Summary

In this chapter, we had a comprehensive look at what variables are, why we need them and how to use them.

We use variables for the temporary storage of data in our scripts. They can be declared as local or global variables using the commands:

- `:local <variable name>`
- `:global <variable name>`

Data assigned to a variable may be any of the types described in chapter 4. Data can be assigned at the time of variable creation using the "`:local`" and "`:global`" commands as follows:

- `:local <variable name> <data value>`
- `:global <variable name> <data value>`

Local variables exist only for the lifetime of a script. Once the script has completed, they no longer exist, and the data they held is no longer available.

Global variables are system-level variables that exist until they are explicitly cleared. They may be declared in a script or on the CLI, and their data is available to other scripts and CLI sessions. Note that global variables are only available to the RouterOS user who created them. Global variables should only be declared within script global scopes, not local scopes.

If a variable already exists, its value may be set using the command:

- `:set <variable name> <data value>`

Variables have the data type "`nothing`" if they are created with no value assigned or have not yet been created. This can be tested using the "`:typeof`" command.

Variable names are best defined using a mix of letters or numbers. Non-alphanumeric characters may also be used, but variable names must always be specified using speech quotes in this scenario, i.e.:

- `:local InterfaceName "ether1-WAN"`
- `:local "Interface_Name" "ether1-WAN"`

The availability of a variable is determined by the current scope of a script. Two types of scope are available:

- Global scope: this exists between the start and endpoints of a script and can encompass other (local) scopes.
- Local scope(s): these exist between any pair of curly braces within a script. They *always* exist within the script's global scope.

Scopes are hierarchical and may contain multiple child scopes.

Chapter 7 – If Statements

One of the most powerful features of any scripting language is the ability to make decisions based on conditions detected during a script's execution. Detecting unexpected or changing conditions allows a script to adapt to its environment.

The RouterOS "`:if`" statement allows our scripts to test for various conditions and take varying logical paths through our code. This allows the script to perform the most appropriate actions based on detected conditions. It provides decision-making capabilities for our code. For example, we might decide to bring up a second interface if a WAN port is detected as being down. Or perhaps we might raise a log error if a remote website or service is detected as unavailable.

We'll explore the power of "`:if`" statements in this chapter.

What Are If Statements? Why Do We Use Them?

Apart from the simplest of scripts, there are few occasions when the logical flow of our code follows a linear path. There may be many decision points where we wish to inspect varying conditions and take appropriate actions based on the state of the environment detected.

"If" statements allow us to test conditions and make relevant decisions in our code to ensure we achieve an appropriate outcome. By providing decision-making capabilities, they add a degree of apparent "smartness" to our scripts. This enables them to handle complex tasks and scenarios.

The basic format of an "`:if`" statement is shown below:

```
:if ( <test for a condition> ) do={
        <run commands if the condition is true >
}
```

An "`:if`" statement initially runs a test to see if a logical condition is true. If the condition is true, it runs the commands in the code block between the curly braces in the "`do={ }`" block.

If the test condition is false, the code block is ignored, and the code flow continues to the code section beyond the curly braces of the "`:if`" statement.

Let's analyse a short example:

```
# filename: ch7-01-basic-if.rsc

# Print a greeting
:put "Hello, I hope you're well.";

# Check if the time is after 6pm (get time in 21:46:04 format)
:local CurrentTime [:system clock get time];
:put "The current time is : $CurrentTime";

# Say good evening if time is after 18:00
:if ($CurrentTime > 18:00) do={
    :put "Good evening!";
}

# Say goodbye
:put "Thanks for visiting, bye!";
```

This rather polite script prints a series of greetings and will additionally wish you a "good evening" if it detects the local time of the MikroTik is past 18:00. Let's analyse it in detail:

- We start with a simple "`Hello, I hope you're well`" greeting printed to the console

- Next, we read the current local time of the MikroTik using the `[:system clock get time]` command. This value is assigned to a variable called "`CurrentTime`"
- We then execute our "`:if`" statement, which checks to see if the current time is after 1800 hours:
    - Note that the test condition used is "`($CurrentTime > 18:00)`". We're testing if the current time is after 1800 hours. One important point to note is that the data type we're using here is the "time" data type. If this were a simple integer or string type, this comparison would not work (*See chapter 4 for a detailed discussion of data types*). Note that this condition is only true between the hours of 18:01 and 23:59. Outside of these hours, the test condition result is false.
    - If the condition is found to be true, an additional statement is printed that says "`Good Evening!`" using the code in the "`do={}`" code block. Note that this code block could contain several commands if required. It is also a local scope, which is worth remembering when dealing with local variables.
- Finally, we printed the message: "`Thanks for visiting, bye!`".
- Let's run this code at two different times of day to see how the output varies. Here is the code run after 18:00 in the evening:

```
[admin@Router] > /import ch7-01-basic-if.rsc
Hello, I hope you're well.
The current time is : 22:17:42
Good evening!
Thanks for visiting, bye!

Script file loaded and executed successfully
[admin@Router] >
```

Now, let's run the same code the next morning and inspect the output again:

```
[admin@Router] > /import ch7-01-basic-if.rsc
Hello, I hope you're well.
The current time is : 06:57:31
Thanks for visiting, bye!

Script file loaded and executed successfully
[admin@Router] >
```

Notice that we get a similar output in both instances, but when the code is run in the evening, we see an additional "`Good Evening!`" greeting. Note that the code run before *and after* the "`:if`" statement is the same in both scenarios. The additional code for the evening time detection only runs after 18:00.

This code could be further improved to detect whether the time is morning, afternoon or evening and produce a variety of appropriate greetings.

Although this is a trivial example, you can hopefully see how we can implement powerful decision logic in our code using "`:if`" statements. We'd use this decision-making capability in real-world examples to take various actions based on detected network conditions.

There are two types of "`:if`" statement that we can use in our code:

- A basic "`:if`" statement: this includes a single block of conditional code that is executed if a condition is found to be true (like the previous examples we reviewed)
- An ":`if-else`" statement: this includes two blocks of conditional code: one is executed if the test condition is true, and the other is executed if it is false.

We'll look at both types in detail next.

Basic "If" Statement

Let's look at a more useful example of how we might use an "`:if`" statement than those we've tried so far. Remember that the basic format of an "`:if`" statement is as follows:

```
:if ( <test for a condition> ) do={
    <run commands if the condition is true >
}
```

We'll use this format in our next example to test several conditions and print appropriate messages based on those tests. We'll monitor the ability of our MikroTik to reach a destination on the Internet using a ping test. Depending on the result, we'll log an appropriate message to indicate the health of our Internet connection.

The example below uses the "`/ping`" RouterOS command to ping a destination IP address. We'll force the ping traffic across the MikroTik's WAN link and send 10 pings. If we get a response to all 10 pings, we'll assume that our Internet connection is in good shape. We'll assume the Internet is down if we get no ping responses. If we get a few ping responses, we'll assume that the Internet connection is degraded.

In reality, this isn't a very robust method of assessing the state of our Internet connection. It's useful as a rudimentary test and demonstrates a useful application of using the "`:if`" statement. For a more robust, real-world test, we'd likely also test the availability of multiple destinations on the Internet and perhaps check the status of our WAN interface.


```
# filename: ch7-02-basic-if.rsc

# Create a variable for the WAN interface name
:local WanInterface "ether1-WAN";

# Create a variable for the number of pings we'd like to send
:local PingCount 10;

# Create a variable for the destination on the Internet to ping
:local DestinationAddress 8.8.8.8;

# Let's try a ping to the Internet across the WAN interface
:local WanPingCount [/ping $DestinationAddress interface=$WanInterface \
    count=$PingCount];

# Let's log the result of our Internet connection test
if ( $WanPingCount = $PingCount) do={
    :log info "The Internet is up.";
}

if ( $WanPingCount = 0) do={
    :log error "The Internet is down.";

}

if ( ($WanPingCount < $PingCount) and ($WanPingCount > 0) ) do={
    :log warning "The Internet connection may be degraded. (Ping \
        result: $WanPingCount/$PingCount)";
}
```

Here is a brief analysis of the script:

- We start by setting a series of local variables:
  - `WanInterface`: name of the MikroTik WAN interface.
  - `PingCount`: number of pings we'll attempt during our test.
  - `DestinationAddress`: the address our test will ping.
- Next, we assign the ping test result across the WAN link to the local variable `WanPingCount`. The result of the "`/ping`" command is the number of successful ping responses received.
- We then execute commands based on one of three "`:if`" statements:
  - If we get a response to all pings (i.e. 100% success), then we log an informational message to say the Internet is up.
  - If we get no responses to our pings, we log an error message that the Internet is down.
  - If we get fewer responses than the number of pings attempted but still get more than zero responses, then we log a warning message that the Internet connection may be degraded.