5.10 While / Until, and User Input

`while` and `until` loops also exist in bash. While loops operate just the same as other languages, and the `until` loop works as a `while` loop that's testCondition is the opposite. The test expressions look similar to those of the `if` command. The while loop's general syntax is as:

```
while [[ testCondition ]]
do
  commandlist
done
```

The same syntax works for the `until` loop:

```
until [[ testCondition ]]
do
  commandlist
done
```

Often times, when you write a `while` loop, it's typically to display a menu and to get some amount of user input. Til now, we haven't had any user input other than writing commands directly to the command line and passing in arguments. To retrieve input from the within a script, we'll need to use the `read` command. In the below toy problem, we'll take in some input and manipulate it!

`whileExample.sh`

```
#!/bin/bash

name="Some name"
again="yes"

while [[ $again = "yes" ]]
do
        echo What\'s your name?
        read name
        allCaps=$(echo $name | tr [a-z] [A-Z])
        echo Hello $name! If I were to shout your name, it would be: $allCaps

        echo Would you like me to read your name and shout it again? \(yes or no would be sufficient!\)
        read again

done
```

```
17
18 echo It has been too fun!
```

Above, we're using a while loop, where our condition for continuing the loop is where the variable `again` must equal `"yes"`. First, though, we want to read in a name, just for fun. We prompt the user, read the name with `read` , and store whatever is read into the `name` variable. From there, we translate it with the command expansions, and store it into `allCaps`, where we then print it out again with echo.

Finally, we prompt the user if they'd like to go through the process again. Any answer that is not yes will stop the loop.

```
1 ./whileExample.sh
```

> What's your name?
>
> Matt
>
> Hello Matt! If I were to shout your name, it would be: MATT
>
> Would you like me to read your name and shout it again? (yes or no would be sufficient!)
>
> no
>
> It has been too fun!

Conversely, were we to use the `until` command:

`untilExample.sh`

```
1  #!/bin/bash
2
3  name="Some name"
4  again="yes"
5
6  until [[ $again = "no" ]]
7  do
8          echo What\'s your name?
9          read name
10         allCaps=$(echo $name | tr [a-z] [A-Z])
11         echo Hello $name! If I were to shout your name, it would be: $allCaps
12
13         echo Would you like me to read your name and shout it again? \(yes or
   no would be sufficient!\)
14         read again
15
16 done
17
```

```
18 echo It has been too fun!
```

Seeing that in action:

```
1 ./untilExample.sh
```

> What's your name?
>
> Matt
>
> Hello Matt! If I were to shout your name, it would be: MATT
>
> Would you like me to read your name and shout it again? (yes or no would be sufficient!)
>
> eh, I dunno
>
> What's your name?
>
> matt...
>
> Hello matt...! If I were to shout your name, it would be: MATT...
>
> Would you like me to read your name and shout it again? (yes or no would be sufficient!)
>
> no
>
> It has been too fun!

Ultimately speaking, the `while` and `until` commands do the same thing, but they're just checking for the opposite boolean values.

5.11 Numerical Expressions

Doing math inside of a shell script can confusing. Suppose we had a quick script:

`badMath.sh`:

```
1  #!/bin/bash
2
3  one=1
4  two=2
5
6  echo one is $one
7  echo two is $two
8
9  three=$one+$two
10 echo $one + $two is: $three
```

Upon running this script, you'd very likely expect to see: `1 + 2 is: 3`. However, the actual output is:

```
./badMath
```

> one is 1
>
> two is 2
>
> 1 + 2 is: 1+2

We've danced around this so far. Every variable stored in bash is string valued. However, there are a number of times in which you'll need to do arithmetic operations. We saw arithmetic expansions earlier with:

```
echo $(( 1 + 2 ))
```

> 3

But you can also use the `let` command to make your code a bit more readable (without having to use expansions):

`goodMath.sh`

```
#!/bin/bash

one=1
two=2

echo one is $one
echo two is $two

let three=$one+$two
echo $one + $two is: $three
```

By prefixing the assignment of `three` with `let`, we are then telling the shell to do the maths!

Numerical Expressions with Conditionals:

When looking to use numerical values in conditionals, there are a number of ways to go about this.

Flags

We've seen one above already using flags:

```
#!/bin/bash
```

```
2
3  directoryCount=$(ls -l | grep ^[d] | wc -l)
4  RED_COLORATION='\033[0;31m' #red color
5  NO_COLORATION='\033[0m'   #no color
6  ORANGE_COLORATION='\033[0;33m' #orange color
7
8  if [[ $directoryCount -gt 10 ]]
9  then
10     printf "${RED_COLORATION}GREATER THAN 10 DIRECTORIES ${NO_COLORATION}
   Please rethink your subdirectory solutions!\n"
11 elif [[ $directoryCount -gt  5 ]]
12 then
13     printf "${ORANGE_COLORATION}Warning! You have $directoryCount
   directories!${NO_COLORATION} You may wish to reconsider this many\n"
14 else
15     echo Not greater than 10
16 fi
```

There are a number of flags for if. By using the correct flags, you can specify that the values you're comparing are integers:

- `-eq`: is equal to
- `-ne`: is not equal to
- `-lt`: is less than
- `-le`: is less than or equal to
- `-gt`: is greater than
- `-ge`: is greater than or equal to

While these flags don't exactly make use of the mathematical notation that we are used to, they do come built in with the if command.

Expansions

Flags are useful, however, often times you may want to just use the mathematical notation that you've become accustomed to. If you were to attempt to use this notation without anything additional, you'd see:

`badMathIf.sh`

```
1  #!/bin/bash
2
3  first=1
4  second=2
5
6  echo give me a number please:
7  read first
```

```
echo give me another number:
read second

if [[ $first > $second  ]]
then
 echo $first is greater than $second!
else
 echo $second is greater than $first!
fi

echo How curious?
```

This notation looks correct at first glance, and it is "correct" in that it's not wrong. It's just not doing exactly what we're expecting:

```
./badMathIf.sh
```

> give me a number please:
>
> 10
>
> give me another number:
>
> 5
>
> 5 is greater than 10!
>
> How curious?

What's happening above is that we're comparing the strings "5" and "10". When you place strings into alphabetical order, you only look at the first letter (and then look at the following letters if the strings have matching characters). Luckily for us, we can use expansion notation to treat these values as numerical ones:

`goodMathIf.sh`

```
#!/bin/bash

first=1
second=2

echo give me a number please:
read first
echo give me another number:
read second

if (( $first > $second  ))
then
```

```
 echo $first is greater than $second!
else
 echo $second is greater than $first!
fi

echo How curious?
```

Notice in the code above that the only thing that has changed is that we've changed the notation of our `if`'s test expression from `[[]]` to `(())`. Be extremely careful that you remember what to surround your test expressions with.

5.12 Break and Continue

When you find yourself in a loop, there are often times certain conditions that you may wish to either exit the loop entirely, or just move onto the next iteration. That's what break and continue are for. `break` is a command that kicks you out of the loop entirely, whereas `continue` moves onto the next iteration.

Suppose we have a loop with a menu:

`continueBreak.sh`

```
#!/bin/bash

function printMenu () {
    echo Please select one of the following:
    echo 1. Reset the variables back to their original values.
    echo 2. See the output of \$first+\$second
    echo 3. See the output of \$\(\(first + second\)\)
    echo 4. Exit the program
    echo
}

originalValueFirst=1
originalValueSecond=2
first=$originalValueFirst
second=$originalValueSecond
userInput='5'

echo This program will print the values of the two variabls \"first\" and
\"second\"
echo After the menu is displayed and the commands are executed,
echo the variables \"first\" and \"second\" will be incremented

```

```
while [[ "infinite" ]]
do
    echo The two values are currently:
    echo first:$first    second:$second
    echo
    printMenu
    read userInput
    echo

    case $userInput in
        1)
           first=$originalValueFirst
           second=$originalValueSecond
           continue
           ;;
        2)
           strVal=$first+$second
           echo first+second \= $strVal
           ;;
        3)
           mathVal=$((first+second))
           echo \$\(\(first+second\)\)\=\> $first + $second \= $mathVal
           ;;
        4) break
           ;;
        *) echo Bad input! Please see the menu.
           continue
           ;;
    esac

    echo Incrementing the values...
    ((first++))
    ((second++))
    echo;echo
done

echo The final values are:
echo first: $first        second:$second
echo Thank you so much for using our script!
```

So in this above script, we have a function for printing a basic menu. We then have our variable list (along with original values for easy resetting). Inside the while loop (that is an infinite loop because it's always returning true for a string with any value) we print the data and then get a user's input. From there we decide what to do.

At option `1`, we reset the data and then continue. What does that mean? When we continue, we go immediately back to the beginning of the while loop! At options `2` and `3` we wind up illustrating how bash commands work. Option `4` breaks out of loop entirely, and moves to line `57`, and the default `*` continues (i.e. moves back to the beginning of the while loop).

For the options that did not `break` or `continue`, we then increment the values with the increment operator within the math expansion!

5.13 Working with Files:

In our earlier script, `versionBump.sh` we initially checked to see fi a file existed with the `-f` flag inside of our if. There are a number of other flags that we can use inside of our if statements to ensure that we can even access a file, write to a file, etc:

- `-r`: Determine if the file is readable
- `-w`: Determine if the file is writable
- `-x`: Determine if the file is executable
- `-f`: Determine if the file is an ordinary file
- `-e`: Determine if the file exists
- `-s`: Determine if the file has a size (that is, if the file has any contents)
- `-o`: Determine if the file is owned by the user
- `-d`: Determine if the file is a directory

Let's write a script to tell us in plain text what a file is:

```
#!/bin/bash


fileName=$1

### File types:
if [ -e ./$fileName ]; then
        echo $fileName exists
fi

if [ -f ./$fileName ]; then
        echo $fileName is an ordinary file
fi

if [ -d ./$fileName ]; then
        echo $fileName is a directory
fi

if [ -o ./$fileName ]; then
```

```
        echo $fileName is owned by $USER
fi


### Contents and permissions
if [ -s ./$fileName ]; then
        echo $fileName has contents
fi

if [ -r ./$fileName ]; then
        echo $fileName is readable
fi

if [ -w ./$fileName ]; then
        echo $fileName is writable
fi

if [ -x ./$fileName ]; then
        echo $fileName is executable
fi

```

If we then attempt to run this file on any other files or directories, we can then in plain text what our permissions are, and whether or not the file exists and is a directory or an ordinary file:

```
./fileType.sh example.c
```

> example.c exists
>
> example.c is an ordinary file
>
> example.c has contents
>
> example.c is readable
>
> example.c is writable

Or for something like a directory:

```
./fileType.sh sampleDir
```

> sampleDir/ exists
>
> sampleDir/ is a directory
>
> sampleDir/ has contents
>
> sampleDir/ is readable

> sampleDir/ is writable
>
> sampleDir/ is executable

What distinguishes an "ordinary file" is that the file is an executable or some text file. Non ordinary files are links and directories.