CHAPTER 11

# TEST SCRIPT LANGUAGE

We will now look at the TSL language. You have already been exposed to this language at various points of this book. All the recorded scripts that WinRunner creates when you perform an operation is in TSL code syntax. Keep in mind that while mastery of TSL is not required for creating automated tests, knowledge of the language helps to enhance the recorded tests and to create highly sophisticated tests. Skillful usage of TSL can limit the need for manual intervention when running your test. It can also make your test less error-prone.

## Test Script Language

TSL is the script language used by WinRunner for recording and executing scripts. In this chapter, I will provide you with the foundational concepts of TSL. Think of these as the building blocks of TSL scripting that will help you learn how to write the language. Another useful resource to review while learning TSL is the WinRunner help system.

---

TIP: To access the WinRunner help system at any time, press the F1 key.

---

The TSL language is very compact containing only a small number of operators and keywords. If you are proficient in any programming language, you will find TSL easy to learn. In fact, you will find it much easier than learning a programming language because TSL is a script language and as such, does not have many of the complex syntax

structures you may find in programming languages. On the other hand, TSL as a script language has considerably less features and capabilities than a programming language.

Table 11.1 shows the list of features that are available in TSL.

| Feature | Description |
|---|---|
| Comments | Allows users to enter human readable information in the test script that will be ignored by the interpreter. |
| Naming Rules | Rules for identifiers within TSL. |
| Data Types | The different types of data values that are supported by the language. |
| Data Storage | Constructs that can be used to store information. |
| Operations | Different type of computational operations. |
| Branching | Avoids executing certain portions of the code unless a condition is met. |
| Loops | Repeats execution of a certain section of code. |
| Functions | Group of statements used to perform some useful functionality. |

*Table 11.1: Parts of the TSL Language.*

## History

TSL stands for Test Script Language and is often referred to by its acronym. Created by Mercury Interactive an used in several of their product line including WinRunner, XRunner and LoadRunner, TSL has similarities in functionality and keywords to the C language and any of its derivatives. Knowledge of similar languages as JavaScript, Perl or Java would aid in your quick understanding of TSL.

## General Syntax Rules

*1. Semi-colons mark the end of a simple statement.*

You may have seen that many of the recorded statements in your test ends with a semicolon. This is important because TSL, like other C-based languages use the semi-colon to specify the end of a statement much like you would use a period to specify the end of a sentence in English. The following code:

```
foo();
bar();
```

can then be written as:

```
foo(); bar();
```

and WinRunner will still execute them properly.

*2. Compound statements use curly braces and not semi-colons.*

Compound statements, i.e. statements that can hold other statements within them do not end with a semi-colon. So, the if statement, which is a compound statement is written as:

```
if ( 1 > 0 ) {
  foo();
}
```

Notice that the content within the if statement has the semi-colon and not the if statement itself. It would be WRONG to write the if statement in the following form:

```
if ( 1 > 0 ); {
  foo();
}
```

*3. TSL is case-sensitive.*

TSL is also a case-sensitive language so pay special attention to the case of the statements that you write. Most of the identifiers, operators and functions within the language utilize lower case. So, when you are unsure, try lower case.

---

TIP: When writing compound statements, it is highly beneficial to indent the content of the compound statement. This makes it easier to identify the items within a compound statements. Though the interpreter does not care about compound statements, it makes the code easier for you to read.

---

We will now look closely at the parts of the TSL language listed in Table 11.1.

## Comments

Comments allow you to include statements within a TSL test and have these statements ignored by the interpreter. These inserted comments

are useful because they provide the reader of the test script with useful information about the test. To create a comment, type the # symbol anywhere on the line and every statement included after this symbol is regarded as a comment. The # symbol can appear as the first character in a line making the whole line a comment, or anywhere in the line making a partial line comment. e.g.

```
# This is a whole line comment

foo();   # This is a partial line comment
```

---

NOTE: When a comment is specified, the font style and color in the editor changes. You can control how the font looks for comments as well as for several other items in the editor by clicking on Tools->Editor Options.

---

Unlike some other languages that you may be used to, TSL does not have multi-line comments. You will have to place the # character at the start of each line you wish to convert into a comment.

---

TIP: It is a good idea to use comments liberally in your test to make it easier for people to understand the assumptions/decisions you made in the test.

---

## Naming Rules

Every language has rules that define what names are acceptable in the language and which names are not. In TSL, several items have to be given names including variables, constants and functions. These names are the identifiers we use to refer to each of these items. The following rules are used for naming items in TSL.

1. *Must begin with a letter or the underscore* - Names like count, strInfo, _data are acceptable but names like 2day, 4get, *count violate this rule.
2. *Cannot contain special characters except the underscore* - Names can only be made up of alphabets, numbers and the underscore. Symbols and other special characters are not allowed.
3. *Cannot match a reserved word* - Words having special meanings to TSL cannot be used as variable names, so names like for, if, in etc. are not allowed.

4. *Must be unique within a context* - This prevents you from giving two items the same name at the same time. This is for the same reasons you avoid given two children in the same house, the same name.
5. *Are case-sensitive* - Meaning that you can declare foo, Foo, and FOO as three different named items without violating rule 4.

In Chapter 6, I described a naming convention that can be used in renaming GUI map objects. The same convention is very applicable here. By naming variables using prefixes of num and str to denote numbers and strings, you can create variables that are easy to reference.

## Data Types

Every script language has different types of data that can be stored when a script is executing. The different types of data are known as data types. While some languages have several data types, TSL has only two. The two data types available in TSL are:

String - for storing alphanumeric data. Strings are denoted by putting double quotes (") around values. Examples include "Hello", "John Q. Public", "101 Curl Drive", "It was the best of times, it was the worst..."

Number - for storing numeric data. Numbers can have negative values, hold decimal places, or be denoted in exponential notation. Examples include 21, -6, 12.31, -4e5, and 4E-5.

Although there are two data types, we do not need to specify the type of a variable during creation. TSL simply uses the context in which we access the data to determine what the data type is. Information is converted easily and flexibly between both types. The following code:

```
foo = "2" + 1;
```

results in foo being assigned the numeric value 3 because TSL sees this as an arithmetic operation. However, the following code:

```
bar = "2" & 1;
```

results in bar being assigned the string value "21". The difference here is that when & (string concatenation) is used, a string operation is performed and the operands are converted to string. However, when + (addition) is used, the operands are converted to numbers and an arithmetic operation is performed. We will look at the different TSL operators later.

## ESCAPE SEQUENCES

String are defined as values enclosed in double quotes e.g.

foo = "Hello";

would hold the value *Hello*. But what if you want to store the value John "Q" Public? To do this, you have to write the code:

foo = "John \"Q\" Public";

Note that we have included a backslash (\) in front of the double quote. This format allows us to store the literal value of the double quote. This is known as using an escape sequence and it is a way to include certain types of characters and values within a string. TSL defines other escape sequences and I have included these in Table 11.2 below.

| Feature | Description |
|---|---|
| \" | Double quote |
| \\ | Backslash |
| \b | Backspace |
| \n | New line |
| \number | Storing octal numbers e.g. \8 is 10 in decimal notation |
| \t | Horizontal tab |
| \v | Vertical tab |

*Table 11.2: TSL escape sequences.*

When you are referring to file names on a computer, pay careful attention to escape sequences. Referring to the file:

C:\newfile.txt

Will be interpreted by the system as

C:
ewfile.txt

Note that the system converted the \n to a new line statement. To prevent this, you will need to escape the \. So, you must use the \ \ sequence instead. This means the file name will be written as:

C:\\newfile.txt

This is a very important item to remember and you will see the usage of the \\ in all instances where I refer to a file.

## Data Storage

This refers to how information is stored in the test during test execution. Computers need to store large amounts of data during test execution. These include:

- dynamic values - values that can change
- fixed value - values that remain the same through test execution
- sets - groups of values.

TSL uses three different types of storage vehicles. These include Variables - for dynamic storage, constants - for fixed values and arrays - for sets. Each of the storage devices are described below:

Variables - Containers that can hold a piece of information. The information being held can be changed over the lifetime of the variable.

Constants - Containers that can hold a piece of information. The information being held cannot be changed.

Arrays - A single container that can be used to hold several pieces of information at the same time. This is used to hold information that are related e.g. names of people, states etc.

Data storage only refers to information that needs to be stored temporarily during test execution. For information that you want to store permanently, you should consider writing to a file or database.

### Variables

| *Syntax:* | class variable [ = value ]; |
|---|---|

In our code, we often need to temporarily store information that we plan to use later. This is similar to when someone tells you a phone number. You may write this phone number down to later use. The piece of paper you wrote the number on is a storage device for the information. Just as you can erase the value on the paper and write a new piece of information, you can similarly change the value in a variable. So, for the multiple pieces of information that your test script will collect and process you will need variables to hold them.

In many programming languages, a variable must be declared before use. But in TSL, variable declaration is optional so you can create a variable simply by using the variable in your code. Whenever the interpreter comes across a new variable being used in your code, it

automatically handles the declaration for you. This is known as implicit declaration. The only exception to this is in functions (discussed later in this chapter) where variable declaration is required.

Declaring a variable is done using the following form:

```
class variable [ = value ];
```

Meaning that that you either write

```
class variable;
```

or

```
class variable = value;
```

The first form simply declares the variable while the second form initializes it within a value.

The word class should be replaced with a value from Table 11.3 below and variable is any name that conforms to all the naming rules I listed earlier.

| Class | Scope | Use in function | Use in test |
|---|---|---|---|
| auto | Local | Yes | No |
| extern | Global | Yes | Yes |
| public | Global | No | Yes |
| static | Local | Yes | Yes |

*Table 11.3: Class values for variable declaration.*

Based on this, the following declares a static variable named numCount and assigns it a value of 7;

```
static numCount = 7;
```

## Constants

| *Syntax:* | **const** CONSTANT = value; |
|---|---|

Constants are very similar to variables because they store values also. The major difference between these two is that the value of a constant is fixed and cannot be changed during code execution. Constants are typically used when an item being described has a fixed value in the real world. E.g.

```
const DAYS_IN_WEEK = 7;
```

This way, instead of writing:

```
foo = 30 / 7;
```

whenever you are doing date based calculations, you can write:

```
foo = 30 / DAY_IN_WEEK;
```

The difference is subtle but ultimately the second form of the code is a little easier to read. Additionally, if you need to change the calculation of days from calendars days (7) to business days (5), you can easily change this in the constant declaration and have the change propagate through your entire code.

## Arrays

Sometimes you may want to store several pieces of related data such as names. One way to do this is to declare different variables for each value e.g.

```
name1 = "John";
name2 = "Jane";
name3 = "Joe";
```

While this would work for your needs, TSL provides a better mechanism for use when dealing with a collection of similar items. This concept, known as arrays, allows you to store several values of related information into the same variable. Each value must be stored within a different index in the variable name. So, the code written above can be rewritten as:

```
names[0] = "John";
names[1] = "Jane";
names[2] = "Joe";
```

The [ ] specify that you are storing information into an array and the number specifies the appropriate index. Indeed, the code looks very similar to what we wrote previously. So, you might ask, what is the benefit of an array? The answer is that using an array, I can easily perform operations on my entire collection of names without having to deal with multiple variables. I can use TSL constructs such as loops to perform operations on the entire array quite easily. The following code:

```
for (i=0; i<3; i++) {
  print(names[i]);
}
```

prints out the three names in my array. If the array had 50 separate values, I would still use a similarly compact amount of code to print all

the values. I simply would just need to change the range that I am instructing the loop to use. If I used 50 different variables, the amount of code I would need to write to print each name would be far greater.

When working with arrays, there are a few rules to remember.

1. Arrays are declared using the following rules:

```
class array[] [ = initialization ];
```

where class is a variable declaration type (see Table 11.3), array is valid name (see Naming Rules), and initialization is a set of initial values. Notice that in this construct, the initialization is optional so I can declare an array as:

```
public names[] = {"John", "Jane", "Joe" };
```

or

```
public names[];
```

2. Arrays can be initialized in 2 ways. Using standard initialization shown below:

```
public names[] = {"John", "Jane", "Joe" };
```

or by using an explicit form of initialization such as:

```
public names[];
names[0] = "John";
names[1] = "Jane";
names[2] = "Joe";
```

3. TSL supports associate arrays allowing you to use strings as your array indexes. For instance:

```
capital["Ohio"] = "Columbus";
```

is valid TSL code that stores *Columbus* within the array capital at the index *Ohio*.

4. Array indexes do not have to start from 0. As shown above, the indexes do not even have to be numbers at all.

5. Arrays can be multidimensional allowing you to store multiple pieces of information at each index. The following code:

```
names["first", 1] = "John";
names["first", 2] = "Jane";
names["middle", 1] = "Q";
names["middle", 2] = "";
names["last", 1] = "Public";
names["last", 2] = "Doe";
```

generates data that can be displayed in the following form shown in Table 11.4.

| names | 1 | 2 |
|---|---|---|
| first | John | Jane |
| middle | Q | |
| last | Public | Doe |

*Table 11.4: The 2-dimensional names array.*

A 1-dimensional array can be represented as a list, a 2-dimensional array can be represented as a table and a 3-dimensional array can be drawn as a cube. While TSL supports arrays with 4 or more dimensions, it is often more difficult to conceptualize these.

## Operations

Operations are the many forms of computations, comparisons etc that can be performed in TSL. The symbols used in an operation are known as operators and the values involved in these operations are known as operands. Most of the operations in TSL are binary operations, meaning they involve two operands. Two of the operations are unary (i.e. involving a single operand), and one operation is tertiary (i.e. involving three operands).

For unary operations, the syntax is in the form:

| | |
|---|---|
| ***Syntax:*** | operator operand1 |

For binary operations, the syntax is in the form:

| | |
|---|---|
| ***Syntax:*** | operand1 operator operand2 |

For tertiary operations, the syntax is in the form:

| | |
|---|---|
| ***Syntax:*** | operand1 operator1 operand2 operator2 operand3 |

is used instead. Many of the operators are made up of 2 symbols combined together. This is possible because of a lack of enough distinct symbols on the keyboard. Whenever you have such an operator e.g. >=, you cannot write this as > =. (Notice the included space).

WinRunner supports six different types of operations. These are:

***Arithmetic***

Arithmetic operations are used for performing calculations. The operands of an arithmetic operation should be numeric values and the result of this operation is numeric. The arithmetic operators available in TSL are shown in Table 11.5:

| Operator | Description | Usage Example |
|---|---|---|
| + | Addition | foo = 4 + 2; # foo has the value 6 |
| - | Subtraction<br>Negation | foo = 4 - 2; # foo has the value 2<br>foo = -4; # foo has the value -4 |
| * | Multiplication | foo = 4 * 2; # foo has the value 8 |
| / | Division | foo = 4 / 2; # foo has the value 2 |
| % | Modulo | foo = 4 % 2; # foo has the value 0 |
| ^ or ** | Exponent | foo = 4 ^ 2; # foo has the value 16 |
| ++ | Increment | foo++; # The value of foo increases by 1 |
| -- | Decrement | foo --; # The value of foo decreases by 1 |

*Table 11.5: Arithmetic operators.*

### *Comparison*

Comparison operators are used to compare 2 values against each other. The two values being compared should be of a similar type e.g. you should compare 2 number against each other and 2 strings against each other. The result of a comparison operation is the Boolean value of 1 (for true) or 0 (for false). Table 11.6 shows the comparison operators available in TSL.

| Operator | Description | Usage Example |
|---|---|---|
| == | Equality | 4 == 2 # the result is 0 (false) |
| != | Inequality | 4 != 2 # the result is 1 (true) |
| > | Greater than | 4 > 2 # the result is 1 (true) |
| < | Less than | 4 < 2 # the result is 0 (false) |
| >= | Greater than or equals | 4 >= 2 # the result is 1 (true) |
| <= | Less than or equals | 4 <= 2 # the result is 0 (false) |

*Table 11.6: Comparison operators.*

### *Logical*

Logical operators exist to allow us to reduce the value of several Boolean operations to a single Boolean answer. The results of Boolean operations can be combined in three ways:

Conjunctions: when both operands in the logical operation are necessary for a condition to be satisfied.

Disjunctions: when only one operand of the logical operation is needed to satisfy a condition.

Negation: When the value of an operand is reversed.

Table 11.7 provides the list of logical operators available in TSL.

| Operator | Description | Usage Example |
|---|---|---|
| && | And (Conjunction) | 4 > 2 && 0 > 2 # the result is 0 (false) |
| \|\| | Or (Disjunction) | 4 > 2 \|\| 0 > 2 # the result is 1 (true) |
| ! | Not (Negation) | !(4 > 2) # the result is 0 (false) |

*Table 11.7: Logical operators.*

### *Concatenation*

Concatenation allows you to join 2 values as strings. The ampersand (&) character is used for concatenation. An example of this operation is:

foo = “Hello ” & “ World”; # foo has the value *Hello World*

### *Assignment*

Assignment statements change the value of a variable. The equal sign (=) is the assignment operator and the operation simply evaluates the expression on the right side of the assignment operator and assigns this value to the storage container on the left side of the assignment operator.

Several shortcut forms also exist for combining an assignment operation with other types of operations. These shortcuts and their expanded forms are listed in Table 11.8 below:

| Operator | Usage Example | Comparable construct |
|---|---|---|
| += | foo += 2; | foo = foo + 2; |
| -= | foo -= 2; | foo = foo – 2; |

| | | |
|---|---|---|
| *= | foo *= 2; | foo = foo * 2; |
| /= | foo /= 2; | foo = foo / 2; |
| %= | foo %= 2; | foo = foo % 2; |
| ^= or **= | foo ^= 2;<br>foo **= 2; | foo = foo ^ 2;<br>foo = foo ** 2; |
| &= | foo &= “Hello”; | foo = foo & “Hello”; |

Table 11.8: Assignment operators.

---

CAUTION: You must not put a space between symbols when operators are made up of two or more symbols.

---

### *Conditional*

Conditional operators are used as a shorthand method for performing assignments based on the result of a condition. The ? and : operators are involved in this operation and they are used in the following syntactic form:

| | |
|---|---|
| ***Syntax:*** | variable = condition ? truevalue : falsevalue; |

In this construct, if the condition evaluates to true, the variable is set to the value indicated by truevalue. However, if the condition is false, the variable is set to falsevalue.

## Branching

Branching statements allow your code to avoid executing certain portions of code unless a specified condition is met. This condition evaluates to a Boolean value of true or false. When the condition evaluates to true, we say that the condition is met and when it evaluates to false, we say the condition is not met.

TSL uses the resulting value of a condition to decide what code to execute. There are two forms of branching statements supported in TSL and these are the if statement and the switch statement.

---

NOTE: In TSL, 1 represents a Boolean value of true, and 0 represents a Boolean value of false.

---

## If Statements

| *Syntax:* | **if** ( condition ) {<br>[ statements; ]<br>[ } **else** {<br>[ elsestatements; ] ]<br>} |
|---|---|

The first form of a branching statement that we will look at is the if statement. This statement allows users to execute code only if a certain condition is met. The syntax for the if statement is shown above, but though it looks intimidating, it isn't. The syntax represents the three different forms of writing the if statement. These forms are:

### The if Form

| *Syntax:* | **if** ( condition ) {<br>[ statements; ]<br>} |
|---|---|

In this form of the if statement, the specified statements are only executed if the condition evaluates to true. If the condition is false then the value is not printed. An example of this is:

```
if ( x > 0 ) {
  pause("Positive");
}
```

This displays the message "Positive" in a dialog when the value of x exceeds 0. This is the simplest form of the if statement.

### The if-else Form

| *Syntax:* | **if** ( condition ) {<br>[ statements; ]<br>} **else** {<br>[ elsestatements; ]<br>} |
|---|---|

In this form, one of the two sets of statement will always be executed. If the specified condition evaluates to true, the statements in the if block are executed. However, if the condition evaluates to false, then the elsestatements are executed.

Since the condition can never be both true and false at the same time, only one set of statements are executed. Therefore, this form is best suited for an either-or situation where you always expect to either execute *statements a* or *statements b*. The following code displays an example of using this form of the if statement:

```
if ( x > 0 ) {
  pause("Positive");
} else {
  pause("Not positive");
}
```

When this TSL code block executes, the message "Positive" is displayed when the value is greater than zero, if the value is zero or less, then the message "Not Positive" is displayed. In all instances, you will always see one message or the other.

### The if-else-if Form

| *Syntax:* | **if** ( condition ) {<br>[ statements; ]<br>} **else if** ( condition ) {<br>[ elseifstatements; ]<br>[ } **else** {<br>[ elsestatements; ] ]<br>} |
|---|---|

This form is best suited for a multiple-choice scenario where you want to execute a single set of statements based on a list of conditions. Notice that there are multiple conditions and the adjacent code is only executed if the entry condition is met. The following code is an example of this form:

```
if ( foo > 0 ) {
  pause("Positive");
}
else if ( foo == 0 ) {
  pause("Zero");
}
else {
  pause("Negative");
}
```

When this executes, only one set of TSL code is executed. The evaluation of the conditions begins from the first line and once a condition evaluates to true, the related code blocks are executed. You

may notice that at any time, a value can only be greater than zero, zero, or less than zero.

You should be careful to ensure that two conditions cannot evaluate to true at the same time, if they do, only the first scenario where the condition evaluates to true is executed as shown in the following code:

```
if ( foo > 0 ) {
  pause("Positive");
}
else if ( foo > 20 ) {
  pause("Greater than 20");
}
else if ( foo == 0 ) {
  pause("Zero");
}
else {
  pause("Negative");
}
```

when foo > 20 is true, foo > 0 is also true. But because foo > 0 is evaluated before foo > 20 (due to its location in the if statement) the message "Greater than 20" can never be seen. You should be careful to avoid this in your code.

### switch statement

| *Syntax:* | **switch** ( testexpression ) {<br>[ case value1[ , value2, … value-n]: [ statements; ] ]<br>[ **default**: [ defaultstatements; ] ]<br>} |
|---|---|

Another branching statement that is supported in TSL is the switch statement. The switch statement uses the value of a single controlling expression to determine the appropriate set of statements to execute. As shown above, the switch statement contains a list of constant values and whichever matches the testexpression determines the set of code to execute.

The execution of a switch statement can be described to run in the following order:

1. The testexpression is evaluated.
2. Each case statement within the switch statement is evaluated and if the constant value matches the value of the testexpression, the embedded statements are executed.

3. If no case statement has a matching value, the embedded statements in the default label are executed.
4. If no case statement has a matching value and there is no default label, no code is executed in the switch statement and execution continues with the next statement after the switch statement.

The following code uses a switch statement to evaluate an input variable and determines if the variable matches Y or N.

```
switch (foo)
{
  case "Y", "y":
      pause("Yes");
      break;
  case "N", "n":
      pause("No");
      break;
  default:
      pause("Unkown value");
}
```

Notice that the case statement has both "Y" and "y" because we wish to do a case insensitive comparison. You may also notice the break statement that is included after each case statement. This is used to prevent run through, if you don't use the break statement, all the code following your case label will be executed.

# Loops

Loops provide the ability to repeat execution of certain statements. These statements are repeatedly executed until an initial condition evaluates to false. TSL supports four different types of loops and these are the while, do..while, for and for..in loops.

We will now look closely at each of these types of loops, but before we do, we will examine the concept of infinite loops.

## Infinte Loops

Because a loop repeats the execution of a statement, it is possible for the loop to keep running without stopping. This is known as an infinite loop and it is one of the biggest problems that can occur with loops. It occurs when the condition that shuts off the loop never evaluates to false. To prevent infinite loops, ensure that:

a. You are modifying the loop variable.

b. The loop condition will eventually evaluate to false.

### The while Loop

| *Syntax:* | **while** ( condition ) {<br>[ statements; ]<br>} |
|---|---|

The while loop is the simplest form of the loop and it simply repeats execution until the condition evaluates to false. The following code shows an example of using the while loop.

```
foo = 1;
while ( foo <= 3 ) {
  pause(foo);
  foo++;
}
```

This code displays the numbers 1, 2 and 3 in a message box on execution. Notice that the loop variable is being incremented using the ++ (increment) operator.

---

NOTE: When using a while loop, you must modify the loop variable so you don't have an infinite loop.

---

### The do..while Loop

| *Syntax:* | **Do**<br>{<br>[ statements; ]<br>} **while** ( condition ); |
|---|---|

A second form of looping statements is the do..while loop. This loop is used when at least one execution iteration is required of the loop. Notice that the condition is located at the bottom of the loop statement so, all statements embedded within the loop have already been executed at least once. The loop simply allows us to determine whether to repeat execution of the loop.

The following example shows how to use a loop to perform the same operation as the while loop above.

```
foo = 1;
do
```

```
{
  pause(foo);
  foo++;
} while ( foo <= 3 );
```

This code displays the values 1, 2, and 3 just as you may expect. Now look at this following example:

```
foo = 10;
do
{
  pause(foo);
  foo++;
} while ( foo <= 3 );
```

In this example, even though the loop variable begins at 10, the loop still executes and displays 10. This is message is only displayed once because in order to continue, the condition must be evaluated and the condition foo <= 3 evaluates to false.

A good way to look at the do..while loop is that it works exactly like a while loop unless in cases when the loop condition initially evaluates to false. In these instances, the while loop will not run at all but the do..while loop will run once.

## The for Loop

| *Syntax:* | **for** ( initializer; condition; modifier ) {<br>[ statements; ]<br>} |
|---|---|

One of the main criticisms of the other forms of loops is that it is often difficult to track the starting value and loop modification statements. This is because the starting value can occur anywhere before the while or do..while statement, and the loop modification can occur anywhere within the while and do..while statement. This can potentially cause errors where you modify your loop variable twice, or possibly not at all.

The for loop avoids both these problems, as shown about, in the definition statement of the loop, the initializer, condition and modifiers are displayed on the same line. This makes it easy to see what value your loop starts from, the condition that ends the loop and how the loop variable is modified after every iteration.

To be clear, the for loop is very similar in nature to the while loop with the only difference being that it enforces a loop management

structure that the while loop lacks. The following code, shows how to use a for loop to rewrite the code we created for our while loop earlier.

```
for ( foo=1; foo <= 3; foo++ ) {
  pause(foo);
}
```

Notice that information about the loop is much easier to read in this structure. Whenever using the for loop, make sure you do not modify the loop variable within the body of the loop. This mistake is easy to make if you are used to working with the while loop.

### The for..in loop

| *Syntax:* | **for** ( variable **in** array ) {<br>  [ statements; ]<br>} |
|---|---|

The final loop statement which we will look at is the for..in statement. This loop differs from the 3 previous loops we have looked at because it is used to iterate over the values that exist within an array and not until a condition becomes false.

As shown from the syntax, the for..in loop repeats for each value within the array and with each execution, the variable is set to a value within the array. To recreate the same operations we have used for the other loops, we would need the following code:

```
bar = {1, 2, 3};
for ( foo in bar ) {
  pause(foo);
}
```

These are the different loops that exist in TSL. You will agree that the difference between each one of them is mostly subtle. In most cases, each loop form can be used in place of the other so, the type of loop you choose to write is mostly based on what you are most comfortable with.

### Loop Modification

When repeating certain operations, there may be instance when you wish to (a) skip execution for the current iteration or (b) stop the loop before the loop condition evaluate to true. TSL provides keywords that can be used to handle these scenarios.

### Skipping a loop iteration

When you want to avoid execution for certain values in your loop, you use the continue statement. This statement simply transfers execution to the next iteration of the loop.

```
foo = 1;
while (  foo < 10 ) {
  print( foo++ );
  if ( foo % 2 == 0 ) continue;
}
```

When the above code executes, the values 1, 3, 5, 7 & 9 are printed. For each even number, the condition `foo % 2 ==0` evaluates to true and so the continue statement is executed. This skips the loop to the next execution.

### Escaping from a Loop

Sometimes in the midst of repeating an activity, you may wish to escape from the loop. For this, you will use the `break` statement which allows you to exit immediately out of the loop and restart program execution on the next line after the loop.

```
foo = 1;
while (  foo < 10 ) {
  print( foo++ );
  if ( foo % 2 == 0 ) break;
}
```

On execution, the code prints the values 1 and then escapes from the loop because the `foo % 2 == 0` condition evaluates to true.

## Functions

Functions are blocks of code containing one or more TSL statements. These code blocks are very useful in performing activities that need to be executed from several different locations in your code. TSL provides many built in functions as well as provides you the ability to create your own user-defined functions.

In TSL, as in most languages, you have the ability to pass information into a function. The function can then process this information. A function also has the ability to return some result from its processing.

There are two steps involved with working with a function. The first step involves creating the function by declaring it. The statements that a function will perform is added in this step and any return values

that will be provided is also added at this time. The next step after a function is declared is to call the function. The function call executes the contents of the previously declared function.

We will begin by looking at the two types of functions and then look at the process involved with defining the functions.

## Built-In Functions

TSL provides a few hundred functions for various uses. While this may sound daunting, thankfully these functions are broken down into groups. Table 11.9 shows the different groups of built-in functions available in TSL. Also, it is important to remember that certain functions may only be available whenever a specific add-in is included within the environment.

You may refer to Chapter 4 on how to include an add-in.

| Function Type | Description |
|---|---|
| Analog | Functions that are used to track mouse motion over the screen as well as record the clicks of the mouse including the buttons that are clicked. These functions can also be used to track keyboard input. |
| Context Sensitive | Functions for handling the various GUI objects that exist within the AUT. These functions are inserted by WinRunner into the test as you perform a matching operation in the AUT. |
| Customization | Functions that allow you to extend the functionality of WinRunner. Using these functions, you can enhance WinRunner in any of the following ways.<br>▪ Add functions to the Function Generator<br>▪ Specify new functions to be used in your test during recording<br>▪ Create functions for interacting with the user interface<br>▪ Create new GUI checkpoints |
| Standard | Functions for performing several operations such as mathematical computation, string manipulation, date/time interactions, array handling etc. |

*Table 11.9: Function Types in TSL.*

## User Defined Functions

When no built in function provides the functionality you want, you may need to create a user-defined function. User-defined functions are different from built-in functions only because you have to define the functions yourself (hence the name user-defined). I'll show you how to define your functions, and then discuss how to invoke both user-defined and built-in functions.

## Defining Functions

| *Syntax:* | [ class ] **function** functionname ( [ [ mode ] parameterlist ] ) {<br>[ statements; ]<br>[ **return** value; ]<br>} |
|---|---|

The syntax structure above shows the structural elements for defining a function. Below, I have provided a detailed description of each item from a function's definition.

class: [optional] Valid values are either static or public. Using public makes the function available to other tests, and in contrast, using static limits access to only the current test or module that defines the function.

functionname: [required] A value conforming to the rules specified in the Naming Rules section defined earlier.

mode: [optional] A value determining the direction of information flow into and out of the function. Each parameter in the parameterlist has a different mode and the valid choices are:

- in – passes a value into the function.
- out – passes a value out of the function.
- inout – passes a value into and out of the function.

parameterlist: [required when a mode is specified] A list of comma separated variables that can be used to pass value into a function and also retrieve values from the function. The mode determines whether the variable is used to pass values into or retrieve values from the function.

Functions can also contain a return statement. The return statement may be used to return a value from the function or to terminate execution of the function. The return statement is optional

NOTE: All variables used in a function must be declared as either the static or auto class.

```
function sum(in var1, inout var2, out var3) {
  auto answer;
  answer = var1 + var2;
  var3 = time_str();
  return answer;
}
```

*Listing 11.1: A TSL function definition.*

## Invoking Functions

| ***Syntax:*** | functionname ( [parameterlist ] ); |
|---|---|

To use a function, the function must be invoked. This is also known as calling a function. The process for calling a built-in function is exactly the same as calling a user-defined function. Before you can call any function, you must know the following details:

*Purpose*: What the function does.

*Function Name*: The name of the function.

*Parameter List*: The list of values to pass into the function. This list is comma separated and should match the number of parameters in the function definition. The mode of each of parameter is also important in determining whether to pass a value or variable to a function. I have listed below what you can pass for each mode.

- in – pass a value or a variable.
- out – pass a variable.
- inout – pass a variable.

*Return Value*: Any value that may be returned from the function. Some functions do not return a value and when they do, you may choose to ignore the return value.

In the code shown in Listing 11.2, I have shown how to invoke the function that was previously defined in Listing 11.1

```
static numVar, numResult, strExecution;
numVar = 20;
numResult = sum( 3, numVar, strExecution);
```

```
4       print("Result: " & numResult);
5       print("Execution Date/Time: " & strExecution);
```

*Listing 11.2: TSL code to invoke a function.*

On execution, this code will display the following in message log window:

```
Result: 23
Execution Date/Time: Thu Apr 26 13:38:41 2007
```

The following describes what happens in each the line of code from Listing 11.2.

Line 1 creates three variables that we will use in this test. Don't forget that variable declaration is optional in TSL.

Line 2 initializes the numVar variable to the value 2.

Line 3 invokes the sum() function. We pass the value 3 to the first parameter of sum() which has a mode of in. The second parameter has an inout mode so we must pass a variable. We pass the value 20 through the variable numVar. The third parameter is mode out so we pass a variable strExecution to this parameter. The value of this variable will be changed in the sum() function. The result of the function call is assigned to our numResult variable.

Line 4 prints out the contents of numResult. This is the sum of the first 2 parameters.

Line 5 prints out the contents of strExecution. This is the date/time when the function was called.

# Useful TSL Function

I have included below some very useful TSL functions. These functions have such repeated usage in test creation that it is beneficial to know them by heart. They are:

call – Executes a WinRunner test.

call_ex – Launches QuickTest Professional and executes a QTP test. QTP must be installed on the machine for this function call to work.

file_compare – Compares the contents of two files and writes the result of this comparison to the test results log.

invoke_application – Used to launch an application. This is the preferred way of launching an application instead of navigating through the Windows program menu.

load – Loads a WinRunner function library into memory for usage.

load_GUI – Loads a GUI Map file into memory for usage. The contents of the GUI Map file can then be used during test execution.

pause – Displays a message in a dialog and pauses test execution until the user dismisses the dialog.

print – Writes out values to the print log. This is useful when you don't want to pause execution, and also don't need the message saved to the test results window.

report_msg – Used to send information to the test result window. This does not affect the result of the test.

set_window – Sets the context window i.e. the window in which all subsequent operations will be performed in. This is perhaps the most imports function in TSL.

tl_step – Marks a step as pass/fail and send the corresponding information to the test result window.

**THINGS TO WATCH OUT FOR WHEN WRITING TSL**

1. Case sensitivity.
2. Use semi-colon to complete every simple statement.
3. Use curly braces with each compound statement.
4. Pair up double quotes, parentheses and curly braces.
5. Use = for assignment and == for equality.

# Summary

In this chapter you have learned about the TSL language. We began by looking at the contents of the language and the different syntax structures that each of the program elements use. Finally, I described the use of functions and provided a list of useful functions that exist in TSL. In the next chapter we will use everything we have learned in created some tests.