tk script library directory (for *wish*), and the parent directory of the Tcl script library directory. For example, on my Macintosh auto_path is a list of these three directories:

```
Disk:System Folder:Extensions:Tool Command Language:tcl8.2
Disk:System Folder:Extensions:Tool Command Language
Disk:System Folder:Extensions:Tool Command Language:tk8.2
```

On my Windows 95 machine the auto_path lists these directories:

```
c:\Program Files\Tcl\lib\Tcl8.2
c:\Program Files\Tcl\lib
c:\Program Files\Tcl\lib\Tk8.2
```

On my UNIX workstation the auto_path lists these directories:

```
/usr/local/tcl/lib/tcl8.2
/usr/local/tcl/lib
/usr/local/tcl/lib/tk8.2
```

The package facility searches these directories and their subdirectories for packages. The easiest way to manage your own packages is to create a directory at the same level as the Tcl library:

```
/usr/local/tcl/lib/welchbook
```

Packages in this location, for example, will be found automatically because the auto_path list includes /usr/local/tcl/lib. You can also add directories to the auto_path explicitly:

```
lappend auto_path directory
```

One trick I often use is to put the directory containing the main script into the auto_path. The following command sets this up:

```
lappend auto_path [file dirname [info script]]
```

If your code is split into bin and lib directories, then scripts in the bin directory can add the adjacent lib directory to their auto_path with this command:

```
lappend auto_path \
    [file join [file dirname [info script]] ../lib]
```

Using Packages

Each script file in a library declares what package it implements with the `package provide` command:

```
package provide name version
```

The *name* identifies the package, and the *version* has a *major.minor* format. The convention is that the minor version number can change and the package implementation will still be compatible. If the package changes in an incompatible way, then the major version number should change. For example, Chapter 17 defines several procedures that use the HTTP network protocol. These include `Http_Open`, `Http_Get`, and `Http_Validate`. The file that contains the procedures starts with this command:

```
package provide Http 1.0
```

Case is significant in package names. In particular, the package that comes with Tcl is named `http` — all lowercase.

More than one file can contribute to the same package simply by specifying the same *name* and *version*. In addition, different versions of the same package can be kept in the same directory but in different files.

An application specifies the packages it needs with the `package require` command:

```
package require name ?version? ?-exact?
```

If the *version* is left off, then the highest available version is loaded. Otherwise the highest version with the same major number is loaded. For example, if the client requires version 1.1, version 1.2 could be loaded if it exists, but versions 1.0 and 2.0 would not be loaded. You can restrict the package to a specific version with the `-exact` flag. If no matching version can be found, then the `package require` command raises an error.

Loading Packages Automatically

The `package require` command depends on an index to record which files implement which packages. The index must be maintained by you, your project librarian, or your system administrator when packages change. The index is computed by the `pkg_mkIndex` command that puts the results into the `pkgIndex.tcl` file in each library directory. The `pkg_mkIndex` command takes the name of a directory and one or more *glob* patterns that specify files within that directory. File name patterns are described on page 115. The syntax is:

```
pkg_mkIndex ?options? directory pattern ?pattern ...?
```

For example:

```
pkg_mkIndex /usr/local/lib/welchbook *.tcl
pkg_mkIndex -direct /usr/local/lib/Sybtcl *.so
```

The `pkg_mkIndex` command sources or loads all the files matched by the pattern, detects what packages they provide, and computes the index. You should be aware of this behavior because it works well only for libraries. If the

pkg_mkIndex command hangs or starts random applications, it is because it sourced an application file instead of a library file.

By default, the index created by pkg_mkIndex contains commands that set up the auto_index array used to automatically load commands when they are first used. This means that code does not get loaded when your script does a package require. If you want the package to be loaded right away, specify the -direct flag to pkg_mkIndex so that it creates an index file with source and load commands. The pkg_mkIndex options are summarized in Table 12–1.

Table 12–1 Options to the pkg_mkIndex command.

-direct	Generates an index with source and load commands in it. This results in packages being loaded directly as a result of package require.
-load *pattern*	Dynamically loads packages that match *pattern* into the slave interpreter used to compute the index. A common reason to need this is with the tcbload package needed to load .tbc files compiled with *TclPro Compiler*.
-verbose	Displays the name of each file processed and any errors that occur.

Packages Implemented in C Code

The files in a library can be either script files that define Tcl procedures or binary files in shared library format that define Tcl commands in compiled code (i.e., a Dynamic Link Library (DLL)). Chapter 44 describes how to implement Tcl commands in C. There is a C API to the package facility that you use to declare the package name for your commands. This is shown in Example 44–1 on page 608. Chapter 37 also describes the Tcl load command that is used instead of source to link in shared libraries. The pkg_mkIndex command also handles shared libraries:

```
pkg_mkIndex directory *.tcl *.so *.shlib *.dll
```

In this example, .so, .shlib, and .dll are file suffixes for shared libraries on UNIX, Macintosh, and Windows systems, respectively. You can have packages that have some of their commands implemented in C, and some implemented as Tcl procedures. The script files and the shared library must simply declare that they implement the same package. The pkg_mkIndex procedure will detect this and set up the auto_index, so some commands are defined by sourcing scripts, and some are defined by loading shared libraries.

If your file servers support more than one machine architecture, such as Solaris and Linux systems, you probably keep the shared library files in machine-specific directories. In this case the auto_path should also list the machine-specific directory so that the shared libraries there can be loaded automatically. If your system administrator configured the Tcl installation properly, this should already be set up. If not, or you have your shared libraries in a non-standard place, you must append the location to the auto_path variable.

Summary of Package Loading

The basic structure of package loading works like this:

- An application does a `package require` command. If the package is already loaded, the command just returns the version number of the already loaded package. If is not loaded, the following steps occur.
- The package facility checks to see if it knows about the package. If it does, then it runs the Tcl scripts registered with the `package ifneeded` command. These commands either load the package or set it up to be loaded automatically when its commands are first used.
- If the package is unknown, the `tclPkgUnknown` procedure is called to find it. Actually, you can specify what procedure to call to do the lookup with the `package unknown` command, but the standard one is `tclPkgUnknown`.
- The `tclPkgUnknown` procedure looks through the `auto_path` directories and their subdirectories for `pkgIndex.tcl` files. It sources those to build an internal database of packages and version information. The `pkgIndex.tcl` files contain calls to `package ifneeded` that specify what to do to define the package. The standard action is to call the `tclPkgSetup` procedure that sets up the `auto_index` so that the commands in the package will be automatically loaded. If you use `-direct` with `pkg_mkIndex`, the script contains `source` and `load` commands instead.
- The `tclPkgSetup` procedure defines the `auto_index` array to contain the correct `source` or `load` commands to define each command in the package. Automatic loading and the `auto_index` array are described in more detail later.

As you can see, there are several levels of processing involved in finding packages. The system is flexible enough that you can change the way packages are located and how packages are loaded. The default scenario is complicated because it uses the delayed loading of source code that is described in the next section. Using the `-direct` flag to `pkg_mkIndex` simplifies the situation. In any case, it all boils down to three key steps:

- Use `pkg_mkIndex` to maintain your index files. Decide at this time whether or not to use direct package loading.
- Put the appropriate `package require` and `package provide` commands in your code.
- Ensure that your library directories, or their parent directories, are listed in the `auto_path` variable.

The package Command

The `package` command has several operations that are used primarily by the `pkg_mkIndex` procedure and the automatic loading facility. These operations are summarized in Table 12–2.

Table 12–2 The `package` command.

`package forget` *package*	Deletes registration information for package.
`package ifneeded` *package* ?*command*?	Queries or sets the command used to set up automatic loading of a package.
`package names`	Returns the set of registered packages.
`package provide` *package version*	Declares that a script file defines commands for *package* with the given *version*.
`package require` *package* ?*version*? ?-exact?	Declares that a script uses *package*. The `-exact` flag specifies that the exact `version` must be loaded. Otherwise, the highest matching version is loaded.
`package unknown` ?*command*?	Queries or sets the *command* used to locate packages.
`package vcompare` *v1 v2*	Compares version *v1* and *v2*. Returns 0 if they are equal, -1 if *v1* is less than *v2*, or 1 if *v1* is greater than *v2*.
`package versions` *package*	Returns which versions of the package are registered.
`package vsatisfies` *v1 v2*	Returns 1 if *v1* is greater or equal to *v2* and still has the same major version number. Otherwise returns 0.

Libraries Based on the `tclIndex` File

You can create libraries without using the `package` command. The basic idea is that a directory has a library of script files, and an index of the Tcl commands defined in the library is kept in a `tclIndex` file. The drawback is that versions are not supported and you may need to adjust the `auto_path` to list your library directory. The main advantage of this approach is that this mechanism has been part of Tcl since the earliest versions. If you currently maintain a library using `tclIndex` files, it will still work.

You must generate the index that records what procedures are defined in the library. The `auto_mkindex` procedure creates the index, which is stored in a file named `tclIndex` that is kept in the script library directory. (Watch out for the difference in capitalization between `auto_mk`**`i`**`ndex` and `pkg_mk`**`I`**`ndex`!) Suppose all the examples from this book are in the directory `/usr/local/tcl/welchbook`. You can make the examples into a script library by creating the `tclIndex` file:

```
auto_mkindex /usr/local/tcl/welchbook *.tcl
```

You will need to update the `tclIndex` file if you add procedures or change any of their names. A conservative approach to this is shown in the next example. It is conservative because it re-creates the index if anything in the library has changed since the `tclIndex` file was last generated, whether or not the change added or removed a Tcl procedure.

Example 12–1 Maintaining a `tclIndex` file.

```
proc Library_UpdateIndex { libdir } {
```

```
    set index [file join $libdir tclIndex]
    if {![file exists $index]} {
        set doit 1
    } else {
        set age [file mtime $index]
        set doit 0
        # Changes to directory may mean files were deleted
        if {[file mtime $libdir] > $age} {
            set doit 1
        } else {
            # Check each file for modification
            foreach file [glob [file join $libdir *.tcl]] {
                if {[file mtime $file] > $age} {
                    set doit 1
                    break
                }
            }
        }
    }
    if { $doit } {
        auto_mkindex $libdir *.tcl
    }
}
```

Tcl uses the `auto_path` variable to record a list of directories to search for unknown commands. To continue our example, you can make the procedures in the book examples available by putting this command at the beginning of your scripts:

```
lappend auto_path /usr/local/tcl/welchbook
```

This has no effect if you have not created the `tclIndex` file. If you want to be extra careful, you can call `Library_UpdateIndex`. This will update the index if you add new things to the library.

```
lappend auto_path /usr/local/tcl/welchbook
Library_UpdateIndex /usr/local/tcl/welchbook
```

This will not work if there is no `tclIndex` file at all because Tcl won't be able to find the implementation of `Library_UpdateIndex`. Once the `tclIndex` has been created for the first time, then this will ensure that any new procedures added to the library will be installed into `tclIndex`. In practice, if you want this sort of automatic update, it is wise to include something like the `Library_UpdateIndex` procedure directly into your application as opposed to loading it from the library it is supposed to be maintaining.

The unknown Command

Automatic loading of Tcl commands is implemented by the `unknown` command. Whenever the Tcl interpreter encounters a command that it does not know about, it calls the `unknown` command with the name of the missing command. The `unknown` command is implemented in Tcl, so you are free to provide your own

mechanism to handle unknown commands. This chapter describes the behavior of the default implementation of unknown, which can be found in the init.tcl file in the Tcl library. The location of the library is returned by the info library command.

How Auto Loading Works

The unknown command uses an array named auto_index. One element of the array is defined for each procedure that can be automatically loaded. The auto_index array is initialized by the package mechanism or by tclIndex files. The value of an auto_index element is a command that defines the procedure. Typical commands are:

```
source [file join $dir bind_ui.tcl]
load [file join $dir mime.so] Mime
```

The $dir gets substituted with the name of the directory that contains the library file, so the result is a source or load command that defines the missing Tcl command. The substitution is done with eval, so you could initialize auto_index with any commands at all. Example 12–2 is a simplified version of the code that reads the tclIndex file.

Example 12–2 Loading a tclIndex file.

```
# This is a simplified part of the auto_load_index procedure.
# Go through auto_path from back to front.
set i [expr [llength $auto_path]-1]
for {} {$i >= 0} {incr i -1} {
    set dir [lindex $auto_path $i]
    if [catch {open [file join $dir tclIndex]} f] {
        # No index
        continue
    }
    # eval the file as a script. Because eval is
    # used instead of source, an extra round of
    # substitutions is performed and $dir gets expanded
    # The real code checks for errors here.
    eval [read $f]
    close $f
}
```

Disabling the Library Facility: auto_noload

If you do not want the unknown procedure to try and load procedures, you can set the auto_noload variable to disable the mechanism:

```
set auto_noload anything
```

Auto loading is quite fast. I use it regularly on applications both large and small. A large application will start faster if you only need to load the code necessary to start it up. As you access more features of your application, the code will load

automatically. Even a small application benefits from auto loading because it encourages you to keep commonly used code in procedure libraries.

Interactive Conveniences

The unknown command provides a few other conveniences. These are used only when you are typing commands directly. They are disabled once execution enters a procedure or if the Tcl shell is not being used interactively. The convenience features are automatic execution of programs, command history, and command abbreviation. These options are tried, in order, if a command implementation cannot be loaded from a script library.

Auto Execute

The unknown procedure implements a second feature: automatic execution of external programs. This makes a Tcl shell behave more like other UNIX shells that are used to execute programs. The search for external programs is done using the standard PATH environment variable that is used by other shells to find programs. If you want to disable the feature all together, set the auto_noexec variable:

```
set auto_noexec anything
```

History

The history facility described in Chapter 13 is implemented by the unknown procedure.

Abbreviations

If you type a unique prefix of a command, unknown recognizes it and executes the matching command for you. This is done after automatic program execution is attempted and history substitutions are performed.

Tcl Shell Library Environment

Tcl searches for its script library directory when it starts up. In early versions of Tcl you had to compile in the correct location, set a Windows registry value, or set the TCL_LIBRARY environment variable to the correct location. Recent versions of Tcl use a standard searching scheme to locate the script library. The search understands the standard installation and build environments for Tcl, and it should eliminate the need to use the TCL_LIBRARY environment variable. On Windows the search for the library used to depend on registry values, but this has also been discontinued in favor of a standard search. In summary, "it should just work." However, this section explains how Tcl finds its script library

so that you can troubleshoot problems.

Locating the Tcl Script Library

The default library location is defined when you configure the source distribution, which is explained on page 642. At this time an initial value for the auto_path variable is defined. (This default value appears in tcl_pkgPath, but changing this variable has no effect once Tcl has started. I just pretend tcl_pkgPath does not exist.) These values are just hints; Tcl may use other directories depending on what it finds in the file system.

When Tcl starts up, it searches for a directory that contains its init.tcl startup script. You can short-circuit the search by defining the TCL_LIBRARY environment variable. If this is defined, Tcl uses it only for its script library directory. However, you should not need to define this with normal installations of Tcl 8.0.5 or later. In my environment I'm often using several different versions of Tcl for various applications and testing purposes, so setting TCL_LIBRARY is never correct for all possibilities. If I find myself setting this environment variable, I know something is wrong with my Tcl installations!

The standard search starts with the default value that is compiled into Tcl (e.g., /usr/local/lib/tcl8.1.) After that, the following directories are examined for an init.tcl file. These example values assume Tcl version 8.1 and patch level 8.1.1:

```
../lib/tcl8.1
../../lib/tcl8.1
../library
../../tcl8.1.1/library
../../../tcl8.1.1/library
```

The first two directories correspond to the standard installation directories, while the last three correspond to the standard build environment for Tcl or Tk. The first directory in the list that contains a valid init.tcl file becomes the Tcl script library. This directory location is saved in the tcl_library global variable, and it is also returned by the info library command.

The primary thing defined by init.tcl is the implementation of the unknown procedure. It also initializes auto_path to contain $tcl_library and the parent directory of $tcl_library. There may be additional directories added to auto_path depending on the compiled in value of tcl_pkgPath.

tcl_findLibrary

A generalization of this search is implemented by tcl_findLibrary. This procedure is designed for use by extensions like Tk and [incr Tcl]. Of course, Tcl cannot use tcl_findLibrary itself because it is defined in init.tcl!

The tcl_findLibrary procedure searches relative to the location of the main program (e.g., *tclsh* or *wish)* and assumes a standard installation or a standard build environment. It also supports an override by an environment vari-

able, and it takes care of sourcing an initialization script. The usage of `tcl_findLibrary` is:

```
tcl_findLibrary base version patch script enVar varName
```

The *base* is the prefix of the script library directory name. The *version* is the main version number (e.g., "8.0"). The *patch* is the full patch level (e.g., "8.0.3"). The *script* is the initialization script to source from the directory. The *enVar* names an environment variable that can be used to override the default search path. The *varName* is the name of a variable to set to name of the directory found by `tcl_findLibrary`. A side effect of `tcl_findLibrary` is to source the script from the directory. An example call is:

```
tcl_findLibrary tk 8.0 8.0.3 tk.tcl TK_LIBRARY tk_library
```

This call first checks to see whether `TK_LIBRARY` is defined in the environment. If so, it uses its value. Otherwise, it searches the following directories for a file named `tk.tcl`. It sources the script and sets the `tk_library` variable to the directory containing that file. The search is relative to the value returned by `info nameofexecutable`:

```
../lib/tk8.0
../../lib/tk8.0
../library
../../tk8.0.3/library
../../../tk8.0.3/library
```

Tk also adds `$tk_library` to the end of `auto_path`, so the other script files in that directory are available to the application:

```
lappend auto_path $tk_library
```

Coding Style

If you supply a package, you need to follow some simple coding conventions to make your library easier to use by other programmers. You can use the namespace facility introduced in Tcl 8.0. You can also use conventions to avoid name conflicts with other library packages and the main application. This section describes the conventions I developed before namespaces were added to Tcl.

A Module Prefix for Procedure Names

The first convention is to choose an identifying prefix for the procedures in your package. For example, the preferences package in Chapter 42 uses `Pref` as its prefix. All the procedures provided by the library begin with `Pref`. This convention is extended to distinguish between private and exported procedures. An exported procedure has an underscore after its prefix, and it is acceptable to call this procedure from the main application or other library packages. Examples include `Pref_Add`, `Pref_Init`, and `Pref_Dialog`. A private procedure is meant for use only by the other procedures in the same package. Its name does not have the underscore. Examples include `PrefDialogItem` and `PrefXres`.

This naming convention precludes casual names like `doit`, `setup`, `layout`, and so on. Without using namespaces, there is no way to hide procedure names, so you must maintain the naming convention for all procedures in a package.

A Global Array for State Variables

You should use the same prefix on the global variables used by your package. You can alter the capitalization; just keep the same prefix. I capitalize procedure names and use lowercase letters for variables. By sticking with the same prefix you identify what variables belong to the package and you avoid conflict with other packages.

Collect state in a global array.

In general, I try to use a single global array for a package. The array provides a convenient place to collect a set of related variables, much as a struct is used in C. For example, the preferences package uses the `pref` array to hold all its state information. It is also a good idea to keep the use of the array private. It is better coding practice to provide exported procedures than to let other modules access your data structures directly. This makes it easier to change the implementation of your package without affecting its clients.

If you do need to export a few key variables from your module, use the underscore convention to distinguish exported variables. If you need more than one global variable, just stick with the prefix convention to avoid conflicts.

The Official Tcl Style Guide

John Ousterhout has published two programming style guides, one for C programming known as *The Engineering Manual* and one for Tcl scripts known as *The Style Guide*. These describe details about file structure as well as naming conventions for modules, procedures, and variables. The *Tcl Style Guide* conventions use Tcl namespaces to separate packages. Namespaces automatically provide a way to avoid conflict between procedure names. Namespaces also support collections of variables without having to use arrays for grouping.

You can find these style guides on the CD-ROM and also in `ftp://ftp.scriptics.com/pub/tcl/doc`. *The Engineering Manual* is distributed as a compressed tar file, `engManual.tar.Z`, that contains sample files as well as the main document. *The Style Guide* is distributed as `styleGuide.ps` (or `.pdf`).

CHAPTER 13

Reflection and Debugging

This chapter describes commands that give you a view into the interpreter. The `history` command and a simple debugger are useful during development and debugging. The `info` command provides a variety of information about the internal state of the Tcl interpreter. The `time` command measures the time it takes to execute a command. Tcl commands discussed are: `clock`, `info`, `history`, and `time`.

Reflection provides feedback to a script about the internal state of the interpreter. This is useful in a variety of cases, from testing to see whether a variable exists to dumping the state of the interpreter. The `info` command provides lots of different information about the interpreter.

The `clock` command is useful for formatting dates as well as parsing date and time values. It also provides high-resolution timer information for precise measurements.

Interactive command history is the third topic of the chapter. The history facility can save you some typing if you spend a lot of time entering commands interactively.

Debugging is the last topic. The old-fashioned approach of adding `puts` commands to your code is often quite useful. For tough problems, however, a real debugger is invaluable. The *TclPro* tools from Scriptics include a high quality debugger and static code checker. The *tkinspect* program is an inspector that lets you look into the state of a Tk application. It can hook up to any Tk application dynamically, so it proves quite useful.

The `clock` Command

The `clock` command has facilities for getting the current time, formatting time values, and scanning printed time strings to get an integer time value. The `clock` command was added in Tcl 7.5. Table 13–1 summarizes the `clock` command:

Table 13–1 The `clock` command.

`clock clicks`	A system-dependent high resolution counter.
`clock format` *`value`* `?-format` *`str`*`?`	Formats a clock value according to *`str`*.
`clock scan` *`string`* `?-base` *`clock`*`? ?-gmt` *`boolean`*`?`	Parses date *`string`* and return seconds value. The *`clock`* value determines the date.
`clock seconds`	Returns the current time in seconds.

The following command prints the current time:

```
clock format [clock seconds]
=> Sun Nov 24 14:57:04  1996
```

The `clock seconds` command returns the current time, in seconds since a starting epoch. The `clock format` command formats an integer value into a date string. It takes an optional argument that controls the format. The format strings contains % keywords that are replaced with the year, month, day, date, hours, minutes, and seconds, in various formats. The default string is:

```
%a %b %d %H:%M:%S %Z %Y
```

Tables 13–2 and 13–3 summarize the `clock` formatting strings:

Table 13–2 Clock formatting keywords.

`%%`	Inserts a %.
`%a`	Abbreviated weekday name (Mon, Tue, etc.).
`%A`	Full weekday name (Monday, Tuesday, etc.).
`%b`	Abbreviated month name (Jan, Feb, etc.).
`%B`	Full month name.
`%c`	Locale specific date and time (e.g., `Nov 24 16:00:59 1996`).
`%d`	Day of month (01 – 31).
`%H`	Hour in 24-hour format (00 – 23).
`%I`	Hour in 12-hour format (01 – 12).
`%j`	Day of year (001 – 366).
`%m`	Month number (01 – 12).
`%M`	Minute (00 – 59).
`%p`	AM/PM indicator.
`%S`	Seconds (00 – 59).
`%U`	Week of year (00 – 52) when Sunday starts the week.
`%w`	Weekday number (Sunday = 0).

Table 13–2 Clock formatting keywords. (Continued)

%W	Week of year (01 – 52) when Monday starts the week.
%x	Locale specific date format (e.g., Feb 19 1997).
%X	Locale specific time format (e.g., 20:10:13).
%y	Year without century (00 – 99).
%Y	Year with century (e.g. 1997).
%Z	Time zone name.

Table 13–3 UNIX-specific clock formatting keywords.

%D	Date as %m/%d/%y (e.g., 02/19/97).
%e	Day of month (1 – 31), no leading zeros.
%h	Abbreviated month name.
%n	Inserts a newline.
%r	Time as %I:%M:%S %p (e.g., 02:39:29 PM).
%R	Time as %H:%M (e.g., 14:39).
%t	Inserts a tab.
%T	Time as %H:%M:%S (e.g., 14:34:29).

The clock clicks command returns the value of the system's highest resolution clock. The units of the clicks are not defined. The main use of this command is to measure the relative time of different performance tuning trials. The following command counts the clicks per second over 10 seconds, which will vary from system to system:

Example 13–1 Calculating clicks per second.

```
set t1 [clock clicks]
after 10000 ;# See page 218
set t2 [clock clicks]
puts "[expr ($t2 - $t1)/10] Clicks/second"
=> 1001313 Clicks/second
```

The clock scan command parses a date string and returns a seconds value. The command handles a variety of date formats. If you leave off the year, the current year is assumed.

Year 2000 Compliance

Tcl implements the standard interpretation of two-digit year values, which is that 70–99 are 1970–1999, 00–69 are 2000–2069. Versions of Tcl before 8.0 did not properly deal with two-digit years in all cases. Note, however, that Tcl is lim-

ited by your system's time epoch and the number of bits in an integer. On Windows, Macintosh, and most UNIX systems, the clock epoch is January 1, 1970. A 32-bit integer can count enough seconds to reach forward into the year 2037, and backward to the year 1903. If you try to `clock scan` a date outside that range, Tcl will raise an error because the seconds counter will overflow or underflow. In this case, Tcl is just reflecting limitations of the underlying system.

If you leave out a date, `clock scan` assumes the current date. You can also use the `-base` option to specify a date. The following example uses the current time as the base, which is redundant:

```
clock scan "10:30:44 PM" -base [clock seconds]
=> 2931690644
```

The date parser allows these modifiers: `year`, `month`, `fortnight` (two weeks), `week`, `day`, `hour`, `minute`, `second`. You can put a positive or negative number in front of a modifier as a multiplier. For example:

```
clock format [clock scan "10:30:44 PM 1 week"]
=> Sun Dec 01 22:30:44  1996
clock format [clock scan "10:30:44 PM -1 week"]
Sun Nov 17 22:30:44  1996
```

You can also use `tomorrow`, `yesterday`, `today`, `now`, `last`, `this`, `next`, and `ago`, as modifiers.

```
clock format [clock scan "3 years ago"]
=> Wed Nov 24 17:06:46  1993
```

Both `clock format` and `clock scan` take a `-gmt` option that uses Greenwich Mean Time. Otherwise, the local time zone is used.

```
clock format [clock seconds] -gmt true
=> Sun Nov 24 09:25:29  1996
clock format [clock seconds] -gmt false
=> Sun Nov 24 17:25:34  1996
```

The `info` Command

Table 13–4 summarizes the `info` command. The operations are described in more detail later.

Table 13–4 The `info` command.

`info args` *`procedure`*	A list of *`procedure`*'s arguments.
`info body` *`procedure`*	The commands in the body of *`procedure`*.
`info cmdcount`	The number of commands executed so far.
`info commands ?`*`pattern`*`?`	A list of all commands, or those matching *`pattern`*. Includes built-ins and Tcl procedures.
`info complete` *`string`*	True if *`string`* contains a complete Tcl command.

Table 13–4 The info command. (Continued)

`info default` *`proc arg var`*	True if *`arg`* has a default parameter value in procedure *`proc`*. The default value is stored into *`var`*.
`info exists` *`variable`*	True if *`variable`* is defined.
`info globals ?`*`pattern`*`?`	A list of all global variables, or those matching *`pattern`*.
`info hostname`	The name of the machine. This may be the empty string if networking is not initialized.
`info level`	The stack level of the current procedure, or 0 for the global scope.
`info level` *`number`*	A list of the command and its arguments at the specified level of the stack.
`info library`	The pathname of the Tcl library directory.
`info loaded ?`*`interp`*`?`	A list of the libraries loaded into the interpreter named *`interp`*, which defaults to the current one.
`info locals ?`*`pattern`*`?`	A list of all local variables, or those matching *`pattern`*.
`info nameofexecutable`	The file name of the program (e.g., of *tclsh* or *wish*).
`info patchlevel`	The release patch level for Tcl.
`info procs ?`*`pattern`*`?`	A list of all Tcl procedures, or those that match *`pattern`*.
`info script`	The name of the file being processed, or the empty string.
`info sharedlibextension`	The file name suffix of shared libraries.
`info tclversion`	The version number of Tcl.
`info vars ?`*`pattern`*`?`	A list of all visible variables, or those matching *`pattern`*.

Variables

There are three categories of variables: *local*, *global*, and *visible*. Information about these categories is returned by the `locals`, `globals`, and `vars` operations, respectively. The local variables include procedure arguments as well as locally defined variables. The global variables include all variables defined at the global scope. The visible variables include locals, plus any variables made visible via `global` or `upvar` commands. A pattern can be specified to limit the returned list of variables to those that match the pattern. The pattern is interpreted according to the rules of `string match`, which is described on page 48:

```
info globals auto*
=> auto_index auto_noexec auto_path
```

Namespaces, which are the topic of the next chapter, partition global variables into different scopes. You query the variables visible in a namespace with:

```
info vars namespace::*
```

Remember that a variable may not be defined yet even though a `global` or `upvar` command has declared it visible in the current scope. Use the `info exists` command to test whether a variable or an array element is defined or not. An example is shown on page 59.

Procedures

You can find out everything about a Tcl procedure with the `args`, `body`, and `default` operations. This is illustrated in the following `Proc_Show` example. The `puts` commands use the `-nonewline` flag because the newlines in the procedure body, if any, are retained:

Example 13–2 Printing a procedure definition.

```
proc Proc_Show {{namepat *} {file stdout}} {
   foreach proc [info procs $namepat] {
      set space ""
      puts -nonewline $file "proc $proc {"
      foreach arg [info args $proc] {
         if [info default $proc $arg value] {
            puts -nonewline $file "$space{$arg $value}"
         } else {
            puts -nonewline $file $space$arg
         }
         set space " "
      }

      # No newline needed because info body may return a
      # value that starts with a newline

      puts -nonewline $file "} {"
      puts -nonewline $file [info body $proc]
      puts $file "}"
   }
}
```

Example 13–3 is a more elaborate example of procedure introspection that comes from the `direct.tcl` file, which is part of the Tcl Web Server described in Chapter 18. This code is used to map URL requests and the associated query data directly into Tcl procedure calls. This is discussed in more detail on page 247. The Web server collects Web form data into an array called `form`. Example 13–3 matches up elements of the `form` array with procedure arguments, and it collects extra elements into an `args` parameter. If a form value is missing, then the default argument value or the empty string is used:

Example 13–3 Mapping form data onto procedure arguments.

```
# cmd is the name of the procedure to invoke
# form is an array containing form values

set cmdOrig $cmd
set params [info args $cmdOrig]

# Match elements of the form array to parameters

foreach arg $params {
    if {![info exists form($arg)]} {
        if {[info default $cmdOrig $arg value]} {
            lappend cmd $value
        } elseif {[string compare $arg "args"] == 0} {
            set needargs yes
        } else {
            lappend cmd {}
        }
    } else {
        lappend cmd $form($arg)
    }
}
# If args is a parameter, then append the form data
# that does not match other parameters as extra parameters

if {[info exists needargs]} {
    foreach {name value} $valuelist {
        if {[lsearch $params $name] < 0} {
            lappend cmd $name $value
        }
    }
}
# Eval the command

set code [catch $cmd result]
```

The `info commands` operation returns a list of all commands, which includes both built-in commands defined in C and Tcl procedures. There is no operation that just returns the list of built-in commands. Example 13–4 finds the built-in commands by removing all the procedures from the list of commands.

Example 13–4 Finding built-in commands.

```
proc Command_Info {{pattern *}} {
    # Create a table of procedures for quick lookup

    foreach p [info procs $pattern] {
        set isproc($p) 1
    }

    # Look for command not in the procedure table
    set result {}
```

```
    foreach c [info commands $pattern] {
        if {![info exists isproc($c)]} {
            lappend result $c
        }
    }
    return [lsort $result]
}
```

The Call Stack

The `info level` operation returns information about the Tcl evaluation stack, or *call stack*. The global level is numbered zero. A procedure called from the global level is at level one in the call stack. A procedure it calls is at level two, and so on. The `info level` command returns the current level number of the stack if no level number is specified.

If a positive level number is specified (e.g., `info level 3`), then the command returns the procedure name and argument values at that level in the call stack. If a negative level is specified, then it is relative to the current call stack. Relative level -1 is the level of the current procedure's caller, and relative level 0 is the current procedure. The following example prints the call stack. The `Call_trace` procedure avoids printing information about itself by starting at one less than the current call stack level:

Example 13–5 Getting a trace of the Tcl call stack.

```
proc Call_Trace {{file stdout}} {
    puts $file "Tcl Call Trace"
    for {set x [expr [info level]-1]} {$x > 0} {incr x -1} {
        puts $file "$x: [info level $x]"
    }
}
```

Command Evaluation

If you want to know how many Tcl commands are executed, use the `info cmdcount` command. This counts all commands, not just top-level commands. The counter is never reset, so you need to sample it before and after a test run if you want to know how many commands are executed during a test.

The `info complete` operation figures out whether a string is a complete Tcl command. This is useful for command interpreters that need to wait until the user has typed in a complete Tcl command before passing it to `eval`. Example 13–6 defines `Command_Process` that gets a line of input and builds up a command. When the command is complete, the command is executed at the global scope. `Command_Process` takes two *callbacks* as arguments. The `inCmd` is evaluated to get the line of input, and the `outCmd` is evaluated to display the results. Chapter 10 describes callbacks why the curly braces are used with `eval` as they are in this example:

Example 13–6 A procedure to read and evaluate commands.

```
proc Command_Process {inCmd outCmd} {
    global command
    append command(line) [eval $inCmd]
    if [info complete $command(line)] {
        set code [catch {uplevel #0 $command(line)} result]
        eval $outCmd {$result $code}
        set command(line) {}
    }
}
proc Command_Read {{in stdin}} {
    if [eof $in] {
        if {$in != "stdin"} {
            close $in
        }
        return {}
    }
    return [gets $in]
}
proc Command_Display {file result code} {
    puts stdout $result
}
while {![eof stdin]} {
    Command_Process {Command_Read stdin} \
        {Command_Display stdout}
}
```


Scripts and the Library

The name of the current script file is returned with the `info script` command. For example, if you use the `source` command to read commands from a file, then `info script` returns the name of that file if it is called during execution of the commands in that script. This is true even if the `info script` command is called from a procedure that is not defined in the script.

Use `info script` to find related files.

I often use `info script` to source or process files stored in the same directory as the script that is running. A few examples are shown in Example 13–7.

Example 13–7 Using `info script` to find related files.

```
# Get the directory containing the current script.
set dir [file dirname [info script]]

# Source a file in the same directory
source [file join $dir helper.tcl]

# Add an adjacent script library directory to auto_path
# The use of ../lib with file join is cross-platform safe.
lappend auto_path [file join $dir ../lib]
```

The pathname of the Tcl library is stored in the `tcl_library` variable, and it is also returned by the `info library` command. While you could put scripts into this directory, it might be better to have a separate directory and use the script library facility described in Chapter 12. This makes it easier to deal with new releases of Tcl and to package up your code if you want other sites to use it.

Version Numbers

Each Tcl release has a version number such as 7.4 or 8.0. This number is returned by the `info tclversion` command. If you want your script to run on a variety of Tcl releases, you may need to test the version number and take different actions in the case of incompatibilities between releases.

The Tcl release cycle starts with one or two alpha and beta releases before the final release, and there may even be a patch release after that. The `info patchlevel` command returns a qualified version number, like 8.0b1 for the first beta release of 8.0. We switched from using "p" (e.g., 8.0p2) to a three-level scheme (e.g., 8.0.3) for patch releases. The patch level is zero for the final release (e.g., 8.2.0). In general, you should be prepared for feature changes during the beta cycle, but there should only be bug fixes in the patch releases. Another rule of thumb is that the Tcl script interface remains quite compatible between releases; feature additions are upward compatible.

Execution Environment

The file name of the program being executed is returned with `info nameofexecutable`. This is more precise than the name in the `argv0` variable, which could be a relative name or a name found in a command directory on your command search path. It is still possible for `info nameofexecutable` to return a relative pathname if the user runs your program as `./foo`, for example. The following construct always returns the absolute pathname of the current program. If `info nameofexecutable` returns an absolute pathname, then the value of the current directory is ignored. The `pwd` command is described on page 115:

```
file join [pwd] [info nameofexecutable]
```

A few operations support dynamic loading of shared libraries, which are described in Chapter 44. The `info sharedlibextension` returns the file name suffix of dynamic link libraries. The `info loaded` command returns a list of libraries that have been loaded into an interpreter. Multiple interpreters are described in Chapter 19.

Cross-Platform Support

Tcl is designed so that you can write scripts that run unchanged on UNIX, Macintosh, and Windows platforms. In practice, you may need a small amount of code that is specific to a particular platform. You can find out information about

the platform via the tcl_platform variable. This is an array with these elements defined:

- tcl_platform(platform) is one of unix, macintosh, or windows.
- tcl_platform(os) identifies the operating system. Examples include MacOS, Solaris, Linux, Win32s (Windows 3.1 with the Win32 subsystem), Windows 95, Windows NT, and SunOS.
- tcl_platform(osVersion) gives the version number of the operating system.
- tcl_platform(machine) identifies the hardware. Examples include ppc (Power PC), 68k (68000 family), sparc, intel, mips, and alpha.
- tcl_platform(isWrapped) indicates that the application has been wrapped up into a single executable with *TclPro Wrapper*. This is not defined in normal circumstances.
- tcl_platform(user) gives the login name of the current user.
- tcl_platform(debug) indicates that Tcl was compiled with debugging symbols.
- tcl_platform(thread) indicates that Tcl was compiled with thread support enabled.

On some platforms a *hostname* is defined. If available, it is returned with the info hostname command. This command may return an empty string.

One of the most significant areas affected by cross-platform portability is the file system and the way files are named. This topic is discussed on page 103.

Tracing Variable Values

The trace command registers a command to be called whenever a variable is accessed, modified, or unset. This form of the command is:

```
trace variable name ops command
```

The *name* is a Tcl variable name, which can be a simple variable, an array, or an array element. If a whole array is traced, the trace is invoked when any element is used according to *ops*. The *ops* argument is one or more of the letters r, for read traces, w, for write traces, and u, for unset traces. The *command* is executed when one of these events occurs. It is invoked as:

```
command name1 name2 op
```

The *name1* argument is the variable or array name. The *name2* argument is the name of the array index, or null if the trace is on a simple variable. If there is an unset trace on an entire array and the array is unset, *name2* is also null. The value of the variable is not passed to the procedure. The traced variable is one level up the Tcl call stack. The upvar, uplevel, or global commands need to be used to make the variable visible in the scope of *command*. These commands are described in more detail in Chapter 7.

A read trace is invoked before the value of the variable is returned, so if it changes the variable itself, the new value is returned. A write trace is called

after the variable is modified. The unset trace is called after the variable is unset.

Read-Only Variables

Example 13–8 uses traces to implement a read-only variable. A variable is modified before the trace procedure is called, so the `ReadOnly` variable is needed to preserve the original value. When a variable is unset, the traces are automatically removed, so the unset trace action reestablishes the trace explicitly. Note that the `upvar` alias (e.g., `var`) cannot be used to set up the trace:

Example 13–8 Tracing variables.

```
proc ReadOnlyVar {varName} {
    upvar 1 $varName var
    global ReadOnly
    set ReadOnly($varName) $var
    trace variable $varName wu ReadOnlyTrace
}
proc ReadOnlyTrace { varName index op } {
    global ReadOnly
    upvar 1 $varName var
    switch $op {
        w {
            set var $ReadOnly($varName)
        }
        u {
            set var $ReadOnly($varName)
            # Re-establish the trace using the true name
            trace variable $varName wu ReadOnlyTrace
        }
    }
}
```

This example merely overrides the new value with the saved value. Another alternative is to raise an error with the `error` command. This will cause the command that modified the variable to return the error. Another common use of `trace` is to update a user interface widget in response to a variable change. Several of the Tk widgets have this feature built into them.

If more than one trace is set on a variable, then they are invoked in the reverse order; the most recent trace is executed first. If there is a trace on an array and on an array element, then the trace on the array is invoked first.

Creating an Array with Traces

Example 13–9 uses an array trace to dynamically create array elements:

Example 13–9 Creating array elements with array traces.

```
# make sure variable is an array
set dynamic() {}
trace variable dynamic r FixupDynamic
proc FixupDynamic {name index op} {
    upvar 1 $name dynArray
    if {![info exists dynArray($index)]} {
        set dynArray($index) 0
    }
}
```

Information about traces on a variable is returned with the `vinfo` option:

```
trace vinfo dynamic
=> {r FixupDynamic}
```

A trace is deleted with the `vdelete` option, which has the same form as the `variable` option. The trace in the previous example can be removed with the following command:

```
trace vdelete dynamic r FixupDynamic
```

Interactive Command History

The Tcl shell programs keep a log of the commands that you type by using a history facility. The log is controlled and accessed via the `history` command. The history facility uses the term *event* to mean an entry in its history log. The events are just commands, and they have an event ID that is their index in the log. You can also specify an event with a negative index that counts backwards from the end of the log. Event -1 is the previous event. Table 13–5 summarizes the Tcl `history` command. In the table, *event* defaults to -1.

In practice you will want to take advantage of the ability to abbreviate the history options and even the name of the `history` command itself. For the command, you need to type a unique prefix, and this depends on what other commands are already defined. For the options, there are unique one-letter abbreviations for all of them. For example, you could reuse the last word of the previous command with `[hist w $]`. This works because a `$` that is not followed by alphanumerics or an open brace is treated as a literal `$`.

Several of the history operations update the history list. They remove the actual `history` command and replace it with the command that resulted from the history operation. The `event` and `redo` operations all behave in this manner. This makes perfect sense because you would rather have the actual command in the history, instead of the history command used to retrieve the command.

Table 13–5 The `history` command.

`history`	Short for `history info` with no *`count`*.
`history add command ?exec?`	Adds the command to the history list. If `exec` is specified, then execute the command.
`history change` *`new`* `?`*`event`*`?`	Changes the command specified by *`event`* to *`new`* in the command history.
`history event ?`*`event`*`?`	Returns the command specified by *`event`*.
`history info ?`*`count`*`?`	Returns a formatted history list of the last *`count`* commands, or of all commands.
`history keep` *`count`*	Limits the history to the last *`count`* commands.
`history nextid`	Returns the number of the next event.
`history redo ?`*`event`*`?`	Repeats the specified command.

History Syntax

Some extra syntax is supported when running interactively to make the history facility more convenient to use. Table 13–6 shows the special history syntax supported by *tclsh* and *wish*.

Table 13–6 Special `history` syntax.

`!!`	Repeats the previous command.
`!`*`n`*	Repeats command number *`n`*. If *`n`* is negative it counts backward from the current command. The previous command is event -1.
`!`*`prefix`*	Repeats the last command that begins with *`prefix`*.
`!`*`pattern`*	Repeats the last command that matches *`pattern`*.
`^`*`old`*`^`*`new`*	Globally replaces *`old`* with *`new`* in the last command.

The next example shows how some of the history operations work:

Example 13–10 Interactive `history` usage.

```
% set a 5
5
% set a [expr $a+7]
12
% history
    1 set a 5
    2 set a [expr $a+7]
    3 history
% !2
19
% !!
```

```
26
% ^7^13
39
% !h
    1 set a 5
    2 set a [expr $a+7]
    3 history
    4 set a [expr $a+7]
    5 set a [expr $a+7]
    6 set a [expr $a+13]
    7 history
```


A Comparison to C Shell History Syntax

The history syntax shown in the previous example is simpler than the history syntax provided by the C shell. Not all of the history operations are supported with special syntax. The substitutions (using `^old^new`) are performed globally on the previous command. This is different from the quick-history of the C shell. Instead, it is like the `!:gs/old/new/` history command. So, for example, if the example had included `^a^b` in an attempt to set `b` to `39`, an error would have occurred because the command would have used `b` before it was defined:

```
set b [expr $b+7]
```

If you want to improve the history syntax, you will need to modify the `unknown` command, which is where it is implemented. This command is discussed in more detail in Chapter 12. Here is the code from the `unknown` command that implements the extra history syntax. The main limitation in comparison with the C shell history syntax is that the `!` substitutions are performed only when `!` is at the beginning of the command:

Example 13–11 Implementing special history syntax.

```
# Excerpts from the standard unknown command
# uplevel is used to run the command in the right context
if {$name == "!!"} {
    set newcmd [history event]
} elseif {[regexp {^!(.+)$} $name dummy event]} {
    set newcmd [history event $event]
} elseif {[regexp {^\^([^^]*)\^([^^]*)\^?$} $name x old new]} {
    set newcmd [history event -1]
    catch {regsub -all -- $old $newcmd $new newcmd}
}
if {[info exists newcmd]} {
    history change $newcmd 0
    return [uplevel $newcmd]
}
```

Debugging

The rapid turnaround with Tcl coding means that it is often sufficient to add a few `puts` statements to your script to gain some insight about its behavior. This solution doesn't scale too well, however. A slight improvement is to add a `Debug` procedure that can have its output controlled better. You can log the information to a file, or turn it off completely. In a Tk application, it is simple to create a text widget to hold the contents of the log so that you can view it from the application. Here is a simple `Debug` procedure. To enable it you need to set the `debug(enable)` variable. To have its output go to your terminal, set `debug(file)` to `stderr`.

Example 13–12 A `Debug` procedure.

```
proc Debug { args } {
    global debug
    if {![info exists debug(enabled)]} {
        # Default is to do nothing
        return
    }
    puts $debug(file) [join $args " "]
}
proc DebugOn {{file {}}} {
    global debug
    set debug(enabled) 1
    if {[string length $file] == 0} {
        set debug(file) stderr
    } else {
        if [catch {open $file w} fileID] {
            puts stderr "Cannot open $file: $fileID"
            set debug(file) stderr
        } else {
            puts stderr "Debug info to $file"
            set debug(file) $fileID
        }
    }
}
proc DebugOff {} {
    global debug
    if {[info exists debug(enabled)]} {
        unset debug(enabled)
        flush $debug(file)
        if {$debug(file) != "stderr" &&
             $debug(file) != "stdout"} {
            close $debug(file)
            unset debug(file)
        }
    }
}
```

Scriptics' TclPro

Scriptics offers a commercial development environment for Tcl called *TclPro*. *TclPro* features an extended Tcl platform and a set of development tools. The Tcl platform includes the popular [incr Tcl], Expect, and TclX extensions. These extensions and Tcl/Tk are distributed in source and binary form for Windows and a variety of UNIX platforms. There is an evaluation copy of TclPro on the CD-ROM. The *TclPro* distribution includes a copy of Tcl/Tk and the extensions that you can use for free. However, you will need to register at the Scriptics web site to obtain an evaluation license for the *TclPro* development tools. Please visit the following URL:

```
http://www.scriptics.com/registration/welchbook.html
```

The current version of TclPro contains these tools:

TclPro Debugger

TclPro Debugger provides a nice graphical user interface with all the features you expect from a traditional debugger. You can set breakpoints, single step, examine variables, and look at the call stack. It understands a subtle issue that can arise from using the `update` command: nested call stacks. It is possible to launch a new Tcl script as a side effect of the `update` command, which pushes the current state onto the execution stack. This shows up clearly in the debugger stack trace. It maintains project state, so it will remember breakpoint settings and other preference items between runs. One of the most interesting features is that it can debug remotely running applications. I use it regularly to debug Tcl code running inside the Tcl Web Server.

TclPro Checker

TclPro Checker is a static code checker. This is a real win for large program development. It examines every line of your program looking for syntax errors and dubious coding practices. It has detailed knowledge of Tcl, Tk, Expect, [incr Tcl], and TclX commands and validates your use of them. It checks that you call Tcl procedures with the correct number of arguments, and can cross-check large groups of Tcl files. It knows about changes between Tcl versions, and it can warn you about old code that needs to be updated.

TclPro Compiler

TclPro Compiler is really just a reader and writer for the byte codes that the Tcl byte-code compiler generates internally. It lets you precompile scripts and save the results, and then load the byte-code later instead of raw source. This provides a great way to hide your source code, if that is important to you. It turns out to save less time than you might think, however. By the time it reads the file from disk, decodes it, and builds the necessary Tcl data structures, it is not much faster than reading a source file and compiling it on the fly.

TclPro Wrapper

TclPro Wrapper assembles a collection of Tcl scripts, data files, and a Tcl/Tk interpreter into a single executable file. This makes distribution of your Tcl application as easy as giving out one file. The Tcl C library has been augmented with hooks in its file system access routines so that a wrapped application can look inside itself for files. The rule is that if you use a relative pathname (i.e., `lib/myfile.dat`), then the wrapped application will look first inside itself for the file. If the file is not found, or if the pathname is absolute (e.g., `/usr/local/lib/myfile.dat`), then Tcl looks on your hard disk for the file. The nice thing about *TclPro Wrapper* is that it handles all kinds of files, not just Tcl source files. It works by concatenating a ZIP file onto the end of a specially prepared Tcl interpreter. *TclPro* comes with pre-built interpreters that include Expect, [incr Tcl], and TclX, or you can build your own interpreter that contains custom C extensions.

Other Tools

The Tcl community has built many interesting and useful tools to help your Tcl development. Only two of them are mentioned below, but you can find many more at the Scriptics Tcl Resource Center:

```
http://www.scriptics.com/resource/
```

The *tkinspect* Program

The *tkinspect* program is a Tk application that lets you look at the state of other Tk applications. It displays procedures, variables, and the Tk widget hierarchy. With *tkinspect* you can issue commands to another application in order to change variables or test out commands. This turns out to be a very useful way to debug Tk applications. It was written by Sam Shen and is available on the CD-ROM. The current FTP address for this is:

```
ftp.neosoft.com:/pub/tcl/sorted/devel/tkinspect-5.1.6.tar.gz
```

The *Tuba* Debugger

Tuba is a debugger written purely in Tcl. It sets breakpoints by rewriting Tcl procedures to contain extra calls to the debugger. A small amount of support code is loaded into your application automatically, and the debugger application can set breakpoints, watch variables, and trace execution. It was written by John Stump and is available on the CD-ROM. The current URL for this package is:

```
http://www.geocities.com/SiliconValley/Ridge/2549/tuba/
```

The `bgerror` Command

When a Tcl script encounters an error during background processing, such

as handling file events or during the command associated with a button, it signals the error by calling the bgerror procedure. A default implementation displays a dialog and gives you an opportunity to view the Tcl call stack at the point of the error. You can supply your own version of bgerror. For example, when my *exmh* mail application gets an error it offers to send mail to me with a few words of explanation from the user and a copy of the stack trace. I get interesting bug reports from all over the world!

The bgerror command is called with one argument that is the error message. The global variable errorInfo contains the stack trace information. There is an example tkerror implementation in the on-line sources associated with this book.

The tkerror Command

The bgerror command used to be called tkerror. When event processing shifted from Tk into Tcl with Tcl 7.5 and Tk 4.1, the name tkerror was changed to bgerror. Backwards compatibility is provided so that if tkerror is defined, then tkerror is called instead of bgerror. I have run into problems with the compatibility setup and have found it more reliable to update my applications to use bgerror instead of tkerror. If you have an application that runs under either Tk 4.0 or Tk 4.1, you can simply define both:

```
proc bgerror [info args tkerror] [info body tkerror]
```

Performance Tuning

The time command measures the execution time of a Tcl command. It takes an optional parameter that is a repetition count:

```
time {set a "Hello, World!"} 1000
=> 28 microseconds per iteration
```

If you need the result of the command being timed, use set to capture the result:

```
puts $log "command: [time {set result [command]}]"
```

Time stamps in a Log

Another way to gain insight into the performance of your script is to generate log records that contain time stamps. The clock seconds value is too coarse, but you can couple it with the clock clicks value to get higher resolution measurements. Use the code shown in Example 13–1 on page 175 to calibrate the clicks per second on your system. Example 13–13 writes log records that contain the current time and the number of clicks since the last record. There will be occasional glitches in the clicks value when the system counter wraps around or is reset by the system clock, but it will normally give pretty accurate results. The Log procedure adds overhead, too, so you should take several measurements in a tight loop to see how long each Log call takes:

Example 13–13 Time Stamps in log records.

```
proc Log {args} {
    global log
    if [info exists log(file)] {
        set now [clock clicks]
        puts $log(file) [format "%s (%d)\t%s" \
            [clock format [clock seconds]] \
            [expr $now - $log(last)] \
            [join $args " "]]
        set log(last) $now
    }
}
proc Log_Open {file} {
    global log
    catch {close $log(file)}
    set log(file) [open $file w]
    set log(last) [clock clicks]
}
proc Log_Flush {} {
    global log
    catch {flush $log(file)}
}
proc Log_Close {} {
    global log
    catch {close $log(file)}
    catch {unset log(file)}
}
```

A more advanced `profile` command is part of the Extended Tcl (TclX) package, which is described in *Tcl/Tk Tools* (Mark Harrison, ed., O'Reilly & Associates, Inc., 1997). The TclX `profile` command monitors the number of calls, the CPU time, and the elapsed time spent in different procedures.

The Tcl Compiler

The built-in Tcl compiler improves performance in the following ways:

- Tcl scripts are converted into an internal byte-code format that is efficient to process. The byte codes are saved so that cost of compiling is paid only the first time you execute a procedure or loop. After that, execution proceeds much faster. Compilation is done as needed, so unused code is never compiled. If you redefine a procedure, it is recompiled the next time it is executed.
- Variables and command arguments are kept in a native format as long as possible and converted to strings only when necessary. There are several native types, including integers, floating point numbers, Tcl lists, byte codes, and arrays. There are C APIs for implementing new types. Tcl is still dynamically typed, so a variable can contain different types during its lifetime.

- Expressions and control structures are compiled into special byte codes, so they are executed more efficiently. Because `expr` does its own round of substitutions, the compiler generates better code if you group expressions with braces. This means that expressions go through only one round of substitutions. The compiler can generate efficient code because it does not have to worry about strange code like:

```
set subexpr {$x+$y}
expr 5 * $subexpr
```

The previous expression is not fully defined until runtime, so it has to be parsed and executed each time it is used. If the expression is grouped with braces, then the compiler knows in advance what operations will be used and can generate byte codes to implement the expression more efficiently.

The operation of the compiler is essentially transparent to scripts, but there are some differences in lists and expressions. These are described in Chapter 51. With lists, the good news is that large lists are more efficient. The problem is that lists are parsed more aggressively, so syntax errors at the end of a list will be detected even if you access only the beginning of the list. There were also some bugs in the code generator in the widely used Tcl 8.0p2 release. Most of these were corner cases like unbraced expressions in `if` and `while` commands. Most of these bugs were fixed in the 8.0.3 patch release, and the rest were cleaned up in Tcl 8.1 with the addition of a new internal parsing package.

CHAPTER 14

Namespaces

Namespaces group procedures and variables into separate name spaces. Namespaces were added in Tcl 8.0. This chapter describes the `namespace` and `variable` commands.

Namespaces provide new scopes for procedures and global variables. Originally Tcl had one global scope for shared variables, local scopes within procedures, and one global namespace for procedures. The single global scope for procedures and global variables can become unmanageable as your Tcl application grows. I describe some simple naming conventions on page 171 that I have used successfully in large programs. The namespace facility is a more elegant solution that partitions the global scope for procedure names and global variables.

Namespaces help structure large Tcl applications, but they add complexity. In particular, command callbacks may have to be handled specially so that they execute in the proper namespace. You choose whether or not you need the extra structure and learning curve of namespaces. If your applications are small, then you can ignore the namespace facility. If you are developing library packages that others will use, you should pick a namespace for your procedures and data so that they will not conflict with the applications in which they are used.

Using Namespaces

Namespaces add new syntax to procedure and variable names. A double colon, `::`, separates the namespace name from the variable or procedure name. You use this syntax to reference procedures and variables in a different namespace. The `namespace import` command lets you name things in other namespaces without

the extra syntax. Namespaces can be nested, so you can create a hierarchy of scopes. These concepts are explained in more detail in the rest of this chapter.

One feature not provided by namespaces is any sort of protection, or a way to enforce access controls between different namespaces. This sort of thing is awkward, if not impossible, to provide in a dynamic language like Tcl. For example, you are always free to use `namespace eval` to reach into any other namespace. Instead of providing strict controls, namespaces are meant to provide structure that enables large scale programming.

The package facility described in Chapter 12 was designed before namespaces. This chapter illustrates a style that ties the two facilities together, but they are not strictly related. It is possible to create a package named `A` that implements a namespace `B`, or to use a package without namespaces, or a namespace without a package. However, it makes sense to use the facilities together.

Example 14–1 repeats the random number generator from Example 7–4 on page 85 using namespaces. The standard naming style conventions for namespaces use lowercase:

Example 14–1 Random number generator using namespaces.

```
package provide random 1.0

namespace eval random {
    # Create a variable inside the namespace
    variable seed [clock seconds]

    # Make the procedures visible to namespace import
    namespace export init random range

    # Create procedures inside the namespace
    proc init { value } {
        variable seed
        set seed $value
    }
    proc random {} {
        variable seed
        set seed [expr ($seed*9301 + 49297) % 233280]
        return [expr $seed/double(233280)]
    }
    proc range { range } {
        expr int([random]*$range)
    }
}
```

Example 14–1 defines three procedures and a variable inside the namespace `random`. From inside the namespace, you can use these procedures and variables directly. From outside the namespace, you use the `::` syntax for namespace qualifiers. For example, the state variable is just `seed` within the namespace, but you use `random::seed` to refer to the variable from outside the

namespace. Using the procedures looks like this:

```
random::random
=> 0.3993355624142661
random::range 10
=> 4
```

If you use a package a lot you can *import* its procedures. A namespace declares what procedures can be imported with the `namespace export` command. Once you import a procedure, you can use it without a qualified name:

```
namespace import random::random
random
=> 0.54342849794238679
```

Importing and exporting are described in more detail later.

Namespace Variables

The `variable` command defines a variable inside a namespace. It is like the `set` command because it can define a value for the variable. You can declare several namespace variables with one `variable` command. The general form is:

```
variable name ?value? ?name value? ...
```

If you have an array, do not assign a value in the `variable` command. Instead, use regular Tcl commands after you declare the variable. You can put any commands inside a `namespace` block:

```
namespace eval foo {
    variable arr
    array set arr {name value name2 value2}
}
```

A namespace variable is similar to a global variable because it is outside the scope of any procedures. Procedures use the `variable` command or qualified names to reference namespace variables. For example, the `random` procedure has a `variable` command that brings the namespace variable into the current scope:

```
variable seed
```

If a procedure has a `variable` command that names a new variable, it is created in the namespace when it is first `set`.

Watch out for conflicts with global variables.

You need to be careful when you use variables inside a namespace block. If you declare them with a `variable` command, they are clearly namespace variables. However, if you forget to declare them, then they will either become namespace variables, or latch onto an existing global variable by the same name. Consider the following code:

```
namespace eval foo {
    variable table
    for {set i 1} {$i <= 256} {incr i} {
    set table($i) [format %c $i]
    }
}
```

If there is already a global variable `i`, then the `for` loop will use that variable. Otherwise, it will create the `foo::i` variable. I found this behavior surprising, but it does make it easier to access global variables like `env` without first declaring them with `global` inside the namespace block.

Qualified Names

A fully qualified name begins with `::`, which is the name for the global namespace. A fully qualified name unambiguously names a procedure or a variable. The fully qualified name works anywhere. If you use a fully qualified variable name, it is *not* necessary to use a `global` command. For example, suppose namespace `foo` has a namespace variable `x`, and there is also a global variable `x`. The global variable `x` can be named with this:

```
::x
```

The `::` syntax does not affect variable substitutions. You can get the value of the global variable `x` with `$::x`. Name the namespace variable `x` with this:

```
::foo::x
```

A partially qualified name does not have a leading `::`. In this case the name is resolved from the current namespace. For example, the following also names the namespace variable `x`:

```
foo::x
```

You can use qualified names with `global`. Once you do this, you can access the variable with its short name:

```
global ::foo::x
set x 5
```

Declaring variables is more efficient than using qualified names.

The Tcl byte-code compiler generates faster code when you declare namespace and global variables. Each procedure context has its own table of variables. The table can be accessed by a direct slot index, or by a hash table lookup of the variable name. The hash table lookup is slower than the direct slot access. When you use the `variable` or `global` command, then the compiler can use a direct slot access. If you use qualified names, the compiler uses the more general hash table lookup.

Command Lookup

A command is looked up first in the current name space. If it is not found there, then it is looked up in the global namespace. This means that you can use all the built-in Tcl commands inside a namespace with no special effort.

You can play games by redefining commands within a namespace. For example, a namespace could define a procedure named `set`. To get the built-in `set` you could use `::set`, while `set` referred to the `set` defined inside namespace. Obviously you need to be quite careful when you do this.

You can use qualified names when defining procedures. This eliminates the

need to put the proc commands inside a namespace block. However, you still need to use namespace eval to create the namespace before you can create procedures inside it. Example 14–2 repeats the random number generator using qualified names. random::init does not need a variable command because it uses a qualified name for seed:

Example 14–2 Random number generator using qualified names.

```
namespace eval random {
    # Create a variable inside the namespace
    variable seed [clock seconds]
}
# Create procedures inside the namespace
proc random::init { seed } {
    set ::random::seed $seed
}
proc random::random {} {
    variable seed
    set seed [expr ($seed*9301 + 49297) % 233280]
    return [expr $seed/double(233280)]
}
proc random::range { range } {
    expr int([random]*$range)
}
```


Nested Namespaces

Namespaces can be nested inside other namespaces. Example 14–3 shows three namespaces that have their own specific variable x. The fully qualified names for these variables are ::foo::x, ::bar::x, and ::bar::foo::x.

Example 14–3 Nested namespaces.

```
namespace eval foo {
    variable x 1     ;# ::foo::x
}
namespace eval bar {
    variable x 2     ;# ::bar::x
    namespace eval foo {
        variable x 3 ;# ::bar::foo::x
    }
    puts $foo::x     ;# prints 3
}
puts $foo::x         ;# prints 1
```

Partially qualified names can refer to two different objects.

In Example 14–3 the partially qualified name foo::x can reference one of two variables depending on the current namespace. From the global scope the name foo::x refers to the namespace variable x inside ::foo. From the ::bar

namespace, `foo::x` refers to the variable `x` inside `::bar::foo`.

If you want to unambiguously name a variable in the current namespace, you have two choices. The simplest is to bring the variable into scope with the `variable` command:

```
variable x
set x something
```

If you need to give out the name of the variable, then you have two choices. The most general solution is to use the `namespace current` command to create a fully qualified name:

```
trace variable [namespace current]::x r \
    [namespace current]::traceproc
```

However, it is simpler to just explicitly write out the namespace as in:

```
trace variable ::myname::x r ::myname::traceproc
```

The drawback of this approach is that it litters your code with references to `::myname::`, which might be subject to change during program development.

Importing and Exporting Procedures

Commands can be imported from namespaces to make it easier to name them. An imported command can be used without its namespace qualifier. Each namespace specifies exported procedures that can be the target of an import. Variables cannot be imported. Note that importing is only a convenience; you can always use qualified names to access any procedure. As a matter of style, I avoid importing names, so I know what package a command belongs to when I'm reading code.

The `namespace export` command goes inside the namespace block, and it specifies what procedures a namespace exports. The specification is a list of `string match` patterns that are compared against the set of commands defined in a namespace. The export list can be defined before the procedures being exported. You can do more than one `namespace export` to add more procedures, or patterns, to the export list for a namespace. Use the `-clear` flag if you need to reset the export list.

```
namespace export ?-clear? ?pat? ?pat? ...
```

Only exported names appear in package indexes.

When you create the `pkgIndex.tcl` package index file with `pkg_mkIndex`, which is described Chapter 12, you should be aware that only exported names appear in the index. Because of this, I often resort to exporting everything. I never plan to import the names, but I do rely on automatic code loading based on the index files. This exports everything:

```
namespace export *
```

The `namespace import` command makes commands in another namespace visible in the current namespace. An `import` can cause conflicts with commands in the current namespace. The `namespace import` command raises an error if there is a conflict. You can override this with the `-force` option. The general form

of the command is:

```
namespace import ?-force? namespace::pat ?namespace::pat?...
```

The *pat* is a `string match` type pattern that is matched against *exported* commands defined in *namespace*. You cannot use patterns to match *namespace*. The *namespace* can be a fully or partially qualified name of a namespace.

If you are lazy, you can import all procedures from a namespace:

```
namespace import random::*
```

The drawback of this approach is that `random` exports an `init` procedure, which might conflict with another module you import in the same way. It is safer to `import` just the procedures you plan on using:

```
namespace import random::random random::range
```

A `namespace import` takes a snapshot.

If the set of procedures in a namespace changes, or if its export list changes, then this has no effect on any imports that have already occurred from that namespace.

Callbacks and Namespaces

Commands like `after`, `bind`, and `button` take arguments that are Tcl scripts that are evaluated later. These *callback* commands execute later in the global scope by default. If you want a callback to be evaluated in a particular namespace, you can construct the callback with `namespace code`. This command does not execute the callback. Instead, it generates a Tcl command that will execute in the current namespace scope when it is evaluated later. For example, suppose `::current` is the current namespace. The `namespace code` command determines the current scope and adds that to the `namespace inscope` command it generates:

```
set callback [namespace code {set x 1}]
=> namespace inscope ::current {set x 1}
# sometime later ...
eval $callback
```

When you evaluate `$callback` later, it executes in the `::current` namespace because of the `namespace inscope` command. In particular, if there is a namespace variable `::current::x`, then that variable is modified. An alternative to using namespace code is to name the variable with a qualified name:

```
set callback {set ::current::x 1}
```

The drawback of this approach is that it makes it tedious to move the code to a different namespace.

If you need substitutions to occur on the command when you define it, use `list` to construct it. Using `list` is discussed in more detail on pages 123 and 387. Example 14–4 wraps up the `list` and the `namespace inscope` into the `code` procedure, which is handy because you almost always want to use `list` when constructing callbacks. The `uplevel` in `code` ensures that the correct namespace is captured; you can use `code` anywhere:

Example 14–4 The code procedure to wrap callbacks.

```
proc code {args} {
    set namespace [uplevel {namespace current}]
    return [list namespace inscope $namespace $args]
}
namespace eval foo {
    variable y "y value" x {}
    set callback [code set x $y]
    => namespace inscope ::foo {set x {y value}}
}
```

The example defines a callback that will set `::foo::x` to `y value`. If you want to set `x` to the value that `y` has at the time of the callback, then you do not want to do any substitutions. In that case, the original `namespace code` is what you want:

```
set callback [namespace code {set x $y}]
=> namespace inscope ::foo {set x $y}
```

If the callback has additional arguments added by the caller, `namespace inscope` correctly adds them. For example, the scrollbar protocol described on page 429 adds parameters to the callback that controls a scrollbar.

Introspection

The `info commands` operation returns all the commands that are currently visible. It is described in more detail on page 179. You can limit the information returned with a `string match` pattern. You can also include a namespace specifier in the pattern to see what is visible in a namespace. Remember that global commands and imported commands are visible, so `info commands` returns more than just what is defined by the namespace. Example 14–5 uses `namespace origin`, which returns the original name of imported commands, to sort out the commands that are really defined in a namespace:

Example 14–5 Listing commands defined by a namespace.

```
proc Namespace_List {{namespace {}}} {
    if {[string length $namespace] == 0} {
        # Determine the namespace of our caller
        set namespace [uplevel {namespace current}]
    }
    set result {}
    foreach cmd [info commands ${namespace}::*] {
        if {[namespace origin $cmd] == $cmd} {
            lappend result $cmd
        }
    }
    return [lsort $result]
}
```

The namespace Command

Table 14–1 summarizes the namespace operations:

Table 14–1 The namespace command.

`namespace current`	Returns the current namespace.
`namespace children ?name? ?pat?`	Returns names of nested namespaces. *name* defaults to current namespace. *pat* is a `string match` pattern that limits what is returned.
`namespace code script`	Generates a `namespace inscope` command that will eval *script* in the current namespace.
`namespace delete name ?name? ...`	Deletes the variables and commands from the specified namespaces.
`namespace eval name cmd ?args? ...`	Concatenates *args*, if present, onto *cmd* and evaluates it in *name* namespace.
`namespace export ?-clear? ?pat? ?pat? ...`	Adds patterns to the export list for current namespace. Returns export list if no patterns.
`namespace forget pat ?pat? ...`	Undoes the `import` of names matching patterns.
`namespace import ?-force? pat ?pat? ...`	Adds the names matching the patterns to the current namespace.
`namespace inscope name cmd ?args? ...`	Appends *args*, if present, onto *cmd* as list elements and evaluates it in *name* namespace.
`namespace origin cmd`	Returns the original name of *cmd*.
`namespace parent ?name?`	Returns the parent namespace of *name*, or of the current namespace.
`namespace qualifiers name`	Returns the part of *name* up to the last `::` in it.
`namespace which ?flag? name`	Returns the fully qualified version of *name*. The *flag* is one of `-command`, `-variable`, or `-namespace`.
`namespace tail name`	Returns the last component of *name*.

Converting Existing Packages to use Namespaces

Suppose you have an existing set of Tcl procedures that you want to wrap in a namespace. Obviously, you start by surrounding your existing code in a `namespace eval` block. However, you need to consider three things: global variables, exported procedures, and callbacks.

- Global variables remain global until you change your code to use `variable` instead of `global`. Some variables may make sense to leave at the global scope. Remember that the variables that Tcl defines are global, including

env, tcl_platform, and the others listed in Table 2–2 on page 30. If you use the upvar #0 trick described on page 86, you can adapt this to namespaces by doing this instead:

```
upvar #0 [namespace current]::$instance state
```

- Exporting procedures makes it more convenient for users of your package. It is not strictly necessary because they can always use qualified names to reference your procedures. An export list is a good hint about which procedures are expected to be used by other packages. Remember that the export list determines what procedures are visible in the index created by pkg_mkIndex.
- Callbacks execute at the global scope. If you use variable traces and variables associated with Tk widgets, these are also treated as global variables. If you want a callback to invoke a namespace procedure, or if you give out the name of a namespace variable, then you must construct fully qualified variable and procedure names. You can hardwire the current namespace:

```
button .foo -command ::myname::callback \
    -textvariable ::myname::textvar
```

or you can use namespace current:

```
button .foo -command [namespace current]::callback \
    -textvariable [namespace current]::textvar
```

[incr Tcl] Object System

The Tcl namespace facility does not provide classes and inheritance. It just provides new scopes and a way to hide procedures and variables inside a scope. There are Tcl C APIs that support hooks in variable name and command lookup for object systems so that they can implement classes and inheritance. By exploiting these interfaces, various object systems can be added to Tcl as shared libraries.

The Tcl namespace facility was proposed by Michael McLennan based on his experiences with [incr Tcl], which is the most widely used object-oriented extension for Tcl. [incr Tcl] provides classes, inheritance, and protected variables and commands. If you are familiar with C++, [incr Tcl] should feel similar. A complete treatment of [incr Tcl] is not made in this book. *Tcl/Tk Tools* (Mark Harrison, O'Reilly & Associates, Inc., 1997) is an excellent source of information. You can find a version of [incr Tcl] on the CD-ROM. The [incr Tcl] home page is:

```
http://www.tcltk.com/itcl/
```

Notes

The final section of this chapter touches on a variety of features of the namespace facility.

Names for Widgets, Images, and Interpreters

There are a number of Tcl extensions that are not affected by the namespaces described in this chapter, which apply only to commands and variable names. For example, when you create a Tk widget, a Tcl command is also created that corresponds to the Tk widget. This command is always created in the global command namespace even when you create the Tk widget from inside a `namespace eval` block. Other examples include Tcl interpreters, which are described in Chapter 19, and Tk images, which are described in Chapter 38.

The `variable` command at the global scope

It turns out that you can use `variable` like the `global` command if your procedures are not inside a namespace. This is consistent because it means "this variable belongs to the current namespace," which might be the global namespace.

Auto Loading and `auto_import`

The following sequence of commands can be used to import commands from the `foo` package:

```
package require foo
namespace import foo::*
```

However, because of the default behavior of packages, there may not be anything that matches `foo::*` after the `package require`. Instead, there are entries in the `auto_index` array that will be used to load those procedures when you first use them. The auto loading mechanism is described in Chapter 12. To account for this, Tcl calls out to a hook procedure called `auto_import`. This default implementation of this procedure searches `auto_index` and forcibly loads any pending procedures that match the import pattern. Packages like [incr Tcl] exploit this hook to implement more elaborate schemes. The `auto_import` hook was first introduced in Tcl 8.0.3.

Namespaces and `uplevel`

Namespaces affect the Tcl call frames just like procedures do. If you walk the call stack with `info level`, the namespace frames are visible. This means that you can get access to all variables with `uplevel` and `upvar`. Level `#0` is still the absolute global scope, outside any namespace or procedure. Try out `Call_Trace` from Example 13–5 on page 180 on your code that uses namespaces to see the effect.

Naming Quirks

When you name a namespace, you are allowed to have extra colons at the end. You can also have two or more colons as the separator between namespace

name components. These rules make it easier to assemble names by adding to the value returned from `namespace current`. These all name the same namespace:

```
::foo::bar
::foo::bar::
::foo:::::::bar
```

The name of the global namespace can be either `::` or the empty string. This follows from the treatment of `::` in namespace names.

When you name a variable or command, a trailing `::` is significant. In the following command a variable inside the `::foo::bar` namespace is modified. The variable has an empty string for its name!

```
set ::foo::bar:: 3
namespace eval ::foo::bar { set {} }
=> 3
```

If you want to embed a reference to a variable just before two colons, use a backslash to turn off the variable name parsing before the colons:

```
set x xval
set y $x\::foo
=> xval::foo
```

Miscellaneous

You can remove names you have imported:

```
namespace forget random::init
```

You can `rename` imported procedures to modify their names:

```
rename range Range
```

You can even move a procedure into another namespace with `rename`:

```
rename random::init myspace::init
```

CHAPTER 15

Internationalization

This chapter describes features that support text processing for different character sets such as ASCII and Japanese. Tcl can read and write data in various character set encodings, but it processes data in a standard character set called Unicode. Tcl has a message catalog that lets you generate different versions of an application for different languages. Tcl commands described are: `encoding` and `msgcat`.

Different languages use different alphabets, or *character sets*. An *encoding* is a standard way to represent a character set. Tcl hides most of the issues associated with encodings and character sets, but you need to be aware of them when you write applications that are used in different countries. You can also write an application using a *message catalog* so that the strings you display to users can be in the language of their choice. Using a message catalog is more work, but Tcl makes it as easy as possible.

Most of the hard work in dealing with character set encodings is done "under the covers" by the Tcl C library. The Tcl C library underwent substantial changes to support international character sets. Instead of using 8-bit bytes to store characters, Tcl uses a 16-bit character set called Unicode, which is large enough to encode the alphabets of all languages. There is also plenty of room left over to represent special characters like ♥ and ⊗.

In spite of all the changes to support Unicode, there are few changes visible to the Tcl script writer. Scripts written for Tcl 8.0 and earlier continue to work fine with Tcl 8.1 and later versions. You only need to modify scripts if you want to take advantage of the features added to support internationalization.

This chapter begins with a discussion of what a character set is and why different codings are used to represent them. It concludes with a discussion of message catalogs.

Character Sets and Encodings

If you are from the United States, you've probably never thought twice about character sets. Most computers use the ASCII encoding, which has 127 characters. That is enough for the 26 letters in the English alphabet, upper case and lower case, plus numbers, various punctuation characters, and control characters like tab and newline. ASCII fits easily in 8-bit characters, which can represent 256 different values.

European alphabets include accented characters like è, ñ, and ä. The ISO Latin-1 encoding is a superset of ASCII that encodes 256 characters. It shares the ASCII encoding in values 0 through 127 and uses the "high half" of the encoding space to represent accented characters as well as special characters like ©. There are several ISO Latin encodings to handle different alphabets, and these share the trick of encoding ASCII in the lower half and other characters in the high half. You might see these encodings referred to as `iso8859-1`, `iso8859-2`, and so on.

Asian character sets are simply too large to fit into 8-bit encodings. There are a number of 16-bit encodings for these languages. If you work with these, you are probably familiar with the "Big 5" or ShiftJIS encodings.

Unicode is an international standard character set encoding. There are both 16-bit Unicode and 32-bit Unicode standards, but Tcl and just about everyone else just use the 16-bit standard. Unicode has the important property that it can encode all the important character sets without conflicts and overlap. By converting all characters to the Unicode encoding, Tcl can work with different character sets simultaneously.

The System Encoding

Computer systems are set up with a standard system encoding for their files. If you always work with this encoding, then you can ignore character set issues. Tcl will read files and automatically convert them from the system encoding to Unicode. When Tcl writes files, it automatically converts from Unicode to the system encoding. If you are curious, you can find out the system encoding with:

```
encoding system
=> cp1252
```

The "cp" is short for "code page," the term that Windows uses to refer to different encodings. On my Unix system, the system encoding is `iso8859-1`.

Do not change the system encoding.

You could also change the system encoding with:

```
encoding system encoding
```

But this is not a good idea. It immediately changes how Tcl passes strings to your operating system, and it is likely to leave Tcl in an unusable state. Tcl automatically determines the system encoding for you. Don't bother trying to set it yourself.

The `encoding names` command lists all the encodings that Tcl knows about. The encodings are kept in files stored in the `encoding` directory under the Tcl script library. They are loaded automatically the first time you use an encoding.

```
lsort [encoding names]
=> ascii big5 cp1250 cp1251 cp1252 cp1253 cp1254 cp1255
cp1256 cp1257 cp1258 cp437 cp737 cp775 cp850 cp852 cp855
cp857 cp860 cp861 cp862 cp863 cp864 cp865 cp866 cp869
cp874 cp932 cp936 cp949 cp950 dingbats euc-cn euc-jp euc-
kr gb12345 gb1988 gb2312 identity iso2022 iso2022-jp
iso2022-kr iso8859-1 iso8859-2 iso8859-3 iso8859-4
iso8859-5 iso8859-6 iso8859-7 iso8859-8 iso8859-9
jis0201 jis0208 jis0212 ksc5601 macCentEuro macCroatian
macCyrillic macDingbats macGreek macIceland macJapan
macRoman macRomania macThai macTurkish macUkraine
shiftjis symbol unicode utf-8
```

The encoding names reflect their origin. The "cp" refers to the "code pages" that Windows uses to manage encodings. The "mac" encodings come from the Macintosh. The "iso," "euc," "gb," and "jis" encodings come from various standards bodies.

File Encodings and `fconfigure`

The conversion to Unicode happens automatically in the Tcl C library. When Tcl reads and writes files, it translates from the current system encoding into Unicode. If you have files in different encodings, you can use the `fconfigure` command to set the encoding. For example, to read a file in the standard Russian encoding (`iso8859-7`):

```
set in [open README.russian]
fconfigure $in -encoding iso8859-7
```

Example 15–1 shows a simple utility I use in *exmh*,* a MIME-aware mail reader. MIME has its own convention for specifying the character set encoding of a mail message that differs slightly from Tcl's naming convention. The procedure launders the name and then sets the encoding. Exmh was already aware of MIME character sets, so it could choose fonts for message display. Adding this procedure and adding two calls to it was all I had to do to adapt *exmh* to Unicode.

Example 15–1 MIME character sets.and file encodings.

```
proc Mime_SetEncoding {file charset} {
    regsub -all {(iso|jis|us)-} $charset {\1} charset
    set charset [string tolower charset]
    regsub usascii $charset ascii charset
    fconfigure $file -encoding $charset
}
```

* The *exmh* home page is http://www.beedub.com/exmh/. It is a wonderful tool that helps me manage tons of e-mail. It is written in Tcl/Tk, of course, and relies on the MH mail system, which limits it to UNIX.

Scripts in Different Encodings

If you have scripts that are not in the system encoding, then you cannot use `source` to load them. However, it is easy to read the files yourself under the proper encoding and use `eval` to process them. Example 15–2 adds a `-encoding` flag to the `source` command. This is likely to become a built-in feature in future versions of Tcl so that commands like `info script` will work properly:

Example 15–2 Using scripts in nonstandard encodings.

```
proc Source {args} {
    set file [lindex $args end]
    if {[llength $args] == 3 &&
            [string equal -encoding [lindex $args 0]]} {
        set encoding [lindex $args 1]
        set in [open $file]
        fconfigure $in -encoding $encoding
        set script [read $in]
        close $in
        return [uplevel 1 $script]
    } elseif {[llength $args] == 1} {
        return [uplevel 1 [list source $file]]
    } else {
        return -code error \
            "Usage: Source ?-encoding encoding? file?"
    }
}
```

Unicode and UTF-8

UTF-8 is an encoding for Unicode. While Unicode represents all characters with 16 bits, the UTF-8 encoding uses either 8, 16, or 24 bits to represent one Unicode character. This variable-width encoding is useful because it uses 8 bits to represent ASCII characters. This means that a pure ASCII string, one with character codes all fewer than 128, is also a UTF-8 string. Tcl uses UTF-8 internally to make the transition to Unicode easier. It allows interoperability with Tcl extensions that have not been made Unicode-aware. They can continue to pass ASCII strings to Tcl, and Tcl will interpret them correctly.

As a Tcl script writer, you can mostly ignore UTF-8 and just think of Tcl as being built on Unicode (i.e., full 16-bit character set support). If you write Tcl extensions in C or C++, however, the impact of UTF-8 and Unicode is quite visible. This is explained in more detail in Chapter 44.

Tcl lets you read and write files in UTF-8 encoding or directly in Unicode. This is useful if you need to use the same file on systems that have different system encodings. These files might be scripts, message catalogs, or documentation. Instead of using a particular native format, you can use Unicode or UTF-8 and read the files the same way on any of your systems. Of course, you will have to set the encoding properly by using `fconfigure` as shown earlier.

The Binary Encoding

If you want to read a data file and suppress all character set transformations, use the `binary` encoding:

```
fconfigure $in -encoding binary
```

Under the binary encoding, Tcl reads in each 8-bit byte and stores it into the lower half of a 16-bit Unicode character with the high half set to zero. During binary output, Tcl writes out the lower byte of each Unicode character. You can see that reading in binary and then writing it out doesn't change any bits. Watch out if you read something in one encoding and then write it out in binary. Any information in the high byte of the Unicode character gets lost!

Tcl actually handles the binary encoding more efficiently than just described, but logically the previous description is still accurate. As described in Chapter 44, Tcl can manage data in several forms, not just strings. When you read a file in binary format, Tcl stores the data as a `ByteArray` that is simply 8 bits of data in each byte. However, if you ask for this data as a string (e.g., with the `puts` command), Tcl automatically converts from 8-bit bytes to 16-bit Unicode characters by setting the high byte to all zeros.

The `binary` command also manipulates data in `ByteArray` format. If you read a file with the binary encoding and then use the `binary` command to process the data, Tcl will keep the data in an efficient form.

The string command also understands the `ByteArray` format, so you can do operations like `string length`, `string range`, and `string index` on binary data without suffering the conversion cost from a `ByteArray` to a UTF-8 string.

Conversions Between Encodings

The `encoding` command lets you convert strings between encodings. The `encoding convertfrom` command converts data in some other encoding into a Unicode string. The `encoding convertto` command converts a Unicode string into some other encoding. For example, the following two sequences of commands are equivalent. They both read data from a file that is in Big5 encoding and convert it to Unicode:

```
fconfigure $input -encoding gb12345
set unicode [read $input]
```

or

```
fconfigure $input -encoding binary
set unicode [encoding convertfrom gb12345 [read $input]]
```

In general, you can lose information when you go from Unicode to any other encoding, so you ought to be aware of the limitations of the encodings you are using. In particular, the `binary` encoding may not preserve your data if it starts out from an arbitrary Unicode string. Similarly, an encoding like `iso-8859-2` may simply not have a representation of a given Unicode character.

The `encoding` Command

Table 15–1 summarizes the encoding command:

Table 15–1 The `encoding` command.

`encoding convertfrom ?`*`encoding`*`? `*`data`*	Converts binary *`data`* from the specified *`encoding`*, which defaults to the system encoding, into Unicode.
`encoding convertto ?`*`encoding`*`? `*`string`*	Converts *`string`* from Unicode into data in the *`encoding`* format, which defaults to the system encoding.
`encoding names`	Returns the names of known encodings.
`encoding system ?`*`encoding`*`?`	Queries or change the system encoding.

Message Catalogs

A message catalog is a list of messages that your application will display. The main idea is that you can maintain several catalogs, one for each language you support. Unfortunately, you have to be explicit about using message catalogs. Everywhere you generate output or display strings in Tk widgets, you need to change your code to go through a message catalog. Fortunately, Tcl uses a nice trick to make this fairly easy and to keep your code readable. Instead of using keys like "message42" to get messages out of the catalog, Tcl just uses the strings you would use by default. For example, instead of this code:

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

A version that uses message catalogs looks like this:

```
puts [msgcat::mc "Hello, World!"]
```

If you have not already loaded your message catalog, or if your catalog doesn't contain a mapping for "Hello, World!", then `msgcat::mc` just returns its argument. Actually, you can define just what happens in the case of unknown inputs by defining your own `msgcat::mcunknown` procedure, but the default behavior is quite good.

The message catalog is implemented in Tcl in the `msgcat` package. You need to use `package require` to make it available to your scripts:

```
package require msgcat
```

In addition, all the procedures in the package begin with "mc," so you can use `namespace import` to shorten their names further. I am not a big fan of `namespace import`, but if you use message catalogs, you will be calling the `msgcat::mc` function a lot, so it may be worthwhile to import it:

```
namespace import msgcat::mc
puts [mc "Hello, World!"]
```

Specifying a Locale

A *locale* identifies a language or language dialect to use in your output. A three-level scheme is used in the locale identifier:

```
language_country_dialect
```

The language codes are defined by the ISO-3166 standard. For example, "en" is English and "es" is Spanish. The country codes are defined by the ISO-639 standard. For example, US is for the United States and UK is for the United Kingdom. The dialect is up to you. The country and dialect parts are optional. Finally, the locale specifier is case insensitive. The following examples are all valid locale specifiers:

```
es
en
en_US
en_us
en_UK
en_UK_Scottish
en_uk_scottish
```

Users can set their initial locale with the `LANG` and `LOCALE` environment variables. If there is no locale information in the environment, then the "c" locale is used (i.e., the C programming language.) You can also set and query the locale with the `msgcat::mclocale` procedure:

```
msgcat::mclocale
=> c
msgcat::mclocale en_US
```

The `msgcat::mcpreferences` procedure returns a list of the user's locale preferences from most specific (i.e., including the dialect) to most general (i.e., only the language). For example:

```
msgcat::mclocale en_UK_Scottish
msgcat::mcpreferences
=> en_UK_Scottish en_UK en
```

Managing Message Catalog Files

A message catalog is simply a Tcl source file that contains a series of `msgcat::mcset` commands that define entries in the catalog. The syntax of the `msgcat::mcset` procedure is:

```
msgcat::mcset locale src-string ?dest-string?
```

The *`locale`* is a locale description like `es` or `en_US_Scottish`. The *`src-string`* is the string used as the key when calling `msgcat::mc`. The *`dest-string`* is the result of `msgcat::mc` when the *`locale`* is in force.

The `msgcat::mcload` procedure should be used to load your message catalog files. It expects the files to be named according to their locale (e.g., `en_US_Scottish.msg`), and it binds the message catalog to the current namespace.

The msgcat::mcload procedure loads files that match the msgcat::mcpreferences and have the .msg suffix. For example, with a locale of en_UK_Scottish, msgcat::mcload would look for these files:

```
en_UK_Scottish.msg en_UK.msg en.msg
```

The standard place for message catalog files is in the msgs directory below the directory containing a package. With this arrangement you can call msgcat::mcload as shown below. The use of info script to find related files is explained on page 181.

```
msgcat::mcload [file join [file dirname [info script]] msgs]
```

The message catalog file is sourced, so it can contain any Tcl commands. You might find it convenient to import the msgcat::mcset procedure. Be sure to use -force with namespace import because that command might already have been imported as a result of loading other message catalog files. Example 15–3 shows three trivial message catalog files:

Example 15–3 Three sample message catalog files.

```
## en.msg
namespace import -force msgcat::mcset

mcset en Hello Hello_en
mcset en Goodbye Goodbye_en
mcset en String String_en
# end of en.msg

## en_US.msg
namespace import -force msgcat::mcset

mcset en_US Hello Hello_en_US
mcset en_US Goodbye Goodbye_en_US
# end of en_US.msg

## en_US_Texan.msg
namespace import -force msgcat::mcset

mcset en_US_Texan Hello Howdy!
# end of en_US_Texan.msg
```

Assuming the files from Example 15–3 are all in the msgs directory below your script, you can load all these files with these commands:

```
msgcat::mclocale en_US_Texan
msgcat::mcload [file join [file dirname [info script]] msgs]
```

The dialect has the highest priority:

```
msgcat::mc Hello
=> Howdy!
```

If the dialect does not specify a mapping, then the country mapping is checked:

```
msgcat::mc Goodbye
=> Goodbye_en_US
```

Finally, the lowest priority is the language mapping:

```
msgcat::mc String
=> String_en
```


Message Catalogs and Namespaces

What happens if two different library packages have conflicting message catalogs? Suppose the `foo` package contains this call:

```
msgcat::set fr Hello Bonjour
```

But the `bar` package contains this conflicting definition:

```
msgcat::mcset fr Hello Ello
```

What happens is that `msgcat::mcset` and `msgcat::mc` are sensitive to the current Tcl namespace. Namespaces are described in detail in Chapter 14. If the `foo` package loads its message catalog while inside the `foo` namespace, then any calls to `msgcat::mc` from inside the `foo` namespace will see those definitions. In fact, if you call `msgcat::mc` from inside any namespace, it will find only message catalog definitions defined from within that namespace.

If you want to share message catalogs between namespaces, you will need to implement your own version of `msgcat::mcunknown` that looks in the shared location. Example 15–4 shows a version that looks in the global namespace before returning the default string.

Example 15–4 Using `msgcat::mcunknown` to share message catalogs.

```
proc msgcat::mcunknown {local src} {
    variable insideUnknown
    if {![info exist insideUnknown]} {

        # Try the global namespace, being careful to note
        # that we are already inside this procedure.

        set insideUnknown true
        set result [namespace eval :: [list \
            msgcat::mc $src \
        ]]
        unset insideUnknown
        return $result
    } else {

        # Being called because the message isn't found
        # in the global namespace

        return $src
    }
}
```

The `msgcat` package

Table 15–2 summarizes the `msgcat` package.

Table 15–2 The msgcat package

`msgcat::mc` *`src`*	Returns the translation of *`src`* according to the current locale and namespace.
`msgcat::mclocale ?`*`locale`*`?`	Queries or set the current *`locale`*.
`msgcat::mcpreferences`	Returns a list of locale preferences ordered from the most specific to the most general.
`msgcat::mcload` *`directory`*	Loads message files for the current locale from *`directory`*.
`msgcat::mcset` *`locale src translation`*	Defines a mapping for the *`src`* string in *`locale`* to the *`translation`* string.
`msgcat::mcunknown` *`locale src`*	This procedure is called to resolve unknown translations. Applications can provide their own implementations.

CHAPTER 16

Event-Driven Programming

This chapter describes event-driven programming using timers and asynchronous I/O facilities. The `after` command causes Tcl commands to occur at a time in the future, and the `fileevent` command registers a command to occur in response to file input/output (I/O). Tcl commands discussed are: `after`, `fblocked`, `fconfigure`, `fileevent`, and `vwait`.

Event-driven programming is used in long-running programs like network servers and graphical user interfaces. This chapter introduces event-driven programming in Tcl. Tcl provides an easy model in which you register Tcl commands, and the system then calls those commands when a particular event occurs. The `after` command is used to execute Tcl commands at a later time, and the `fileevent` command is used to execute Tcl commands when the system is ready for I/O. The `vwait` command is used to wait for events. During the wait, Tcl automatically calls Tcl commands that are associated with different events.

The event model is also used when programming user interfaces using Tk. Originally, event processing was associated only with Tk. The event loop moved from Tk to Tcl in the Tcl 7.5/Tk 4.1 release.

The Tcl Event Loop

An event loop is built into Tcl. Tcl checks for events and calls out to handlers that have been registered for different types of events. Some of the events are processed internally to Tcl. You can register Tcl commands to be called in response to events. There are also C APIs to event loop, which are described on page 689. Event processing is active all the time in Tk applications. If you do not use Tk, you can start the event loop with the `vwait` command as shown in Example 16–2 on page 220. The four event classes are handled in the following order:

- Window events. These include keystrokes and button clicks. Handlers are set up for these automatically by the Tk widgets, and you can register window event handlers with the `bind` command described in Chapter 26.
- File events. The `fileevent` command registers handlers for these events.
- Timer events. The `after` command registers commands to occur at specific times.
- Idle events. These events are processed when there is nothing else to do. The Tk widgets use idle events to display themselves. The `after idle` command registers a command to run at the next idle time.

The `after` Command

The `after` command sets up commands to happen in the future. In its simplest form, it pauses the application for a specified time, in milliseconds. The example below waits for half a second:

```
after 500
```

During this time, the application *does not* process events. You can use the `vwait` command as shown on page 220 to keep the Tcl event loop active during the waiting period. The `after` command can register a Tcl command to occur after a period of time, in milliseconds:

```
after milliseconds cmd arg arg...
```

The `after` command treats its arguments like `eval`; if you give it extra arguments, it concatenates them to form a single command. If your argument structure is important, use `list` to build the command. The following example always works, no matter what the value of `myvariable` is:

```
after 500 [list puts $myvariable]
```

The return value of `after` is an identifier for the registered command. You can cancel this command with the `after cancel` operation. You specify either the identifier returned from `after`, or the command string. In the latter case, the event that matches the command string exactly is canceled.

Table 16–1 summarizes the `after` command:

Table 16–1 The `after` command.

`after` *`milliseconds`*	Pauses for *`milliseconds`*.
`after` *`ms arg ?arg...?`*	Concatenates the *`args`* into a command and executes it after *`ms`* milliseconds. Immediately returns an ID.
`after cancel` *`id`*	Cancels the command registered under *`id`*.
`after cancel` *`command`*	Cancels the registered *`command`*.
`after idle` *`command`*	Runs *`command`* at the next idle moment.
`after info ?`*`id`*`?`	Returns a list of IDs for outstanding `after` events, or the command associated with *`id`*.

The fileevent Command

The fileevent command registers a procedure that is called when an I/O channel is ready for read or write events. For example, you can open a pipeline or network socket for reading, and then process the data from the pipeline or socket using a command registered with fileevent. The advantage of this approach is that your application can do other things, like update the user interface, while waiting for data from the pipeline or socket. Network servers use fileevent to manage connections to many clients. You can use fileevent on stdin and stdout, too. Using network sockets is described in Chapter 17.

The command registered with fileevent uses the regular Tcl commands to read or write data on the I/O channel. For example, if the pipeline generates line-oriented output, you should use gets to read a line of input. If you try and read more data than is available, your application may block waiting for more input. For this reason, you should read one line in your fileevent handler, assuming the data is line-oriented. If you know the pipeline will generate data in fixed-sized blocks, then you can use the read command to read one block.

The fconfigure command, which is described on page 221, can put a channel into nonblocking mode. This is not strictly necessary when using fileevent. The pros and cons of nonblocking I/O are discussed later.

End of file makes a channel readable.

You should check for end of file in your read handler because it will be called when end of file occurs. It is important to close the channel inside the handler because closing the channel automatically unregisters the handler. If you forget to close the channel, your read event handler will be called repeatedly.

Example 16–1 shows a read event handler. A pipeline is opened for reading and its command executes in the background. The Reader command is invoked when data is available on the pipe. When end of file is detected a variable is set, which signals the application waiting with vwait. Otherwise, a single line of input is read and processed. The vwait command is described on the next page. Example 22–1 on page 316 also uses fileevent to read from a pipeline.

Example 16–1 A read event file handler.

```
proc Reader { pipe } {
    global done
    if {[eof $pipe]} {
        catch {close $pipe}
        set done 1
        return
    }
    gets $pipe line
    # Process the line here...
}
set pipe [open "|some command"]
fileevent $pipe readable [list Reader $pipe]
vwait done
```

There can be at most one read handler and one write handler for an I/O channel. If you register a handler and one is already registered, then the old registration is removed. If you call `fileevent` without a command argument, it returns the currently registered command, or it returns the empty string if there is none. If you register the empty string, it deletes the current file handler. Table 16–2 summarizes the `fileevent` command.

Table 16–2 The `fileevent` command.

`fileevent` *`fileId`* `readable` `?`*`command`*`?`	Queries or registers *`command`* to be called when *`fileId`* is readable.
`fileevent` *`fileId`* `writable` `?`*`command`*`?`	Queries or registers *`command`* to be called when *`fileId`* is writable.

The `vwait` Command

The `vwait` command waits until a variable is modified. For example, you can set variable `x` at a future time, and then wait for that variable to be set with `vwait`.

```
set x 0
after 500 {set x 1}
vwait x
```

Waiting with `vwait` causes Tcl to enter the event loop. Tcl will process events until the variable `x` is modified. The `vwait` command completes when some Tcl code runs in response to an event and modifies the variable. In this case the event is a timer event, and the Tcl code is simply:

```
set x 1
```

In some cases `vwait` is used only to start the event loop. Example 16–2 sets up a file event handler for `stdin` that will read and execute commands. Once this is set up, `vwait` is used to enter the event loop and process commands until the input channel is closed. The process exits at that point, so the `vwait` variable `Stdin(wait)` is not used:

Example 16–2 Using `vwait` to activate the event loop.

```
proc Stdin_Start {prompt} {
    global Stdin
    set Stdin(line) ""
    puts -nonewline $prompt
    flush stdout
    fileevent stdin readable [list StdinRead $prompt]
    vwait Stdin(wait)
}
proc StdinRead {prompt} {
    global Stdin
    if {[eof stdin]} {
        exit
```

```
    }
    append Stdin(line) [gets stdin]
    if {[info complete $Stdin(line)]} {
        catch {uplevel #0 $Stdin(line)} result
        puts $result
        puts -nonewline $prompt
        flush stdout
        set Stdin(line) {}
    } else {
        append Stdin(line) \n
    }
}
```


The fconfigure Command

The fconfigure command sets and queries several properties of I/O channels. The default settings for channels are suitable for most cases. If you do event-driven I/O you may want to set your channel into nonblocking mode. If you handle binary data, you should turn off end of line and character set translations. You can query the channel parameters like this:

```
fconfigure stdin
-blocking 1 -buffering none -buffersize 4096 -encoding
iso8859-1 -eofchar {} -translation lf
```

Table 16–3 summarizes the properties controlled by fconfigure. They are discussed in more detail below.

Table 16–3 I/O channel properties controlled by fconfigure.

-blocking	Blocks until I/O channel is ready: 0 or 1.
-buffering	Buffer mode: none, line, or full.
-buffersize	Number of characters in the buffer.
-eofchar	Special end of file character. Control-z (\x1a) for DOS. Null otherwise.
-encoding	The character set encoding.
-error	Returns the last POSIX error message associated with a channel.
-translation	End of line translation: auto, lf, cr, crlf, binary.
-mode	Serial devices only. Format: *baud*,*parity*,*data*,*stop*
-peername	Sockets only. IP address of remote host.
-peerport	Sockets only. Port number of remote host.

Nonblocking I/O

By default, I/O channels are *blocking*. A gets or read will wait until data is available before returning. A puts may also wait if the I/O channel is not ready to

accept data. This behavior is all right if you are using disk files, which are essentially always ready. If you use pipelines or network sockets, however, the blocking behavior can hang up your application.

The `fconfigure` command can set a channel into *nonblocking mode*. A `gets` or `read` command may return immediately with no data. This occurs when there is no data available on a socket or pipeline. A `puts` to a nonblocking channel will accept all the data and buffer it internally. When the underlying device (i.e., a pipeline or socket) is ready, then Tcl automatically writes out the buffered data. Nonblocking channels are useful because your application can do something else while waiting for the I/O channel. You can also manage several nonblocking I/O channels at once. Nonblocking channels should be used with the `fileevent` command described earlier. The following command puts a channel into nonblocking mode:

```
fconfigure fileID -blocking 0
```

It is not strictly necessary to put a channel into nonblocking mode if you use `fileevent`. However, if the channel is in blocking mode, then it is still possible for the `gets` or `read` done by your `fileevent` procedure to block. For example, an I/O channel might have some data ready, but not a complete line. In this case, a `gets` would block, unless the channel is nonblocking. Perhaps the best motivation for a nonblocking channel is the buffering behavior of a nonblocking `puts`. You can even `close` a channel that has buffered data, and Tcl will automatically write out the buffers as the channel becomes ready. For these reasons, it is common to use a nonblocking channel with `fileevent`. Example 16–3 shows a `fileevent` handler for a nonblocking channel. As described above, the `gets` may not find a complete line, in which case it doesn't read anything and returns -1.

Example 16–3 A read event file handler for a nonblocking channel.

```
set pipe [open "|some command"]
fileevent $pipe readable [list Reader $pipe]
fconfigure $pipe -blocking 0
proc Reader { pipe } {
    global done
    if {[eof $pipe]} {
        catch {close $pipe}
        set done 1
        return
    }
    if {[gets $pipe line] < 0} {
        # We blocked anyway because only part of a line
        # was available for input
    } else {
        # Process one line
    }
}
vwait done
```

The fblocked Command

The fblocked command returns 1 if a channel does not have data ready. Normally the fileevent command takes care of waiting for data, so I have seen fblocked useful only in testing channel implementations.

Buffering

By default, Tcl buffers data, so I/O is more efficient. The underlying device is accessed less frequently, so there is less overhead. In some cases you may want data to be visible immediately and buffering gets in the way. The following turns off all buffering:

```
fconfigure fileID -buffering none
```

Full buffering means that output data is accumulated until a buffer fills; then a write is performed. For reading, Tcl attempts to read a whole buffer each time more data is needed. The read-ahead for buffering will not block. The -buffersize parameter controls the buffer size:

```
fconfigure fileID -buffering full -buffersize 8192
```

Line buffering is used by default on stdin and stdout. Each newline in an output channel causes a write operation. Read buffering is the same as full buffering. The following command turns on line buffering:

```
fconfigure fileID -buffering line
```

End of Line Translations

On UNIX, text lines end with a newline character (\n). On Macintosh they end with a carriage return (\r). On Windows they end with a carriage return, newline sequence (\r\n). Network sockets also use the carriage return, newline sequence. By default, Tcl accepts any of these, and the line terminator can even change within a channel. All of these different conventions are converted to the UNIX style so that once read, text lines always end with a newline character (\n). Both the read and gets commands do this conversion. By default, text lines are generated in the platform-native format during output.

The default behavior is almost always what you want, but you can control the translation with fconfigure. Table 16–4 shows settings for -translation:

Table 16–4 End of line translation modes.

binary	No translation at all.
lf	UNIX-style, which also means no translations.
cr	Macintosh style. On input, carriage returns are converted to newlines. On output, newlines are converted to carriage returns.
crlf	Windows and Network style. On input, carriage return, newline is converted to a newline. On output, a newline is converted to a carriage return, newline.
auto	The default behavior. On input, all end of line conventions are converted to a newline. Output is in native format.

End of File Character

In DOS file systems, there may be a Control-z character (`\x1a`) at the end of a text file. By default, this character is ignored on the Windows platform if it occurs at the end of the file, and this character is output when you close the file. You can turn this off by specifying an empty string for the end of file character:

```
fconfigure fileID -eofchar {}
```

Serial Devices

The `-mode` attribute specifies the baud rate, parity mode, the number of data bits, and the number of stop bits:

```
set tty [open /dev/ttya]
fconfigure $tty -mode
=> 9600,0,8,2
```

If you need to set additional attributes of the serial channel, you will have to write a command in C that makes the system calls you need. If you are familiar with serial devices, you know there are quite a few possibilities!

Windows has some special device names that always connect you to the serial line devices when you use `open`. They are `com1` through `com8`. The system console is named `con`. The null device is `nul`.

UNIX has names for serial devices in `/dev`. The serial devices are `/dev/ttya`, `/dev/ttyb`, and so on. The system console is `/dev/console`. The current terminal is `/dev/tty`. The null device is `/dev/null`.

Macintosh needs a special command to open serial devices. This is provided by a third-party extension that you can find at the Tcl Resource Center under:

```
http://www.scriptics.com/resource/software/extensions/macintosh/
```

Character Set Encodings

Tcl automatically converts various character set encodings into Unicode internally. It cannot automatically detect the encoding for a file or network socket, however, so you need to use `fconfigure -encoding` if you are reading data that is not in the system's default encoding. Character set issues are explained in more detail in Chapter 15.

Configuring Read-Write Channels

If you have a channel that is used for both input and output, you can set the channel parameters independently for input and output. In this case, you can specify a two-element list for the parameter value. The first element is for the input side of the channel, and the second element is for the output side of the channel. If you specify only a single element, it applies to both input and output. For example, the following command forces output end of line translations to be `crlf` mode, leaves the input channel on automatic, and sets the buffer size for both input and output:

```
fconfigure pipe -translation {auto crlf} -buffersize 4096
```

CHAPTER 17

Socket Programming

This chapter shows how to use sockets for programming network clients and servers. Advanced I/O techniques for sockets are described, including nonblocking I/O and control over I/O buffering. Tcl commands discussed are: `socket`, `fconfigure`, and `http::geturl`.

Sockets are network communication channels. The sockets described in this chapter use the TCP network protocol, although you can find Tcl extensions that create sockets using other protocols. TCP provides a reliable byte stream between two hosts connected to a network. TCP handles all the issues about routing information across the network, and it automatically recovers if data is lost or corrupted along the way. TCP is the basis for other protocols like Telnet, FTP, and HTTP.

A Tcl script can use a network socket just like an open file or pipeline. Instead of using the Tcl `open` command, you use the `socket` command to open a socket. Then you use `gets`, `puts`, and `read` to transfer data. The `close` command closes a network socket.

Network programming distinguishes between clients and servers. A server is a process or program that runs for long periods of time and controls access to some resource. For example, an FTP server governs access to files, and an HTTP server provides access to hypertext pages on the World Wide Web. A client typically connects to the server for a limited time in order to gain access to the resource. For example, when a Web browser fetches a hypertext page, it is acting as a client. The extended examples in this chapter show how to program the client side of the HTTP protocol.

The Scotty extension supports many network protocols.

The Scotty Tcl extension provides access to other network protocols like UDP, DNS, and RPC. It also supports the SNMP network management protocol and the MIB database associated with SNMP. Scotty is a great extension pack-

age that is widely used for network management applications. Its home page is:

```
http://wwwsnmp.cs.utwente.nl/~schoenw/scotty/
```

Client Sockets

A client opens a socket by specifying the *host address* and *port number* for the server of the socket. The host address gives the network location (i.e., which computer), and the port selects a particular server from all the possible servers that may be running on that host. For example, HTTP servers typically use port 80, while FTP servers use port 20. The following example shows how to open a client socket to a Web server:

```
set s [socket www.scriptics.com 80]
```

There are two forms for host names. The previous example uses a *domain name*: `www.scriptics.com`. You can also specify raw IP addresses, which are specified with four dot-separated integers (e.g., `128.15.115.32`). A domain name is mapped into a raw IP address by the system software, and it is almost always a better idea to use a domain name in case the IP address assignment for the host changes. This can happen when hosts are upgraded or they move to a different part of the network. As of Tcl 8.2, there is no direct access from Tcl to the DNS service that maps host names to IP addresses. You'll need to use Scotty to get DNS access.

Some systems also provide symbolic names for well-known port numbers. For example, instead of using `20` for the FTP service, you can use `ftp`. On UNIX systems, the well-known port numbers are listed in the file named `/etc/services`.

Client Socket Options

The `socket` command accepts some optional arguments when opening the client-side socket. The general form of the command is:

```
socket ?-async? ?-myaddr address? ?-myport myport? host port
```

Ordinarily the address and port on the client side are chosen automatically. If your computer has multiple network interfaces, you can select one with the `-myaddr` option. The *`address`* value can be a domain name or an IP address. If your application needs a specific client port, it can choose one with the `-myport` option. If the port is in use, the `socket` command will raise an error.

The `-async` option causes connection to happen in the background, and the `socket` command returns immediately. The socket becomes writable when the connection completes, or fails. You can use `fileevent` to get a callback when this occurs. This is shown in Example 17–1. If you use the socket before the connection completes, and the socket is in blocking mode, then Tcl automatically blocks and waits for the connection to complete. If the socket is in nonblocking mode, attempts to use the socket return immediately. The `gets` and `read` commands would return -1, and `fblocked` would return 1 in this situation.

In some cases, it can take a long time to open the connection to the server. Usually this occurs when the server host is down, and it may take longer than

you want for the connection to time out. The following example sets up a timer with `after` so that you can choose your own timeout limit on the connection:

Example 17–1 Opening a client socket with a timeout.

```
proc Socket_Client {host port timeout} {
   global connected
   after $timeout {set connected timeout}
   set sock [socket -async $host $port]
   fileevent $sock w {set connected ok}
   vwait connected
   if {$connected == "timeout"} {
      return -code error timeout
   } else {
      return $sock
   }
}
```


Server Sockets

A TCP server socket allows multiple clients. The way this works is that the `socket` command creates a *listening socket*, and then new sockets are created when clients make connections to the server. Tcl takes care of all the details and makes this easy to use. You simply specify a port number and give the socket command a *callback* to execute when a client connects to your server socket. The callback is just a Tcl command. A simple example is shown below:

Example 17–2 Opening a server socket.

```
set listenSocket [socket -server Accept 2540]
proc Accept {newSock addr port} {
   puts "Accepted $newSock from $addr port $port"
}
vwait forever
```

The `Accept` command is the callback. With server sockets, Tcl adds additional arguments to the callback before it calls it. The arguments are the new socket connection, and the host and port number of the remote client. In this simple example, `Accept` just prints out its arguments.

The `vwait` command puts Tcl into its event loop so that it can do the background processing necessary to accept connections. The `vwait` command will wait until the `forever` variable is modified, which won't happen in this simple example. The key point is that Tcl processes other events (e.g., network connections and other file I/O) while it waits. If you have a Tk application (e.g., *wish*), then it already has an event loop to handle window system events, so you do not need to use `vwait`. The Tcl event loop is discussed on page 217.

Server Socket Options

By default, Tcl lets the operating system choose the network interface used for the server socket, and you simply supply the port number. If your computer has multiple interfaces, you may want to specify a particular one. Use the `-myaddr` option for this. The general form of the command to open server sockets is:

```
socket -server callback ?-myaddr address? port
```

The last argument to the `socket` command is the server's port number. For your own unofficial servers, you'll need to pick port numbers higher than 1024 to avoid conflicts with existing services. UNIX systems prevent user programs from opening server sockets with port numbers less than 1024. If you use 0 as the port number, then the operating system will pick the listening port number for you. You must use `fconfigure` to find out what port you have:

```
fconfigure $sock -sockname
=> ipaddr hostname port
```

The Echo Service

Example 17–3 The echo service.

```
proc Echo_Server {port} {
    global echo
    set echo(main) [socket -server EchoAccept $port]
}
proc EchoAccept {sock addr port} {
    global echo
    puts "Accept $sock from $addr port $port"
    set echo(addr,$sock) [list $addr $port]
    fconfigure $sock -buffering line
    fileevent $sock readable [list Echo $sock]
}
proc Echo {sock} {
    global echo
    if {[eof $sock] || [catch {gets $sock line}]} {
        # end of file or abnormal connection drop
        close $sock
        puts "Close $echo(addr,$sock)"
        unset echo(addr,$sock)
    } else {
        if {[string compare $line "quit"] == 0} {
            # Prevent new connections.
            # Existing connections stay open.
            close $echo(main)
        }
        puts $sock $line
    }
}
```

The echo server accepts connections from clients. It reads data from the clients and writes that data back. The example uses `fileevent` to wait for data from the client, and it uses `fconfigure` to adjust the buffering behavior of the network socket. You can use Example 17–3 as a template for more interesting services.

The `Echo_Server` procedure opens the socket and saves the result in `echo(main)`. When this socket is closed later, the server stops accepting new connections but existing connections won't be affected. If you want to experiment with this server, start it and wait for connections like this:

```
Echo_Server 2540
vwait forever
```

The `EchoAccept` procedure uses the `fconfigure` command to set up line buffering. This means that each `puts` by the server results in a network transmission to the client. The importance of this will be described in more detail later. A complete description of the `fconfigure` command is given in Chapter 16. The `EchoAccept` procedure uses the `fileevent` command to register a procedure that handles I/O on the socket. In this example, the `Echo` procedure will be called whenever the socket is readable. Note that it is not necessary to put the socket into nonblocking mode when using the `fileevent` callback. The effects of nonblocking mode are discussed on page 221.

`EchoAccept` saves information about each client in the `echo` array. This is used only to print out a message when a client closes its connection. In a more sophisticated server, however, you may need to keep more interesting state about each client. The name of the socket provides a convenient handle on the client. In this case, it is used as part of the array index.

The `Echo` procedure first checks to see whether the socket has been closed by the client or there is an error when reading the socket. The `if` expression only performs the `gets` if the `eof` does not return true:

```
if {[eof $sock] || [catch {gets $sock line}]} {
```

Closing the socket automatically clears the `fileevent` registration. If you forget to close the socket upon the end of file condition, the Tcl event loop will invoke your callback repeatedly. It is important to close it when you detect end of file.

Example 17–4 A client of the echo service.

```
proc Echo_Client {host port} {
    set s [socket $host $port]
    fconfigure $s -buffering line
    return $s
}
set s [Echo_Client localhost 2540]
puts $s "Hello!"
gets $s
=> Hello!
```

In the normal case, the server simply reads a line with `gets` and then writes it back to the client with `puts`. If the line is "quit," then the server closes its main socket. This prevents any more connections by new clients, but it doesn't affect any clients that are already connected.

Example 17–4 shows a sample client of the `Echo` service. The main point is to ensure that the socket is line buffered so that each `puts` by the client results in a network transmission. (Or, more precisely, each newline character results in a network transmission.) If you forget to set line buffering with `fconfigure`, the client's `gets` command will probably hang because the server will not get any data; it will be stuck in buffers on the client.

Fetching a URL with HTTP

The HyperText Transport Protocol (HTTP) is the protocol used on the World Wide Web. This section presents a procedure to fetch pages or images from a server on the Web. Items in the Web are identified with a Universal Resource Location (URL) that specifies a host, port, and location on the host. The basic outline of HTTP is that a client sends a URL to a server, and the server responds with some header information and some content data. The header information describes the content, which can be hypertext, images, postscript, and more.

Example 17–5 Opening a connection to an HTTP server.

```
proc Http_Open {url} {
    global http
    if {![regexp -nocase {^(http://)?([^:/]+)(:([0-9])+)?(/.*)} \
            $url x protocol server y port path]} {
        error "bogus URL: $url"
    }
    if {[string length $port] == 0} {
        set port 80
    }
    set sock [socket $server $port]
    puts $sock "GET $path HTTP/1.0"
    puts $sock "Host: $server"
    puts $sock "User-Agent: Tcl/Tk Http_Open"
    puts $sock ""
    flush $sock
    return $sock
}
```

The `Http_Open` procedure uses `regexp` to pick out the server and port from the URL. This regular expression is described in detail on page 149. The leading `http://` is optional, and so is the port number. If the port is left off, then the standard port 80 is used. If the regular expression matches, then a `socket` command opens the network connection.

The protocol begins with the client sending a line that identifies the com-

mand (GET), the path, and the protocol version. The path is the part of the URL after the server and port specification. The rest of the request is lines in the following format:

key: *value*

The Host identifies the server, which supports servers that implement more than one server name. The User-Agent identifies the client program, which is often a browser like *Netscape Navigator* or *Internet Explorer*. The key-value lines are terminated with a blank line. This data is flushed out of the Tcl buffering system with the flush command. The server will respond by sending the URL contents back over the socket. This is described shortly, but first we consider proxies.

Proxy Servers

A *proxy* is used to get through firewalls that many organizations set up to isolate their network from the Internet. The proxy accepts HTTP requests from clients inside the firewall and then forwards the requests outside the firewall. It also relays the server's response back to the client. The protocol is nearly the same when using the proxy. The difference is that the complete URL is passed to the GET command so that the proxy can locate the server. Example 17–6 uses a proxy if one is defined:

Example 17–6 Opening a connection to an HTTP server.

```
# Http_Proxy sets or queries the proxy
proc Http_Proxy {{new {}}} {
    global http
    if ![info exists http(proxy)] {
        return {}
    }
    if {[string length $new] == 0} {
        return $http(proxy):$http(proxyPort)
    } else {
        regexp {^([^:]+):([0-9]+)$} $new x \
            http(proxy) http(proxyPort)
    }
}

proc Http_Open {url {cmd GET} {query {}}} {
    global http
    if {![regexp -nocase {^(http://)?([^:/]+)(:([0-9])+)?(/.*)} \
            $url x protocol server y port path]} {
        error "bogus URL: $url"
    }
    if {[string length $port] == 0} {
        set port 80
    }
    if {[info exists http(proxy)] &&
            [string length $http(proxy)]} {
        set sock [socket $http(proxy) $http(proxyPort)]
```

```
        puts $sock "$cmd http://$server:$port$path HTTP/1.0"
    } else {
        set sock [socket $server $port]
        puts $sock "$cmd $path HTTP/1.0"
    }
    puts $sock "User-Agent: Tcl/Tk Http_Open"
    puts $sock "Host: $server"
    if {[string length $query] > 0} {
        puts $sock "Content-Length: [string length $query]"
        puts $sock ""
        puts $sock $query
    }
    puts $sock ""
    flush $sock
    fconfigure $sock -blocking 0
    return $sock
}
```

The HEAD Request

In Example 17–6, the `Http_Open` procedure takes a `cmd` parameter so that the user of `Http_Open` can perform different operations. The `GET` operation fetches the contents of a URL. The `HEAD` operation just fetches the description of a URL, which is useful to validate a URL. The `POST` operation transmits query data to the server (e.g., values from a form) and also fetches the contents of the URL. All of these operations follow a similar protocol. The reply from the server is a status line followed by lines that have key-value pairs. This format is similar to the client's request. The reply header is followed by content data with `GET` and `POST` operations. Example 17–7 implements the `HEAD` command, which does not involve any reply data:

Example 17–7 `Http_Head` validates a URL.

```
proc Http_Head {url} {
    upvar #0 $url state
    catch {unset state}
    set state(sock) [Http_Open $url HEAD]
    fileevent $state(sock) readable [list HttpHeader $url]
    # Specify the real name, not the upvar alias, to vwait
    vwait $url\(status)
    catch {close $state(sock)}
    return $state(status)
}
proc HttpHeader {url} {
    upvar #0 $url state
    if {[eof $state(sock)]} {
        set state(status) eof
        close $state(sock)
        return
    }
    if {[catch {gets $state(sock) line} nbytes]} {
```

```
            set state(status) error
            lappend state(headers) [list error $nbytes]
            close $state(sock)
            return
        }
        if {$nbytes < 0} {
            # Read would block
            return
        } elseif {$nbytes == 0} {
            # Header complete
            set state(status) head
        } elseif {![info exists state(headers)]} {
            # Initial status reply from the server
            set state(headers) [list http $line]
        } else {
            # Process key-value pairs
            regexp {^([^:]+): *(.*)$} $line x key value
            lappend state(headers) [string tolower $key] $value
        }
    }
```


The Http_Head procedure uses Http_Open to contact the server. The HttpHeader procedure is registered as a fileevent handler to read the server's reply. A global array keeps state about each operation. The URL is used in the array name, and upvar is used to create an alias to the name (upvar is described on page 86):

```
upvar #0 $url state
```

You cannot use the upvar alias as the variable specified to vwait. Instead, you must use the actual name. The backslash turns off the array reference in order to pass the name of the array element to vwait, otherwise Tcl tries to reference url as an array:

```
vwait $url\(status)
```

The HttpHeader procedure checks for special cases: end of file, an error on the gets, or a short read on a nonblocking socket. The very first reply line contains a status code from the server that is in a different format than the rest of the header lines:

```
code message
```

The code is a three-digit numeric code. 200 is OK. Codes in the 400's and 500's indicate an error. The codes are explained fully in RFC 1945 that specifies HTTP 1.0. The first line is saved with the key http:

```
set state(headers) [list http $line]
```

The rest of the header lines are parsed into key-value pairs and appended onto state(headers). This format can be used to initialize an array:

```
array set header $state(headers)
```

When HttpHeader gets an empty line, the header is complete and it sets the state(status) variable, which signals Http_Head. Finally, Http_Head returns the status to its caller. The complete information about the request is still in the global array named by the URL. Example 17–8 illustrates the use of Http_Head:

Example 17–8 Using `Http_Head`.

```
set url http://www.sun.com/
set status [Http_Head $url]
=> eof
upvar #0 $url state
array set info $state(headers)
parray info
info(http)          HTTP/1.0 200 OK
info(server)        Apache/1.1.1
info(last-modified) Nov ...
info(content-type)  text/html
```

The GET and POST Requests

Example 17–9 shows `Http_Get`, which implements the `GET` and `POST` requests. The difference between these is that `POST` sends query data to the server after the request header. Both operations get a reply from the server that is divided into a descriptive header and the content data. The `Http_Open` procedure sends the request and the query, if present, and reads the reply header. `Http_Get` reads the content.

The descriptive header returned by the server is in the same format as the client's request. One of the key-value pairs returned by the server specifies the `Content-Type` of the URL. The content-types come from the MIME standard, which is described in RFC 1521. Typical content-types are:

- `text/html` — HyperText Markup Language (HTML), which is introduced in Chapter 3.
- `text/plain` — plain text with no markup.
- `image/gif` — image data in GIF format.
- `image/jpeg` — image data in JPEG format.
- `application/postscript` — a postscript document.
- `application/x-tcl` — a Tcl program! This type is discussed in Chapter 20.

Example 17–9 `Http_Get` fetches the contents of a URL.

```
proc Http_Get {url {query {}}} {
   upvar #0 $url state        ;# Alias to global array
   catch {unset state}        ;# Aliases still valid.
   if {[string length $query] > 0} {
      set state(sock) [Http_Open $url POST $query]
   } else {
      set state(sock) [Http_Open $url GET]
   }
   set sock $state(sock)
   fileevent $sock readable [list HttpHeader $url]

   # Specify the real name, not the upvar alias, to vwait
   vwait $url\(status)
```

```
        set header(content-type) {}
        set header(http) "500 unknown error"
        array set header $state(headers)

        # Check return status.
        # 200 is OK, other codes indicate a problem.
        regsub "HTTP/1.. " $header(http) {} header(http)
        if {![string match 2* $header(http)]} {
            catch {close $sock}
            if {[info exists header(location)] &&
                    [string match 3* $header(http)]} {
                # 3xx is a redirection to another URL
                set state(link) $header(location)
                return [Http_Get $header(location) $query]
            }
            return -code error $header(http)
        }
        # Set up to read the content data
        switch -glob -- $header(content-type) {
            text/*    {
                # Read HTML into memory
                fileevent $sock readable [list HttpGetText $url]
            }
            default   {
                # Copy content data to a file
                fconfigure $sock -translation binary
                set state(filename) [File_TempName http]
                if [catch {open $state(filename) w} out] {
                    set state(status) error
                    set state(error) $out
                    close $sock
                    return $header(content-type)
                }
                set state(fd) $out
                fcopy $sock $out -command [list HttpCopyDone $url]
            }
        }
        vwait $url\(status)
        return $header(content-type)
    }
```

`Http_Get` uses `Http_Open` to initiate the request, and then it looks for errors. It handles redirection errors that occur if a URL has changed. These have error codes that begin with `3`. A common case of this error is when a user omits the trailing slash on a URL (e.g., `http://www.scriptics.com`). Most servers respond with:

```
302 Document has moved
Location: http://www.scriptics.com/
```

If the `content-type` is text, then `Http_Get` sets up a `fileevent` handler to read this data into memory. The socket is in nonblocking mode, so the read handler can read as much data as possible each time it is called. This is more efficient than using `gets` to read a line at a time. The text will be stored in the

state(body) variable for use by the caller of Http_Get. Example 17–10 shows the HttpGetText fileevent handler:

Example 17–10 HttpGetText reads text URLs.

```
proc HttpGetText {url} {
    upvar #0 $url state
    if {[eof $state(sock)]} {
        # Content complete
        set state(status) done
        close $state(sock)
    } elseif {[catch {read $state(sock)} block]} {
        set state(status) error
        lappend state(headers) [list error $block]
        close $state(sock)
    } else {
        append state(body) $block
    }
}
```

The content may be in binary format. This poses a problem for Tcl 7.6 and earlier. A null character will terminate the value, so values with embedded nulls cannot be processed safely by Tcl scripts. Tcl 8.0 supports strings and variable values with arbitrary binary data. Example 17–9 uses fcopy to copy data from the socket to a file without storing it in Tcl variables. This command was introduced in Tcl 7.5 as unsupported0, and became fcopy in Tcl 8.0. It takes a callback argument that is invoked when the copy is complete. The callback gets additional arguments that are the bytes transferred and an optional error string. In this case, these arguments are added to the url argument specified in the fcopy command. Example 17–11 shows the HttpCopyDone callback:

Example 17–11 HttpCopyDone is used with fcopy.

```
proc HttpCopyDone {url bytes {error {}}} {
    upvar #0 $url state
    if {[string length $error]} {
        set state(status) error
        lappend state(headers) [list error $error]
    } else {
        set state(status) ok
    }
    close $state(sock)
    close $state(fd)
}
```

The user of Http_Get uses the information in the state array to determine the status of the fetch and where to find the content. There are four cases to deal with:

- There was an error, which is indicated by the `state(error)` element.
- There was a redirection, in which case, the new URL is in `state(link)`. The client of `Http_Get` should change the URL and look at its state instead. You can use `upvar` to redefine the alias for the `state` array:

```
upvar #0 $state(link) state
```

- There was text content. The content is in `state(body)`.
- There was another content-type that was copied to `state(filename)`.

The `fcopy` Command

The `fcopy` command can do a complete copy in the background. It automatically sets up `fileevent` handlers, so you do not have to use `fileevent` yourself. It also manages its buffers efficiently. The general form of the command is:

```
fcopy input output ?-size size? ?-command callback?
```

The `-command` argument makes `fcopy` work in the background. When the copy is complete or an error occurs, the *`callback`* is invoked with one or two additional arguments: the number of bytes copied, and, in the case of an error, it is also passed an error string:

```
fcopy $in $out -command [list CopyDone $in $out]
proc CopyDone {in out bytes {error {}} {
    close $in ; close $out
}
```

With a background copy, the `fcopy` command transfers data from *`input`* until end of file or *`size`* bytes have been transferred. If no `-size` argument is given, then the copy goes until end of file. It is not safe to do other I/O operations with *`input`* or *`output`* during a background `fcopy`. If either *`input`* or *`output`* gets closed while the copy is in progress, the current copy is stopped. If the *`input`* is closed, then all data already queued for *`output`* is written out.

Without a `-command` argument, the `fcopy` command reads as much as possible depending on the blocking mode of *`input`* and the optional *`size`* parameter. Everything it reads is queued for output before `fcopy` returns. If *`output`* is blocking, then `fcopy` returns after the data is written out. If *`input`* is blocking, then `fcopy` can block attempting to read *`size`* bytes or until end of file.

The `http` Package

The standard Tcl library includes an `http` package that is based on the code I wrote for this chapter. This section documents the package, which has a slightly different interface. The library version uses namespaces and combines the `Http_Get`, `Http_Head`, and `Http_Post` procedures into a single `http::geturl` procedure. The examples in this chapter are still interesting, but you should look at `http.tcl` in the Tcl library, which I also wrote. Definitely use the standard `http` package for your production code.

http::config

The http::config command is used to set the proxy information, timeouts, and the User-Agent and Accept headers that are generated in the HTTP request. You can specify the proxy host and port, or you can specify a Tcl command that is run to determine the proxy. With no arguments, http::config returns the current settings:

```
http::config
=> -accept */* -proxyfilter httpProxyRequired -proxyhost
{} -proxyport {} -timeout unlimited
-useragent {Tcl http client package 2.0}
```

If you specify just one option, its value is returned:

```
http::config -proxyfilter
=> httpProxyRequired
```

You can set one or more options:

```
http::config -proxyhost webcache.eng -proxyport 8080
```

The default proxy filter just returns the -proxyhost and -proxyport values if they are set. You can supply a smarter filter that picks a proxy based on the host in the URL. The proxy filter is called with the hostname and should return a list of two elements, the proxy host and port. If no proxy is required, return an empty list.

The -timeout value limits the time the transaction can take. Its value is unlimited for no timeout, or a milliseconds value. You can specify 500, for example, to have a half-second timeout.

http::geturl

The http::geturl procedure does a GET, POST, or HEAD transaction depending on its arguments. By default, http::geturl blocks until the request completes and it returns a token that represents the transaction. As described below, you use the token to get the results of the transaction. If you supply a -command *callback* option, then http::geturl returns immediately and invokes *callback* when the transaction completes. The callback is passed the token that represents the transaction. Table 17–1 lists the options to http::geturl:

Table 17–1 Options to the http::geturl command.

-blocksize *num*	Block size when copying to a channel.
-channel *fileID*	The *fileID* is an open file or socket. The URL data is copied to this channel instead of saving it in memory.
-command *callback*	Calls *callback* when the transaction completes. The token from http::geturl is passed to *callback*.
-handler *command*	Called from the event handler to read data from the URL.

Table 17–1 Options to the `http::geturl` command. (Continued)

`-headers` *list*	The *list* specifies a set of headers that are included in the HTTP request. The list alternates between header keys and values.
`-progress` *command*	Calls *command* after each block is copied to a channel. It gets called with three parameters: *command token totalsize currentsize*
`-query` *codedstring*	Issues a POST request with the *codedstring* form data.
`-timeout` *msec*	Aborts the request after *msec* milliseconds have elapsed.
`-validate` *bool*	If *bool* is true, a HEAD request is made.

For simple applications you can simply block on the transaction:

```
set token [http::geturl www.beedub.com/index.html]
=> http::1
```

The leading `http://` in the URL is optional. The return value is a token that is also the name of a global array that contains state about the transaction. Names like `http::1` are used instead of using the URL as the array name. You can use `upvar` to convert the return value from `http::geturl` to an array variable:

```
upvar #0 $token state
```

By default, the URL data is saved in `state(body)`. The elements of the state array are described in Table 17–2:

Table 17–2 Elements of the `http::geturl` state array.

`body`	The contents of the URL.
`currentsize`	The current number of bytes transferred.
`error`	An explanation of why the transaction was aborted.
`http`	The HTTP reply status.
`meta`	A list of the keys and values in the reply header.
`status`	The current status: `pending`, `ok`, `eof`, or `reset`.
`totalsize`	The expected size of the returned data.
`type`	The content type of the returned data.
`url`	The URL of the request.

A handful of access functions are provided so that you can avoid using the state array directly. These are listed in Table 17–3:

Table 17–3 The `http` support procedures.

`http::data $token`	Returns `state(body)`.
`http::status $token`	Returns `state(status)`.
`http::error $token`	Returns `state(error)`.
`http::code $token`	Returns `state(http)`.
`http::wait $token`	Blocks until the transaction completes.
`http::cleanup $token`	Unsets the state array named by `$token`.

You can take advantage of the asynchronous interface by specifying a command that is called when the transaction completes. The callback is passed the token returned from `http::geturl` so that it can access the transaction state:

```
http::geturl $url -command [list Url_Display $text $url]
proc Url_Display {text url token} {
    upvar #0 $token state
    # Display the url in text
}
```

You can have `http::geturl` copy the URL to a file or socket with the `-channel` option. This is useful for downloading large files or images. In this case, you can get a progress callback so that you can provide user feedback during the transaction. Example 17–12 shows a simple downloading script:

Example 17–12 Downloading files with `http::geturl`.

```
#!/usr/local/tclsh8.0
if {$argc < 2} {
    puts stderr "Usage: $argv0 url file"
    exit 1
}
package require http
set url [lindex $argv 0]
set file [lindex $argv 1]
set out [open $file w]
proc progress {token total current} {
    puts -nonewline "."
}
http::config -proxyhost webcache.eng -proxyport 8080
set token [http::geturl $url -progress progress \
    -headers {Pragma no-cache} -channel $out]
close $out
# Print out the return header information
puts ""
upvar #0 $token state
puts $state(http)
foreach {key value} $state(meta) {
    puts "$key: $value"
}
exit 0
```

http::formatQuery

If you specify form data with the `-query` option, then `http::geturl` does a `POST` transaction. You need to encode the form data for safe transmission. The `http::formatQuery` procedure takes a list of keys and values and encodes them in `x-www-url-encoded` format. Pass this result as the query data:

```
http::formatQuery name "Brent Welch" title "Tcl Programmer"
=> name=Brent+Welch&title=Tcl+Programmer
```

http::reset

You can cancel an outstanding transaction with `http::reset`:

```
http::reset $token
```

This is done automatically when you setup a `-timeout` with `http::config`.

http::cleanup

When you are done with the data returned from `http::geturl`, use the `http::cleanup` procedure to unset the state variable used to store the data.

Basic Authentication

Web pages are often password protected. The most common form of this uses a protocol called Basic Authentication, which is not very strong, but easy to implement. With this scheme, the server responds to an HTTP request with a 401 error status and a `Www-Authenticate` header, which specifies the authentication protocol the server wants to use. For example, the server response can contain the following information:

```
HTTP/1.0 401 Authorization Required
Www-Authenticate: Basic realm="My Pages"
```

The *realm* is meant to be an authentication domain. In practice, it is used in the string that gets displayed to the user as part of the password prompt. For example, a Web browser will display this prompt:

```
Enter the password for My Pages at www.beedub.com
```

After getting the user name and password from the user, the Web browser tries its HTTP request again. This time it includes an `Authorization` header that contains the user name and password encoded with *base64 encoding*. There is no encryption at all — anyone can decode the string, which is why this is not a strong form of protection. The Tcl Web Server includes a `base64.tcl` file that has `Base64_Encode` and `Base64_Decode` procedures. Example 17–13 illustrates the Basic Authentication protocol. `http::geturl` takes a `-headers` option that lets you pass additional headers in the request.

Example 17–13 Basic Authentication using `http::geturl`.

```
proc BasicAuthentication {url promptProc} {
    set token [http::geturl $url]
    http::wait $token
    if {[string match *401* [http::code $token]]} {
        upvar #0 $token data

        # Extract the realm from the Www-Authenticate line

        array set reply $data(meta)
        if {[regexp {realm=(.*)} $reply(Www-Authenticate) \
                x realm]} {

            # Call back to prompt for username, password

            set answer [$promptProc $realm]
            http::cleanup $token

            # Encode username:password and pass this in
            # the Authorization header

            set auth [Base64_Encode \
                [lindex $answer 0]:[lindex $answer 1]]
            set token [http::geturl $url -headers \
                [list Authorization "Basic $auth"]]
            http::wait $token
        }
    }
    return $token
}
```

Example 17–13 takes a `promptProc` argument that is the name of a procedure to call to get the username and password. This procedure could display a Tk dialog box, or prompt for user input from the terminal. In practice, you probably already know the username and password. In this case, you can skip the initial challenge–response steps and simply supply the `Authorization` header on the first request:

```
http::geturl $url -headers \
    [list Authorization \
        "Basic [Base64_Encode $username:$password]"]
```

CHAPTER 18

TclHttpd Web Server

This chapter describes TclHttpd, a Web server built entirely in Tcl. The Web server can be used as a standalone server, or it can be embedded into applications to Web-enable them. TclHttpd provides a Tcl+HTML template facility that is useful for maintaining site-wide look and feel, and an application-direct URL that invokes a Tcl procedure in an application.

TclHttpd started out as about 175 lines of Tcl that could serve up HTML pages and images. The Tcl `socket` and I/O commands make this easy. Of course, there are lots of features in Web servers like Apache or Netscape that were not present in the first prototype. Steve Uhler took my prototype, refined the HTTP handling, and aimed to keep the basic server under 250 lines. I went the other direction, setting up a modular architecture, adding in features found in other Web servers, and adding some interesting ways to connect TclHttpd to Tcl applications.

Today TclHttpd is used both as a general-purpose Web server, and as a framework for building server applications. It implements `www.scriptics.com`, including the Tcl Resource Center and Scriptics' electronic commerce facilities. It is also built into several commercial applications such as license servers and mail spam filters. Instructions for setting up the TclHttpd on your platform are given toward the end of the chapter, on page 266. It works on Unix, Windows, and Macintosh. Using TclHttpd, you can have your own Web server up and running quickly.

This chapter provides an overview of the server and several examples of how you can use it. The chapter is not an exhaustive reference to every feature. Instead, it concentrates on a very useful subset of server features that I use the most. There are references to Tcl files in the server's implementation, which are all found in the `lib` directory of the distribution. You may find it helpful to read the code to learn more about the implementation. You can find the source on the CD-ROM.

Integrating TclHttpd with your Application

The bulk of this chapter describes the various ways you can extend the server and integrate it into your application. TclHttpd is interesting because, as a Tcl script, it is easy to add to your application. Suddenly your application has an interface that is accessible to Web browsers in your company's intranet or the global Internet. The Web server provides several ways you can connect it to your application:

- *Static pages* — As a "normal" Web server, you can serve static documents that describe your application.
- *Domain handlers* — You can arrange for all URL requests in a section of your Web site to be handled by your application. This is a very general interface where you interpret what the URL means and what sort of pages to return to each request. For example, `http://www.scriptics.com/resource` is implemented this way. The URL past `/resource` selects an index in a simple database, and the server returns a page describing the pages under that index.
- *Application-Direct URLs* — This is a domain handler that maps URLs onto Tcl procedures. The form query data that is part of the HTTP GET or POST request is automatically mapped onto the parameters of the application-direct procedure. The procedure simply computes the page as its return value. This is an elegant and efficient alternative to the CGI interface. For example, in TclHttpd the URLs under `/status` report various statistics about the Web server's operation.
- *Document handlers* — You can define a Tcl procedure that handles all files of a particular type. For example, the server has a handler for CGI scripts, HTML files, image maps, and HTML+Tcl template files.
- *HTML+Tcl Templates* — These are Web pages that mix Tcl and HTML markup. The server replaces the Tcl using the `subst` command and returns the result. The server can cache the result in a regular HTML file to avoid the overhead of template processing on future requests. Templates are a great way to maintain common look and feel to a family of Web pages, as well as to implement more advanced dynamic HTML features like self-checking forms.

TclHttpd Architecture

Figure 18–1 shows the basic components of the server. At the core is the `Httpd` module (`httpd.tcl`), which implements the server side of the `HTTP` protocol. The "d" in Httpd stands for *daemon*, which is the name given to system servers on `UNIX`. This module manages network requests, dispatches them to the `Url` module, and provides routines used to return the results to requests.

The `Url` module (`url.tcl`) divides the Web site into *domains*, which are subtrees of the URL hierarchy provided by the server. The idea is that different domains may have completely different implementations. For example, the Docu-

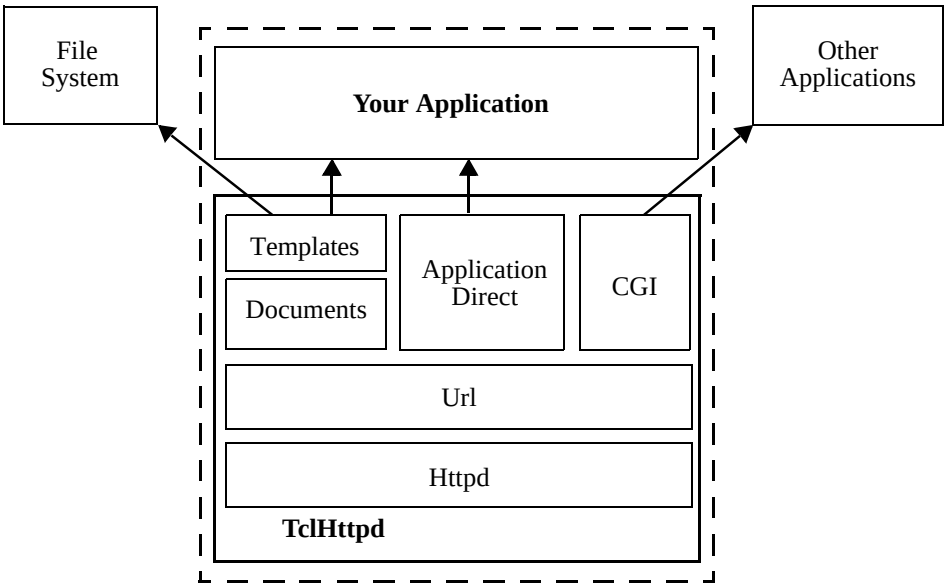

Fig. 18–1 The dotted box represents one application that embeds TclHttpd. Document templates and Application Direct URLs provide direct connections from an HTTP request to your application.

ment domain (`doc.tcl`) maps its URLs into files and directories on your hard disk, while the Application-Direct domain (`direct.tcl`) maps URLs into Tcl procedure calls within your application. The CGI domain (cgi.tcl) maps URLs onto other programs that compute Web pages.

Domain Handlers

You can implement new kinds of domains that provide your own interpretation of a URL. This is the most flexible interface available to extend the Web server. You provide a callback that is invoked to handle every request in a domain, or subtree, of the URL hierarchy. The callback interprets the URL, computes the page content, and returns the data using routines from the `Httpd` module.

Example 18–1 defines a simple domain that always returns the same page to every request. The domain is registered with the `Url_PrefixInstall` command. The arguments to `Url_PrefixInstall` are the URL prefix and a callback that is called to handle all URLs that match that prefix. In the example, all URLs that have the prefix `/simple` are dispatched to the `SimpleDomain` procedure.

Example 18–1 A simple URL domain.

```
Url_PrefixInstall /simple [list SimpleDomain /simple]

proc SimpleDomain {prefix sock suffix} {
    upvar #0 Httpd$sock data

    # Generate page header

    set html "<title>A simple page</title>\n"
    append html "<h1>$prefix$suffix</h1>\n"
    append html "<h1>Date and Time</h1>\n"
    append html [clock format [clock seconds]]
    # Display query data

    if {[info exist data(query)]} {
        append html "<h1>Query Data</h1>\n"
        append html "<table>\n"
        foreach {name value} [Url_DecodeQuery $data(query)] {
            append html "<tr><td>$name</td>\n"
            append html "<td>$value</td></tr>\n"
        }
        append html "</table>\n"
    }
    Httpd_ReturnData $sock text/html $html
}
```

The SimpleDomain handler illustrates several properties of domain handlers. The sock and suffix arguments to SimpleDomain are appended by Url_Dispatch when it invokes the domain handler. The suffix parameter is the part of the URL after the prefix. The prefix is passed in as part of the callback definition so the domain handler can recreate the complete URL. For example, if the server receives a request for the URL /simple/page, then the prefix is /simple, the suffix is /request.

Connection State and Query Data

The sock parameter is a handle on the socket connection to the remote client. This variable is also used to name a state variable that the Httpd module maintains about the connection. The name of the state array is Httpd$sock, and SimpleDomain uses upvar to get a more convenient name for this array (i.e., data):

```
upvar #0 Httpd$sock data
```

An important element of the state array is the query data, data(query). This is the information that comes from HTML forms. The query data arrives in an encoded format, and the Url_DecodeQuery procedure is used to decode the data into a list of names and values. Url_DecodeQuery is similar to Cgi_List from Example 11–5 on page 154 and is a standard function provided by url.tcl.

Returning Results

Finally, once the page has been computed, the `Httpd_ReturnData` procedure is used to return the page to the client. This takes care of the HTTP protocol as well as returning the data. There are three related procedures, `Httpd_ReturnFile`, `Httpd_Error`, and `Httpd_Redirect`. These are summarized in Table 18–1 on page 259.

Application Direct URLs

The Application Direct domain implementation provides the simplest way to extend the Web server. It hides the details associated with query data, decoding URL paths, and returning results. All you do is define Tcl procedures that correspond to URLs. Their arguments are automatically matched up to the query data as shown in Example 13–3 on page 179. The Tcl procedures compute a string that is the result data, which is usually HTML. That's all there is to it.

The `Direct_Url` procedure defines a URL prefix and a corresponding Tcl command prefix. Any URL that begins with the URL prefix will be handled by a corresponding Tcl procedure that starts with the Tcl command prefix. This is shown in Example 18–2:

Example 18–2 Application Direct URLs.

```
Direct_Url /demo Demo

proc Demo {} {
    return "<html><head><title>Demo page</title></head>\n\
        <body><h1>Demo page</h1>\n\
        <a href=/demo/time>What time is it?</a>\n\
        <form action=/demo/echo>\n\
        Data: <input type=text name=data>\n\
        <br>\n\
        <input type=submit name=echo value='Echo Data'>\n\
        </form>\n\
        </body></html>"
}
proc Demo/time {{format "%H:%M:%S"}} {
    return [clock format [clock seconds] -format $format]
}
proc Demo/echo {args} {
    # Compute a page that echoes the query data

    set html "<head><title>Echo</title></head>\n"
    append html "<body><table>\n"
    foreach {name value} $args {
        append html "<tr><td>$name</td><td>$value</td></tr>\n"
    }
    append html "</tr></table>\n"
    return $html
}
```

Example 18–2 defines /demo as an Application Direct URL domain that is implemented by procedures that begin with Demo. There are just three URLs defined:

```
/demo
/demo/time
/demo/echo
```

The /demo page displays a hypertext link to the /demo/time page and a simple form that will be handled by the /demo/echo page. This page is static, and so there is just one return command in the procedure body. Each line of the string ends with:

```
\n\
```

This is just a formatting trick to let me indent each line in the procedure, but not have the line indented in the resulting string. Actually, the \-newline will be replaced by one space, so each line will be indented one space. You can leave those off and the page will display the same in the browser, but when you view the page source you'll see the indenting. Or you could not indent the lines in the string, but then your code looks somewhat odd.

The /demo/time procedure just returns the result of clock format. It doesn't even bother adding <html>, <head>, or <body> tags, which you can get away with in today's browsers. A simple result like this is also useful if you are using programs to fetch information via HTTP requests.

Using Query Data

The /demo/time procedure is defined with an optional format argument. If a format value is present in the query data, then it overrides the default value given in the procedure definition.

The /demo/echo procedure creates a table that shows its query data. Its args parameter gets filled in with a name-value list of all query data. You can have named parameters, named parameters with default values, and the args parameter in your application-direct URL procedures. The server automatically matches up incoming form values with the procedure declaration. For example, suppose you have an application direct procedure declared like this:

```
proc Demo/param { a b {c cdef} args} { body }
```

You could create an HTML form that had elements named a, b, and c, and specified /demo/param for the ACTION parameter of the FORM tag. Or you could type the following into your browser to embed the query data right into the URL:

```
/demo/param?a=5&b=7&c=red&d=%7ewelch&e=two+words
```

In this case, when your procedure is called, a is 5, b is 7, c is red, and the args parameter becomes a list of:

```
d ~welch e {two words}
```

The %7e and the + are special codes for nonalphanumeric characters in the query data. Normally, this encoding is taken care of automatically by the Web browser when it gets data from a form and passes it to the Web server. However,

if you type query data directly or format URLs with complex query data in them, then you need to think about the encoding. Use the `Url_Encode` procedure to encode URLs that you put into Web pages.

If parameters are missing from the query data, they either get the default values from the procedure definition or the empty string. Consider this example:

```
/demo/param?b=5
```

In this case `a` is `""`, `b` is `5`, `c` is `cdef`, and `args` is an empty list.

Returning Other Content Types

The default content type for application direct URLs is `text/html`. You can specify other content types by using a global variable with the same name as your procedure. (Yes, this is a crude way to craft an interface.) Example 18–3 shows part of the `faces.tcl` file that implements an interface to a database of picons — personal icons — that is organized by user and domain names. The idea is that the database contains images corresponding to your e-mail correspondents. The `Faces_ByEmail` procedure, which is not shown, looks up an appropriate image file. The application direct procedure is `Faces/byemail`, and it sets the global variable `Faces/byemail` to the correct value based on the filename extension. This value is used for the `Content-Type` header in the result part of the `HTTP` protocol.

Example 18–3 Alternate types for Application Direct URLs.

```
Direct_Url /faces Faces
proc Faces/byemail {email} {
    global Faces/byemail
    set filename [Faces_ByEmail $email]
    set Faces/byemail [Mtype $filename]
    set in [open $filename]
    fconfigure $in -translation binary
    set X [read $in]
    close $in
    return $X
}
```

Document Types

The Document domain (`doc.tcl`) maps URLs onto files and directories. It provides more ways to extend the server by registering different document type handlers. This occurs in a two-step process. First, the type of a file is determined by its suffix. The `mime.types` file contains a map from suffixes to `MIME` types such as `text/html` or `image/gif`. This map is controlled by the `Mtype` module in `mtype.tcl`. Second, the server checks for a Tcl procedure with the appropriate name:

```
Doc_mimetype
```

The matching procedure, if any, is called to handle the URL request. The procedure should use routines in the `Httpd` module to return data for the request. If there is no matching `Doc_`*`mimetype`* procedure, then the default document handler uses `Httpd_ReturnFile` and specifies the Content Type based on the file extension:

```
Httpd_ReturnFile $sock [Mtype $path] $path
```

You can make up new types to support your application. Example 18–4 shows the pieces needed to create a handler for a fictitious document type `application/myjunk` that is invoked to handle files with the `.junk` suffix. You need to edit the `mime.types` file and add a document handler procedure to the server:

Example 18–4 A sample document type handler.

```
# Add this line to mime.types
application/myjunk      .junk

# Define the document handler procedure
#   path is the name of the file on disk
#   suffix is part of the URL after the domain prefix
#   sock is the handle on the client connection

proc Doc_application/myjunk {path suffix sock} {
    upvar #0 Httpd$sock data
    # data(url) is more useful than the suffix parameter.

    # Use the contents of file $path to compute a page
    set contents [somefunc $path]

    # Determine your content type
    set type text/html

    # Return the page
    Httpd_ReturnData $sock $type $data
}
```

As another example, the HTML+Tcl templates use the `.tml` suffix that is mapped to the `application/x-tcl-template` type. The TclHttpd distribution also includes support for files with a `.snmp` extension that implements a template-based Web interface to the Scotty SNMP Tcl extension.

HTML + Tcl Templates

The template system uses HTML pages that embed Tcl commands and Tcl variable references. The server replaces these using the `subst` command and returns the results. The server comes with a general template system, but using `subst` is so easy you could create your own template system. The general template framework has these components:

- Each `.html` file has a corresponding `.tml` template file. This feature is enabled with the `Doc_CheckTemplates` command in the server's configuration file. Normally, the server returns the `.html` file unless the corresponding `.tml` file has been modified more recently. In this case, the server processes the template, caches the result in the `.html` file, and returns the result.
- A dynamic template (e.g., a form handler) must be processed each time it is requested. If you put the `Doc_Dynamic` command into your page, it turns off the caching of the result in the `.html` page. The server responds to a request for a `.html` page by processing the `.tml` page. Or you can just reference the `.tml` file directly. If you do this, the server always processes the template.
- The server creates a `page` global Tcl variable that has context about the page being processed. Table 18–7 lists the elements of the page array.
- The server initializes the `env` global Tcl variable with similar information, but in the standard way for CGI scripts. Table 18–8 lists the elements of the `env` array that are set by `Cgi_SetEnv` in `cgi.tcl`.
- The server supports per-directory "`.tml`" files that contain Tcl source code. These files are designed to contain procedure definitions and variable settings that are shared among pages. The name of the file is simply "`.tml`", with nothing before the period. This is a standard way to hide files in UNIX, but it can be confusing to talk about the per-directory "`.tml`" files and the *`file`*`.tml` templates that correspond to *`file`*`.html` pages. The server will source the "`.tml`" files in all directories leading down to the directory containing the template file. The server compares the modify time of these files against the template file and will process the template if these "`.tml`" files are newer than the cached `.html` file. So, by modifying the "`.tml`" file in the root of your URL hierarchy, you invalidate all the cached `.html` files.
- The server supports a script library for the procedures called from templates. The `Doc_TemplateLibrary` procedure registers this directory. The server adds the directory to its `auto_path`, which assumes you have a `tclIndex` or `pkgIndex.tcl` file in the directory so that the procedures are loaded when needed.

Where to put your Tcl Code

There are three places you can put the code of your application: directly in your template pages, in the per-directory "`.tml`" files, or in the library directory.

The advantage of putting procedure definitions in the library is that they are defined one time but executed many times. This works well with the Tcl bytecode compiler. The disadvantage is that if you modify procedures in these files, you have to explicitly source them into the server for these changes to take effect. The `/debug/source` URL described on page 264 is handy for this chore.

The advantage of putting code into the per-directory "`.tml`" files is that changes are picked up immediately with no effort on your part. The server auto-

matically checks if these files are modified and sources them each time it processes your templates. However, that code is run only one time, so the byte-code compiler just adds overhead.

I try to put as little code as possible in my *file*.tml template files. It is awkward to put lots of code there, and you cannot share procedures and variable definitions easily with other pages. Instead, my goal is to have just procedure calls in the template files, and put the procedure definitions elsewhere. I also avoid putting if and foreach commands directly into the page.

Templates for Site Structure

The next few examples show a simple template system used to maintain a common look at feel across the pages of a site. Example 18–5 shows a simple one-level site definition that is kept in the root .tml file. This structure lists the title and URL of each page in the site:

Example 18–5 A one-level site structure.

```
set site(pages) {
    Home                /index.html
    "Ordering Computers"/ordering.html
    "New Machine Setup" /setup.html
    "Adding a New User" /newuser.html
    "Network Addresses" /network.html
}
```

Each page includes two commands, SitePage and SiteFooter, that generate HTML for the navigational part of the page. Between these commands is regular HTML for the page content. Example 18–6 shows a sample template file:

Example 18–6 A HTML + Tcl template file.

```
[SitePage "New Machine Setup"]
This page describes the steps to take when setting up a new
computer in our environment. See
<a href=/ordering.html>Ordering Computers</a>
for instructions on ordering machines.
<ol>
<li>Unpack and setup the machine.
<li>Use the Network control panel to set the IP address
and hostname.
<!-- Several steps omitted -->
<li>Reboot for the last time.
</ol>
[SiteFooter]
```

The SitePage procedure takes the page title as an argument. It generates HTML to implement a standard navigational structure. Example 18–7 has a simple implementation of SitePage:

Example 18–7 `SitePage` template procedure.

```
proc SitePage {title} {
   global site
   set html "<html><head><title>$title</title></head>\n"
   append html "<body bgcolor=white text=black>\n"
   append html "<h1>$title</h1>\n"
   set sep ""
   foreach {label url} $site(pages) {
      append html $sep
      if {[string compare $label $title] == 0} {
         append html "$label"
      } else {
         append html "<a href='$url'>$label</a>"
      }
      set sep " | "
   }
   return $html
}
```

The `foreach` loop that computes the simple menu of links turns out to be useful in many places. Example 18–8 splits out the loop and uses it in the `SitePage` and `SiteFooter` procedures. This version of the templates creates a left column for the navigation and a right column for the page content:

Example 18–8 `SiteMenu` and `SiteFooter` template procedures.

```
proc SitePage {title} {
   global site
   set html "<html><head><title>$title</title></head>\n\
      <body bgcolor=$site(bg) text=$site(fg)>\n\
      <!-- Two Column Layout -->\n\
      <table cellpadding=0>\n\
      <tr><td>\n\
      <!-- Left Column -->\n\
      <img src='$site(mainlogo)'>\n\
      <font size=+1>\n\
      [SiteMenu <br> $site(pages)]\n\
      </font>\n\
      </td><td>\n\
      <!-- Right Column -->\n\
      <h1>$title</h1>\n\
      <p>\n"
   return $html
}
proc SiteFooter {} {
   global site
   set html "<p><hr>\n\
      <font size=-1>[SiteMenu | $site(pages)]</font>\n\
      </td></tr></table>\n"
   return $html
}
proc SiteMenu {sep list} {
```

```
    global page
    set s ""
    set html ""
    foreach {label url} $list {
        if {[string compare $page(url) $url] == 0} {
            append html $s$label
        } else {
            append html "$s<a href='$url'>$label</a>"
        }
        set s $sep
    }
    return $html
}
```

Of course, a real site will have more elaborate graphics and probably a two-level, three-level, or more complex tree structure that describes its structure.You can also define a family of templates so that each page doesn't have to fit the same mold. Once you start using templates, it is fairly easy to change both the template implementation and to move pages around among different sections of your Web site.

There are many other applications for "macros" that make repetitive HTML coding chores easy. Take, for example, the link to `/ordering.html` in Example 18–6. The proper label for this is already defined in `$site(pages)`, so we could introduce a `SiteLink` procedure that uses this:

Example 18–9 The `SiteLink` procedure.

```
proc SiteLink {label} {
    global site
    array set map $site(pages)
    if {[info exist map($label)]} {
        return "<a href='$map($label)'>$label</a>"
    } else {
        return $label
    }
}
```

If your pages embed calls to `SiteLink`, then you can change the URL associated with the page name by changing the value of `site(pages)`. If this is stored in the top-level `".tml"` file, the templates will automatically track the changes.

Form Handlers

HTML forms and form-handling programs go together. The form is presented to the user on the client machine. The form handler runs on the server after the user fills out the form and presses the submit button. The form presents input widgets like radiobuttons, checkbuttons, selection lists, and text entry fields. Each of these widgets is assigned a name, and each widget gets a value based on

the user's input. The form handler is a program that looks at the names and values from the form and computes the next page for the user to read.

CGI is a standard way to hook external programs to Web servers for the purpose of processing form data. CGI has a special encoding for values so that they can be transported safely. The encoded data is either read from standard input or taken from the command line. The CGI program decodes the data, processes it, and writes a new HTML page on its standard output. Chapter 3 describes writing CGI scripts in Tcl.

TclHttpd provides alternatives to CGI that are more efficient because they are built right into the server. This eliminates the overhead that comes from running an external program to compute the page. Another advantage is that the Web server can maintain state between client requests in Tcl variables. If you use CGI, you must use some sort of database or file storage to maintain information between requests.

Application Direct Handlers

The server comes with several built-in form handlers that you can use with little effort. The `/mail/forminfo` URL will package up the query data and mail it to you. You use form fields to set various mail headers, and the rest of the data is packaged up into a Tcl-readable mail message. Example 18–10 shows a form that uses this handler. Other built-in handlers are described starting at page 263.

Example 18–10 Mail form results with `/mail/forminfo`.

```
<form action=/mail/forminfo method=post>
   <input type=hidden name=sendto value=mailreader@my.com>
   <input type=hidden name=subject value="Name and Address">
   <table>
      <tr><td>Name</td><td><input name=name></td></tr>
      <tr><td>Address</td><td><input name=addr1></td></tr>
      <tr><td> </td><td><input name=addr2></td></tr>
      <tr><td>City</td><td><input name=city></td></tr>
      <tr><td>State</td><td><input name=state></td></tr>
      <tr><td>Zip/Postal</td><td><input name=zip></td></tr>
      <tr><td>Country</td><td><input name=country></td></tr>
   </table>
</form>
```

The mail message sent by `/mail/forminfo` is shown in Example 18–11.

Example 18–11 Mail message sent by `/mail/forminfo`.

```
To: mailreader@my.com
Subject: Name and Address

data {
   name   {Joe Visitor}
```

```
    addr1  {Acme Company}
    addr2  {100 Main Street}
    city   {Mountain View}
    state  California
    zip    12345
    country   USA
}
```

It is easy to write a script that strips the headers, defines a `data` procedure, and uses `eval` to process the message body. Whenever you send data via e-mail, if you format it with Tcl list structure, you can process it quite easily. The basic structure of such a mail reader procedure is shown in Example 18–12:

Example 18–12 Processing mail sent by `/mail/forminfo`.

```
# Assume the mail message is on standard input

set X [read stdin]

# Strip off the mail headers, when end with a blank line
if {[regsub {.*?\n\ndata} $X {data} X] != 1} {
    error "Malformed mail message"
}
proc data {fields} {
    foreach {name value} $fields {
        # Do something
    }
}
# Process the message. For added security, you may want
# do this part in a safe interpreter.
eval $X
```

Template Form Handlers

The drawback of using application-direct URL form handlers is that you must modify their Tcl implementation to change the resulting page. Another approach is to use templates for the result page that embed a command that handles the form data. The `Mail_FormInfo` procedure, for example, mails form data. It takes no arguments. Instead, it looks in the query data for `sendto` and `subject` values, and if they are present, it sends the rest of the data in an e-mail. It returns an HTML comment that flags that mail was sent.

When you use templates to process form data, you need to turn off result caching because the server must process the template each time the form is submitted. To turn off caching, embed the `Doc_Dynamic` command into your form handler pages, or set the `page(dynamic)` variable to 1. Alternatively, you can simply post directly to the *`file`*`.tml` page instead of to the *`file`*`.html` page.

Self Posting Forms

This section illustrates a self-posting form. This is a form on a page that posts the form data to back to the same page. The page embeds a Tcl command to check its own form data. Once the data is correct, the page triggers a redirect to the next page in the flow. This is a powerful technique that I use to create complex page flows using templates. Of course, you need to save the form data at each step. You can put the data in Tcl variables, use the data to control your application, or store it into a database. TclHttpd comes with a `Session` module, which is one way to manage this information. For details you should scan the `session.tcl` file in the distribution.

Example 18–13 shows the `Form_Simple` procedure that generates a simple self-checking form. Its arguments are a unique ID for the form, a description of the form fields, and the URL of the next page in the flow. The field description is a list with three elements for each field: a required flag, a form element name, and a label to display with the form element. You can see this structure in the template shown in Example 18–14 on page 258. The procedure does two things at once. It computes the HTML form, and it also checks if the required fields are present. It uses some procedures from the `form` module to generate form elements that retain values from the previous page. If all the required fields are present, it discards the HTML, saves the data, and triggers a redirect by calling `Doc_Redirect`.

Example 18–13 A self-checking form procedure.

```
proc Form_Simple {id fields nextpage} {
    global page
    if {![form::empty formid]} {
        # Incoming form values, check them
        set check 1
    } else {
        # First time through the page
        set check 0
    }
    set html "<!-- Self-posting. Next page is $nextpage -->\n"
    append html "<form action=\"$page(url)\" method=post>\n"
    append html "<input type=hidden name=formid value=$id>\n"
    append html "<table border=1>\n"
    foreach {required key label} $fields {
        append html "<tr><td>"
        if {$check && $required && [form::empty $key]} {
            lappend missing $label
            append html "<font color=red>*</font>"
        }
        append html "</td><td>$label</td>\n"
        append html "<td><input [form::value $key]></td>\n"
        append html "</tr>\n"
    }
    append html "</table>\n"
    if {$check} {
        if {![info exist missing]} {
```

```
            # No missing fields, so advance to the next page.
            # In practice, you must save the existing fields
            # at this point before redirecting to the next page.

            Doc_Redirect $nextpage
        } else {
            set msg "<font color=red>Please fill in "
            append msg [join $missing ", "]
            append msg "</font>"
            set html <p>$msg\n$html
        }
    }
    append html "<input type=submit>\n</form>\n"
    return $html
}
```

Example 18–14 shows a page template that calls `Form_Simple` with the required field description.

Example 18–14 A page with a self-checking form.

```
<html><head>
   <title>Name and Address Form</title>
</head>
<body bgcolor=white text=black>
   <h1>Name and Address</h1>
   Please enter your name and address.
   [myform::simple nameaddr {
      1 name    "Name"
      1 addr1   "Address"
      0 addr2"  "Address"
      1 city    "City"
      0 state   "State"
      1 zip     "Zip Code"
      0 country "Country"
   } nameok.html]
</body></html>
```

The `form` package

TclHttpd comes with a `form` package (`form.tcl`) that is designed to support self-posting forms. The `Form_Simple` procedure uses `form::empty` to test if particular form values are present in the query data. For example, it tests to see whether the `formid` field is present so that the procedure knows whether or not to check for the rest of the fields. The `form::value` procedure is useful for constructing form elements on self-posting form pages. It returns:

`name="`*`name`*`" value="`*`value`*`"`

The *`value`* is the value of form element *`name`* based on incoming query data, or just the empty string if the query value for *`name`* is undefined. As a result, the

form can post to itself and retain values from the previous version of the page. It is used like this:

```
<input type=text [form::value name]>
```

The `form::checkvalue` and `form::radiovalue` procedures are similar to `form::value` but designed for checkbuttons and radio buttons. The `form::select` procedure formats a selection list and highlights the selected values. The `form::data` procedure simply returns the value of a given form element. These are summarized in Table 18–6 on page 261.

Programming Reference

This section summarizes many of the more useful functions defined by the server. These tables are not complete, however. You are encouraged to read through the code to learn more about the features offered by the server.

Table 18–1 summarizes the `Httpd` functions used when returning pages to the client.

Table 18–1 `Httpd` support procedures.

`Httpd_Error` *`sock code`*	Returns a simple error page to the client. The *`code`* is a numeric error code like 404 or 500.
`Httpd_ReturnData` *`sock type data`*	Returns a page with Content-Type *`type`* and content *`data`*.
`Httpd_ReturnFile` *`sock type file`*	Returns a *`file`* with Content-Type *`type`*.
`Httpd_Redirect` *`newurl sock`*	Generates a 302 error return with a Location of *`newurl`*.
`Httpd_SelfUrl` *`url`*	Expands *`url`* to include the proper `http://` *`server:port`* prefix to reference the current server.

Table 18–2 summarizes a few useful procedures provided by the `Url` module (`url.tcl`). The `Url_DecodeQuery` is used to decode query data into a Tcl-friendly list. The `Url_Encode` procedure is useful when encoding values directly into URLs. URL encoding is discussed in more detail on page 247.

Table 18–2 `Url` support procedures.

`Url_DecodeQuery` *`query`*	Decodes a `www-url-encoded` query string and return a name, value list.
`Url_Encode` *`value`*	Returns *`value`* encoded according to the www-url-encoded standard.
`Url_PrefxInstall` *`prefix callback`*	Registers *`callback`* as the handler for all URLs that begin with *`prefix`*. The callback is invoked with two additional arguments: *`sock`*, the handle to the client, and *`suffix`*, the part of the URL after *`prefix`*.

The Doc module provides procedures for configuration and generating responses, which are summarized in Tables 18–3 and 18–4, respectively.

Table 18–3 Doc procedures for configuration.

`Doc_Root ?`*`directory`*`?`	Sets or queries the *`directory`* that corresponds to the root of the URL hierarchy.
`Doc_AddRoot` *`virtual directory`*	Maps the file system *`directory`* into the URL subtree starting at *`virtual`*.
`Doc_ErrorPage` *`file`*	Specifies a *`file`* relative to the document root used as a simple template for error messages. This is processed by `DocSubstSystem` file in `doc.tcl`.
`Doc_CheckTemplates` *`how`*	If *`how`* is 1, then `.html` files are compared against corresponding `.tml` files and regenerated if necessary.
`Doc_IndexFile` *`pattern`*	Registers a file name *`pattern`* that will be searched for the default index file in directories.
`Doc_NotFoundPage` *`file`*	Specifies a *`file`* relative to the document root used as a simple template for page not found messages. This is processed by `DocSubstSystem` file in `doc.tcl`.
`Doc_PublicHtml` *`dirname`*	Defines the directory used for each users home directory. When a URL like *`~user`* is specified, the *`dirname`* under their home directory is accessed.
`Doc_TemplateLibrary` *`directory`*	Adds *`directory`* to the `auto_path` so the source files in it are available to the server.
`Doc_TemplateInterp` *`interp`*	Specifies an alternate interpreter in which to process document templates (i.e., `.tml` files.)
`Doc_Webmaster ?`*`email`*`?`	Sets or queries the *`email`* for the Webmaster.

Table 18–4 Doc procedures for generating responses.

`Doc_Error` *`sock errorInfo`*	Generates a 500 response on *`sock`* based on the template registered with `Doc_ErrorPage`. *`errorInfo`* is a copy of the Tcl error trace after the error.
`Doc_NotFound` *`sock`*	Generates a 404 response on *`sock`* by using the template registered with `Doc_NotFoundPage`.
`Doc_Subst` *`sock file`* `?`*`interp`*`?`	Performs a `subst` on the file and return the resulting page on *`sock`*. *`interp`* specifies an alternate Tcl interpreter.

The Doc module also provides procedures for cookies and redirects that are useful in document templates. These are described in Table 18–5.

Table 18–5 Doc procedures that support template processing.

`Doc_Coookie` *name*	Returns the cookie *name* passed to the server for this request, or the empty string if it is not present.
`Doc_Dynamic`	Turns off caching of the HTML result. Meant to be called from inside a page template.
`Doc_IsLinkToSelf` *url*	Returns 1 if the *url* is a link to the current page.
`Doc_Redirect` *newurl*	Raises a special error that aborts template processing and triggers a page redirect to *newurl*.
`Doc_SetCookie -name` *name* `-value` *value* `-path` *path* `-domain` *domain* `-expires` *date*	Sets cookie *name* with the given *value* that will be returned to the client as part of the response. The *path* and *domain* restrict the scope of the cooke. The *date* sets an expiration date.

Table 18–6 describes the `form` module that is useful for self-posting forms, which are discussed on page 257.

Table 18–6 The `form` package.

`form::data` *name*	Returns the value of the form value *name*, or the empty string.
`form::empty` *name*	Returns 1 if the form value *name* is missing or zero length.
`form::value` *name*	Returns `name="`*name*`" value="`*value*`"`, where *value* comes from the query data, if any.
`form::checkvalue` *name value*	Returns `name="`*name*`" value="`*value*`" CHECKED`, if *value* is present in the query data for *name*. Otherwise, it just returns `name="`*name*`" value="`*value*`"`.
`form::radiovalue` *name value*	Returns `name="`*name*`" value="`*value*`" CHECKED`, if the query data for *name* is equal to *value*. Otherwise, it just returns `name="`*name*`" value="`*value*`"`.
`form::select` *name valuelist args*	Generates a `select` form element with name *name*. The *valuelist* determines the `option` tags and values, and *args* are optional parameters to the main `select` tag.

Table 18–7 shows the initial elements of the `page` array that is defined during the processing of a template.

Table 18–7 Elements of the `page` array.

`query`	The decoded query data in a name, value list.
`dynamic`	If 1, the results of processing the template are not cached in the corresponding `.html` file.
`filename`	The file system pathname of the requested file (e.g., `/usr/local/htdocs/tclhttpd/index.html`).

Table 18–7 Elements of the `page` array. (Continued)

`template`	The file system pathname of the template file (e.g., `/usr/local/htdocs/tclhttpd/index.tml`).
`url`	The part of the url after the server name (e.g., `/tclhttpd/index.html`).
`root`	A relative path from the template file back to the root of the URL tree. This is useful for creating relative links between pages in different directories.

Table 18–8 shows the elements of the `env` array. These are defined during CGI requests, application-direct URL handlers, and page template processing:

Table 18–8 Elements of the `env` array.

`AUTH_TYPE`	Authentication protocol (e.g., `Basic`).
`CONTENT_LENGTH`	The size of the query data.
`CONTENT_TYPE`	The type of the query data.
`DOCUMENT_ROOT`	File system pathname of the document root.
`GATEWAY_INTERFACE`	Protocol version, which is `CGI/1.1`.
`HTTP_ACCEPT`	The Accept headers from the request.
`HTTP_AUTHORIZATION`	The Authorization challenge from the request.
`HTTP_COOKIE`	The cookie from the request.
`HTTP_FROM`	The From: header of the request.
`HTTP_REFERER`	The Referer indicates the previous page.
`HTTP_USER_AGENT`	An ID string for the Web browser.
`PATH_INFO`	Extra path information after the template file.
`PATH_TRANSLATED`	The extra path information appended to the document root.
`QUERY_STRING`	The form query data.
`REMOTE_ADDR`	The client's IP address.
`REMOTE_USER`	The remote user name specified by Basic authentication.
`REQUEST_METHOD`	GET, POST, or HEAD.
`REQUEST_URI`	The complete URL that was requested.
`SCRIPT_NAME`	The name of the current file relative to the document root.
`SERVER_NAME`	The server name, e.g., `www.beedub.com`.
`SERVER_PORT`	The server's port, e.g., 80.
`SERVER_PROTOCOL`	The protocol (e.g., `http` or `https`).
`SERVER_SOFTWARE`	A software version string for the server.

Standard Application-Direct URLs

The server has several modules that provide application-direct URLs. These application-direct URLs lets you control the server or examine its state from any Web browser. You can look at the implementation of these modules as examples for your own application.

Status

The `/status` URL is implemented in the `status.tcl` file. The status module implements the display of hit counts, document hits, and document misses (i.e., documents not found). The `Status_Url` command enables the application-direct URLs and assigns the top-level URL for the status module. The default configuration file contains this command:

```
Status_Url /status
```

Assuming this configuration, the following URLs are implemented:

Table 18–9 Status application-direct URLs.

`/status`	Main status page showing summary counters and hit count histograms.
`/status/doc`	Shows hit counts for each page. This page lets you sort by name or hit count, and limit files by patterns.
`/status/hello`	A trivial URL that returns "hello".
`/status/notfound`	Shows miss counts for URLs that users tried to fetch.
`/status/size`	Displays an estimated size of Tcl code and Tcl data used by the TclHttpd program.
`/status/text`	This is a version of the main status page that doesn't use the graphical histograms of hit counts.

Debugging

The `/debug` URL is implemented in the `debug.tcl` file. The debug module has several useful URLs that let you examine variable values and other internal state. It is turned on with this command in the default configuration file:

```
Debug_Url /debug
```

Table 18–10 lists the `/debug` URLs. These URLs often require parameters that you can specify directly in the URL. For example, the `/debug/echo` URL echoes its query parameters:

```
http://yourserver:port/debug/echo?name=value&name2=val2
```

Table 18–10 Debug application-direct URLs.

`/debug/after`	Lists the outstanding `after` events.
`/debug/dbg`	Connects to *TclPro Debugger*. This takes a `host` and `port` parameter. You need to install `prodebug.tcl` from *TclPro* into the server's script library directory.
`/debug/echo`	Echoes its query parameters. Accepts a `title` parameter.
`/debug/errorInfo`	Displays the `errorInfo` variable along with the server's version number and Webmaster e-mail. Accepts `title` and `errorInfo` arguments.
`/debug/parray`	Displays a global array variable. The name of the variable is specified with the `aname` parameter.
`/debug/pvalue`	A more general value display function. The name of the variable is specified with the `aname` parameter. This can be a variable name, an array name, or a pattern that matches several variable names.
`/debug/raise`	Raises an error (to test error handling). Any parameters become the error string.
`/debug/source`	Sources a file from either the server's main library directory or the `Doc_TemplateLibrary` directory. The file is specified with the `source` parameter.

The sample URL tree that is included in the distribution includes the file `htdocs/hacks.html`. This file has several small forms that use the `/debug` URLs to examine variables and source files. Example18–15 shows the implementation of `/debug/source`. You can see that it limits the files to the main script library and to the script library associated with document templates. It may seem dangerous to have these facilities, but I reason that because my source directories are under my control, it cannot hurt to reload any source files. In general, the library scripts contain only procedure definitions and no global code that might reset state inappropriately. In practice, the ability to tune (i.e., fix bugs) in the running server has proven useful to me on many occasions. It lets you evolve your application without restarting it!

Example 18–15 The `/debug/source` application-direct URL implementation.

```
proc Debug/source {source} {
    global Httpd Doc
    set source [file tail $source]
    set dirlist $Httpd(library)
    if {[info exists Doc(templateLibrary)]} {
        lappend dirlist $Doc(templateLibrary)
    }
    foreach dir $dirlist {
        set file [file join $dir $source]
        if [file exists $file] {
            break
```

```
        }
    }
    set error [catch {uplevel #0 [list source $file]} result]
    set html "<title>Source $source</title>\n"
    if {$error} {
        global errorInfo
        append html "<H1>Error in $source</H1>\n"
        append html "<pre>$result<p>$errorInfo</pre>"
    } else {
        append html "<H1>Reloaded $source</H1>\n"
        append html "<pre>$result</pre>"
    }
    return $html
}
```


Administration

The `/admin` URL is implemented in the `admin.tcl` file. The admin module lets you load URL redirect tables, and it provides URLs that reset some of the counters maintained by the server. It is turned on with this command in the default configuration file:

```
Admin_Url /admin
```

Currently, there is only one useful admin URL. The `/admin/redirect/reload` URL sources the file named `redirect` in the document root. This file is expected to contain a number of `Url_Redirect` commands that establish URL redirects. These are useful if you change the names of pages and want the old names to still work.

The administration module has a limited set of application-direct URLs because the simple application-direct mechanism doesn't provide the right hooks to check authentication credentials. The HTML+Tcl templates work better with the authentication schemes.

Sending Email

The `/mail` URL is implemented in the `mail.tcl` file. The mail module implements various form handlers that e-mail form data. Currently, it is UNIX-specific because it uses `/usr/lib/sendmail` to send the mail. It is turned on with this command in the default configuration file:

```
Mail_Url /mail
```

The application-direct URLs shown in Table 18–11 are useful form handlers. You can specify them as the `ACTION` parameter in your `<FORM>` tags. The mail module provides two Tcl procedures that are generally useful. The `MailInner` procedure is the one that sends mail. It is called like this:

```
MailInner sendto subject from type body
```

The `sendto` and `from` arguments are e-mail addresses. The `type` is the Mime type (e.g., `text/plain` or `text/html`) and appears in a `Content-Type` header. The `body` contains the mail message without any headers.

Table 18–11 Application-direct URLS that e-mail form results.

`/mail/bugreport`	Sends e-mail with the `errorInfo` from a server error. It takes an `email` parameter for the destination address and an `errorInfo` parameter. Any additional arguments get included into the message.
`/mail/forminfo`	Sends e-mail containing form results. It requires these parameters: `sendto` for the destination address, `subject` for the mail subject, `href` and `label` for a link to display on the results page. Any additional arguments are formatted with the Tcl `list` command for easy processing by programs that read the mail.
`/mail/formdata`	This is an older form of `/mail/forminfo` that doesn't format the data into Tcl lists. It requires only the `email` and `subject` parameters. The rest are formatted into the message body.

The `Mail_FormInfo` procedure is designed for use in HTML+Tcl template files. It takes no arguments but instead looks in current query data for its parameters. It expects to find the same arguments as the `/mail/forminfo` direct URL. Using a template with `Mail_FormInfo` gives you more control over the result page than posting directly to `/mail/forminfo`, and is illustrated in Example 18–10 on page 255.

The TclHttpd Distribution

Get the TclHttpd distribution from the CD-ROM, or find it on the Internet at:

```
ftp://ftp.scriptics.com/pub/tcl/httpd/
http://www.scriptics.com/tclhttpd/
```

Quick Start

Unpack the tar file or the zip file, and you can run the server from the `httpd.tcl` script in the `bin` directory. On UNIX:

```
tclsh httpd.tcl -port 80
```

This command will start the Web server on the standard port (80). By default it uses port 8015 instead. If you run it with the `-help` flag, it will tell you what command line options are available. If you use *wish* instead of *tclsh,* then a simple Tk user interface is displayed that shows how many hits the server is getting.

On Windows you can double-click the `httpd.tcl` script to start the server. It will use *wish* and display the user interface. Again it will start on port 8015. You will need to create a shortcut that passes the `-port` argument, or edit the associated configuration file to change this. Configuring the server is described later.

Once you have the server running, you can connect to it from your Web browser. Use this URL if you are running on the default (nonstandard) port:

```
http://hostname:8015/
```

If you are running without a network connection, you may need to specify `127.0.0.1` for the hostname. This is the "localhost" address and will bypass the network subsystem.

```
http://127.0.0.1:8015/
```

Inside the Distribution

The TclHttpd distribution is organized into the following directories:

- `bin` — This has sample start-up scripts and configuration files. The `httpd.tcl` script runs the server. The `tclhttpd.rc` file is the standard configuration file. The `minihttpd.tcl` file is the 250-line version. The `torture.tcl` file has some scripts that you can use to fetch many URLs at once from a server.
- `lib` — This has all the Tcl sources. In general, each file provides a package. You will see the `package require` commands partly in `bin/httpd.tcl` and partly in `bin/tclhttpd.rc`. The idea is that only the core packages are required by `httpd.tcl`, and different applications can tune what packages are needed by adjusting the contents of `tclhttpd.rc`.
- `htdocs` — This is a sample URL tree that demonstrates the features of the Web server. There is also some documentation there. One directory to note is `htdocs/libtml`, which is the standard place to put site-specific Tcl scripts used with the Tcl+HTML template facility.
- `src` — There are a few C source files for a some optional packages. These have been precompiled for some platforms, and you can find the compiled libraries back under `lib/Binaries` in platform-specific subdirectories.

Server Configuration

TclHttpd configures itself with three main steps. The first step is to process the command line arguments described in Table 18–12. The arguments are copied into the `Config` Tcl array. Anything not specified on the command line gets a default value. The next configuration step is to source the configuration file. The default configuration file is named `tclhttpd.rc` in the same directory as the start-up script (i.e., `bin/tclhttpd.rc`). This file can override command line arguments by setting the `Config` array itself. This file also has application-specific `package require` commands and other Tcl commands to initialize the application. Most of the Tcl commands used during initialization are described in the rest of this section. The final step is to actually start up the server. This is done back in the main `httpd.tcl` script. For example, to start the server for the document tree under `/usr/local/htdocs` and your own e-mail address as Webmaster, you can execute this command to start the server:

```
tclsh httpd.tcl -docRoot /usr/local/htdocs -webmaster welch
```

Alternatively, you can put these settings into a configuration file, and start the server with that configuration file:

```
tclsh httpd.tcl -config mytclhttpd.rc
```

In this case, the `mytclhttpd.rc` file could contain these commands to hard-wire the document root and Webmaster e-mail. In this case, the command line arguments *cannot* override these settings:

```
set Config(docRoot) /usr/local/htdocs
set Config(webmaster) welch
```

Command Line Arguments

There are several parameters you may need to set for a standard Web server. These are shown below in Table 18–12. The command line values are mapped into the `Config` array by the `httpd.tcl` start-up script.

Table 18–12 Basic TclHttpd Parameters.

Parameter	Command Option	Config Variable
Port number. The default is `8015`.	`-port` *`number`*	`Config(port)`
Server name. The default is `[info hostname]`.	`-name` *`name`*	`Config(name)`
IP address. The default is `0`, for "any address".	`-ipaddr` *`address`*	`Config(ipaddr)`
Directory of the root of the URL tree. The default is the `htdocs` directory.	`-docRoot` *`directory`*	`Config(docRoot)`
User ID of the TclHttpd process. The default is `50`. (UNIX only.)	`-uid` *`uid`*	`Config(uid)`
Group ID of the TclHttpd process. The default is `100`. (UNIX only.)	`-gid` *`gid`*	`Config(gid)`
Webmaster e-mail. The default is `webmaster`.	`-webmaster` *`email`*	`Config(webmaster)`
Configuration file. The default is `tclhttpd.rc`.	`-config` *`filename`*	`Config(file)`
Additional directory to add to the `auto_path`.	`-library` *`directory`*	`Config(library)`

Server Name and Port

The name and port parameters define how your server is known to Web browsers. The URLs that access your server begin with:

```
http://name:port/
```

If the port number is 80, you can leave out the port specification. The call that starts the server using these parameters is found in `httpd.tcl` as:

```
Httpd_Server $Config(name) $Config(port) $Config(ipaddr)
```

Specifying the IP address is necessary only if you have several network interfaces (or several IP addresses assigned to one network interface) and want the server to listen to requests on a particular network address. Otherwise, by default, server accepts requests from any network interface.

User and Group ID

The user and group IDs are used on UNIX systems with the `setuid` and `setgid` system calls. This lets you start the server as root, which is necessary to listen on port 80, and then switch to a less privileged user account. If you use Tcl+HTML templates that cache the results in HTML files, then you need to pick an account that can write those files. Otherwise, you may want to pick a very unprivileged account.

The `setuid` function is available through the TclX (Extended Tcl) `id` command, or through a `setuid` extension distributed with TclHttpd under the src directory. If do not have either of these facilities available, then the attempt to change user ID gracefully fails. See the `README` file in the `src` directory for instructions on compiling and installing the extensions found there.

Webmaster Email

The Webmaster e-mail address is used for automatic error reporting in the case of server errors. This is defined in the configuration file with the following command:

```
Doc_Webmaster $Config(webmaster)
```

If you call `Doc_Webmaster` with no arguments, it returns the e-mail address you previously defined. This is useful when generating pages that contain `mailto:` URLs with the Webmaster address.

Document Root

The document root is the directory that contains the static files, templates, CGI scripts, and so on that make up your Web site. By default the httpd.tcl script uses the *htdocs* directory next to the directory containing *httpd.tcl*. It is worth noting the trick used to locate this directory:

```
file join [file dirname [info script]] ../htdocs
```

The `info script` command returns the full name of the `http.tcl` script, `file dirname` computes its directory, and `file join` finds the adjacent directory. The path `../htdocs` works with `file join` on any platform. The default location of the configuration file is found in a similar way:

```
file join [file dirname [info script]] tclhttpd.rc
```

The configuration file initializes the document root with this call:

```
Doc_Root $Config(docRoot)
```

If you need to find out what the document root is, you can call Doc_Root with no arguments and it returns the directory of the document root. If you want to add additional document trees into your Web site, you can do that with a call like this in your configuration file:

```
Doc_AddRoot directory urlprefix
```

Other Document Settings

The `Doc_IndexFile` command sets a pattern used to find the index file in a directory. The command used in the default configuration file is:

```
Doc_IndexFile index.{htm,html,tml,subst}
```

If you invent other file types with different file suffixes, you can alter this pattern to include them. This pattern will be used by the Tcl `glob` command.

The `Doc_PublicHtml` command is used to define "home directories" on your HTML site. If the URL begins with ~username, then the Web server will look under the home directory of username for a particular directory. The command in the default configuration file is:

```
Doc_PublicHtml public_html
```

For example, if my home directory is `/home/welch`, then the URL `~welch` maps to the directory `/home/welch/public_html`. If there is no `Doc_PublicHtml` command, then this mapping does not occur.

You can register two special pages that are used when the server encounters an error and when a user specifies an unknown URL. The default configuration file has these commands:

```
Doc_ErrorPage error.html
Doc_NotFoundPage notfound.html
```

These files are treated like templates in that they are passed through `subst` in order to include the error information or the URL of the missing page. These are pretty crude templates compared to the templates described earlier. You can count only on the `Doc` and `Httpd` arrays being defined. Look at the `Doc_SubstSystemFile` in `doc.tcl` for the truth about how these files are processed.

Document Templates

The template mechanism has two main configuration options. The first specifies an additional library directory that contains your application-specific scripts. This lets you keep your application-specific files separate from the TclHttpd implementation. The command in the default configuration file specifies the libtml directory of the document tree:

```
Doc_TemplateLibrary [file join $Config(docRoot) libtml]
```

You can also specify an alternate Tcl interpreter in which to process the templates. The default is to use the main interpreter, which is named {} accord-

ing to the conventions described in Chapter 19.

```
Doc_TemplateInterp {}
```

Log Files

The server keeps standard format log files. The `Log_SetFile` command defines the base name of the log file. The default configuration file uses this command:

```
Log_SetFile /tmp/log$Config(port)_
```

By default the server rotates the log file each night at midnight. Each day's log file is suffixed with the current date (e.g., `/tmp/log`*`port`*`_990218`.) The error log, however, is not rotated, and all errors are accumulated in `/tmp/log`*`port`*`_error`.

The log records are normally flushed every few minutes to eliminate an extra I/O operation on each HTTP transaction. You can set this period with `Log_FlushMinutes`. If minutes is 0, the log is flushed on every HTTP transaction. The default configuration file contains:

```
Log_FlushMinutes 1
```

CGI Directories

You can register a directory that contains CGI programs with the `Cgi_Directory` command. This command has the interesting effect of forcing all files in the directory to be executed as CGI scripts, so you cannot put normal HTML files there. The default configuration file contains:

```
Cgi_Directory /cgi-bin
```

This means that the `cgi-bin` directory under the document root is a CGI directory. If you supply another argument to `Cgi_Directory`, then this is a file system directory that gets mapped into the URL defined by the first argument. You can also put CGI scripts into other directories and use the `.cgi` suffix to indicate that they should be executed as CGI scripts.

The `cgi.tcl` file has some additional parameters that you can tune only by setting some elements of the `Cgi` Tcl array. See the comments in the beginning of that file for details.

CHAPTER 19

Multiple Interpreters and Safe-Tcl

This chapter describes how to create more than one Tcl interpreter in your application. A child interpreter can be made safe so that it can execute untrusted scripts without compromising your application or your computer. Command aliases, hidden commands, and shared I/O channels enable communication among interpreters. Tcl command described is: `interp`.

Safe-Tcl was invented by Nathaniel Borenstein and Marshall Rose so that they could send Tcl scripts via e-mail and have the recipient safely execute the script without worry of viruses or other attacks. Safe-Tcl works by removing dangerous commands like `exec` and `open` that would let an untrusted script damage the host computer. You can think of this restricted interpreter as a "padded cell" in which it is safe to execute untrusted scripts. To continue the analogy, if the untrusted code wants to do anything potentially unsafe, it must ask permission. This works by adding additional commands, or *aliases*, that are implemented by a different Tcl interpreter. For example, a `safeopen` command could be implemented by limiting file space to a temporary directory that is deleted when the untrusted code terminates.

The key concept of Safe-Tcl is that there are two Tcl interpreters in the application, a trusted one and an untrusted (or "safe") one. The trusted interpreter can do anything, and it is used for the main application (e.g., the Web browser or e-mail user interface). When the main application receives a message containing an untrusted script, it evaluates that script in the context of the untrusted interpreter. The restricted nature of the untrusted interpreter means that the application is safe from attack. This model is much like user mode and kernel mode in a multiuser operating system like UNIX or Windows/NT. In these systems, applications run in user mode and trap into the kernel to access resources like files and the network. The kernel implements access controls so that users cannot read and write each other's files, or hijack network services. In Safe-Tcl the application implements access controls for untrusted scripts.

The dual interpreter model of Safe-Tcl has been generalized in Tcl 7.5 and made accessible to Tcl scripts. A Tcl script can create other interpreters, destroy them, create command aliases among them, share I/O channels among them, and evaluate scripts in them.

The `interp` Command

The `interp` command is used to create and manipulate interpreters. The interpreter being created is called a *slave*, and the interpreter that creates it is called the *master*. The master has complete control over the slave. The `interp` command is summarized in Table 19–1.

Table 19–1 The `interp` command.

`interp aliases` *`slave`*	Lists aliases that are defined in *`slave`*.
`interp alias` *`slave cmd1`*	Returns target command and arguments for the alias *`cmd1`* in *`slave`*.
`interp alias` *`slave cmd1 master cmd2 arg ...`*	Defines *`cmd1`* in *`slave`* that is an alias to *`cmd2`* in *`master`* with additional *`args`*.
`interp create ?-safe?` *`slave`*	Creates an interpreter named *`slave`*.
`interp delete` *`slave`*	Destroys interpreter *`slave`*.
`interp eval` *`slave cmd args ...`*	Evaluates *`cmd`* and *`args`* in *`slave`*.
`interp exists` *`slave`*	Returns 1 if *`slave`* is an interpreter, else 0.
`interp expose` *`slave cmd`*	Exposes hidden command *`cmd`* in *`slave`*.
`interp hide` *`slave cmd`*	Hides *`cmd`* from *`slave`*.
`interp hidden` *`slave`*	Returns the commands hidden from *`slave`*.
`interp invokehidden` *`slave cmd arg ...`*	Invokes hidden command *`cmd`* and *`args`* in *`slave`*.
`interp issafe` *`slave`*	Returns 1 if *`slave`* was created with `-safe` flag.
`interp marktrusted` *`slave`*	Clears the `issafe` property of *`slave`*.
`interp share` *`master file slave`*	Shares the I/O descriptor named *`file`* in *`master`* with *`slave.`*
`interp slaves` *`master`*	Returns the list of slave interpreters of *`master`*.
`interp target` *`slave cmd`*	Returns the name of the interpreter that is the target of alias *`cmd`* in *`slave`*.
`interp transfer` *`master file slave`*	Transfers the I/O descriptor named *`file`* from *`master`* to *`slave.`*

Creating Interpreters

Here is a simple example that creates an interpreter, evaluates a couple of commands in it, and then deletes the interpreter:

Example 19–1 Creating and deleting an interpreter.

```
interp create foo
=> foo
interp eval foo {set a 5}
=> 5
set sum [interp eval foo {expr $a + $a}]
=> 10
interp delete foo
```


In Example 19–1 the interpreter is named `foo`. Two commands are evaluated in the `foo` interpreter:

```
set a 5
expr $a + $a
```

Note that curly braces are used to protect the commands from any interpretation by the main interpreter. The variable `a` is defined in the `foo` interpreter and does not conflict with variables in the main interpreter. The set of variables and procedures in each interpreter is completely independent.

The Interpreter Hierarchy

A slave interpreter can itself create interpreters, resulting in a hierarchy. The next examples illustrates this, and it shows how the grandparent of an interpreter can reference the grandchild by name. The example uses `interp slaves` to query the existence of child interpreters.

Example 19–2 Creating a hierarchy of interpreters.

```
interp create foo
=> foo
interp eval foo {interp create bar}
=> bar
interp create {foo bar2}
=> foo bar2
interp slaves
=> foo
interp slaves foo
=> bar bar2
interp delete bar
=> interpreter named "bar" not found
interp delete {foo bar}
```

The example creates `foo`, and then it creates two children of `foo`. The first one is created by `foo` with this command:

```
interp eval foo {interp create bar}
```

The second child is created by the main interpreter. In this case, the grandchild must be named by a two-element list to indicate that it is a child of a child. The same naming convention is used when the grandchild is deleted:

```
interp create {foo bar2}
interp delete {foo bar2}
```

The `interp slaves` operation returns the names of child (i.e., slave) interpreters. The names are relative to their parent, so the slaves of `foo` are reported simply as `bar` and `bar2`. The name for the current interpreter is the empty list, or `{}`. This is useful in command aliases and file sharing described later. For security reasons, it is not possible to name the master interpreter from within the slave.

The Interpreter Name as a Command

After interpreter *`slave`* is created, a new command is available in the main interpreter, also called *`slave`*, that operates on the child interpreter. The following two forms are equivalent most operations:

```
slave operation args ...
interp operation slave args ...
```

For example, the following are equivalent commands:

```
foo eval {set a 5}
interp eval foo {set a 5}
```

And so are these:

```
foo issafe
interp issafe foo
```

However, the operations `delete`, `exists`, `share`, `slaves`, `target`, and `transfer` cannot be used with the per interpreter command. In particular, there is no `foo delete` operation; you must use `interp delete foo`.

If you have a deep hierarchy of interpreters, the command corresponding to the slave is defined only in the parent. For example, if a master creates `foo`, and `foo` creates `bar`, then the master must operate on `bar` with the `interp` command. There is no "`foo bar`" command defined in the master.

Use `list` with `interp eval`

The `interp eval` command treats its arguments like `eval`. If there are extra arguments, they are all concatenated together first. This can lose important structure, as described in Chapter 10. To be safe, use `list` to construct your commands. For example, to safely define a variable in the slave, you should do this:

```
interp eval slave [list set var $value]
```

Safe Interpreters

A child can be created either safe (i.e., untrusted) or fully functional. In the examples so far, the children have been trusted and fully functional; they have all the basic Tcl commands available to them. An interpreter is made safe by eliminating certain commands. Table 19–2 lists the commands removed from safe interpreters. As described later, these commands can be used by the master on behalf of the safe interpreter. To create a safe interpreter, use the `-safe` flag:

```
interp create -safe untrusted
```

Table 19–2 Commands hidden from safe interpreters.

`cd`	Changes directory.
`exec`	Executes another program.
`exit`	Terminates the process.
`fconfigure`	Sets modes of an I/O stream.
`file`	Queries file attributes.
`glob`	Matches on file name patterns.
`load`	Dynamically loads object code.
`open`	Opens files and process pipelines.
`pwd`	Determines the current directory.
`socket`	Opens network sockets.
`source`	Loads scripts.

A safe interpreter does not have commands to manipulate the file system and other programs (e.g., `cd`, `open`, and `exec`). This ensures that untrusted scripts cannot harm the host computer. The `socket` command is removed so that untrusted scripts cannot access the network. The `exit`, `source`, and `load` commands are removed so that an untrusted script cannot harm the hosting application. Note that commands like `puts` and `gets` are *not* removed. A safe interpreter can still do I/O, but it cannot create an I/O channel. We will show how to pass an I/O channel to a child interpreter on page 281.

The initial state of a safe interpreter is very safe, but it is too limited. The only thing a safe interpreter can do is compute a string and return that value to the parent. By creating command aliases, a master can give a safe interpreter controlled access to resources. A *security policy* implements a set of command aliases that add controlled capabilities to a safe interpreter. We will show, for example, how to provide limited network and file system access to untrusted slaves. Tcl provides a framework to manage several security policies, which is described in Chapter 20.

Command Aliases

A *command alias* is a command in one interpreter that is implemented by a command in another interpreter. The master interpreter installs command aliases in its slaves. The command to create an alias has the following general form:

```
interp alias slave cmd1 target cmd2 ?arg arg ...?
```

This creates *cmd1* in *slave* that is an alias for *cmd2* in *target*. When *cmd1* is invoked in *slave*, *cmd2* is invoked in *target*. The alias mechanism is transparent to the slave. Whatever *cmd2* returns, the slave sees as the return value of *cmd1*. If *cmd2* raises an error, the error is propagated to the slave.

Name the current interpreter with {}.

If *target* is the current interpreter, name it with {}. The empty list is the way to name yourself as the interpreter. This is the most common case, although *target* can be a different slave. The *slave* and *target* can even be the same interpreter.

The arguments to *cmd1* are passed to *cmd2*, after any additional arguments to *cmd2* that were specified when the alias was created. These hidden arguments provide a safe way to pass extra arguments to an alias. For example, it is quite common to pass the name of the slave to the alias. In Example 19–3, `exit` in the interpreter `foo` is an alias that is implemented in the current interpreter (i.e., {}). When the slave executes `exit`, the master executes:

```
interp delete foo
```

Example 19–3 A command alias for `exit`.

```
interp create foo
interp alias foo exit {} interp delete foo
interp eval foo exit
# Child foo is gone.
```

Alias Introspection

You can query what aliases are defined for a child interpreter. The `interp aliases` command lists the aliases; the `interp alias` command can also return the value of an alias, and the `interp target` command tells you what interpreter implements an alias. These are illustrated in the following examples:

Example 19–4 Querying aliases.

```
proc Interp_ListAliases {name out} {
    puts $out "Aliases for $name"
    foreach alias [interp aliases $name] {
        puts $out [format "%-20s => (%s) %s" $alias \
                [interp target $name $alias] \
                [interp alias $name $alias]]
    }
}
```

Example 19–4 generates output in a human readable format. Example 19–5 generates the aliases as Tcl commands that can be used to re-create them later:

Example 19–5 Dumping aliases as Tcl commands.

```
proc Interp_DumpAliases {name out} {
   puts $out "# Aliases for $name"
   foreach alias [interp aliases $name] {
      puts $out [format "interp alias %s %s %s %s" \
         $name $alias [list [interp target $name $alias]] \
         [interp alias $name $alias]]
   }
}
```


Hidden Commands

The commands listed in Table 19–2 are *hidden* instead of being completely removed. A hidden command can be invoked in a slave by its master. For example, a master can load Tcl scripts into a slave by using its hidden `source` command:

```
interp create -safe slave
interp invokehidden slave source filename
```

Without hidden commands, the master has to do a bit more work to achieve the same thing. It must open and read the file and `eval` the contents of the file in the slave. File operations are described in Chapter 9.

```
interp create -safe slave
set in [open filename]
interp eval slave [read $in]
close $in
```

Hidden commands were added in Tcl 7.7 in order to better support the Tcl/Tk browser plug-in described in Chapter 20. In some cases, hidden commands are strictly necessary; it is not possible to simulate them any other way. The best examples are in the context of Safe-Tk, where the master creates widgets or does potentially dangerous things on behalf of the slave. These will be discussed in more detail later.

A master can hide and expose commands using the `interp hide` and `interp expose` operations, respectively. You can even hide Tcl procedures. However, the commands inside the procedure run with the same privilege as that of the slave. For example, if you are really paranoid, you might not want an untrusted interpreter to read the clock or get timing information. You can hide the `clock` and `time` commands:

```
interp create -safe slave
interp hide slave clock
interp hide slave time
```

You can remove commands from the slave entirely like this:

```
interp eval slave [list rename clock {}]
interp eval slave [list rename time {}]
```

Substitutions

You must be aware of Tcl parsing and substitutions when commands are invoked in other interpreters. There are three cases corresponding to `interp eval`, `interp invokehidden`, and command aliases.

With `interp eval` the command is subject to a complete round of parsing and substitutions in the target interpreter. This occurs after the parsing and substitutions for the `interp eval` command itself. In addition, if you pass several arguments to `interp eval`, those are concatenated before evaluation. This is similar to the way the `eval` command works as described in Chapter 19. The most reliable way to use `interp eval` is to construct a list to ensure the command is well structured:

```
interp eval slave [list cmd arg1 arg2]
```

With hidden commands, the command and arguments are taken directly from the arguments to `interp invokehidden`, and there are no substitutions done in the target interpreter. This means that the master has complete control over the command structure, and nothing funny can happen in the other interpreter. For this reason you should not create a list. If you do that, the whole list will be interpreted as the command name! Instead, just pass separate arguments to `interp invokehidden` and they are passed straight through to the target:

```
interp invokehidden slave command arg1 arg2
```

Never `eval` alias arguments.

With aliases, all the parsing and substitutions occur in the slave before the alias is invoked in the master. The alias implementation should never `eval` or `subst` any values it gets from the slave to avoid executing arbitrary code.

For example, suppose there is an alias to open files. The alias does some checking and then invokes the hidden `open` command. An untrusted script might pass `[exit]` as the name of the file to open in order to create mischief. The untrusted code is hoping that the master will accidentally `eval` the filename and cause the application to exit. This attack has nothing to do with opening files; it just hopes for a poor alias implementation. Example 19–6 shows an alias that is not subject to this attack:

Example 19–6 Substitutions and hidden commands.

```
interp alias slave open {} safeopen slave
proc safeopen {slave filename {mode r}} {
   # do some checks, then...
   interp invokehidden $slave open $filename $mode
}
interp eval slave {open \[exit\]}
```

The command in the slave starts out as:

```
open \[exit\]
```

The master has to quote the brackets in its `interp eval` command or else the slave will try to invoke `exit` because of command substitution. Presumably `exit` isn't defined, or it is defined to terminate the slave. Once this quoting is done, the value of `filename` is `[exit]` and it is not subject to substitutions. It is safe to use `$filename` in the `interp invokehidden` command because it is only substituted once, in the master. The hidden `open` command also gets `[exit]` as its filename argument, which is never evaluated as a Tcl command.

I/O from Safe Interpreters

A safe child interpreter cannot open files or network sockets directly. An alias can create an I/O channel (i.e., open a file or socket) and give the child access to it. The parent can share the I/O channel with the child, or it can transfer the I/O channel to the child. If the channel is shared, both the parent and the child can use it. If the channel is transferred, the parent no longer has access to the channel. In general, transferring an I/O channel is simpler, but sharing an I/O channel gives the parent more control over an unsafe child. The differences are illustrated in Example 19–7 and Example 19–9.

There are three properties of I/O channels that are important to consider when choosing between sharing and transferring: the name, the seek offset, and the reference count.

- The name of the I/O channel (e.g., `file4`) is the same in all interpreters. If a parent transfers a channel to a child, it can close the channel by evaluating a `close` command in the child. Although names are shared, an interpreter cannot attempt I/O on a channel to which it has not been given access.
- The seek offset of the I/O channel is shared by all interpreters that share the I/O channel. An I/O operation on the channel updates the seek offset for all interpreters that share the channel. This means that if two interpreters share an I/O channel, their output will be cleanly interleaved in the channel. If they both read from the I/O channel, they will get different data. Seek offsets are explained in more detail on page 114.
- A channel has a reference count of all interpreters that share the I/O channel. The channel remains open until all references are closed. When a parent transfers an I/O channel, the reference count stays the same. When a parent shares an I/O channel, the reference count increments by one. When an interpreter closes a channel with `close`, the reference count is decremented by one. When an interpreter is deleted, all of its references to I/O channels are removed.

The syntax of commands to share or transfer an I/O channel is:

```
interp share interp1 chanName interp2
interp transfer interp1 chanName interp2
```

In these commands, *chanName* exists in *interp1* and is being shared or transferred to *interp2*. As with command aliases, if *interp1* is the current interpreter, name it with {}.

The following example creates a temporary file for an unsafe interpreter. The file is opened for reading and writing, and the slave can use it to store data temporarily.

Example 19–7 Opening a file for an unsafe interpreter.

```
proc TempfileAlias {slave} {
    set i 0
    while {[file exists Temp$slave$i]} {
        incr i
    }
    set out [open Temp$slave$i w+]
    interp transfer {} $out $slave
    return $out
}
proc TempfileExitAlias {slave} {
    foreach file [glob -nocomplain Temp$slave*] {
        file delete -force $file
    }
    interp delete $slave
}
interp create -safe foo
interp alias foo Tempfile {} TempfileAlias foo
interp alias foo exit {} TempfileExitAlias foo
```

The `TempfileAlias` procedure is invoked in the parent when the child interpreter invokes `Tempfile`. `TempfileAlias` returns the name of the open channel, which becomes the return value from `Tempfile`. `TempfileAlias` uses `interp transfer` to pass the I/O channel to the child so that the child has permission to access the I/O channel. In this example, it would also work to invoke the hidden `open` command to create the I/O channel directly in the slave.

Example 19–7 is not fully safe because the unsafe interpreter can still overflow the disk or create a million files. Because the parent has transferred the I/O channel to the child, it cannot easily monitor the I/O activity by the child. Example 19–9 addresses these issues.

The Safe Base

An safe interpreter created with `interp create -safe` has no script library environment and no way to source scripts. Tcl provides a *safe base* that extends a raw safe interpreter with the ability to source scripts and packages which are described in Chapter 12. The safe base also defines an `exit` alias that terminates the slave like the one in Example 19–7. The safe base is implemented as Tcl scripts that are part of the standard Tcl script library. Create an interpreter that uses the safe base with `safe::interpCreate`:

```
safe::interpCreate foo
```

The safe base has `source` and `load` aliases that only access directories on an *access path* defined by the master interpreter. The master has complete control over what files can be loaded into a slave. In general, it would be all right to source any Tcl program into an untrusted interpreter. However, untrusted scripts might learn things from the error messages they get by sourcing arbitrary files. The safe base also has versions of the `package` and `unknown` commands that support the library facility. Table 19–3 lists the Tcl procedures in the safe base:

Table 19–3 The safe base master interface.

`safe::interpCreate ?`*`slave`*`? ?`*`options`*`?`	Creates a safe interpreter and initialize the security policy mechanism.
`safe::interpInit` *`slave ?options?`*	Initializes a safe interpreter so it can use security policies.
`safe::interpConfigure` *`slave ?options?`*	Options are `-accessPath` *`pathlist`*, `-nostatics`, `-deleteHook` *`script`*, `-nestedLoadOk`.
`safe::interpDelete` *`slave`*	Deletes a safe interpreter.
`safe::interpAddToAccessPath` *`slave directory`*	Adds a directory to the slave's access path.
`safe::interpFindInAccessPath`	Maps from a directory to the token visible in the slave for that directory.
`safe::setLogCmd ?`*`cmd arg ... ?`*	Sets or queries the logging command used by the safe base.

Table 19–4 lists the aliases defined in a safe interpreter by the safe base.

Table 19–4 The safe base slave aliases.

`source`	Loads scripts from directories in the access path.
`load`	Loads binary extensions from the slaves access path.
`file`	Only the `dirname`, `join`, `extension`, `root`, `tail`, `pathname`, and `split` operations are allowed.
`exit`	Destroys the slave interpreter.

Security Policies

A *security policy* defines what a safe interpreter can do. Designing security policies that are secure is difficult. If you design your own, make sure to have your colleagues review the code. Give out prizes to folks who can break your policy. Good policy implementations are proven with lots of review and trial attacks.

The good news is that Safe-Tcl security policies can be implemented in relatively small amounts of Tcl code. This makes them easier to analyze and get correct. Here are a number of rules of thumb:

- Small policies are better than big, complex policies. If you do a lot of complex processing to allow or disallow access to resources, chances are there are holes in your policy. Keep it simple.
- Never `eval` arguments to aliases. If an alias accepts arguments that are passed by the slave, you must avoid being tricked into executing arbitrary Tcl code. The primary way to avoid this is never to `eval` arguments that are passed into an alias. Watch your expressions, too. The `expr` command does an extra round of substitutions, so brace all your expressions so that an attacker cannot pass `[exit]` where you expect a number!
- Security policies do not compose. Each time you add a new alias to a security policy, it changes the nature of the policy. Even if *`alias1`* and *`alias2`* are safe in isolation, there is no guarantee that they cannot be used together to mount an attack. Each addition to a security policy requires careful review.

Limited Socket Access

The `Safesock` security policy provides limited socket access. The policy is designed around a simple table of allowed hosts and ports. An untrusted interpreter can connect only to addresses listed in the table. For example, I would never let untrusted code connect to the *sendmail*, *ftp*, or *telnet* ports on my hosts. There are just too many attacks possible on these ports. On the other hand, I might want to let untrusted code fetch a URL from certain hosts, or connect to a database server for an intranet application. The goal of this policy is to have a simple way to specify exactly what hosts and ports a slave can access. Example 19–8 shows a simplified version of the `Safesock` security policy that is distributed with Tcl 8.0.

Example 19–8 The `Safesock` security policy.

```
# The index is a host name, and the
# value is a list of port specifications, which can be
# an exact port number
# a lower bound on port number: N-
# a range of port numbers, inclusive: N-M
array set safesock {
    sage.eng      3000-4000
    www.sun.com   80
    webcache.eng  {80 8080}
    bisque.eng    {80 1025-}
}
proc Safesock_PolicyInit {slave} {
    interp alias $slave socket {} SafesockAlias $slave
}
```

```
proc SafesockAlias {slave host port} {
    global safesock
    if ![info exists safesock($host)] {
        error "unknown host: $host"
    }

    foreach portspec $safesock($host) {
        set low [set high ""]
        if {[regexp {^([0-9]+)-([0-9]*)$} $portspec x low high]} {
            if {($low <= $port && $high == "") ||
                    ($low <= $port && $high >= $port)} {
                set good $port
                break
            }
        } elseif {$port == $portspec} {
            set good $port
        }
    }

    if [info exists good] {
        set sock [interp invokehidden $slave socket $host $good]
        interp invokehidden $slave fconfigure $sock \
            -blocking 0
        return $sock
    }
    error "bad port: $port"
}
```

The policy is initialized with `Safesock_PolicyInit`. The name of this procedure follows a naming convention used by the safe base. In this case, a single alias is installed. The alias gives the slave a `socket` command that is implemented by `SafesockAlias` in the master.

The alias checks for a port that matches one of the port specifications for the host. If a match is found, then the `invokehidden` operation is used to invoke two commands in the slave. The `socket` command creates the network connection, and the `fconfigure` command puts the socket into nonblocking mode so that `read` and `gets` by the slave do not block the application:

```
set sock [interp invokehidden $slave socket $host $good]
interp invokehidden $slave fconfigure $sock -blocking 0
```

The `socket` alias in the slave does not conflict with the hidden `socket` command. There are two distinct sets of commands, hidden and exposed. It is quite common for the alias implementation to invoke the hidden command after various permission checks are made.

The Tcl Web browser plug-in ships with a slightly improved version of the `Safesock` policy. It adds an alias for `fconfigure` so that the `http` package can set end of line translations and buffering modes. The `fconfigure` alias does not let you change the blocking behavior of the socket. The policy has also been extended to classify hosts into trusted and untrusted hosts based on their address. A different table of allowed ports is used for the two classes of hosts. The classification is done with two tables: One table lists patterns that match trusted

hosts, and the other table lists hosts that should not be trusted even though they match the first table. The improved version also lets a downloaded script connect to the Web server that it came from. The Web browser plug-in is described in Chapter 20.

Limited Temporary Files

Example 19–9 improves on Example 19–7 by limiting the number of temporary files and the size of the files. It is written to work with the safe base, so it has a `Tempfile_PolicyInit` that takes the name of the slave as an argument. `TempfileOpenAlias` lets the child specify a file by name, yet it limits the files to a single directory.

The example demonstrates a shared I/O channel that gives the master control over output. `TempfilePutsAlias` restricts the amount of data that can be written to a file. By sharing the I/O channel for the temporary file, the slave can use commands like `gets`, `eof`, and `close`, while the master does the `puts`. The need for shared I/O channels is somewhat reduced by hidden commands, which were added to Safe-Tcl more recently than shared I/O channels. For example, the `puts` alias can either write to a shared channel after checking the file size, or it can invoke the hidden `puts` in the slave. This alternative is shown in Example 19–10.

Example 19–9 The `Tempfile` security policy.

```
# Policy parameters:
#  directory is the location for the files
#  maxfile is the number of files allowed in the directory
#  maxsize is the max size for any single file.

array set tempfile {
   maxfile      4
   maxsize      65536
}
# tempfile(directory) is computed dynamically based on
# the source of the script

proc Tempfile_PolicyInit {slave} {
   global tempfile
   interp alias $slave open {} \
      TempfileOpenAlias $slave $tempfile(directory) \
         $tempfile(maxfile)
   interp alias $slave puts {} TempfilePutsAlias $slave \
      $tempfile(maxsize)
   interp alias $slave exit {} TempfileExitAlias $slave
}
proc TempfileOpenAlias {slave dir maxfile name {m r} {p 0777}} {
   global tempfile
   # remove sneaky characters
   regsub -all {|/:} [file tail $name] {} real
   set real [file join $dir $real]
```

```
        # Limit the number of files
        set files [glob -nocomplain [file join $dir *]]
        set N [llength $files]
        if {($N >= $maxfile) && (\
                [lsearch -exact $files $real] < 0)} {
            error "permission denied"
        }
        if [catch {open $real $m $p} out] {
            return -code error "$name: permission denied"
        }
        lappend tempfile(channels,$slave) $out
        interp share {} $out $slave
        return $out
    }
    proc TempfileExitAlias {slave} {
        global tempfile
        interp delete $slave
        if [info exists tempfile(channels,$slave)] {
            foreach out $tempfile(channels,$slave) {
                catch {close $out}
            }
            unset tempfile(channels,$slave)
        }
    }
    # See also the puts alias in Example 22–4 on page 327
    proc TempfilePutsAlias {slave max chan args} {
        # max is the file size limit, in bytes
        # chan is the I/O channel
        # args is either a single string argument,
        # or the -nonewline flag plus the string.

        if {[llength $args] > 2} {
            error "invalid arguments"
        }
        if {[llength $args] == 2} {
            if {![string match -n* [lindex $argv 0]]} {
                error "invalid arguments"
            }
            set string [lindex $args 1]
        } else {
            set string [lindex $args 0]\n
        }
        set size [expr [tell $chan] + [string length $string]]
        if {$size > $max} {
            error "File size exceeded"
        } else {
            puts -nonewline $chan $string
        }
    }
```

The `TempfileAlias` procedure is generalized in Example 19–9 to have parameters that specify the directory, name, and a limit to the number of files allowed. The `directory` and `maxfile` limit are part of the alias definition. Their existence is transparent to the slave. The slave specifies only the name and

access mode (i.e., for reading or writing.) The `Tempfile` policy can be used by different slave interpreters with different parameters.

The master is careful to restrict the files to the specified directory. It uses `file tail` to strip off any leading pathname components that the slave might specify. The `tempfile(directory)` definition is not shown in the example. The application must choose a directory when it creates the safe interpreter. The `Browser` security policy described on page 300 chooses a directory based on the name of the URL containing the untrusted script.

The `TempfilePutsAlias` procedure implements a limited form of `puts`. It checks the size of the file with `tell` and measures the output string to see if the total exceeds the limit. The limit comes from a parameter defined when the alias is created. The file cannot grow past the limit, at least not by any action of the child interpreter. The `args` parameter is used to allow an optional `-nonewline` flag to `puts`. The value of `args` is checked explicitly instead of using the `eval` trick described in Example 10–3 on page 127. Never `eval` arguments to aliases or else a slave can attack you with arguments that contain embedded Tcl commands.

The master and slave share the I/O channel. The name of the I/O channel is recorded in `tempfile`, and `TempfileExitAlias` uses this information to close the channel when the child interpreter is deleted. This is necessary because both parent and child have a reference to the channel when it is shared. The child's reference is automatically removed when the interpreter is deleted, but the parent must close its own reference.

The shared I/O channel lets the master use `puts` and `tell`. It is also possible to implement this policy by using hidden `puts` and `tell` commands. The reason `tell` must be hidden is to prevent the slave from implementing its own version of `tell` that lies about the seek offset value. One advantage of using hidden commands is that there is no need to clean up the tempfile state about open channels. You can also layer the puts alias on top of any existing puts implementation. For example, a script may define `puts` to be a procedure that inserts data into a text widget. Example 19–10 shows the difference when using hidden commands.

Example 19–10 Restricted `puts` using hidden commands.

```
proc Tempfile_PolicyInit {slave} {
    global tempfile
    interp alias $slave open {} \
        TempfileOpenAlias $slave $tempfile(directory) \
            $tempfile(maxfile)
    interp hide $slave tell
    interp alias $slave tell {} TempfileTellAlias $slave
    interp hide $slave puts
    interp alias $slave puts {} TempfilePutsAlias $slave \
        $tempfile(maxsize)
    # no special exit alias required
}
proc TempfileOpenAlias {slave dir maxfile name {m r} {p 0777}} {
```

```
        # remove sneaky characters
        regsub -all {|/:} [file tail $name] {} real
        set real [file join $dir $real]
        # Limit the number of files
        set files [glob -nocomplain [file join $dir *]]
        set N [llength $files]
        if {($N >= $maxfile) && (\
                [lsearch -exact $files $real] < 0)} {
            error "permission denied"
        }
        if [catch {interp invokehidden $slave \
                open $real $m $p} out] {
            return -code error "$name: permission denied"
        }
        return $out
    }
    proc TempfileTellAlias {slave chan} {
        interp invokehidden $slave tell $chan
    }
    proc TempfilePutsAlias {slave max chan args} {
        if {[llength $args] > 2} {
            error "invalid arguments"
        }
        if {[llength $args] == 2} {
            if {![string match -n* [lindex $args 0]]} {
                error "invalid arguments"
            }
            set string [lindex $args 1]
        } else {
            set string [lindex $args 0]\n
        }
        set size [interp invokehidden $slave tell $chan]
        incr size [string length $string]
        if {$size > $max} {
            error "File size exceeded"
        } else {
            interp invokehidden $slave \
                puts -nonewline $chan $string
        }
    }
```

Safe `after` Command

The `after` command is unsafe because it can block the application for an arbitrary amount of time. This happens if you only specify a time but do not specify a command. In this case, Tcl just waits for the time period and processes no events. This will stop all interpreters, not just the one doing the `after` command. This is a kind of *resource attack*. It doesn't leak information or damage anything, but it disrupts the main application.

Example 19–11 defines an alias that implements `after` on behalf of safe interpreters. The basic idea is to carefully check the arguments, and then do the `after` in the parent interpreter. As an additional feature, the number of out-

standing `after` events is limited. The master keeps a record of each `after` event scheduled. Two IDs are associated with each event: one chosen by the master (i.e., `myid`), and the other chosen by the `after` command (i.e., `id`). The master keeps a map from `myid` to `id`. The map serves two purposes: The number of map entries counts the number of outstanding events. The map also hides the real `after` ID from the slave, which prevents a slave from attempting mischief by specifying invalid after IDs to `after cancel`. The `SafeAfterCallback` is the procedure scheduled. It maintains state and then invokes the original callback in the slave.

Example 19–11 A safe `after` command.

```
# SafeAfter_PolicyInit creates a child with
# a safe after command

proc SafeAfter_PolicyInit {slave max} {
    # max limits the number of outstanding after events
    global after
    interp alias $slave after {} SafeAfterAlias $slave $max
    interp alias $slave exit {} SafeAfterExitAlias $slave
    # This is used to generate after IDs for the slave.
    set after(id,$slave) 0
}

# SafeAfterAlias is an alias for after. It disallows after
# with only a time argument and no command.

proc SafeAfterAlias {slave max args} {
    global after
    set argc [llength $args]
    if {$argc == 0} {
        error "Usage: after option args"
    }
    switch -- [lindex $args 0] {
        cancel {
            # A naive implementation would just
            # eval after cancel $args
            # but something dangerous could be hiding in args.
            set myid [lindex $args 1]
            if {[info exists after(id,$slave,$myid)]} {
                set id $after(id,$slave,$myid)
                unset after(id,$slave,$myid)
                after cancel $id
            }
            return ""
        }
        default {
            if {$argc == 1} {
                error "Usage: after time command args..."
            }
            if {[llength [array names after id,$slave,*]]\
                    >= $max} {
                error "Too many after events"
```

```
            }
            # Maintain concat semantics
            set command [concat [lrange $args 1 end]]
            # Compute our own id to pass the callback.
            set myid after#[incr after(id,$slave)]
            set id [after [lindex $args 0] \
               [list SafeAfterCallback $slave $myid $command]]
            set after(id,$slave,$myid) $id
            return $myid
        }
    }
}

# SafeAfterCallback is the after callback in the master.
# It evaluates its command in the safe interpreter.

proc SafeAfterCallback {slave myid cmd} {
    global after
    unset after(id,$slave,$myid)
    if [catch {
        interp eval $slave $cmd
    } err] {
        catch {interp eval $slave bgerror $error}
    }
}

# SafeAfterExitAlias is an alias for exit that does cleanup.

proc SafeAfterExitAlias {slave} {
    global after
    foreach id [array names after id,$slave,*] {
        after cancel $after($id)
        unset after($id)
    }
    interp delete $slave
}
```

CHAPTER 20

Safe-Tk and the Browser Plugin

This chapter describes Safe-Tk that lets untrusted scripts display and manipulate graphical user interfaces. The main application of Safe-Tk is the Tcl/Tk plugin for Web browsers like Netscape Navigator and Internet Explorer.

Safe-Tk supports network applets that display user interfaces. The main vehicle for Safe-Tk is a plugin for Netscape Navigator and Internet Explorer. The plugin supports Tcl applets, or *Tclets*, that are downloaded from the Web server and execute inside a window in a Web browser. For the most part, Tcl/Tk applications can run unchanged in the plugin. However, security policies place some restrictions on Tclets. The plugin supports multiple security policies, so Tclets can do a variety of interesting things in a safe manner.

The current version of the plugin uses Tcl/Tk 8.0. You can configure the plugin to use an existing *wish* application to host the Tcl applets, or the plugin can load the Tcl/Tk shared libraries and everything runs in the browser process. You can use a custom *wish* that has extensions built in or dynamically loaded. This gives intranet applications of the plugin the ability to access databases and other services that are not provided by the Tcl/Tk core. With the security policy mechanism you can still provide mediated access to these resources. This chapter describes how to set up the plugin.

The source code of the plugin is freely available. You can recompile the plugin against newer versions of Tcl/Tk, or build custom plugins that have your own Tcl extensions built in. One particularly active plugin user is NASA, which maintains and distributes an enhanced version of the plugin. You can find them from the main plugin Web site at:

```
http://www.scriptics.com/plugin/
```

Tk in Child Interpreters

A child interpreter starts out with just the core Tcl commands. It does not include Tk or any other extensions that might be available to the parent interpreter. This is true whether or not the child interpreter is declared safe. You add extensions to child interpreters by using a form of the `load` command that specifies an interpreter:

```
load {} Tk child
```

Normally, `load` takes the name of the library file that contains the extension. In this case, the Tk package is a *static package* that is already linked into the program (e.g., *wish* or the plugin), so the file name is the empty string. The `load` command calls the Tk initialization procedure to register all the Tcl commands provided by Tk.

Embedding Tk Windows

By default, a slave interpreter that loads Tk gets a new top-level window. *Wish* supports a `-use` command line option that directs Tk to use an existing window as dot. You can use this to embed an application within another. For example, the following commands run a copy of *Wish* that uses the `.embed` toplevel as its main window:

```
toplevel .embed
exec wish -use [winfo id .embed] somescript.tcl &
```

More often, embedding is used with child interpreters. If the interpreter is *not* safe, you can set the `argv` and `argc` variables in the slave before loading Tk:

```
interp create trustedTk
interp eval trustedTk \
     [list set argv [list -use [winfo id .embed]]]
interp eval trustedTk [list set argc 2]
load {} Tk trustedTk
```

If the child interpreter is safe, then you cannot set `argv` and `argc` directly. The easiest way to pass `-use` to a safe interpreter is with the `safe::loadTk` command:

```
safe::interpCreate safeTk
safe::loadTk safeTk -use [winfo id .embed]
```

When Tk is loaded into a safe interpreter, it calls back into the master interpreter and evaluates the `safe::TkInit` procedure. The job of this procedure is to return the appropriate `argv` value for the slave. The `safe::loadTk` procedure stores its additional arguments in the `safe::tkInit` variable, and this value is retrieved by the `safe::TkInit` procedure and returned to the slave. This protocol is used so that a safe interpreter cannot attempt to hijack the windows of its master by constructing its own `argv` variable!

Safe-Tk Restrictions

When Tk is loaded into a safe interpreter, it hides several Tk commands. Primarily these are hidden to prevent *denial of service* attacks against the main process. For example, if a child interpreter did a global `grab` and never released it, all input would be forever directed to the child. Table 20–1 lists the Tk commands hidden by default from a safe interpreter. The Tcl commands that are hidden in safe interpreters are listed on page 277.

Table 20–1 Tk commands omitted from safe interpreters.

`bell`	Rings the terminal bell.
`clipboard`	Accesses the `CLIPBOARD` selection.
`grab`	Directs input to a specified widget.
`menu`	Creates and manipulates menus, because menus need `grab`.
`selection`	Manipulates the selection.
`send`	Executes a command in another Tk application.
`tk`	Sets the application name.
`tk_chooseolor`	Color choice dialog.
`tk_getOpenFile`	File open dialog.
`tk_getSaveFile`	File save dialog.
`tk_messageBox`	Simple dialog boxes.
`toplevel`	Creates a detached window.
`wm`	Controls the window manager.

If you find these restrictions limiting, you can restore commands to safe interpreters with the `interp expose` command. For example, to get menus and toplevels working, you could do:

```
interp create -safe safeTk
foreach cmd {grab menu menubutton toplevel wm} {
    interp expose safeTk $cmd
}
```

Instead of exposing the command directly, you can also construct aliases that provide a subset of the features. For example, you could disable the `-global` option to grab. Aliases are described in detail in Chapter 19.

The Browser plugin defines a more elaborate configuration system to control what commands are available to slave interpreters. You can have lots of control, but you need to distribute the *security policies* that define what Tclets can do in the plugin. Configuring security policies for the plugin is described later.

The Browser Plugin

The HTML EMBED tag is used to put various objects into a Web page, including a Tcl program. For example:

```
<EMBED src=eval.tcl width=400 height=300>
```

The width and height are interpreted by the plugin as the size of the embedded window. The src specifies the URL of the program. These parameter names (e.g., width) are case sensitive and should be lowercase. In the above example, eval.tcl is a relative URL, so it should be in the same directory as the HTML file that has the EMBED tag. The window size is fixed in the browser, which is different from normal toplevels in Tk. The plugin turns off geometry propagation on your main window so that your Tclet stays the size allocated.

There are also "full window" Tclets that do not use an EMBED tag at all. Instead, you just specify the .tcl file directly in the URL. For example, you can type this into your browser, or use it as the HREF parameter in a URL link:

```
http://www.beedub.com/plugin/bike.tcl
```

In this case, the plugin occupies the whole browser window and will resize as you resize the browser window.

The embed_args and plugin Variables

The parameters in the EMBED tag are available to the Tcl program in the embed_args variable, which is an array with the parameter names as the index values. For example, the string for a ticker-tape Tclet can be passed in the EMBED tag as the string parameter, and the Tclet will use $embed_args(string) as the value to display:

```
<EMBED src=ticker.tcl width=400 height=50 string="Hello World">
```

Note that HTML tag parameters are case sensitive. Your Tclet may want to map all the parameter names to lowercase for convenience:

```
foreach {name value} [array get embed_args] {
    set embed_args([string tolower $name]) $value
}
```

The plugin array has version, patchLevel, and release elements that identify the version and release date of the plugin implementation.

Example Plugins

The plugin home page is a great place to find Tclet examples. There are several plugins done by the Tcl/Tk team at Sunlabs, plus links to a wide variety of Tclets done on the Net.

```
http://www.scriptics.com/plugin/
```

I wrote a cute little plugin that calculates the effective wheel diameter of multigear bicycles. Brian Lewis, who built the Tcl 8.0 byte-code compiler, explained to me the concept and how important this information is to bicycle

enthusiasts. I put together a Tclet that displays the gear combinations on a Tk canvas and lets you change the number of gears and their size. You can find the result at:

```
http://www.beedub.com/plugin/bike.html
```


Setting Up the plugin

There are plugin versions for UNIX, Windows, and Macintosh. The installation scripts take care of installing the plugin in the correct locations, which are described in the next sections about each platform. The plugin and the security policies that are distributed with it will continue to be updated. You can get the latest version from the Tcl/Tk Web site, `http://www.scriptics.com/plugin/`. If that URL changes, you can find an up-to-date pointer under `http://www.beedub.com/book/`. The plugin may already be installed at your site. Bring up the `About Plugins` dialog under `Help` in your browser to see if the Tcl/Tk plugin is listed.

The plugin is composed of the following parts, although the location of these files varies somewhat among platforms:

- The plugin shared libraries (i.e., DLLs). The Web browser dynamically loads the plugin implementation when it needs to execute a Tclet embedded in a Web page. There is a standard directory that the browser scans for the libraries that implement plugins.
- The Tcl/Tk script libraries. The plugin needs the standard script libraries that come with Tcl and Tk, plus it has its own scripts that complete its implementation. Each platform has a plugin script directory with these subdirectories: `tcl`, `tk`, `plugin`, `config`, `safetcl`, and `utils`. The plugin implementation is in the `plugin` directory.
- The security policies. These are kept in a `safetcl` directory that is a peer of the Tcl script library.
- The trust configuration. This defines what Tclets can use which security policies. This is in a `config` directory that is a peer of the Tcl script library.
- Local hooks. Local customization is supported by two hooks, `siteInit` and `siteSafeInit`. The `siteInit` procedure is called from the plugin when it first loads, and `siteSafeInit` is called when each applet is initialized. It is called with the name of the slave interpreter and the list of arguments from the `<EMBED>` tag. You can provide these as scripts that get loaded from the `auto_path` of the master interpreter. Chapter 12 describes how to manage script libraries found in the `auto_path`. The plugin also sources a personal start up script in which you can define `siteInit` and `siteSafeInit`. This script is `~/.pluginrc` on UNIX and `plugin/tclplugin.rc` on Windows and Macintosh.

Plugin Environment Variables

The plugin can be configured to run Tcl/Tk directly in the browser process,

or to run with a small library in the browser that communicates with an external *wish* application. The default is to run in process. The advantage of the external process is that you can use custom *wish* shells that can load various extensions. Table 20–2 shows the environment variables used to control the plugin configuration.

Table 20–2 Plugin Environment Variables

TCL_PLUGIN_INPROCESS	If this is defined and 1, then Tcl/Tk is loaded directly into the browser. Otherwise, the plugin forks *wish*.
TCL_PLUGIN_WISH	This names the *wish* executable used to run Tclets. This must be version 8.0 or higher to properly support embedding.
TCL_PLUGIN_CONSOLE	If this is set to 1, then a console is opened when a Tclet is loaded. The console prompts for Tcl commands that are evaluated in the master interpreter. If the value is something other than 1, then it is taken to be a script (e.g., *TkCon*) that implements a console.
TCL_PLUGIN_LOGWINDOW	If 1, various status messages from the plugin are displayed.
TCL_PLUGIN_LOGFILE	If defined, this file captures the log output.

UNIX Configuration

Netscape looks in each user's `~/.netscape/plugins` for the shared libraries that implement plugins. It also looks in a `plugins` directory under its main directory, which will vary from site to site. You can define a search path for plugins with the `NXP_PLUGIN_PATH` environment variable. The plugin script library is in `~/.tclplug/2.0/plugin`. You can change this default location by setting the `TCL_PLUGIN_DIR` environment variable. Once the plugin finds its script library, it assumes that the Tcl and Tk script directories, the security policies, and the trust map are in peer directories.

Windows Configuration

The default location for plugins is in the `PLUGINS` directory of the Netscape installation. The Tcl/Tk plugin also works in Internet Explorer from the same location. The script libraries are found under `C:\TCLPLUG\2.0`. You can change this location by setting the registry variable:

```
Software\Sun\Tcl Plugin\2.0\Directory
```

Macintosh Configuration

Installing the plugin on the Macintosh is a three-step process. In step one, you unpack the initial download file, which creates another installer file. In step two, you run that installer, which puts files into two locations. You specify one folder that will hold the documentation and the Netscape plugin. The rest of the plugin goes into a `plugin` folder under the `Tool Command Language` folder in your

system's Extensions folder. It does not conflict with any existing Tcl/Tk installations you may already have. In step three, you complete the process by moving the Tcl Plugin file into the Plugins directory of your Netscape installation.

The current version of the Macintosh plugin is limited to Netscape. Version 4 works better than Netscape 3, and Internet Explorer is not supported at all.

Security Policies and Browser Plugin

Tclets run in a safe interpreter that is set up with the *safe base* facilities described on page 282. This limits a Tclet to a display-only application. To do something more interesting, you must grant the Tclet more privilege. The extra functions are bundled together into a *security policy*, which is implemented as a set of command aliases. Unlike a Java applet, a Tclet can choose from different security policies. A few standard security policies are distributed with the plugin, and these are described below. You can also create custom security policies to support intranet applications. You can even choose to grant certain Tclets the full power of Tcl/Tk. The policy command is used to request a security policy:

```
policy name
```

The policies that are part of the standard plugin distribution are described below. The home, inside, and outside policies all provide limited network access. They differ in what set of hosts are accessible. The default trust configuration lets any Tclet request the home, inside, or outside policy.

- home. This provides a socket and fconfigure commands that are limited to connecting to the host from which the Tclet was downloaded. You can specify an empty string for the host argument to socket to connect back to the home host. This policy also supports open and file delete that are similar to the Tempfile policy shown in Example 19–9 on page 286. This provides limited local storage that is inside a directory that is, by default, private to the Tclet. Files in the private directory persist after the Tclet exits, so it can maintain long term state. Tclets from the same server can share the directory by putting the same prefix=*partialurl* argument in their EMBED tag. The *partialurl* must be a prefix of the Tclet's URL. Finally, the home policy automatically provides a browser package that is described later.
- inside. This is just like the home policy, except that the site administrator controls a table of hosts and ports to which untrusted slaves can connect with socket. A similar set of tables control what URLs can be accessed with the browser package. This is similar to the Safesock policy shown in Example 19–8 on page 284. The set of hosts is supposed to be inside the firewall. The local file storage used by this policy is distinct from that used by the home and outside policies. This is true even if Tclets try to share by using the prefix=*partialurl* parameter.
- outside. This is just like the home and inside policies, except that the set of hosts is configured to be outside the firewall. The local file storage used by

this policy is distinct from that used by the `home` and `inside` policies.

- `trusted`. This policy restores all features of Tcl and Tk. This policy lets you launch all your Tcl and Tk applications from the Web browser. The default trust map settings do not allow this for any Tclet. The trust map configuration is described later.
- `javascript`. This policy provides a superset of the `browser` package that lets you invoke arbitrary Javascript and to write HTML directly to frames. This does not have the limited socket or temporary file access that the `home`, `inside`, and `outside` policies have. However, the `javascript` policy places no restrictions on the URLs you can fetch, plus it lets Tclets execute Javascript, which may have its own security risks. The default trust map settings do not allow this for any Tclet.

The Browser Package

The `browser` package is bundled with several of the security policies. It makes many features of the Web browser accessible to Tclets. They can fetch URLs and display HTML in frames. However, the `browser` package has some risks associated with it. HTTP requests can be used to transmit information, so a Tclet using the policy could leak sensitive information if it can fetch a URL outside the firewall. To avoid information leakage, the `inside`, `outside`, and `home` policies restrict the URL that can be fetched with `browser::getURL`. Table 20–3 lists the aliases defined by the `browser` package.

Table 20–3 Aliases defined by the `browser` package.

`browser::status` *`string`*	Displays *`string`* in the browser status window.
`browser::getURL` *`url`* ?*`timeout`*? ?*`newcallback`*? ?*`writecallback`*? ?*`endcallback`*?	Fetches *`url`*, if allowed by the security policy. The *`callbacks`* occur before, during, and after the *`url`* data is returned.
`browser::displayURL` *`url frame`*	Causes the browser to display *`url`* in *`frame`*.
`browser::getForm` *`url data`* ?*`raw`*? ?*`timeout`*? ?*`newcallback`*? ?*`writecallback`*? ?*`endcallback`*?	Posts data to url. The callbacks are the same as for browser::getURL. If *`raw`* is 0, then data is a name value list that gets encoded automatically. Otherwise, it is assumed to be encoded already.
`browser::displayForm` *`url frame data`* ?*`raw`*?	Posts *`data`* to *`url`* and displays the result in *`frame`*. The *`raw`* argument is the same as in `browser::getForm`.

The `browser::getURL` function uses the browser's built-in functions, so it understands proxies and supports `ftp:`, `http:`, and `file:` urls. Unfortunately, the `browser::getURL` interface is different from the `http::geturl` interface. It uses a more complex callback scheme that is due to the nature of the browser's

built-in functions. If you do not specify any callbacks, then the call blocks until all the data is received, and then that data is returned. The callback functions are described in Table 20–4.

Table 20–4 The browser::getURL callbacks.

`newcallback` *`name stream url mimetype datemodified size`*	This is called when data starts to arrive from *`url`*. The *`name`* identifies the requesting Tclet, and the *`stream`* identifies the connection. The *`mimetype`*, *`datemodified`*, and *`size`* parameters are attributes of the returned data.
`writecallback` *`name stream size data`*	This is called when *`size`* bytes of *`data`* arrive for Tcllet *`name`* over *`stream`*.
`endcallback` *`name stream reason data`*	This is called when the request has completed, although there may be some final bytes in *`data`*. The reason is one of: `EOF`, `NETWOR_ERROR`, `USER_BREAK`, or `TIMEOUT`.

Configuring Security Policies

There are three aspects to the plugin security policy mechanism: *policies*, *features*, and *trust maps*. A policy is an umbrella for a set of features that are allowed for certain Tclets based on the trust map. A feature is a set of commands and aliases that are defined for a safe interpreter that requests a policy. The trust map is a filter based on the URL of the Tclet. In the future, trust may bet determined by digital signatures instead of URLs. The trust map determines whether a Tclet can request a given policy.

Security Policies are configured for each client.

Remember that the configuration files affect the client machine, which is the workstation that runs the Web browser. If you create Tclets that require custom security policies, you have the burden of distributing the configuration files to clients that will use your Tclets. You also have the burden of convincing them that your security policy is safe!

The `config/plugin.cfg` File

The main configuration file is the `config/plugin.cfg` file in the plugin distribution. This file lists what features are supported by the plugin, and it defines the URL filters for the trust map.

The configuration file is defined into sections with a `section` command. The policies section defines which Tclets can use which security policies. For example, the default configuration file contains these lines in the policies section:

```
section policies
    allow home
    disallow intercom
    disallow inside
```

```
        disallow outside
        disallow trusted
        allow javascript ifallowed trustedJavaScriptURLS \
              $originURL
```

This configuration grants all Tclets the right to use the `home` policy, disallows all Tclets from using the `intercom, inside`, `outside`, and `trusted` policies, and grants limited access to the `javascript` policy.

If you are curious, the configuration files are almost Tcl, but not quite. I lost an argument about that one, so these are stylized configuration files that follow their own rules. For example, the `originURL` variable is not defined in the configuration file but is a value that is tested later when the Tclet is loaded. I'll just give examples here and you can peer under the covers if you want to learn how they are parsed.

The `ifallowed` clause depends on another section to describe the trust mapping for that policy. For the `javascript` policy, the `config/plugin.cfg` file contains:

```
section trustedJavascriptURLs
    allow http://sunscript.sun.com:80/plugin/javascript/*
```

Unfortunately, this server isn't running anymore, so you may want to add the Scriptics Web server to your own configuration:

```
    allow http://www.scriptics.com:80/plugin/javascript/*
```

You can use a combination of `allow` and `disallow` rules in a section. The arguments to `allow` and `disallow` are URL `string match` patterns, and they are processed in order. For example, you could put a liberal `allow` rule followed by `disallow` rules that restrict access, or vice versa. It is probably safest to explicitly list each server that you trust.

Policy Configuration Files

Each security policy has a configuration file associated with it. For example, the `outside` policy uses the file `outside.cfg` file in the `config` directory. This file specifies what hosts and ports are accessible to Tclets using the outside policy. For the `inside` and `outside` policies, the configuration files are similar in spirit to the `safesock` array used to configure the `Safesock` security policy shown on page 284. There are a set of allowed hosts and ports, and a set of excluded hosts. The excluded hosts are an exception list. If a host matches the included set but also matches the excluded set, it is not accessible. There is an included and excluded set for URLs that affect `browser::geturl`. The settings from the `Tempfile` policy shown on page 286 are also part of the `home`, `inside`, and `outside` configuration files. The configuration files are well commented, and you should read through them to learn about the configuration options for each security policy.

Security Policy Features

The aliases that make up a security policy are organized into sets called *features*. The features are listed in the main `config/plugin.cfg` configuration file:

```
variable featuresList {url stream network persist unsafe}
```

In turn, each security policy configuration file lists what features are part of the policy. For example, the `config/home.cfg` file lists these features:

```
section features
    allow url
    allow network
    allow persist unless {[string match {UNKNOWN *} \
        [getattr originURL]]}
```

Each feature is implemented in a file in the `safetcl` directory of the distribution. For example, the url feature is implemented in `safetcl/url.tcl`. The code in these files follows some conventions in order to work with the configuration mechanism. Each one is implemented inside a namespace that is a child of the `safefeature` namespace (e.g., `safefeature::url`). It must implement an `install` procedure that is called to initialize the feature for a new Tclet. It is inside this procedure that the various allow/disallow rules are checked. The `cfg::allowed` command supports the rule language used in the `.cfg` files.

Creating New Security Policies

This book does not describe the details of the configuration language or the steps necessary to create a new security policy. There are several manual pages distributed with the plugin that explain these details. They can be found on the Web at:

```
http://www.scriptics.com/plugin/man/
```

If you are serious about tuning the existing security policies or creating new ones, you should read the existing feature implementations in detail. As usual, modifying a working example is the best way to proceed! I think it is a very nice property of the plugin that its security policies are implemented in Tcl source code that is clearly factored out from the rest of the Tcl/Tk and plugin implementation. With a relatively small amount of code, you can create custom security policies that grant interesting abilities to Tclets.