UNIT – 5

- TCL is string based scripting language and also a procedural language
- The language is commonly used for GUIs and testing
- In TCL by default everything is string
- TCL is shell application that reads TCL command from its standard input or from a file and gives desired results.
- TCL program should have .tcl extension

Example:

```
#!/usr/bin/tclsh
Puts "Hello World"
```

Script execution

```
$ chmod +x helloworld.tcl
$ ./helloworld.tcl
```

Output: Hello World

In TCL, "Puts" Command is used to print messages to the console .syntax of puts is below

puts?-nonewline? ?channelId? string

- **Nonewline**: This optional parameter suppresses the newline character by default command. It puts a newline to each string
- **Channelid**: This parameter used for standard input channel (stdin) and standard output channel (stdout).

Ex:

```
%puts "Hello World"
% Hello World
%puts stdout  "Hello World"
% Hello World
```

TCL Substitution type

There are three kinds of substitutions in TCL

1. Command substitution
2. Variable substitution
3. Backslash substitution

Command substitution

Square brackets are used for command substitution.

Example:

```
% puts [expr 1*3]
% 3
```

Here the command between the square brackets is evaluated first. The results is returned .”expr” used for performing the arithmetic calculation.

Variable substitution

TCL performs variable substitution with the help of **$** sign.
Example:-

```
#!/usr/bin/tclsh
set  a  10
puts  a
puts  $a
```

Here we create a variable called “a” and set value “10” to it.

- puts a : It will print string “a” but not the value of ‘a’ to the console
- puts $a : It will print the value of ‘a’ to the console

Let’s execute and verify it. You will get the output as below.

```
$ ./substitution.tcl
a
10
```

Backslash substitution
In Tcl, the backslash is used for escaping special characters as well as for spreading long commands across multiple lines. Any character immediately following the backslash will stand without substitution. In the example below, you can see special character ” “, remains after the backslash.

```
#!/usr/bin/tclsh
puts "This  is my  \"car\"
$ ./backslashsubstitution.tcl
This is my "car"
```

NOTE: To comment any string in TCL "#" is used. All characters after the "#" are ignored by tclsh shell command.

TCL Variable

A variable is an identifier which holds a value. In other words, a variable is a reference to a computer memory, where the value is stored.

Variables are created by "set command" and all variable names are case sensitive. It means hello, Hello, HELLO all are different in TCL. Look at some example for case sensitive variable.

```
set  name  Techoit
set  Name  Technoit_1
set  NAME  Technoit_2
```

Output:-

```
puts $name
Technoit

puts  $Name
Technoit_1

puts $NAME
Technoit_2
```

Creating TCL Variables

To create variables in TCL, you need to use **"set"** command

```
Set a 10
```

To obtain the value of variable have to use "$" symbol like

```
put $a
10
```

So we get the value of variable 'a' as 10.

TCL Command Info exist

The "set" command is used to create and read variables as shown above. The unset command is used to destroy a variable. The "info exists" command returns 1 if varName

exists as a variable (or an array element) in the current context, otherwise returns 0. (see example below).

There are various “info” command in TCL like “info exists”, “info functions”, “info global”, and so on. Here we will see an example of “info exists”.

Ex:

```
set a 20
puts $a
20

puts [info exists a]
1

unset a
puts [info exists a]
0
```

Different braces and their behavior

{} -> Curly braces

Curly braces in TCL group words together to become arguments. Curly braces are used to define a block that’s deferred – in other words, it may be run AFTER the rest of the command on the current line. Characters within braces are passed to a command exactly as written.

Some points to remember

1. Variable substitution is not allowed inside {} braces
2. It used to create list data type

Example:

```
set x 10
puts {$x}
$x
set number {1 2 3 4 5} -> Here number is a list data type
puts $number
```

OUTPUT: 1 2 3 4 5

[ ] square braces

Square brackets are used to create nested command. Simply put, output of one command passed as argument to another command. Square brackets are used to define a block

that's run BEFORE the rest of the command on the current line, and the result is substituted into the line.

Ex: –

```
set x 10
puts "y : [set y [set x 10]]"
puts "x : $x"
```

OUTPUT→

```
y : 10
x : 10
```

( ) round braces

This command is used to create array data type and also indicate operator precedence.

```
set a(1) 10
set a(2) 20
```

Here "a" is an array with value 10 and 20. See below commands to print keys, key value pairs and values of array.

```
puts [array get a] ->  To print key value pairs we use this command
1 10 2 20
puts [array names a]  -> To print only keys
1 2
puts $a(1)   -> To print first value of array
10
puts $a(2)  -> To print second value of array
20
```

For print "Nth" value of array a, we will use Puts $a(N)

TCL Command-line arguments

Items of data passed to a script from the command line are known as arguments. The number of command line arguments to a Tcl script is passed as the global variable **argc** . The name of a Tcl script is passed to the script as the global variable **argv0** , and the rest of the command line arguments are passed as a list in **argv**.

TCL has 3 pre-defined variables such as

```
$argc   -> indicates the number of arguments passed to the script
$argv   -> indicates list of arguments
$argv0  -> indicates the name of script
```

Ex:

```
arg-script.tcl
#!/usr/bin/tclsh<
puts   "number of arguments passed to the scripts : $argc"
puts  "list of arguments are passed to the script: $argv"
puts  "the name of scripts is: $argv0"
$ ./arg-script.tcl  10 20 30
```

Output

- Number of arguments passed to the scripts: 3
- List of arguments are passed to the script: 10 20 30
- The name of the script is : arg-script.tcl

TCL Expression and Operator

Expression is constructed from operands and operators. It's evaluated with "expr" command. Operators are evaluated based on precedence and associativity. TCL language has built-in operators as below

Operator Category	Symbol	Precedence/Associativity
Arithmetic Operator	+ - * / %	Left to Right
Relational Operator	== != < > <= >=	Left to Right
Logical Operator	&& || !	Left to Right
Bitwise Operator	& | ^ ~	Left to Right
Ternary Operator	?:	Right to Left
Shift Operator	<< >>	Left to Right
String Comparison Operator	eq ne	Left to Right
Exponentiation Operator	**	Left to Right
List Operator	In ni	Left to Right

Arithmetic Operator

A TCL expression consists of a combination of operands, operators, and parentheses. Let see example of Arithmetic operators in TCL

+ Add two or more operands

```
set a 10
set b 20
puts [expr $a + $b]
30
```

- Subtracts two or more operands

```
set a 20
set b 10
puts [expr $a - $b]
10
```

***Multiply two or more operands**

```
set a 20
set b 10
puts [expr $a * $b]
200
```

/ Divide numerator by denumerator

```
set a 20
set b 10
puts [expr $a / $b]
2
```

% Modulus operator divides numerator by de-numerator but returns reminder

```
set a 20
set b 10
puts [expr $a % $b]
0
```

Relational Operator

Checks if the value of left operand is greater than the value of the right operand. If yes, then condition becomes true and return 1 else return 0.

```
set a 20
set b 10
puts [expr $a > $b]
1
```

Check if the value of left operand is less than the value of the right operand. If yes, then condition becomes true and return 1 else return 0

```
set a 10
set b 20
puts [expr $a < $b]
1
```

>= Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true and return 1 else return 0

```
set a 20
set b 10
puts [expr $a >= $b]
1
```

<= Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true and return 1 else return 0

```
set a 20
set b 10
puts [expr $a <= $b]
0
```

!= Checks if the values of two operands are equal or not, if values are not equal then condition becomes true and return 1 else return 0

```
set a 20
set b 10
puts [expr $a != $b]
1
```

== Checks if the values of two operands are equal or not, if yes then condition becomes true and return 1 else return 0

```
set a 20
set b 10
puts [expr $a == $b]
0
```

Logical Operator

&& If both the operands are non-zero, then condition becomes true and return 1 else return 0

```
set a 20
set b 10
puts [expr $a && $b]
1
```

|| If any of the two operands is non-zero, then condition becomes true and return 1 else return 0

```
set a 0
set b 10
puts [expr $a || $b]
1
```

! Used to reverse the result of any expression. Here in the output, you can see the value of 'a' has now become 1 from 0. While the value of 'b' has become 0 from 1.

```
set a 0
set b 1
puts [expr !$a]
1
puts [expr !$b]
0
```

Bitwise Operator

& (bitwise and) perform bit by bit operation and follow the below table for operation.

A	B	A & B
0	0	0
0	1	0
1	1	1
1	0	0

Ex:

```
set A 10
set B 20
```

Follow the sequence to convert decimal to binary number
128 64 32 16 8 4 2 1
10 binary equivalents will be
128 64 32 16 8 4 2 1 à 0 0 0 0 1 0 1 0
20 binary equivalents will be
128 64 32 16 8 4 2 1 à 0 0 0 1 0 1 0 0

So now as per above tabular rules
A & B will be 0 0 0 0 0 0 0 0

| (bitwise or) perform bit by bit operation and follow the below table

A	B	A | B
0	0	0
0	1	1
1	1	1
1	0	1

Ex:

set A 10
set B 20

Follow the sequence to convert decimal to binary number
128 64 32 16 8 4 2 1
10 binary equivalents will be
128 64 32 16 8 4 2 1 à 0 0 0 0 1 0 1 0
20 binary equivalents will be
128 64 32 16 8 4 2 1 à 0 0 0 1 0 1 0 0

So now as per above tabular rules

A | B will be 0 0 0 1 1 1 1 0
^ (bitwise exclusive or) perform bit by bit operation and follow the below table

A	B	A ^ B
0	0	0
0	1	1
1	1	0
1	0	1

Ex:

set A 10
set B 20

Follow the sequence to convert decimal to binary number
128 64 32 16 8 4 2 1
10 binary equivalents will be

128 64 32 16 8 4 2 1 à 0 0 0 0 1 0 1 0

20 binary equivalents will be
128 64 32 16 8 4 2 1 à 0 0 0 1 0 1 0 0

So now as per above tabular rules
A ^ B will be 0 0 0 1 1 1 1 0 à 30

~ (bitwise negation) operator changes each 1 to 0 and 0 to 1, follow the table as reference

A	~A
0	1
1	0

```
set A 7
puts [expr ~$A]
-8
```

Ternary Operator (?:)

Syntax is

condition-expression? expression_1: expression_2

If condition-exp is true, exp1 is evaluated and the result is returned. If the cond-exp is false, exp2 is evaluated and its result is returned. In our example, exp1 is true as the value of A is greater than 6.

```
set A 7
set result [expr $A > 6 ? true : false]
puts $result
true
```

Shift Operator

Shift operator is denoted by either << left shift operator, or by the >> right shift operator. For << left shift operator, the left operands value is moved left by the number of bits specified by the right operand.

```
set A 7
set result [expr $A << 2]
puts $result
```

For the >> right shift operator, the left operands value is moved right by the number of bits specified by the right operand.

```
set A 7
set result [expr $A >> 2]
puts $result
```

String Comparison Operator

String comparison operator compares the value of both operands. If the value of operand are same, then it will return 1 else return 0. In example value for both A and B is 7, therefore result return 1.

```
set A 7
set B 7
set result [expr $A eq $B]
puts $result
1
```

Ne (if value of both operand are different then it will return 1 else return 0)

```
set A 7
set B 8
set result [expr $A ne $B]
puts $result
1
```

Exponentiation operator

Pow () and ** both are same. It always returns floating value.

** indicates the power to the desired operand.

```
set A 7
set result [expr $A ** 2]
puts $result
49
```

List Operator

If the required value found in the defined list, it returns 1 else return 0. In example value 1 exists in variable 'a' hence it will return 1.

```
set a {1 2 3}
if {1 in $a} {
puts "ok"
} else {
puts "fail"
```

```
}
Output: ok
```

ni, if the required value found in the defined list then it will return 0 else return 1.

```
set a {1 2 3}
if {1 ni $a} {
puts "ok"
} else {
puts "fail"
}
Output: fail
```

TCL flow control and decision making

There are various flow control and decision making command which are used to alter the flow of a program. Program executions always start from the top of source file to the bottom.

If statement consists of Boolean expression followed by one or more statements.

If ... statement
Syntax

```
if expr ?then? body
```

if expr is evaluated to true, then the body of command is executed.

```
set age 10

if {$age < 20} {
puts "Age is less than 20"
}

Output: Age is less than 20
```

If ... else statement
Syntax

If expression ? then body_1 else body_2

If expression is evaluated to true, then it will return body_1 else it will return body_2

```
set age 10
if {$age < 20} {
puts "Age is less than 20"
} else {
Puts "Age is greater than 20"
}
```

output: Age is less than 20

Nested if..else statement

It means one if or else..if statement can be put inside another if or else..if statements.

Syntax

```
If  {expression_1} {
Body_1
If {expression_2} {
Body_2
}
}
```

Ex:

```
set a 10
set b 20

if {$a == 10} {
# if expression_1 is true then it will go to expression_2
if {$b == 20} {
#if expression_2 is true then it will print the below string
puts "value of a is 10 and b is 20"
}
}
```

o/p: value of a is 10 and b is 20

Switch statement

The switch statement enables a variable to be tested for equality against a list of values. It evaluates the list of values and returns the result of that evaluation. If no values matches then default values will be returned.

Example:

```
#!/usr/bin/tclsh
# switch_cmd.tcl
set domain x
switch $domain {
   x { puts "x" }
   y { puts "y" }
   z { puts "z" }
   default { puts "unknown" }
}
```

Nested switch
Nested switch statement means switch statement inside a switch statement.

Syntax :

```
switch <switchingstring1> {
  <matchstring1> {
    body1
    switch <switchingstring2> {
      <matchstring2> {
        body2
       }
       ...
   switch <switchingstringN> {
        <matchStringN> {
         bodyN
        }
      }
  }
```

Example: In the following example, value of a is 100, and with the same code we switch statement for another value of b is 200. The out will show value for both a and b.

```
#!/usr/bin/tclsh
set a 100
set b 200
switch $a {
  100 {
    puts "The value of a is $a"
    switch $b {
      200 {
        puts "The value of b is $b"
      }
    }
  }
}
```

Output:

```
The value of a is 100
The value of b is 200
```

TCL Loop statement

Loop statement allows executing a statement or group of statement multiple times. Tcl provides the following types of looping statement.

While command

When a given condition is true then a statement or group of statement repeats which are in the loop body.

Syntax:

```
While  {condition} {
   Statements
}
```

Ex :

```
#!/usr/bin/tclsh
Set a 10
While {$a < 12} {
   Puts "a is $a"
   incr a
}
```

Output

```
a is 10
a is 11
```

In the above example, "incr" built-in command is used. It means the value of 'a' will be increased by 1 till the maximum value (<12).

For command

It executes a sequence of statements multiple times based upon a counter value. It is automatically increased or decreased during each repetition of the loop.

Syntax

```
For {start} {test} {next} {
Body
}
```

Example: In below example value of 'i' is set to 0 and incremented till value <4.

```
#!/usr/bin/tclsh
for {set i 0} {$i < 4} {incr i} {
put $i
}
```

Output

```
0
1
2
3
```

Tcl Data Structures are:

Arrays.
Namespaces.
Procedures.
Interpreters.
Dictionaries.

Arrays:

An array is a systematic arrangement of a group of elements using indices. The syntax for the conventional array is shown below.

set ArrayName(Index) value

An example for creating simple array is shown below.

```
#!/usr/bin/tclsh
set languages(0) Tcl
set languages(1) “C Language”
puts  $languages(0)
puts $languages(1)
```

When the above code is executed, it produces the following result

```
Tcl
C Language
```

Size of Array

The syntax for calculating size array is shown below.

```
[array size variablename]
```

An example for printing the size is shown below.

```
#!/usr/bin/tclsh
set languages(0) Tcl
set languages(1) “C Language”
puts [array size languages]
```

When the above code is executed, it produces the following result

```
2
```

Array Iteration

Though, array indices can be non-continuous like values specified for index 1 then index 10 and so on. But, in case they are continuous, we can use array iteration to access elements of the array. Simple array iteration for printing elements of the array is shown below.

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

```
set languages(0) Tcl
set languages(1) “C Language”
for {set index 0} {$index<[array size languages]} {incr index} {
      puts “languages($index) : $languages($index)”
}
```

When the above code is executed, it produces the following result

```
languages(0) : Tcl
languages(1) : C Language
```

Namespaces:

Namespace is a container for set of identifiers that is used to group variables and procedures. Namespaces are available from Tcl version 8.0. Before the introduction of the namespaces, there was single global scope. Now with namespaces, we have additional partitions of global scope.

Creating Namespace

Namespaces are created using the **namespace** command. A simple example for creating namespace is shown below

```
#!/usr/bin/tclsh
namespace eval MyMath {
   #Create a variable inside the namespace
   variable myResult
}
#Create procedures inside the namespace
proc MyMath::Add {a b } {
set MyMath::myResult [expr $a+$b]
}
MyMath::Add 10 23
puts $::MyMath::myResult
```

When the above code is executed, it produces the following result

```
33
```

In the above program, you can see there is a namespace with a variable **myResult** and a procedure **Add.** This makes it possible to create variables and procedures with the same names under different namespaces.

Nested Namespaces

Tcl allows nesting of namespaces. A simple example for nesting namespaces is given below

```
#!/usr/bin/tclsh
namespace eval MyMath {
   #Create a variable inside the namespace
   variable myResult
}
namespace eval extendedMath{
# Create a variable inside the namespace
namespace eval MyMath {
#Create a variable inside the namespace
   variable myResult
}
set::MyMath::myResult “test1”
puts $::MyMath::myResult
set::extendedMath::MyMath::myResult “test2”
puts $::extendedMath::MyMath::myResult
```

When the above code is executed, it produces the following result

```
test1
test2
```

Importing and Exporting Namespace

You can see in the previous namespace examples, we use a lot of scope resolution operator and it's more complex to use. We can avoid this by importing and exporting namespaces. An example is given below

```
#!/usr/bin/tclsh
namespace eval MyMath{
#Create a variable inside the namespace
variable myResult
namespace export Add
}
#Create procedures inside the namespace
proc MyMath::Add {a b} {
return [expr $a + $b]
}
namespace import MyMath::*
puts [Add 10 30]
```

When the above code is executed, it produces the following result

Forget Namespace

You can remove an imported namespace by using **forget** subcommand. A simple example is shown below

```
#!/usr/bin/tclsh
namespace eval MyMath{
#Create a variable inside the namespace
variable myResult
namespace export Add
}
#Create procedures inside the namespace
proc MyMath::Add {a b} {
return [expr $a + $b]
}
namespace import MyMath::*

puts [Add 10 30]

namespace forget MyMath::*
```

When the above code is executed, it produces the following result −

```
40
```

Trapping errors:

Error handling in Tcl is provided with the help of **error** and **catch** commands. The syntax for each of these commands is shown below.

Error syntax:

```
error message info code
```

In the above error command syntax, message is the error message, info is set in the global variable errorInfo and code is set in the global variable errorCode.

Catch Syntax

```
catch script resultVarName
```

In the above catch command syntax, script is the code to be executed, resultVarName is variable that holds the error or the result. The catch command returns 0 if there is no error, and 1 if there is an error.

An example for simple error handling is shown below

```
#!/usr/bin/tclsh
 proc Div {a b} {
      if {b == 0} {
```

```
            error “Error generated by error” “Info String for error” 401
          } else {
            return [expr $a/$b]
          }
}
if {[catch {puts “Result = [Div 10 0]”} errmsg]} {
          puts “ErrorMsg: $errmsg”
          puts “ErrorCode: $errorCode”
          puts “ErrorInfo” \n&errorInfo\n”
}
if {[catch {puts “Result = [Div 10 2]”} errmsg]} {
puts “ErrorMsg: $errmsg”
          puts “ErrorCode: $errorCode”
          puts “ErrorInfo” \n&errorInfo\n”
}
```

When the above code is executed, it produces the following result

```
ErrorMsg: Error generated by error
ErrorCode: 401
ErrorInfo:
Info String for error
(procedure "Div" line 1)
invoked from within
"Div 10 0"

Result = 5
```

As you can see in the above example, we can create our own custom error messages. Similarly, it is possible to catch the error generated by Tcl. An example is shown below

```
#!/usr/bin/tclsh
catch {set file [open myNonexistingfile.txt]} result
puts “ErrorMsg: $result”
puts “ErrorCode: $errorCode”
puts “ErrorInfo:\n$errorInfo\n”
```

When the above code is executed, it produces the following result

```
ErrorMsg: couldn't open "myNonexistingfile.txt": no such file or directory
ErrorCode: POSIX ENOENT {no such file or directory}
ErrorInfo:
    couldn't open "myNonexistingfile.txt": no such file or directory
while executing
"open myNonexistingfile.txt"
```

Strings:

The primitive data-type of Tcl is string and often we can find quotes on Tcl as string only language. These strings can contain alphanumeric character, just numbers, Boolean, or even binary data. Tcl uses 16 bit unicode characters and alphanumeric characters can contain letters including non-Latin characters, number or punctuation.

Boolean value can be represented as 1, yes or true for true and 0, no, or false for false.

String Representations

Unlike other languages, in Tcl, you need not include double quotes when it's only a single word. An example can be

```
#!/usr/bin/tclsh
set myVariable hello
puts $myVariable
```

When the above code is executed, it produces the following result

```
hello
```

When we want to represent multiple strings, we can use either double quotes or curly braces. It is shown below

```
#!/usr/bin/tclsh
set myVariable “hello world”
puts $myVariable
set myVariable {hello world}
puts $myVariable
```

When the above code is executed, it produces the following result

```
hello world
hello world
```

String Escape Sequence

A character literal can be a plain character (e.g., 'x'), an escape sequence (e.g., '\t'), or a universal character (e.g., '\u02C0').

There are certain characters in Tcl when they are preceded by a backslash they will have special meaning and they are used to represent like newline (\n) or tab (\t). Here, you have a list of some of such escape sequence codes

Escape sequence	Meaning

\\	\ character
\'	' character
\"	" character
\?	? character
\a	Alert or bell
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab

Following is the example to show a few escape sequence characters

```
#!/usr/bin/tclsh

Puts “”Hello/tWorld/nSt. Mary’s
```

When the above code is compiled and executed, it produces the following result −

```
Hello World
Tutorialspoint
```

String Command

The list of subcommands for string command is listed in the following table

S.No.	Methods & Description
1	**compare** string1 string2 Compares string1 and string2 lexo graphically. Returns 0 if equal, -1 if string1 comes before string2, else 1.
2	**first** string1 string2 Returns the index first occurrence of string1 in string2. If not found, returns -1.
3	**index** string index Returns the character at index.
4	**last** string1 string2 Returns the index last occurrence of string1 in string2. If not found, returns -1.
5	**length** string Returns the length of string.
6	**match pattern** string Returns 1 if the string matches the pattern.
7	**range** string index1 index2 Return the range of characters in string from index1 to index2.
8	**tolower** string Returns the lowercase string.
9	**toupper** string Returns the uppercase string.

10	**trim** string ?trimcharacters? Removes trimcharacters in both ends of string. The default trimcharacters is whitespace.
11	**trimleft** string ?trimcharacters? Removes trimcharacters in left beginning of string. The default trimcharacters is whitespace.
12	**trimright** string ?trimcharacters? Removes trimcharacters in left end of string. The default trimcharacters is whitespace.
13	**wordend** findstring index Return the index in findstring of the character after the word containing the character at index.
14	**wordstart** findstring index Return the index in findstring of the first character in the word containing the character at index.

Examples of some commonly used Tcl string sub commands are given below.

String Comparison

```
#!/usr/bin/tclsh
set s1 “Hello”
set s2 “World”
set s3 “World”
puts [string compare $s1 $s2]
if {[string campare $s2 $s3]==0} {
        puts “String\’s1\’ and \’s2\’ are same.”
}
if {[string campare $s $s2]==-1} {
        puts “String\’s1\’ comes before \’s2\’.”;
}
if {[string compare $s2 $s1]==1} {
puts “String \’s2\’ comes after \’s1\’.”
}
```

When the above code is compiled and executed, it produces the following result

```
-1
String 's1' and 's2' are same.
String 's1' comes before 's2'.
String 's2' comes after 's1'.
```

Index of String

```
#!/usr/bin/tclsh
set s1 “Hello World”
set s2 “o”
puts “First occurrence of $s2 in s1”
puts [string first $s2 $s1]
puts “Character at index 0 in  s1”
puts [string index $s1 0]
puts “Last occurrence of $s2 in s1”
puts [string last $s2 $s1]
puts “Word end index in s1”
puts [string wordend $s1 20]
puts “Word start index in s1”
puts [string wordstart $ s1 20]
```

When the above code is compiled and executed, it produces the following result

```
First occurrence of o in s1
4
Character at index 0 in s1
H
Last occurrence of o in s1
7
Word end index in s1
11
Word start index in s1
6
```

Length of String

```
#!/usr/bin/tclsh
set s1 “Hello World”
puts “Length of string s1 is”
puts [string length $s1]
```

When the above code is compiled and executed, it produces the following result

```
Length of string s1
11
```

Handling Cases

```
#!/usr/bin /tclsh
set s1 “Hello World”
puts “Upper  case String of s1”
puts [string toupper $s1]
puts “Lower  case String of s1”
puts [string tolower $s1]
```

When the above code is compiled and executed, it produces the following result

```
Uppercase string of s1
HELLO WORLD
Lowercase string of s1
hello world
```

Trimming Characters

```
!#/usr/bin/tclsh
set s1 “Hello World”
set s2 “World”
puts  “Trim right $s2 in $s1”
puts [string trimright $s1 $s2]
set s2 “Hello”
puts “Trim left $s2 in $s1”
puts [string trimleft $s1 $s2]
set s1 “Hello World”
set s2 “ “
puts “Trim characters s1 on both sides of s2”
puts [string trim $s1 $s2]
```

When the above code is compiled and executed, it produces the following result

```
Trim right World in Hello World
Hello
Trim left Hello in Hello World
World
Trim characters s1 on both sides of s2
Hello World
```

Matching Strings

```
#!/usr/bin/tclsh
set s1 "test@test.com"
set s2 "*@*.com"
puts "Matching pattern s2 in s1"
puts [stringmatch "*@*.com"  $s1]
puts "Matching pattern  tcl in s1"
puts [stringmatch {tcl} $s1]
```

When the above code is compiled and executed, it produces the following result

```
Matching pattern s2 in s1
1
Matching pattern tcl in s1
0
```

Append Command

```
#!/usr/bin/tclsh
set  s1 Hello
append s1 " World"
puts $s1
```

When the above code is compiled and executed, it produces the following result

```
Hello World
```

Format command

The following table shows the list of format specifiers available in Tcl

Specifier	Use
%s	String representation
%d	Integer representation
%f	Floating point representation
%e	Floating point representation with mantissa-exponent form
%x	Hexa decimal representation

Some simple examples are given below

```
#!/usr/bin/tclsh
puts [format “%f” 43.5]
puts [format “%e 43.5]
puts [format “%d%s” 4 tuts]
puts [format “%s” “Tcl Language”]
puts [format “%x” 40]
```

When the above code is compiled and executed, it produces the following result

```
43.500000
4.350000e+01
4 tuts
Tcl Language
28
```

Scan command

Scan command is used for parsing a string based to the format specifier. Some examples are shown below.

```
#!usr/bin/tclsh
puts [scan “90” {%[0-9]} m]
puts [scan “abc” {%[a-z]} m]
puts [scan “abc” {%[A-Z]} m]
puts [scan “ABC” {%[A-Z]} m]
```

When the above code is compiled and executed, it produces the following result

```
1
1
0
1
```

Files

Tcl supports file handling with the help of the built in commands open, read, puts, gets, and close.

A file represents a sequence of bytes, does not matter if it is a text file or binary file.

Opening Files

Tcl uses the open command to open files in Tcl. The syntax for opening a file is as follows

```
open fileName accessMode
```

Here, **filename** is string literal, which you will use to name your file and **accessMode** can have one of the following values −

S.No.	Mode & Description
1	**R** Opens an existing text file for reading purpose and the file must exist. This is the default mode used when no accessMode is specified.
2	**W** Opens a text file for writing, if it does not exist, then a new file is created else existing file is truncated.
3	**A** Opens a text file for writing in appending mode and file must exist. Here, your program will start appending content in the existing file content.
4	**r+** Opens a text file for reading and writing both. File must exist already.
5	**w+** Opens a text file for reading and writing both. It first truncate the file to zero length if it exists otherwise create the file if it does not exist.
6	**a+** Opens a text file for reading and writing both. It creates the file if it does not exist. The reading will start from the beginning, but writing can only be appended.

Closing a File

To close a file, use the close command. The syntax for close is as follows

```
close fileName
```

Any file that has been opened by a program must be closed when the program finishes using that file. In most cases, the files need not be closed explicitly; they are closed automatically when File objects are terminated automatically.

Writing a File

Puts command is used to write to an open file.

puts $filename "text to write"

A simple example for writing to a file is shown below.

```
#!/usr/bin/tclsh
set fp [open “input.txt” w+]
puts $fp “test”
close $fp
```

When the above code is compiled and executed, it creates a new file **input.txt** in the directory that it has been started under (in the program's working directory).

Reading a File

Following is the simple command to read from a file

```
set file_data [read $fp]
```

A complete example of read and write is shown below

```
#!/usr/bin/tclsh
set fp [open “input.txt” w+]
puts $fp “test”
close %fp
set fp [open “input.txt” r]
set file_data [read $fp]
puts $file_data
close $fp
```

When the above code is compiled and executed, it reads the file created in previous section and produces the following result

```
test
```

Here is another example for reading file till end of file line by line

```
#!/usr/bin/tclsh
set fp[open “input.txt” w+]
puts $fp “test\ntest”
```

```
close %fp
set fp[open “input.txt” r]
while { [gets $fp data]>=0} {
        puts $data
}
Close $fp
```

When the above code is compiled and executed, it reads the file created in previous section and produces the following result

```
test
test
```

Event driven programs:

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.

In a graphical user interface things happen at unpredictable times. A user might for example select a row in a table while the program is communicating with a remote server. These two events may be unrelated, and the order in which they occur might not have particular meaning for the program. Moreover, an interface typically should remain responsive even when the program is waiting for data on some channel. In event-driven programming the program maintains a queue of events that have occurred and responds to those events as it can. Tcl provides an event loop which starts when one calls vwait. Tk itself calls vwait on startup, making it unnecessary for the main script to do it.

Major classes of events in Tcl/Tk

- GUI events - Tk's *raison d'etre*. Note that this includes send and various wm subcommands as well as bind and the *-command* options to various widgets...
- File events - These occur when it becomes possible to either read from or write to a channel (especially to a pipe, socket or serial port.)
- Timer events - Great for scheduling things for happening later, and brilliant for use with periodic activity.
- Idle events - Useful for when you wish to put something off until there is nothing to do except wait for events (mainly used in the management of display updates...)
- Virtual events - Comes in handy when a certain program state which is reached should call a proc at a higher abstraction level of the application program. It is not good practice to call a high level proc from a lower level. event generate is helpful because (a) the programmer specifies a name for the virtual event (hence a low level software module is consistent in having defined names for ALL internal

resources which can be used by apps) and (b) the event can simply be ignored when not needed.

Nuts and Bolts Internet Programming

The public Internet is a worldwide computer network, that is, a network that interconnects millions of computing devices throughout the world.

In Internet world, all of these devices are called **hosts** or **end systems**. **Hosts** or **end systems** are connected together by **communication links**. They are indirectly connected to each other through intermediate switching devices know as **packed switches**.

Different links can transmit data at different rates, with the **transmission rate** of a link measured in bits/second.

Packet switches come in many shapes and flavors, but two most prominent types in today's Internet are **routers** and **link-layer switches**. Both types of switches forward packets toward their ultimate destinations.

From the sending end system to the receiving end system, the sequence of communication links and packet switches traversed by a packet is known as a **route** or **path** through the network. Rather than provide a *dedicated* path between communicating end systems, the Internet uses a technique known as **packet switching** that allows multiple communicating end systems to share a path, or parts of a path, at the same time.

End systems access the Internet through **Internet Service Provider (ISPs)** including residential ISPs.End systems, packet switches, and other pieces of the Internet, run **protocols** that control the sending and receiving of information within the INternet, The **Transmission Control Protocol (TCP)** and the **Internet Protocol(IP)** are two of the most important protocols in the Internet. The IP protocols specifices the format of the packets that are sent and received among routers and end systems. The Internet's principal protocols are collectively known ad **TCP/IP**.

Internet standards are developed by the Internet Engineering Task Force(IETF). The IETF standards documents are called **request for comments (RFCs)**. RFCs started out as general request for comments to resolve architecture problems that faced the precursor to the Internet. These private networks are often referred to as **intranets**, as they use the same types of host, routers, links, and protocols as the public Internet.

Security Issues

The application provides a security model that restricts the execution of the applet. For example, the application might be a Web browser and the applet might be a program that animates the content of a Web page. Or, an application might consist of only a security model with no other functionality, and it might be used to run large "applets" that implement major applications such as spreadsheets or word processors.

The security issues associated with applets fall into three major groups: integrity attacks, privacy attacks, and denial of service attacks. These are discussed individually in the subsections below, followed by a discussion of risk management in general.

Integrity attacks

A malicious applet may try to modify or delete information in the environment in unauthorized ways. For example, it might attempt to transfer funds from one account to another in a bank, or it might attempt to delete files on a user's personal machine. In order to prevent this kind of attack, applets must be denied almost all operations that modify the state of the host environment. Occasionally, it may be desirable to permit the applet to make a few The Safe-Tcl Security Model DRAFT, limited modifications; for example, if the applet is an editor, it might be given write access to the file being edited.

Privacy attacks

The second form of attack consists of information theft or leakage: a malicious applet may try to read private information from the host environment and transmit it to a conspirator outside the environment. Information disclosed in this way may have direct value to the recipient, such as business information that could affect the price of a company's stock, or its disclosure could damage the party from which it was taken, for example, if it describes an individual's treatment for substance abuse. One approach to the privacy problem is to prevent applets from accessing sensitive information at all.

However, this approach would also prevent applets from performing many useful functions. For example, this approach would prevent applets from helping to display, analyze, and edit sensitive information. A more desirable approach is to give applets access to sensitive information but prevent them from transmitting the information outside the host environment. This approach is called flow control.

In principle, it might seem possible for the security model to analyze the flow of information through the applet and prevent information read from sensitive sources from being written to insecure I/O ports. This might be done by analyzing the application statically to detect illegal flows. Or, it might be done using a technique called data tainting , in which data is tagged with information about its sensitivity.

Data read from a sensitive source is tagged with a high sensitivity level, and attempts to write that data to an insecure I/O port are denied. Unfortunately, these forms of flow control are hard to implement and use.

Denial of service

The third form of attack consists of denial of service, where the applet attempts to interfere with the normal operation of the host system. For example, it might consume all the available file space, cover the screen with windows so that the user cannot interact with any other applications, or exercise a bug to crash its application.

Denial-of-service attacks are less severe from a security standpoint than integrity or privacy attacks, yet they are harder to prevent.

They are less severe because they don't do lasting damage; in the worst case, the effects can be eliminated by killing the application and freeing any extraneous resources that it allocated.

Managing risk

It is unlikely that any security policy can completely eliminate all security threats. For example, any bug in an application gives a malicious applet the opportunity to deny service by crashing the application. In addition, there exist subtle techniques for signaling information that make it nearly impossible to implement perfect flow control for privacy.

Attempts to completely eliminate the risks would restrict applets to such a degree that they would not be able to perform any useful functions. Thus, security models like Safe-Tcl do not try to eliminate security risks entirely.

Instead, they attempt to reduce the risks to a manageable level, so that the benefits provided by applets are greater than the costs incurred by security attacks.

C Interface

Introduction. The Tcl API (also called' Tcl library' or' Tcl C interface') is a huge set of C functions which can be used to create binary extensions to the Tcl language. The functions cover the whole range of Tcl and are, if used right, platform-independent.

The file tcl.h should be considered a public declaration of the Tcl C functions, defines/macros, and structures which a developer can *safely* depend on. Most of the actual function declarations are found in the tclDecls.h file; this has to do with the Stubs mechanism.

Tk-Visual Kits

Tk refers to Toolkit and it provides cross platform GUI widgets, which helps you in building a Graphical User Interface. It was developed as an extension to Tcl scripting language by John Ousterhout. Tk remained in development independently from Tcl with version being different to each other, before, it was made in sync with Tcl in v8.0.

Features of Tk

It is cross platform with support for Linux, Mac OS, Unix, and Microsoft Windows operating systems.

- It is an open source.
- It provides high level of extendibility.
- It is customizable.
- It is configurable.
- It provides a large number of widgets.
- It can be used with other dynamic languages and not just Tcl.
- GUI looks identical across platforms.

Applications Built in Tk

Large successful applications have been built in Tcl/Tk.

- Dashboard Soft User Interface
- Forms GUI for Relational DB
- Ad Hoc GUI for Relational DB
- Software/Hardware System Design
- Xtask - Task Management
- Musicology with Tcl and Tk
- Calender app
- Tk mail
- Tk Debugger

Fundamental Concepts of Tk

Tk adds about 35 Tcl commands that let you create and manipulate widgets in a graphical user interface.

Tk works with the X window system, Windows, and Macintosh. The same script can run unchanged on all of these major platforms.

Tk provides a set of Tcl commands that create and manipulate widgets.

A widget is a window in a graphical user interface that has a particular appearance and behavior.

Widget types include buttons, scrollbars, menus, and text windows.

Tk also has a general-purpose drawing widget called a canvas that lets you create lighter-weight items such as lines, boxes, and bitmaps.

Tk widgets are organized in a hierarchy. To an application, the window hierarchy means that there is a primary window, and inside that window there can be a number of children windows. The children windows can contain more windows, and so on.

Just as a hierarchical file system has directories (i.e., folders) that are containers for files and directories, a hierarchical window system uses windows as containers for other windows.

Widgets are under the control of a geometry manager that controls their size and location on the screen.

Until a geometry manager learns about a widget, it will not be mapped onto the screen and you will not see it.

Tk has powerful geometry managers that make it very easy to create nice screen layouts.

The main trick with any geometry manager is that you use frame widgets as containers for other widgets. One or more widgets are created and then arranged in a frame by a geometry manager.

By putting frames within frames you can create complex layouts. There are three different geometry managers you can use in Tk: grid, pack, and place.

Tk by example

Button

A button is a widget that is typically used by a GUI programmer to receive a simple command from the user - a "press here to submit" type idea is typically represented by a button.

```
for {set i 0} {$i<5} {incr i} {
   grid [button .b$i -text "Button $i" -command "runMyProc $i"] }
```

Events & Binding

Little stop watch

```
#! /usr/bin/tclsh
package require Tk
option add *Button.padY 0       ;# to make it look better on Windows
option add *Button.borderWidth 1
#-------------------------------------------------- testing i18n
package require msgcat
namespace import msgcat::mc msgcat::mcset
mcset de Start Los
mcset de Stop  Halt
mcset de Zero  Null
mcset fr Start Allez
mcset fr Stop  Arrêtez
mcset fr Zero  ???
mcset zh Start \u8DD1
mcset zh Stop  \u505C
mcset zh Zero  ???
msgcat::mclocale en ;# edit this line for display language
#------------------------------------------------------------ UI
button .start -text [mc Start] -command Start
label  .time -textvar time -width 9 -bg black -fg yellow -font "Sans 20"
set time 00:00.00
button .stop -text [mc Stop] -command Stop
button .zero -text [mc Zero] -command Zero
set state 0
bind . <Key-space> {
   if {$state} {.stop invoke
   } else {
```

```
        .start invoke
    }
}
bind . <Key-0> {
    .zero invoke
}
eval pack [winfo children .] -side left -fill y
#-------------------------------------------------------- procedures
proc every {ms body} {eval $body; after $ms [namespace code [info level 0]]}

proc Start {} {
    if {$::time eq {00:00.00}} {
        set ::time0 [clock clicks -milliseconds]
    }
    every 20 {
        set m [expr {[clock clicks -milliseconds] - $::time0}]
        set ::time [format %2.2d:%2.2d.%2.2d \
            [expr {$m/60000}] [expr {($m/1000)%60}] [expr {$m%1000/10}]]
        incr ::titleskip
        if {$::titleskip >= 12} {
            wm title . "Timer $::time"
            set ::titleskip 0
        }
    }
    .start config -state disabled
    set ::state 1
}
proc Stop {} {
    if {[llength [after info]]} {
```

```
        after cancel [after info]
    }
    .start config -state normal
    set ::state 0
}
proc Zero {} {
    set ::time 00:00.00
    set ::time0 [clock clicks -milliseconds]
}
```