Introduction to Programming with Tcl

By Shyam Pather

Information and Telecommunication Technology Center, University of Kansas

If you have reached this page from a search engine, you may want to check out the main Tcl/Tk page for other information and versions of this tutorial.

How to use this Document

This document is intended as an introductory tutorial to Tcl, and not as an exhaustive reference. I have attempted to highlight the most important aspects of the Tcl language, so as to provide some base material from which to get started. I have not included any material on the Tk toolkit, since there are already several Tk tutorials available on the internet. If your purpose in using Tcl/Tk is to write graphical user interfaces, then you might be tempted to rush out and learn the Tk commands and pick up the Tcl language as you go along. However, it is my belief that a solid foundation in the Tcl language is essential to writing successful Tcl/Tk code. Therefore, I would strongly recommend taking the time to go through this document and its examples before attempting to use Tk.

Most of the Tcl commands presented in this document can be used in many different ways, only a few of which are illustrated here. For this reason, I recommend reading the manual pages for all the commands referred to in this tutorial. For more complete and descriptive coverage of Tcl and Tk, refer to John Ousterhout's book Tcl and the Tk Toolkit, described on the Tcl/Tk Information page.

Introduction to Tcl

Tcl was originally intended to be a reusable command language. Its developers had been creating a number of interactive tools, each requiring its own command language. Since they were more interested in the tools themselves than the command languages they would employ, these command languages were constructed quickly, without regard to proper design.

After implementing several such "quick-and-dirty" command languages and experiencing problems with each one, they decided to concentrate on implementing a general-purpose, robust command language that could easily be integrated into new applications. Thus Tcl (Tool Command Language) was born.

Since that time, Tcl has been widely used as a scripting language. In most cases, Tcl is used in combination with the Tk ("Tool Kit") library, a set of commands and procedures that make it relatively easy to program graphical user interfaces in Tcl.
One of Tcl's most useful features is its extensibility. If an application requires some functionality not offered by standard Tcl, new Tcl commands can be implemented using the C language, and integrated fairly easily. Since Tcl is so easy to extend, many people have written extension packages for some common tasks, and made these freely available on the internet. (For more information, see the Tcl/Tk Information page).

### Tcl Programming Basics

The main difference between Tcl and languages such as C, is that Tcl is an *interpreted* rather than a *compiled* language. Tcl programs are simply scripts consisting of Tcl commands that are processed by a Tcl interpreter at run time. One advantage that this offers is that Tcl programs can themselves generate Tcl scripts that can be evaluated at a later time. This can be useful, for example, when creating a graphical user interface with a command button that needs to perform different actions at different times.

The next several sections describe the essential elements of Tcl programs. Each section is accompanied by a series of examples, and a sample Tcl interpreter that you can use to try out the examples yourself.

#### Variables and Variable Substitution

Variables in Tcl, as in most other languages, can be thought of as boxes in which various kinds of data can be stored. These boxes, or variables, are given names, which are then used to access the values stored in them.

Unlike C, Tcl does not require that variables be declared before they are used. Tcl variables are simply created when they are first assigned values, using the `set` command. Although they do not have to be deleted, Tcl variables can be deleted using the `unset` command.

The value stored in a variable can be accessed by prefacing the name of the variable with a dollar sign ("\$"). This is known as variable substitution, and is illustrated in the examples below.

Tcl is an example of a "weakly typed" language. This simply means that almost any type of data can be stored in any variable. For example, the same variable can be used to store a number, a date, a string, or even another Tcl script.

---

**Example 1.1:**

```tcl
set foo "john"
puts "Hi my name is $foo"
```

**Output:** Hi my name is john

Example 1.1 illustrates the use of variable substitution. The value "john" is assigned to the variable "foo", whose value is then substituted for "$foo". Note that variable substitution can occur within a string. The `puts` command (described in a later section) is used to display the string.

---

**Example 1.2:**

```tcl
set month 2
```

---
set day 3
set year 97
set date "$month:$day:$year"
puts $date

Output: 2:3:97

Here variable substitution is used in several places: The values of the variables "month", "day", and "year" are substituted in the `set` command that assigns the value of the "date" variable, and the value of the "date" variable is then substituted in the line that displays the output.

---

**Example 1.3:**

```tcl
set foo "puts hi"
eval $foo
```

Output: hi

In this example, the variable "foo" holds another (small) Tcl script that simply prints the word "hi". The value of the variable "foo" is substituted into an `eval` command, which causes it to be evaluated by the Tcl interpreter (the `eval` command will be described in greater detail in a later section).

**Try it yourself:**

Feel free to type in any of the above examples in this Tcl interpreter:

---

**Expressions**

Tcl allows several types of expressions, including mathematical expressions, and relational expressions. Tcl expressions are usually evaluated using the `expr` command, as illustrated in the examples below.

---

**Example 2.1:**

```tcl
expr 0 == 1
```
Output: 0

---

**Example 2.2:**

```tcl
expr 1 == 1
```

Output: 1

---

Examples 2.1 and 2.2 illustrate the use of relational expressions with the `expr` command. The first expression evaluates to 0 (false) since 0 does not equal 1, whereas the second expression evaluates to 1 (true), since, obviously, 1 does equal 1. The relational operator "==" is used to do the comparison.

---

**Example 2.3:**

```tcl
expr 4 + 5
```

Output: 9

---

Example 2.3 shows how to use the `expr` statement to evaluate an arithmetic expression. Here the result is simply the sum of 4 and 5. Tcl offers a rich set of arithmetic and relational operators, each of which is described in the `expr` manual page.

---

**Example 2.4:**

```tcl
expr sin(2)
```

Output: 0.909297

---

This example shows that the `expr` statement can be used to evaluate the result of a mathematical function, in this case, the sine of an angle. Tcl offers many such mathematical functions, also described on the `expr` manual page.

---

**Try it yourself:**

Feel free to type in any of the above examples in this Tcl interpreter:
Command Substitution

Just as variable substitution is used to substitute the value of a variable into a Tcl script, command substitution can be used to replace a Tcl command with the result it returns. Consider the following example:

Example 3.1:

```tcl
puts "I am [expr 10 * 2] years old, and my I.Q. is [expr 100 - 25]"
```

Output: I am 20 years old, and my I.Q. is 75

As this example shows, square brackets are used to achieve command substitution: The text between the square brackets is evaluated as a Tcl script, and its result is then substituted in its place. In this case, command substitution is used to place the results of two mathematical expressions into a string. Command substitution is often used in conjunction with variable substitution, as shown in Example 3.2:

Example 3.2:

```tcl
set my_height 6.0
puts "If I was 2 inches taller, I would be [expr $my_height + (2.0 / 12.0)] feet tall"
```

Output: If I was 2 inches taller, I would be 6.16667 feet tall

In this example, the value of the variable "my_height" is substituted inside the angle brackets before the command is evaluated. This is a good illustration of Tcl's one-pass recursive parsing mechanism. When evaluating a statement, the Tcl interpreter, makes one pass over it, and in doing so makes all the necessary substitutions. Once this is done, the interpreter then evaluates the resulting expression. If, during its pass over the expression, the interpreter encounters square brackets (indicating that command substitution is to be performed), it recursively parses the script inside the square brackets in the same manner. For more information on one-pass parsing, refer to Matt Peters' document on the topic.

Try it yourself:
Feel free to type in any of the above examples in this Tcl interpreter:

**Control Flow**

In all but the simplest scripts, some mechanism is needed to control the flow of execution. Tcl offers decision-making constructs (if-else and switch statements) as well as looping constructs (while, for, and foreach statements), both of which can alter the flow of execution in response to some condition. The following examples serve to illustrate these constructs.

---

**Example 4.1:**

```tcl
set my_planet "earth"
if {$my_planet == "earth"} {
    puts "I feel right at home."
} elseif {$my_planet == "venus"} {
    puts "This is not my home."
} else {
    puts "I am neither from Earth, nor from Venus."
}
set temp 95
if {$temp < 80} {
    puts "It's a little chilly."
} else {
    puts "Warm enough for me."
}
```

**Output:**

I feel right at home.

Warm enough for me.

Example 4.1 makes two uses of the if-statement. It sets the value of the variable "my_planet" to "earth", and then uses an if-statement to choose which statement to print. The general syntax of the if-statement is as follows:

if test1 body1 ?elseif test2 body2 elseif ...? ?else bodyn?

If the test1 expression evaluates to a true value, then body1 is executed. If not, then if there are any elseif
clauses present, their test expressions are evaluated and, if true, their bodies are executed. If any one of the tests is made successfully, after its corresponding body is executed, the `if`-statement terminates, and does not make any further comparisons. If there is an `else` clause present, its body is executed if no other test succeeds.

Another decision-making construct is the `switch`-statement. It is a simplification of the `if`-statement that is useful when one needs to take one of several actions depending on the value of a variable whose possible values are known. This is illustrated in Example 4.2, which uses a switch statement to print a sentence, depending on the value of a variable "num_legs".

Example 4.2:

```tcl
set num_legs 4
switch $num_legs {
    2 {puts "It could be a human."}
    4 {puts "It could be a cow."}
    6 {puts "It could be an ant."}
    8 {puts "It could be a spider."}
    default {puts "It could be anything."}
}
```

Output:

It could be a cow.

The `switch`-statement has two general forms (both of which are described in detail in the manual page), but the form used here is as follows:

```tcl
switch ?options? string {pattern body ?pattern body ...?}
```

Basically, the `string` argument is compared to each of the `patterns` and if a comparison succeeds, the corresponding `body` is executed, after which the switch statement returns. The pattern "default", if present, is always matched, and thus its body always executed if none of the earlier comparisons succeed.

It is often useful to execute parts of a program repeatedly, until some condition is met. In order to facilitate this, Tcl offers three looping constructs: the `while`, `for`, and `foreach` statements, each of which is shown in the examples below.

Example 4.3:

```tcl
for {set i 0} {$i < 10} {incr i 1} {
    puts "In the for loop, and i == $i"
}
```

Output:

In the for loop, and i == 0
In the for loop, and i == 1
In the for loop, and i == 2
In the for loop, and i == 3
In the for loop, and i == 4
In the for loop, and i == 5
In the for loop, and i == 6
In the for loop, and i == 7
In the for loop, and i == 8
In the for loop, and i == 9

The general syntax for the **for**-loop is as follows:

```
for init test reinit body
```

The `init` argument is a Tcl script that initializes a looping variable. In the `for`-loop used in Example 4.3, the looping variable was called "i", and the `init` argument simply set it to 0. The `test` argument is a Tcl script which will be evaluated to decide whether or not to enter the body of the `for`-loop. Each time this script evaluates to a true value, the body of the loop is executed. The first time this script evaluates to false, the loop terminates. The `reinit` argument specifies a script that will be called after each time the body is executed. In Example 4.3, the `reinit` script increments the value of the looping variable, "i". Thus, `for`-loop in this example executes its body 10 times, before its `test` script evaluates to false, causing the loop to terminate.

**Example 4.4:**

```
set i 0
while {($i < 10)} {
    puts "In the while loop, and i == $i"
    incr i 1
}
```

**Output:**

In the while loop, and i == 0
In the while loop, and i == 1
In the while loop, and i == 2
In the while loop, and i == 3
In the while loop, and i == 4
In the while loop, and i == 5
In the while loop, and i == 6
In the while loop, and i == 7
In the while loop, and i == 8
In the while loop, and i == 9
Example 4.4 illustrates the use of a `while`-loop, the general syntax of which follows the form:

```
while test body
```

The basic concept behind the `while`-loop is that while the script specified by the `test` argument evaluates to a true value, the script specified by the `body` argument is executed. The `while` loop in Example 4.4 accomplishes the same effect as the `for`-loop in Example 4.3. A looping variable, "i", is again initialized to 0 and incremented each time the loop is executed. The loop terminates when the value of "i" reaches 10.

Note, that in the case of the `while`-loop, the initialization and re-initialization of the looping variable are not part of the `while`-statement itself. Therefore, the initialization of the variable is done before the `while`-loop, and the reinitialization is incorporated into its body. If these statements were left out, the code would probably still run, but with unexpected results.

---

**Example 4.5:**

```
foreach vowel {a e i o u} {
    puts "$vowel is a vowel"
}
```

**Output:**

```
a is a vowel
e is a vowel
i is a vowel
o is a vowel
u is a vowel
```

The `foreach`-loop, illustrated in Example 4.5, operates in a slightly different manner to the other types of Tcl loops described in this section. Whereas `for`-loops and `while`-loops execute while a particular condition is true, the `foreach`-loop executes once for each element of a fixed list. The general syntax for the `foreach`-loop is:

```
foreach varName list body
```

The variable specified by `varName` takes on each of the values in the `list` in turn, and the `body` script is executed each time. In Example 4.5, the variable "vowel" takes on each of the values in the list `"{a e i o u}"` (Tcl list structure will be discussed in more detail in a later section), and for each value, the body of the loop is executed, resulting in one printed statement each time.

**Try it yourself:**

Feel free to type in any of the above examples in this Tcl interpreter:
Procedures

Procedures in Tcl serve much the same purpose as functions in C. They may take arguments, and may return values. The basic syntax for defining a procedure is:

```
proc name argList body
```

Once a procedure is created, it is considered to be a command, just like any other built-in Tcl command. As such, it may be called using its name, followed by a value for each of its arguments. The return value from a procedure is equivalent to the result of a built-in Tcl command. Thus, command substitution can be used to substitute the return value of a procedure into another expression.

By default, the return value from a procedure is the result of the last command in its body. However, to return another value, the `return` command may be used. If an argument is given to the `return` command, then the value of this argument becomes the result of the procedure. The `return` command may be used anywhere in the body of the procedure, causing the procedure to exit immediately.

Example 5.1:

```
proc sum_proc {a b} {
    return [expr $a + $b]
}

proc magnitude {num} {
    if {$num > 0} {
        return $num
    }
    set num [expr $num * (-1)]
    return $num
}

set num1 12
set num2 14
set sum [sum_proc $num1 $num2]

puts "The sum is $sum"
puts "The magnitude of 3 is [magnitude 3]"
puts "The magnitude of -2 is [magnitude -2]"
```

Output:

```
The sum is 26
```
The magnitude of 3 is 3
The magnitude of -2 is 2

This example first creates two procedures, "sum_proc" and "magnitude". "sum_proc" takes two arguments, and simply returns the value of their sum. "magnitude" returns the absolute value of a number. After the procedure definitions, three global variables are created. The last of these, "sum" is assigned the return value of the procedure "sum_proc", called with the values of the variables "num1" and "num2" as arguments. The "magnitude" procedure is then called twice, first with "3" as an argument, then with "-2".

The "sum_proc" procedure uses the `expr` command to calculate the sum of its arguments. The result of the `expr` command is substituted into the `return` statement, making it the return value for the procedure.

The "magnitude" procedure makes use of an `if`-statement to take different actions, depending on the sign of its argument. If the number is positive, its value is returned, and the procedure exits immediately, skipping all the rest of its code. Otherwise, the number is multiplied by -1 to obtain its magnitude, and this value is returned. The same effect could be achieved by moving the statement that multiplies the value by -1 into an `else`-clause, but the purpose of this example was to illustrate the use of the `return` statement at several locations within a procedure.

Inside the body of a procedure, new variables may be created with the `set` command as normal. However, these variables will be local to the procedure, and will no longer be accessible once the procedure returns. If access to global variables is needed inside a procedure, these may be accessed by means of the `global` keyword, as described in Example 5.2.

---

**Example 5.2:**

```tcl
proc dumb_proc {} {
    set myvar 4
    puts "The value of the local variable is $myvar"
    global myglobalvar
    puts "The value of the global variable is $myglobalvar"
}
set myglobalvar 79
dumb_proc
```

Output:

The value of the local variable is 4
The value of the global variable is 79

The procedure "dumb_proc" achieves no special purpose, and is simply designed to illustrate the use of the `global` keyword to access global variables. It takes no arguments, and as such its argument list is empty. Note that even though the procedure takes no arguments, the empty list structure must still be included.

The procedure first creates a local variable, "myvar", sets its value to "4", and then displays it. Then it uses the `global` keyword to gain access to a global variable named "myglobalvar". The value of this global variable is then printed.
After the procedure definition, a global variable "myglobalvar" is created, and assigned a value of "79". The procedure "dumb_proc" is then called, resulting in the output shown above.

**Try it yourself:**

Feel free to type in any of the above examples in this Tcl interpreter:

---

**Lists**

Lists in Tcl provide a simple means by which to group collections of items, and deal with the collection as a single entity. When needed, the single items in the group can be accessed individually. Lists are represented in Tcl as strings with a specified format. As such, they can be used in any place where strings are normally allowed. The elements of a list are also strings, and therefore any form of data that can be represented by a string can be included in a list (allowing lists to be nested within one another).

The following examples will illustrate many important list commands:

---

**Example 6.1:**

```tcl
set simple_list "John Joe Mary Susan"
puts [lindex $simple_list 0]
puts [lindex $simple_list 2]
```

**Output:**

John
Mary

Example 6.1 creates a simple list of four elements, each of which consists of one word. The `lindex` command is then used to extract two of the elements in the list: the 0th element and the 2nd element. Note that list indexing is zero-based. It is also important to see that the `lindex` command, along with most other list commands, takes an actual list as its first argument, not the name of a variable containing a list. Thus the value of the variable "simple_list" is substituted into the `lindex` command.

---

**Example 6.2:**
set simple_list2 "Mike Sam Heather Jennifer"
set compound_list [list $simple_list $simple_list2]
puts $compound_list
puts [llength $compound_list]

Output:

{John Joe Mary Susan} {Mike Sam Heather Jennifer}
2

Example 6.2 is a continuation of Example 6.1, and assumes the variable "simple_list" (created in Example 6.1) still exists. In this example, a new variable called "simple_list2" is created, and assigned the value of another simple four-element list. A compound list is then formed by using the list command, which simply forms a list from its arguments. The list command ensures that proper list structure is observed, even when its arguments themselves are lists, or other complex structures. Displaying the value of "compound_list" shows that it is a list of two elements, each of which is itself a list of four elements. The llength command is used to obtain the length of the list, "compound_list", which is 2 in this case.

This example highlights two ways in which to create lists in Tcl: by explicitly listing the elements within quotes, and by using the list command. Explicitly listing the elements works well when each of the elements is a single word. However, if the elements contain whitespaces, then maintaining proper list structure becomes a little more tricky. For these cases, the list command proves very useful.

Example 6.3:

set mylist "Mercury Venus Mars"
puts $mylist
set mylist [linsert $mylist 2 Earth]
puts $mylist
lappend mylist Jupiter
puts $mylist

Output:

Mercury Venus Mars
Mercury Venus Earth Mars
Mercury Venus Earth Mars Jupiter

In example 6.3, a simple list of 3 items is created, and assigned to the variable "mylist". The linsert command is then used to insert a new item into this list. Note that, as with the llength command, the linsert command takes an actual list as its first argument, not the name of a variable containing a list. The linsert command returns a list that is the same as the list it was passed, except that the specified item is inserted in the appropriate position. This return value needs to be assigned back to the variable "mylist" in order for the list stored in that variable to change.

One list command that does not behave in this way is the lappend command. It takes the name of a variable as its first argument, and appends its subsequent arguments onto the list stored in that variable. Thus the value of the variable is modified directly. Understanding the difference between the way the lappend command works, and the way that commands such as linsert work is fundamental to using lists correctly.
The list commands presented here are only a small subset of those available. Refer to the manual pages, or one of the other Tcl/Tk references for a complete description of all list commands.

**Try it yourself:**

Feel free to type in any of the above examples in this Tcl interpreter:

---

**Arrays**

Another way of grouping data in Tcl is to use arrays. Arrays are simply collections of items in which each item is given a unique index by which it may be accessed. As with all other Tcl variables, arrays need not be declared before they are used, and, unlike arrays in C, their size need not be specified either.

An individual element of an array may be referred to by using the array name, followed immediately by the index of the element, enclosed in parentheses. Array elements are treated much like any other Tcl variables. They are created by means of the `set` command, and their values can be substituted using the dollar sign ("$"), as is the case with other variables.

---

**Example 7.1:**

```tcl
set myarray(0) "Zero"
set myarray(1) "One"
set myarray(2) "Two"

for {set i 0} {$i < 3} {incr i 1} {
    puts $myarray($i)
}
```

**Output:**

Zero
One
Two

---

In Example 7.1, an array called "myarray" is created and initialized. Note that no special code is required to create the array because it is created by the `set` statement that assigns a value to its first element. The
for-loop simply prints out the value stored in each element of the array. Note the use of variable substitution in the array index and the array name.

Example 7.2:

```tcl
set person_info(name) "Fred Smith"
set person_info(age) "25"
set person_info(occupation) "Plumber"

foreach thing {name age occupation} {
    puts "$thing == $person_info($thing)"
}
```

Output:

name == Fred Smith
age == 25
occupation == Plumber

Example 7.2 illustrates one of the unique features of Tcl arrays: array indices need not be integers. In fact, array indices can take on any string value. In this case, the array "person_info" is created with three elements. The indices for the elements are "name", "age", and "occupation". The foreach-loop simply displays each of the elements in the array.

Using arrays with named indices is one of the ways to abstract objects in Tcl. In Example 7.2, the "person_info" array can be thought of as an "object" describing a person. Each of the elements in the array then describes a fundamental attribute of the object.

One problem with using named indices with arrays is that one needs to remember the names of all the elements in order to traverse the array. In Example 7.2, for example, the names of all the elements had to be listed explicitly. In a case such as this one, in which there are only three elements, this does not present much of a problem. However, if the array contained many more elements, explicitly listing them each time the array had to be traversed would lead to very messy code. The array Tcl command, illustrated in Example 7.3, provides a means to get around this problem.

Example 7.3:

```tcl
set person_info(name) "Fred Smith"
set person_info(age) "25"
set person_info(occupation) "Plumber"

foreach thing [array names person_info] {
    puts "$thing == $person_info($thing)"
}
```

Output:

occupation == Plumber
Example 7.3 produces essentially the same result as Example 7.2, but it makes use of the *array* command to obtain the names of the elements in the array, instead of listing them explicitly. The array elements are displayed in a different order than they were in Example 7.2, simply because the *array* command returns the names of the elements in a different order than the one in which they were explicitly listed previously.

The general purpose of the *array* command is to retrieve various pieces of information about an array (such as its size or the names of its elements), and perform other operations (such as searching) on it. The general syntax of the *array* command is:

```
array option arrayName ?arg arg ...?
```

The *option* argument specifies which array operation to perform. In the case of Example 7.3, the *option* argument is given the value "names", which causes the *array* command to return a list of the names of the elements in the array given by the *arrayName* argument. For a complete list of the allowed values of the *option* argument, and well as a description of the corresponding operations, refer to the manual page for the *array* command.

**Try it yourself:**

Feel free to type in any of the above examples in this Tcl interpreter:

---

**Strings**

Since strings are the most prevalent data type in Tcl, it makes sense that Tcl provides a rich set of functions for manipulating them. Most string operations are done by means of the *string* command, which takes the following general form:

```
string option arg ?arg arg ...?
```

The *string* command actually performs several different functions, and the *option* argument is used to differentiate between them. Example 8.1 creates a string and then uses the *string* command to manipulate it, and obtain information about it.

**Example 8.1:**
set str "This is a string"
puts "The string is: $str"
puts "The length of the string is: [string length $str]"
puts "The character at index 3 is: [string index $str 3]"
puts "The characters from index 4 through 8 are: [string range $str 4 8]"
puts "The index of the first occurrence of letter "i" is: [string first i $str]"

Output:

The string is: This is a string
The length of the string is: 16
The character at index 3 is: s
The characters from index 4 through 8 are: is a
The index of the first occurrence of letter "i" is: 2

In Example 8.1, a variable called "str" is created, and initialized to the value, "This is a string". The string command is then used with various options to obtain various pieces of information about the string. Refer to the manual page for the string Command for a complete listing and explanation of the various options. Also, there are several other string-related commands worth exploring, such as format, regexp, regsub, and scan.

Try it yourself:

Feel free to type in any of the above examples in this Tcl interpreter:

Input/Output

Most input and output operations in Tcl are done by means of the puts and gets commands. Most of the examples in this document have made use of the puts command to display output on the console. In a similar manner, the gets command can be used to wait for input from the console, and optionally store it in a variable. Its general syntax has the following form:

gets channelId ?varName?

The first argument to gets is the name of an open channel from which to read data, and can be thought of as a file descriptor in the C sense. If the varName argument is specified, gets stores the data it reads in that variable, and returns the number of bytes read. If varName is not specified, then gets simply returns the data...
Example 9.1:

puts -nonewline "Enter your name: "
set bytesread [gets stdin name]
puts "Your name is $name, and it is $bytesread bytes long"

Output: (note that user input is shown in italics)

Enter your name: Shyam
Your name is Shyam, and it is 5 bytes long

Example 9.1 makes use of both the puts and gets commands. The puts command is used with the -nonewline flag to suppress the trailing newline that it normally appends to its output. A variable, "bytesread", is then assigned the result of a gets command that reads from the channel "stdin" (the standard input), and stores the data it reads in the variable, "name". Thus "bytesread" ends up storing the number of bytes of user input read from the console.

In Example 9.1, gets was used to read from the channel "stdin" (created automatically when the Tcl interpreter is started) which corresponds to the standard input. The puts command can also be used with a channel identifier to write to a specific channel. However, if no channel identifier is passed to puts, it writes to the standard output (this is the way puts has been used throughout this document). In addition to the standard input and output, channels can also be created to read from other types of files. As illustrated by Example 9.2, the open command can be used to open a channel to a file, and obtain an appropriate identifier for the channel. This identifier can then be passed to gets to read from the file, or puts to write to the file.

Example 9.2: (Note: This example will not work in the Tcl interpreter included in this page. Feel free to cut and paste the text of this example into a Tcl interpreter on your host system to see it work).

set f [open "/tmp/myfile" "w"]
puts $f "We live in Texas. It's already 110 degrees out here."
puts $f "456"
close $f

Output: (none)

This example uses the open command to open a channel to a file called "/tmp/myfile". The syntax of the open command can take on three forms, one of which is:

open name ?access?

The access argument specifies what type of access (for example, read-only access or read-write access) to the file given by name is desired. See the manual page for the open command for a complete description of the access modes. In this case, write-only access is desired, so the value "w" is given for the access
argument.

The open command returns a channel identifier that can be used with `gets` and `puts` to read and write from the file. In Example 9.2, this identifier is stored in the variable, "f". The `puts` command is then used to write two strings to the file, and then the `close` command is used to close the file.

Example 9.3 reads the file created in Example 9.2, and displays its contents.

---

**Example 9.3:** (Note: This example will not work in the Tcl interpreter included in this page. Feel free to cut and paste the text of this example into a Tcl interpreter on your host system to see it work).

```tcl
set f [open "/tmp/myfile" "r"]
set line1 [gets $f]
set len_line2 [gets $f line2]
oclose $f
puts "line 1: $line1"
puts "line 2: $line2"
puts "Length of line 2: $len_line2"
```

Output:

```
line 1: We live in Texas. It's already 110 degrees out here.
line 2: 456
Length of line 2: 3
```

The file, "/tmp/myfile", is opened in read-only mode with the `open` command. The `gets` command is then used with the channel identifier returned by `open` to read from the file. The first call to `gets` does not give it the name of a variable in which to store the data it reads, so this data is returned instead. Command substitution is used to store it in the variable, "line1". The second call to `gets` tells it to store the data it reads in the variable, "line2". Therefore, `gets` would return the number of bytes it read, which, by means of command substitution, is stored in the variable "len_line2". Since all the data has been read, the file is then closed.

In this case, it was known that the file contained only two lines of data. If the length of the file was not known, the `eof` command could be used with a while loop to read until the end of the file was reached.

**Try it yourself:**

Feel free to type in any of the above examples in this Tcl interpreter:
Other Miscellaneous Tcl Commands

eval

As described earlier, Tcl uses a one-pass parsing mechanism when evaluating scripts. It is sometimes useful, however, to have the interpreter make more than one pass over a script before evaluating it. Being able to force the interpreter to parse a script more than once allows one to store Tcl scripts in variables, and have them be evaluated at a later time. This is shown in Example 10.1:

Example 10.1:

```tcl
set foo "set a 22"
eval $foo
puts $a
```

Output:

```
22
```

The variable "foo" is set to the value "set a 22", which is itself a Tcl script. Next, the value of the variable "foo" is substituted into the `eval` command. The `eval` command simply passes its arguments through the Tcl interpreter for another round of parsing. When the interpreter encounters the statement "eval $foo", the first round of parsing simply substitutes the value of the variable "foo" in the place of "$foo", resulting in the expression "eval set a 22". The `eval` command then sends its arguments, "set a 22", through the interpreter again, resulting in the variable "a" being created and assigned the value "22".

One might be tempted to think that the use of the `eval` command could be avoided and simply replaced with the statement,

```tcl
$foo
```

This does not work because, on encountering the statement "$foo", the interpreter simply replaces it with the value stored in the variable "foo", and then considers its parsing work done. So, "$foo" gets replaced by "set a 22", but the interpreter never parses "set a 22", which it needs to do to make sense of the components of the statement (it needs to realize that "set" corresponds to a built in command, and that it is being passed two arguments, "a" and "22") and evaluate it correctly.

catch
When an error occurs in a Tcl command, the entire script of which it is a part is halted, and an error message is displayed. However, instead of halting the whole Tcl script, it may be useful to simply display a friendly error message and continue execution of the Tcl script.

The `catch` command prevents Tcl's error handling mechanisms from executing (and thus halting execution) and simply returns a meaningful value when an error occurs. This allows the program to define its own behaviour in the case of an error.

---

**Example 10.2:** (Note: This example will not work in the Tcl interpreter included in this page. Feel free to cut and paste the text of this example into a Tcl interpreter on your host system to see it work).

```tcl
set retval [catch {set f [open "nosuchfile" "r"]}]
if {$retval == 1} {
    puts "An error occurred"
}
```

Output: (this output occurs if there is no file named "nosuchfile" in the current directory).

An error occurred

The `catch` command is given a Tcl script as an argument. It evaluates this script, and if an error occurs, it returns 1, otherwise it returns 0. In Example 10.2, the script passed to `catch` tries to open a file named "nosuchfile". Assuming that no file with this name exists in the current directory, the `open` command should return an error. Since it occurs within a `catch` statement, the normal Tcl error handling routines do not get invoked, and the `catch` command simply returns 1. This return value is assigned to the variable "retval", which is checked to determine whether or not to print the error message. The `catch` command can be used in many different ways, only one of which is shown here. Refer to the manual page for a more complete description.

**Try it yourself:**

Feel free to type in any of the above examples in this Tcl interpreter: