Flow Control - Part 2

Hold on to your hats. This lesson is going to be a big one!

More Branching

In the previous lesson on flow control we learned about the **if** command and how it is used to alter program flow based on a command's exit status. In programming terms, this type of program flow is called *branching* because it is like traversing a tree. We come to a fork in the tree and the evaluation of a condition determines which branch we take.

There is a second and more complex kind of branching called a *case*. A case is multiple-choice branch. Unlike the simple branch, where we take one of two possible paths, a case supports several possible outcomes based on the evaluation of a value.

We can construct this type of branch with multiple **if** statements. In the example below, we evaluate some input from the user:

```
#!/bin/bash

read -p "Enter a number between 1 and 3 inclusive > " character
if [ "$character" = "1" ]; then
    echo "You entered one."
elif [ "$character" = "2" ]; then
    echo "You entered two."
elif [ "$character" = "3" ]; then
    echo "You entered three."
else
    echo "You did not enter a number between 1 and 3."
fi
```

Not very pretty.

Fortunately, the shell provides a more elegant solution to this problem. It provides a built-in command called **case**, which can be used to construct an equivalent program:

```
#!/bin/bash

read -p "Enter a number between 1 and 3 inclusive > " character
case $character in
    1 ) echo "You entered one."
        ;;
    2 ) echo "You entered two."
        ;;
```

```
    3 ) echo "You entered three."
        ;;
    * ) echo "You did not enter a number between 1 and 3."
esac
```

The **case** command has the following form:

```
case word in
    patterns ) commands ;;
esac
```

case selectively executes statements if word matches a pattern. We can have any number of patterns and statements. Patterns can be literal text or wildcards. We can have multiple patterns separated by the "|" character. Here is a more advanced example to show how this works:

```
#!/bin/bash

read -p "Type a digit or a letter > " character
case $character in
                                # Check for letters
    [[:lower:]] | [[:upper:]] ) echo "You typed the letter $character"
                                ;;

                                # Check for digits
    [0-9] )                     echo "You typed the digit $character"
                                ;;

                                # Check for anything else
    * )                         echo "You did not type a letter or a digi
esac
```

Notice the special pattern "*". This pattern will match anything, so it is used to catch cases that did not match previous patterns. Inclusion of this pattern at the end is wise, as it can be used to detect invalid input.

Loops

The final type of program flow control we will discuss is called *looping*. Looping is repeatedly executing a section of a program based on the exit status of a command. The shell provides three commands for looping: **while**, **until** and **for**. We are going to cover **while** and **until** in this lesson and **for** in a upcoming lesson.

The **while** command causes a block of code to be executed over and over, as long as the exit status of a specified command is true. Here is a simple example of a program that counts from zero to nine:

```
#!/bin/bash

number=0
while [ "$number" -lt 10 ]; do
```

```
    echo "Number = $number"
    number=$((number + 1))
done
```

On line 3, we create a variable called `number` and initialize its value to 0. Next, we start the **`while`** loop. As we can see, we have specified a command that tests the value of `number`. In our example, we test to see if `number` has a value less than 10.

Notice the word `do` on line 4 and the word `done` on line 7. These enclose the block of code that will be repeated as long as the exit status remains zero.

In most cases, the block of code that repeats must do something that will eventually change the exit status, otherwise we will have what is called an *endless loop*; that is, a loop that never ends.

In the example, the repeating block of code outputs the value of `number` (the **`echo`** command on line 5) and increments `number` by one on line 6. Each time the block of code is completed, the test command's exit status is evaluated again. After the tenth iteration of the loop, `number` has been incremented ten times and the **`test`** command will terminate with a non-zero exit status. At that point, the program flow resumes with the statement following the word `done`. Since `done` is the last line of our example, the program ends.

The **`until`** command works exactly the same way, except the block of code is repeated as long as the specified command's exit status is false. In the example below, notice how the expression given to the **`test`** command has been changed from the **`while`** example to achieve the same result:

```
#!/bin/bash

number=0
until [ "$number" -ge 10 ]; do
    echo "Number = $number"
    number=$((number + 1))
done
```

Building a Menu

A common user interface for text-based programs is a *menu*. A menu is a list of choices from which the user can pick.

In the example below, we use our new knowledge of loops and cases to build a simple menu driven application:

```
#!/bin/bash

selection=
until [ "$selection" = "0" ]; do
    echo "
    PROGRAM MENU
    1 - Display free disk space
    2 - Display free memory
```

```
    0 - exit program
"
    echo -n "Enter selection: "
    read selection
    echo ""
    case $selection in
        1 ) df ;;
        2 ) free ;;
        0 ) exit ;;
        * ) echo "Please enter 1, 2, or 0"
    esac
done
```

The purpose of the **until** loop in this program is to re-display the menu each time a selection has been completed. The loop will continue until selection is equal to 0, the "exit" choice. Notice how we defend against entries from the user that are not valid choices.

To make this program better looking when it runs, we can enhance it by adding a function that asks the user to press the Enter key after each selection has been completed, and clears the screen before the menu is displayed again. Here is the enhanced example:

```
#!/bin/bash

press_enter()
{
    echo -en "\nPress Enter to continue"
    read
    clear
}

selection=
until [ "$selection" = "0" ]; do
    echo "
    PROGRAM MENU
    1 - display free disk space
    2 - display free memory

    0 - exit program
"
    echo -n "Enter selection: "
    read selection
    echo ""
    case $selection in
        1 ) df ; press_enter ;;
        2 ) free ; press_enter ;;
        0 ) exit ;;
        * ) echo "Please enter 1, 2, or 0"; press_enter
    esac
done
```

When your computer hangs...

We have all had the experience of an application *hanging*. Hanging is when a program suddenly seems to stop and become unresponsive. While you might think that the program has stopped, in most cases, the program is still running but its program logic is stuck in an endless loop.

Imagine this situation: you have an external device attached to your computer, such as a USB disk drive but you forgot to turn it on. You try and use the device but the application hangs instead. When this happens, you could picture the following dialog going on between the application and the interface for the device:

```
Application:    Are you ready?
Interface:  Device not ready.

Application:    Are you ready?
Interface:  Device not ready.

Application:    Are you ready?
Interface:  Device not ready.
```

and so on, forever.

Well-written software tries to avoid this situation by instituting a *timeout*. This means that the loop is also counting the number of attempts or calculating the amount of time it has waited for something to happen. If the number of tries or the amount of time allowed is exceeded, the loop exits and the program generates an error and exits.
