Shell Scripting Part I

Tyler Hutcheon

Shell Scripting

Overview

- The Linux shell can control the system with commands and perform file operations of start applications

- You can create a file that includes shell commands and start this file like an application
  - This is called a **shell script**

- A Linux system offers different shell types
  - Scripts written for one shell may not work in another shell

- Most Linux OSes use the **bash** shell as the default
    - BASH = Bourne Again SHell

Shell Scripting

Overview

- Reasons why Shell Scripting will make your life better
  - You can automate many daily system tasks
    - Reduces time and effort to manage the system
  - They control the boot procedure and other system functions
  - Shell scripting is relatively easy to learn
  - They run on any system with BASH installed
- There are also some minor disadvantages
  - Shell scripts are slow compared to other languages
  - They can use a lot of CPU power

Shell Scripting

The Secret

- Shell Scripting is very, very simple.

- Take a bunch of commands that you want to run on the command line and put them in a file.
  - That’s a shell script!

- Where it gets complicated is when you begin using more powerful features of your shell, like variables, loops, functions and other powerful commands.
  - But these are all things that you can do on the command line normally.

- In fact, the better you become at shell scripting, the better you will be at the Linux command prompt.

Shell Scripting

Objectives

- Basic Script Elements
- Use Control Structures

Basic Script Elements

Basic Script Elements

Flow Charts for Scripts

- Flow charts are a way to visualize the programming elements of the script
- Flow charts assist the author in laying down the steps required to achieve the desired goals
- Flow charts make is clear which constructs are needed
- Flow charts provide a clear outline of what you expect the script to do

Basic Script Elements

The Basic Rules of Shell Scripting

- A shell script is an ASCII text file (traditionally ending in .sh)
- It contains commands to be executed in order (one at a time)
- Scripts are not "executed", the commands you write in the script are not machine code, the CPU would have no idea how to run them.
- Instead, they are "Interpreted": read by another program, BASH, an Interpreter, which translates the requests into machine code that can be run on the processor
    - This is typically a slow process, but we don't notice it when we type things in at the command line.
    - When you "execute" a script, it reads the first line of the file to see what it is, and based on the configuration there, runs BASH instead, and has BASH interpret the script.

Basic Script Elements

The Basic Rules of Shell Scripting

- To run scripts, there are 3 main options:

1. Run the interpreter and provide the file it should read:

   Ex: **bash script.sh** or **sh script.sh**

2. Make it executable, and execute it: (note the "./")

   Ex:
   ```
   chmod 755 script.sh
   ./script.sh
   ```

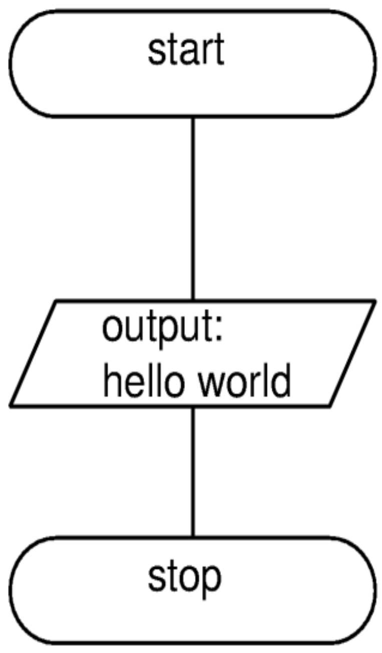
3. Make it executable, put it in the $PATH, and execute it:

   Ex:
   ```
   chmod 755 script.sh
   mv script.sh ~/bin
   script.sh
   ```

Basic Script Elements

The Basic Rules of Shell Scripting

- The flow chart for a simple script
  - The 3 elements of a shell script with the corresponding code.



```
#!/bin/bash

# This script ...

echo "Hello world!"

exit 0
```

Basic Script Elements

The Basic Rules of Shell Scripting

- The 3 elements of a shell script

-**Start:** The first line of any shell script must be the ***shebang***: **#!**

-Specifies which shell program to execute the script **/bin/bash**

-Comments begin with a “**#**” character

-**Commands:** The commands follow, run sequentially

-**Stop:** Here is where you do any cleanup

-Defines the script's exit status where **0** indicates **success** and **1** indicates **failure**

-You can query the exit status with **echo $?**

Basic Script Elements

Troubleshooting

- The first issue you can run into is that the file is in DOS format (meaning the line returns are **\r\n** (carriage return + newline) or **^M** (carriage return)
    - Linux and UNIX use **\n** (newline) as a standard line break.
- This confuses BASH.
    - You will see odd syntax errors, or simply "Bad Interpreter"

- The first line must begin with "**#!**", must be followed by the path and filename of the interpreter, and must end with a line return ("**\n**")
    - To fix, first try the command "**dos2unix**".
    - If that doesn't work, make sure your first line is written properly, with a proper newline at the end.

Basic Script Elements

Example 1 – hello.sh

Basic Script Elements

How to Develop Scripts that Read User Input

- You can use the **read** command to read user input

- It takes a variable as an argument where it stores the input

-Example: **read VARIABLE**

- Useful if you want to ask the user a question, or to pause the script until the user presses enter

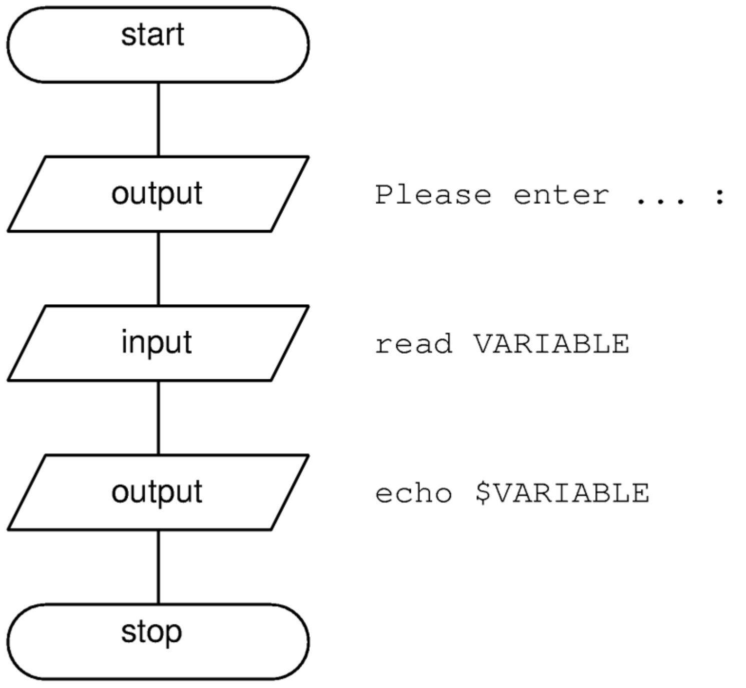
- Example:

```
echo “What is your name?”
read NAME
```

Basic Script Elements

How to Develop Scripts that Read User Input

- Flow chart for a Script that Reads User Input



```
Please enter ... :

read VARIABLE

echo $VARIABLE
```

Basic Script Elements

Example 2 – name1.sh

Basic Script Elements

How to Develop Scripts that Read User Input

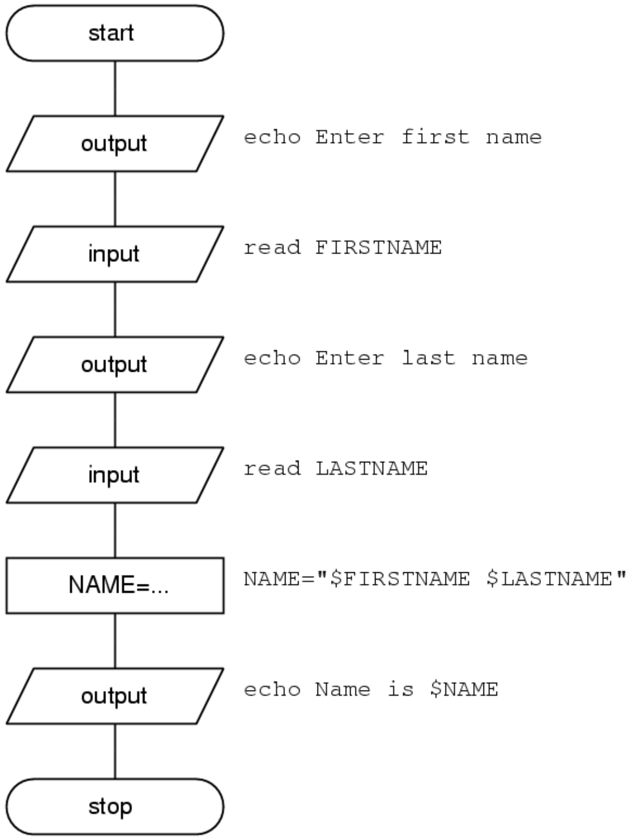
- Instead of “read”, you can also accept command-line input:
- **./script.sh this is a “list of items” on the “command line”**

- $0=“./script.sh”   (i.e. the name of the script, as it was executed)
- $1=“this”
- $2=“is”
- $3=“a”
- $4=“list of items”
- $5=“on”
- $6=“the”
- $7=“command line”

Basic Script Elements

Perform Basic Script Operations with Variables

- Flow chart showing how a string value can be assigned to a variable



```
echo Enter first name
read FIRSTNAME
echo Enter last name
read LASTNAME
NAME="$FIRSTNAME $LASTNAME"
echo Name is $NAME
```

Basic Script Elements

Perform Basic Script Operations with Variables

•Create a variable NAME

-NAME=”$FIRSTNAME $LASTNAME”

-Combine two variables and assign the combined value to another variable

-To assign a value to a variable use the name of the variable

-To use the value of a variable put $ before the name

-Variable names are in UPPER CASE by convention

-Variables defined in a script will be Local variables. They will only be accessible to the script. Once the script stops, the variables are no longer available.

-Variables can be exported, allowing sub-shells to access them:

-export NAME

-… or …

-export NAME=“$FIRSTNAME $LASTNAME”

Basic Script Elements

Example 3 - name2.sh / find.sh

Basic Script Elements

User Input

- Beware of user input! Where possible, avoid using user input at all. When you do use it, check it first, to be sure it contains something valid.

- When in doubt, always quote filenames and pathnames that aren’t statically defined.
  - rm –rf $INPUTFILE
  - Vs.
  - rm –rf “$INPUTFILE”
    - What if $INPUTFILE=“/home/ username/”? (with a space?)

- Also, use the least powerful command to solve your problem (is the –rf really necessary?)

Basic Script Elements

Debugging

- Debugging BASH scripts is very easy. There are two methods:
- 1. Run bash with "-x" to debug. Before running any command, it will print out the command that it is running with a + sign before it.

```
bash -x /path/to/file.sh
sh -x /path/to/file.sh
#!/bin/bash -x
```

- 2. Use echo. If you are going to use any dangerous command (rm, userdel, mv, etc), ***ALWAYS ALWAYS ALWAYS*** put "echo" before it until you have completely debugged your script. Using echo there will cause it to print out the command it would run, instead of actually running it.

```
echo rm -rf "$FILE"
echo userdel "$USER"
echo mv "$FILE" "$DESTINATION"
```

Basic Script Elements

System Variables

- System Environment Variables (should not change or set these):

$HOSTNAME	- The server's hostname
$HOSTTYPE / $MACHTYPE	- The server's type (eg: x86_64 / x86_64-redhat-linux-gnu)
$TERM	- Your terminal mode (Linux, VT100, etc)
$USER / $LOGNAME	- Your username
$ID / $UID	- Your numeric userid
$SHELL	- Your shell path and filename
$MAIL	- Where your mailbox is
$PATH	- The binary search path
$PWD	- Your present working directory
$OLDPWD	- Your previous working directory
$LANG	- Your ISO-coded language (eg: en_US.UTF-8)
$HOME	- Your home directory
$_	- The full path of the last command run
$PS1	- The command prompt

- There may also be others. Use **env** or **set** to view the current environment variables.

Basic Script Elements

Use Command Substitution

- Means that the output of a command is used in a shell command line or in a shell script
- The command is always included in backticks **`` `command` ``**
- **Examples:**

```
#!/bin/bash
echo “Today is `date +%m/%d/%Y`”
```

```
#!/bin/bash
TODAY=`date +%m/%d/%Y`
echo “Today is $TODAY”
```

Basic Script Elements

Example 4 - info.sh

Basic Script Elements

Use Arithmetic Operations

- Shell scripts often use values assigned to variables for calculations

- The Bourne shell is limited is some ways
- Can only perform operations with whole numbers (integers)
- All values are assigned 64-bit values
  - Thus possible values range from -2^63 to +2^63 – 1

- There are 4 different ways to do math in BASH
  - For the following, assume we want to add 10 to $B and store it in $A

Basic Script Elements

Use Arithmetic Operations

- Method 1 – Declare
  - Declare is a BASH built-in. This means it will be fast.
  - The declare" lines define variables $A and $B as being Integers.

```
declare -i A
declare -i B
A=B+10
```

  - In this case, $B doesn't need the "$" because strings can't be assigned to A.

- Method 2 – Using Parenthesis
  - This is also a BASH built-in. The braces tell BASH that it needs to interpolate the contents arithmetically.

```
A=$((B+10))
A=$[$B+10]
```

  - Both formatting options here will work equally as well.

Basic Script Elements

Use Arithmetic Operations

- Method 3 – let
  - Let is another BASH built-in. The “let” tells BASH that it needs to interpolate the result arithmetically.
  - **let A=“$B + 10”**

- Method 4 - **External command expr**
  - Bourne shell compatible
  - Slower than built-in commands
  - Only works with **+**, **-**, *****, **/**, **%**
  - **A=`expr $B + 10`**

Basic Script Elements

Example 5 – sum.sh

Use Control Structures

Use Control Structures

Create Basic Branches With the if command

- Basic **if** command

```
if condition
    then
        commands
fi
```

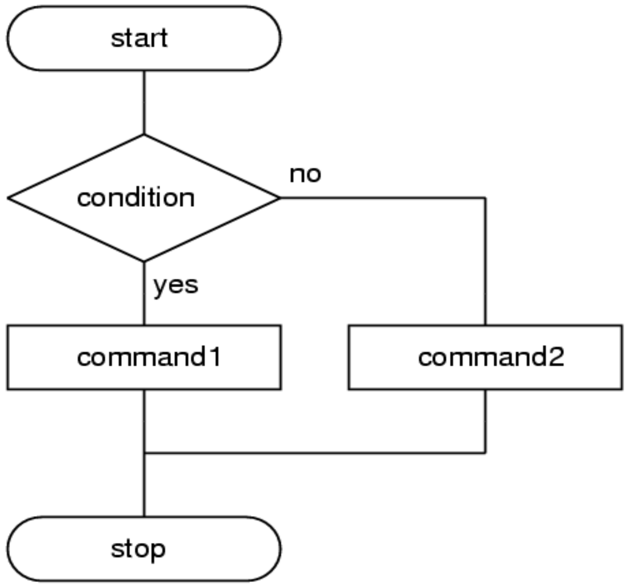
- Extended **if** command

```
if condition
    then
        command1
    else
        command2
fi
```

Use Control Structures

Create Basic Branches With the if command

- A flow chart of the **if** command



Use Control Structures

Create Basic Branches With the if command

- Command structure

- Must begin with **if** and end with **fi**
- **condition** is a command that is run.
    - The condition is successful / true if the error code is 0.
    - The condition is unsuccessful / false if the error code is >=1

True	**=**	**Successful**	**=**	**0**
False	**=**	**Unsuccessful**	**=**	**1**

- If the exit status is not zero or the condition is not true, the shell goes to the end of the branch
- If an **else** statement is present then it goes to the end of the **else** statement
- When used in a shell script, individual commands must follow immediately after a command separator
- You can run this on the command line as-is, or use semicolons in place of line returns to fit it on a single line:

» **If condition; then commands; fi**

Use Control Structures

Create Basic Branches With the if command

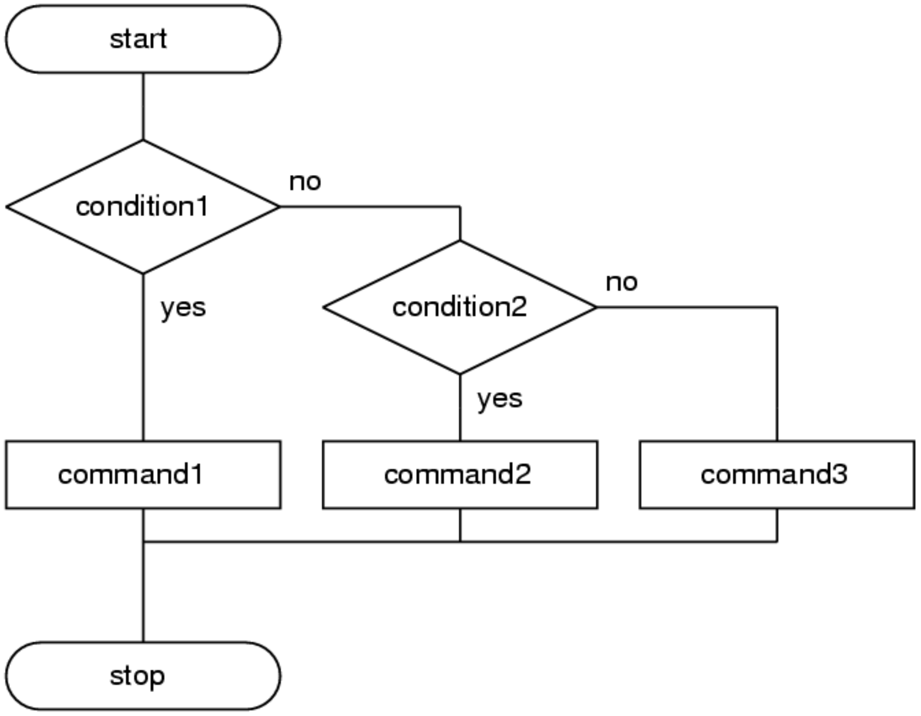
- Several **if** branches are often nested in each other
- **elif** represents the sequence “else if”
- Example:

```
if condition1
   then
      command1
   elif condition2
   then
        command2
      else
        command3
fi
```

Use Control Structures

Create Basic Branches With the if command

- A flow chart of the **elif** command



Use Control Structures

Create Basic Branches With the if command

- There are several ways to use the Bash shell to successively execute several commands
  - Includes using the separators **&&** and **||**
  - Executes a second command depending upon the success or failure of the first
  - **command1 && command2**
    - Executes command2 if command1 exits with success
  - **command1 || command2**
    - Executes command2 if command1 exits with failure
  - These separators are short forms of an **if** branch

Use Control Structures

Create Basic Branches With the if command

- Alternative if + test syntax
  - if [“$VARIABLE” = “value”]
  - then
    - Command1
    - Command2
  - fi
  - The space between the bracket and the condition is important.

  - if [[-e *.sh]]
  - then
    - Command1
    - Command2
  - fi
  - Double brackets support shell expansion of * and ? – but it must **not** be quoted.

Use Control Structures

Example 6 - file_check.sh

Use Control Structures

Build Multiple Branches with a case statement

- Can create multiple branches with **case**

- The expression contained in a variable is compared with a number of expressions and a command is executed for each expression matched

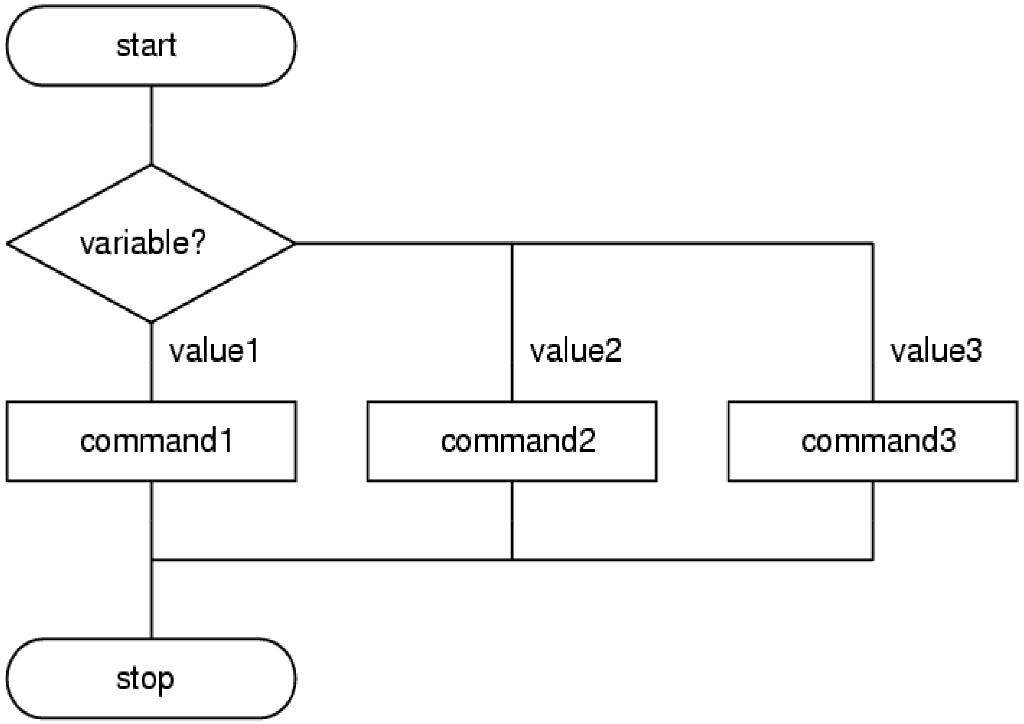
- Basic **case** statement structure

```
case $variable in
     value1)  command1;;
     value2)  command2;;
     value3)  command3;;
     *) command 4;;
esac
```

Use Control Structures

Build Multiple Branches with a case statement

- Basic **case** statement flow chart



Use Control Structures

Build Multiple Branches with a case statement

- To allow for several expressions to be matched within one branch, you simply list the expressions on the same line separated by a |

- The Asterisk * is often used as the last expression to cover all cases not matched by any other alternatives

- In the expressions you can supply alternatives to ensure exact matches. You provide these alternatives in brackets (basic regex!)

```
case “$CREATURE” in
     [dD]og  |  [cC]at  |  [mM]ouse)   echo “4 legs!”;;
     [sS]pider )                       echo “6 legs!”;;
     *)                                echo “???”
esac
```

Use Control Structures

Example 7 - yes_no.sh

Use Control Structures

Create Loops Using the while and until Commands

- The purpose of a loop is to test a certain condition and to execute a given command while the condition is true (**while loop**) or until the condition becomes true (**until loop**)
- The structure of a **while** loop:

```
while condition
do
  commands
done
```

- The structure of an **until** loop:

```
until condition
do
  commands
done
```

Use Control Structures

Create Loops Using the while and until Commands

- The **while** loop remains operative
- WHILE the condition’s exit status is true
- (Will keep looping if the condition’s exit status is zero, stops looping when the condition’s exit status is non-zero)

- The **until** loop remains operative
- UNTIL the condition is true.
- (Will keep looping if the condition’s exit status is non-zero, stops looping when the condition’s exit status is zero)

- These commands are opposite of each other. Some people always use one, some people always use the other, others will use both, to avoid “negative” statements (**while x is not equal to y** vs. **until x = y**)

Use Control Structures

Example 8 - counter1.sh / counter2.sh

Use Control Structures

Process Lists with the for loop

- A **for** loop allows you to run a set of commands once for each item in a list.
- When you write a for loop, you dream up a variable name, called an iterator (and therefore generally called $i, even though it’s not uppercase per convention).
- The list of items is generally passed as space-separated list of words or numbers. If there are 10 items in the list, the commands between **do** and **done** will be run 10 times.
- On each cycle of the loop, the iterator variable will be set to the next item in the list.
- If you don’t supply a list, **for** will use the input variables: $1, $2, $3, ...
  - These are the command line parameters passed to the script

Use Control Structures

Process Lists with the for loop

- The structure of a **for** loop

```
for VARIABLE in item1 item2 item3
  do
      commands
  done
```

- Example

```
for i in 1 2 3 4 5 6 7 8
  do
      ping -c1 W0110DaAbcXyz0$i
  done
```

Use Control Structures

Process Lists with the **for** loop

- To run this manually on the command prompt, you can type it as-is with line breaks, or, more commonly, you can use a semicolon as a separator:
   - for i in 1 2 3 4; do ping -c1 W0110DaAbcXyz0$i; done

- C-style for loops are also possible. This loop will create variable $a, loop while $a is less than or equal to $LIMIT, incrementing $a on each loop:

```
LIMIT=10
for ((a=1; a <= LIMIT ; a++))
do
    echo -n "$a "
done
```

Use Control Structures

Example 9 – lowercase1.sh

Use Control Structures

Interrupt Loop Processing (a.k.a Spaghetti Code)

- Use the **continue** command to exit from the current iteration of a loop and resume with the next iteration of the loop
- **continue** = skip the rest of the commands in this loop

- The **break** command is another way to introduce a new condition within a loop

- **break** = stop the entire loop right now.

- These create “Spaghetti Code”, it makes the script unconventional and much harder to read. Avoid these where possible, and if you use them, document it well.

Use Control Structures

Interrupt Loop Processing (a.k.a Spaghetti Code)

- Example of using the **continue** command

```
for FILE in `ls *.docx`
  do
    if test -e “/Documents/$FILE”
     then
       echo "The file $FILE exists."
       continue
    fi
    cp “$FILE” /Documents
  done
```

Use Control Structures

Interrupt Loop Processing (a.k.a Spaghetti Code)

- Example of the way to avoid using the continue command

```
for FILE in `ls *.docx`
  do
    if test -e “/Documents/$FILE”
     then
       echo "The file $FILE exists."
     else
      cp “$FILE” /Documents
    fi
done
```

Use Control Structures

Example 10 – lowercase2.sh