Software Systems

Lectures Week 3

Bash

Expressions – Control Structures – Developer Techniques

Prof. Joseph Vybihal

Computer Science

McGill University

Readings

- Chapter 2 from textbook
- Bash and command-line help:
  - http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html
  - http://ss64.com/bash/

Part 1

Bash Expressions

Variables

- There are four kinds of variables in a shell scripts:
  - Environment Variable : these variables are used to customize the operating system and the shell (use SETENV).
  - User-created : these are script variables (use SET or simply declare the variable: X = 5).
  - Positional Parameters : command line parameters (using $1, $2, etc.).
  - Session Variables: defined by the OS or Web Server when the user logs in.

Positional Parameters

- Passing parameters from the command-line prompt to the script.
  - Eg (no arguments): ./script
  - Eg (arguments) : ./script 5 2 Bob

- From within the script you can access the 5, 2, Bob using the $position notation.
  - X = $1 puts 5 into variable X
  - Y = $2 puts 2 into variable Y
  - Z = $3 puts Bob into variable Z

Prompt: ./bashfilename 5 10 bob

$0 = ./bashfilename

$1 = 5

$2 = 10

$3 = bob

Example

The command ls with positional parameters.

Create the script:
>> vi myls

```
#!/bin/bash
ls -l-a $1
```

Just the sha-bang and the ls command, but using the positional parameter

Run the script:
>> ./myls *.doc

What happens internally:
ls -l-a *.doc

Changing *.doc at the command line with *.txt will then list all the txt files.

Positional Variables

- System parameters:
  - $# : number of arguments on the command line
  - $- : options supplied to the shell
  - $? : exit value of the last command executed
  - $$: process number of the current process
  - $! : process number of the last command done in background
- Command-line parameters:
  - $n : where n is from 1 through 9, reading left to right
  - $0 : the name of the current shell or program
  - $* : all arguments on the command line ”$1 $2 ... $9” as a string
  - $@ : all arguments on the command line (”$1”, ”$2”, . . ., ”$9”) as an array

Script using all positional parameters

•The following script will print out the positional variables:

```
#!/bin/sh
echo "$#:" $#
echo '$-:' $-
echo '$?:' $?
echo '$$:' $$
echo '$!:' $!
echo '$3:' $3
echo '$0:' $0
echo '$*:' $*
echo '$@:' $@
```

What is displayed if this is executed at the prompt?

>> ./pos 3 5 7 8 9

Simple Script Example

•The following script gathers information about the system and stores it in a file specified at the command line.

```
#!/bin/sh
# Assume program name is info.sh
uname -a > $1
date >> $1
who >> $1
```

•The output was as follows :

```
[jvybihal][~/cs206] ./info.sh output.txt

[jvybihal][~/cs206] cat output.txt
Linux rogue 2.6.12-gentoo-r4 #1 SMP ...
Thu Aug 10 10:57:38 EDT 2006
jvybihal pts/0 Aug 10 08:04 (dz2.cs.mcgill.ca)
```

User-created Variables

Declaring and using regular program variables

Declaring in Java is type specific:

```
int x;
x = 10;
```

Script variables work like Java variables except they are un typed.

Declaring in Scripts is type free:

```
x=10
x=”bob”
```

Scripts are sensitive to white spaces.

User-created Variables

Using variables:

```
x="Bob"
echo $x
ls $x
y=$x
```

Note:

echo $y	echo y	echo "y"
Print contents of Y.	Undefined/Unclear.	Print the letter y.

$ vi test.sh

```
# test.sh
x=10
y = 20
echo $x
echo $y
```

Left spaces before and after the equals

$ chmod +x test.sh
$ ./test.sh

```
./test.sh: 2: ./test.sh: y: not found
10
```

Output from running the script.
Notice the error message for y.

Bash Variables and Expressions

- Bash is untyped
  - This means that variables can be assigned to anything
  - This means that variables do not need to be declared
- Bash variables and strings look similar
  - Therefore the need for the $ operator, example:
    - X = 5; X = “Bob”; X = Bob ← the last Bob could be a string or var
    - Solution: X = “Bob”; X = $Bob ← first is a string the second is a var
- Bash expressions have type information
  - Bash uses the square brackets [] and the round brackets () to distinguish type information
  - This is important because we can write command line commands and they can appear as strings or expressions

Mixed Examples

```
X=5             # assign the integer 5
X="Bob"         # assign the string Bob
X=`ls`          # assign to X a list of file names
set `date`      # $0=day, $1=month, $2=year,…
grep $1 `ls`    # in all the files see if $1 exists
```

Capturing Complex Output

- Some commands, such as date, have output that require an extra bit of parsing to use.

```
Sun Aug 13 11:42:38 EDT 2006
```

- You can use the set command to capture and parse the output.

```
set `date`
```

- The output will be stored in $1, $2, $3, etc.
- Note that using set will erase any data you might already have in $n.

Example of set

- The following script executes the date command and outputs the parsed result.

```
#!/usr/bin/sh
set `date`
echo "Time: $4 $5"
echo "Day: $1"
echo "Date: $3 $2 $6"
```

- The output would be as follows:

```
Time: 12:45:54 EDT
Day: Sun
Date: 13 Aug 2006
```

Expressions

- Command-line tools
  - Integer tool (**expr**, with +, -, *, /, =, !=, %, >, <, etc.)
    - expr 5 + 2
    - expr 5 = 5
    - expr $x = $y
    - sum=`expr 5 + 2`
  - BASIC CALCULATOR --- **bc**
    - If you need to use fractions, use bc to evaluate arithmetic expressions using C programming language syntax.
    - echo $[3/4]
      at the command prompt, it would return 0 because bash only uses integers when answering.
    - Use bc:
      echo 3/4 | bc
      X= `echo 3/4 | bc`

Expressions

- Bash language syntax
  - On the command line (or a shell) try this:
    - echo 1 + 1
  - If you expected to see '2' you'll be disappointed.
  - What if you want BASH to evaluate some numbers you have? The solution is this:
    - echo $((1+1))
  - This is to evaluate an arithmetic expression. You can achieve the same effect also like this:
    - echo $[1+1]
    - echo $[“$x”+”$y”]
    - echo $[[$x+$y]] # improved version of the square bracket

Expressions

- Bash language syntax
  - $(command) is the same as `command`
    - echo `ls | grep “bob”`
    - echo $(ls | grep “bob”)
  - $((expression))
    - Will evaluate the expression instead of a command (see last slide)
  - ${} expression tool
    - animal=“cat”
    - echo $animals # No such variable as “animals”.
    - echo ${animal}s
      - cats
    - echo $animal_food # No such variable as “animal_food”.
    - echo ${animal}_food
      - cat_food

Operators

- The ones you are used to:
  - = != + - * / %
- New ones:
  - =~ for regular expressions: X=[[$ANSWER =~ ^y(es)?$]]
  - *,? basic pattern matching: X=[[$ANSWER = y*]]

Quotes

None ---- up to the version of the shell

' ----- as is

“ ---- pre-process first

` ---- exec command, turn ans into string

“ “ “ ---- we will talk about later...

In Bash the ' and the “ are interpreted as the same.

Double Quote

- Preprocess interpretation
  - X='abc\n'
  - Y="abc\n"
  - echo X
    - Outputs: abc\n
  - echo Y
    - Outputs: abc
      <new line>
  - Escape characters:
    - \n new line
    - \t tab
    - \b back space
    - \a alert
    - \\ display a back slash

In Bash you must use:

echo –e "abc\n"

To see the effect.

Writing to STDOUT

The `echo` command writes to STDOUT

echo “text with default carriage return”

echo –e “text with back slash pre processing”

echo –n “text without default carriage return”

What does this print”

echo “hello” echo –n “hello” echo –e “hello\n”

Reading from STDIN

- The `read` command allows you to read a string from STDIN.
- That string is then stored in the specified variable.

```
#!/bin/bash

echo "What is your name?"
read name
echo "Your name is $name."        bob
echo 'Your name is $name.'        $name
```

Question

- Write a script that will tell you if your friend is currently logged into the system.

- Write a script that calculates the number of days you have been alive. Assume 365 days for every year. Do not consider leap years. Ask the user for their age.


```
./days
Years:
20
```

```
#! bash
echo “Years:”
read age
x=`expr $age * 365`
echo $x

echo `expr $age * 365`

echo $(( $age * 365 ))
```

```
vi abc.txt
./days < abc.txt
```

Part 2

Bash Control Structures

Conditionals in Bash

The test Command

- The test command is used to evaluate a condition.
- The test command can evaluate condition at the file, string or integer level.

Syntax:

```
if test EXPRESSION
if [ EXPRESSION ]        ← Bash
if ( EXPRESSION )
```

File Tests

Example: if (-r .cshrc)

- -r file: true if it exists and is readable
- -w file: true if it exists and is writeable
- -x file: true if it exists and is executable
- -f file: true if it exists and is a regular file
- -d name: true if it exists and is a directory
- -h or -L file: true if exists & a symbolic link
- and many more . . .

String Tests

Example: if ($x)

- -z string : true if the string length is zero
- -n string : true if the string length is non-zero
- string1 = string2 : true if strings are identical
- string1 != string2 : true if strings not identical
- string : true if string is not NULL

Integer Tests

- n1 -eq n2 : true if integers n1 and n2 are equal
- n1 -ne n2 : true if integers n1 and n2 are not equal
- n1 -gt n2 : true if integer n1 is greater than integer n2
- n1 -ge n2 : true if int n1 is greater than or equal to int n2
- n1 -lt n2 : true if integer n1 is less than integer n2
- n1 -le n2 : true if int n1 is less than or equal to int n2

Condition Examples

If [-f /var/log/messages]

If [$? –eq 0]

If ! Grep $USER /etc/passwd

If [“$num” –gt “150”]

If [“$(whoami)” != ‘root’]

Control Structures

The if Statement

- Bash if-statement behaves similar to Java

```
if _condition_
then
    _code_
elif _condition_
then
    _code_
else
    _code_
fi
```

Example

- The following example program can be used to add or subtract two numbers.

```
#!/bin/sh
if test $1 = “add”
then
   result=‘expr $2 + $3‘
elif test $1 = “sub”
then
   result=‘expr $2 - $3‘
else
   result=0
fi
echo "The result is $result \n"
```

```
% ./calc add 5 10
```

Example

```
#!/bin/bash
# Count=99
if [ $1 -eq 100 ]
then
  echo "Count is 100"
elif [ $1 -gt 100 ]
then
  echo "Count is greater than 100"
else
  echo "Count is less than 100"
fi
```

```
% ./bla 50
```

The case Statement

- A case statement is similar to a Java switch statement.

```
case condition in

    condition1) action1;;
    condition2) action2;;
    condition3 | condition4) action3;;

    *) else_action;;

esac
```

Example

```
$ ./prog add  5  2
      $1  $2  $3
```

- The following example program is a reformatting of the if example, but with a case statement.

```
#!/bin/bash
case $1 in
“add” | “addition”) result=‘expr $2 + $3‘;;
“sub” | “subtraction”) result=‘expr $2 - $3‘;;
*) result=0;;
esac
echo "The result is $result \n"
```

Bash Example

```
$ cat yorno.sh
#!/bin/bash

echo -n "Do you agree with this? [yes or no]: "
read yno
case $yno in

        [yY] | [yY][Ee][Ss] )
                echo "Agreed"
                ;;

        [nN] | [nN][Oo] )
                echo "Not agreed, you can't proceed the installation";
                exit 1
                ;;
        *) echo "Invalid input"
            ;;
esac

$ ./yorno.sh
Do you agree with this? [yes or no]: YES
Agreed
```

The for Loop

- The for loop is similar to a Java *iterator*.
- It allows you to iterate (loop) over a list of strings.

```
for var in list
do
    actions
done
```

Example

- The following script executes the file command for each file in the specified path.

```
#!/usr/bin/sh
for i in `ls $1`
do
    file $i;
done
```

```
$ ./bla *.*
```


```
$ ls *.doc
F1.doc   f2.doc f3.doc
$ ./bla *.doc
```

```
for i in “f1.doc f2.doc f3.doc”
do
    Do some stuff
done
```

1. do some stuff with i = “f1.doc”
2. do some stuff with i = “f2.doc”

Example

```
$ cat for2.sh
i=1
weekdays="Mon Tue Wed Thu Fri"
for day in "$weekdays"
do
 echo "Weekday $((i++)) : $day"
done

$ ./for2.sh
Weekday 1 : Mon
Weekday 2 : Tue
Weekday 3 : Wed
Weekday 4 : Thu
Weekday 5 : Fri
```

Example

```
$ cat for5.sh
i=1
for file in /etc/[abcd]*.conf
do
 echo "File $((i++)) : $file"
done

$ ./for5.sh
File 1 : /etc/asound.conf
File 2 : /etc/autofs_ldap_auth.conf
File 3 : /etc/cas.conf
File 4 : /etc/cgconfig.conf
File 5 : /etc/cgrules.conf
File 6 : /etc/dracut.conf
```

The while Statement

- Very similar to Java

```
while condition
do
    actions
    [continue]
    [break]
done
```

Using parameters

- The following script will pad a file with zeros.

```
#!/usr/bin/bash

i=`wc -c < $1`;
while test $i -lt $2
do
    echo -n "0" >> $1;
    i=`wc -c < $1`;
done
```

```
% ./fill ass1.c 100
```

Wrap

```
$ more dext
#!/bin/sh

if [ $# = 0 ]
    then
        echo "Usage: $0 name"
        exit 1
fi

user_input="$1"
grep -i "$user_input" << DIRECTORY
    John Doe     555.232.0000    johnd@somedomain.com
    Jenny Great 444.6565.1111   jg@new.somecollege.edu
    David Nice   999.111.3333    david_nice@xyz.org
    Don Carr     555.111.3333    dcarr@old.hoggie.edu
    Masood Shah 666.010.9820    shah@Garments.com.pk
    Jim Davis    777.000.9999    davis@great.advisor.edu
    Art Pohm     333.000.8888    art.pohm@great.professor.edu
    David Carr   777.999.2222    dcarr@net.net.gov
DIRECTORY

exit 0
$ dext
```

```
$ ./dext Jenny
Jenny Great 444.6565.1111 jg@
$
```

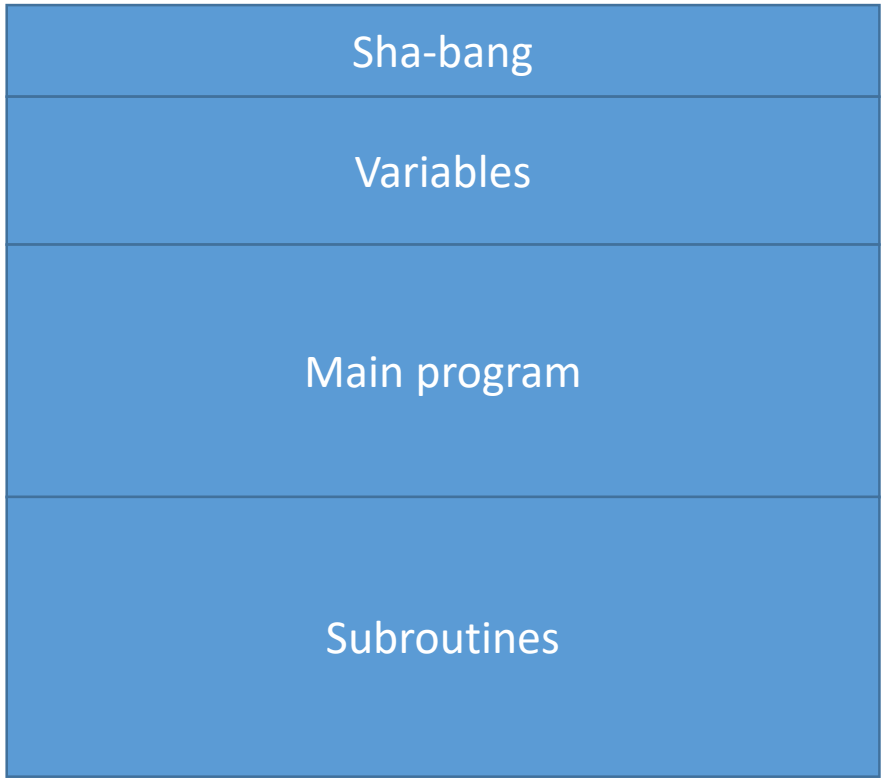
grep "$1" < filename.txt

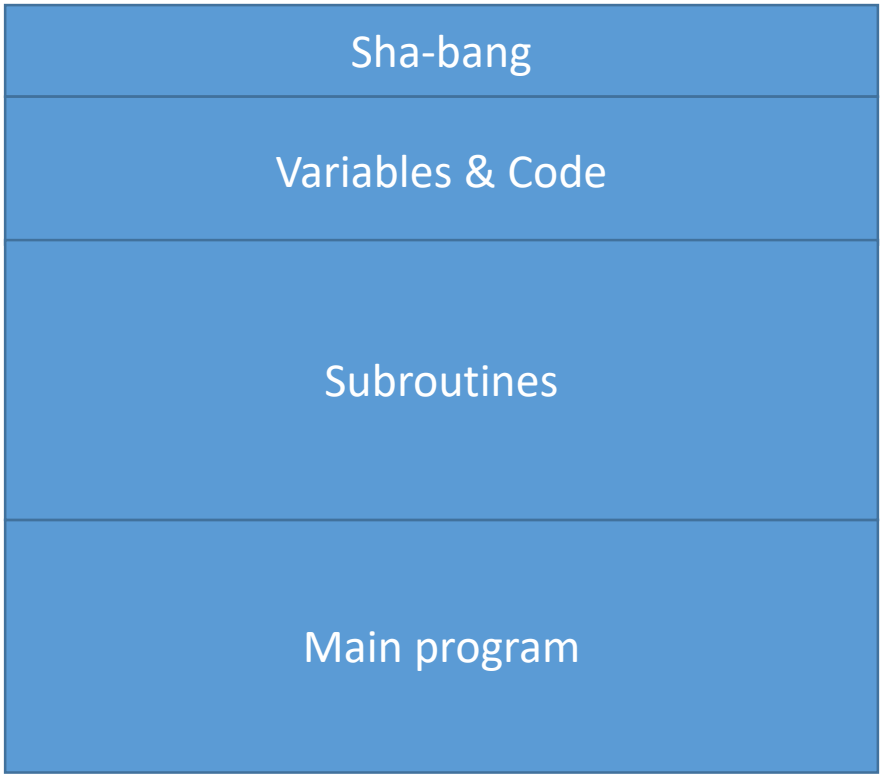
Grep $x "john doe jenny great....."

Functions

- Bash functions are limited
  - Passing parameters is possible through positional variables
  - Returning results is limited
  - Scope rules apply
    - Scope rules work as in other languages
    - Global variables can be used as a means to return values and pass values to a function

Bash File Structure





Scope Rules

A variable is identified in the following order based on where is was used and where is was created:

1st Block

Defined within a control structure

2nd Local

Defined within a function

3rd Global

Defined at the top of the script


```
$ cat dext
#!/bin/sh

if [ $# = 0 ]
    then
        echo "Usage: $0 name"
        exit 1
fi

OutputData()
{
    echo "Info about $user_input"
    (grep -i "$user_input" | sort) << DIRECTORY
    John Doe      555.232.0000    johnd@somedomain.com
    Jenny Great   444.6565.1111   jg@new.somecollege.edu
    David Nice    999.111.3333    david_nice@xyz.org
    Don Carr      555.111.3333    dcarr@old.hoggie.edu
    Masood Shah   666.010.9820    shah@Garments.com.pk
    Jim Davis     777.000.9999    davis@great.advisor.edu
    Art Pohm      333.000.8888    art.pohm@great.professor.edu
    David Carr    777.999.2222    dcarr@net.net.gov
DIRECTORY
    echo                  # A blank line between two records
}

# As long as there is at least one command line argument (name), take the
# first name, call the OutputData function to search the DIRECTORY and
# display the line(s) containing the name, shift this name left by one
# position, and repeat the process.

while [ $# != 0 ]
do
    user_input="$1"       # Get the next command line argument (name)
    OutputData            # Display info about the next name
    shift                 # Get the following name
done
```

Functions

Notice that the function is in the middle of the program.

Notice that global variables are being used to pass parameters.

Notice how the function is called.

Notice the “shift” command at the end of the while loop.


```
#!/bin/bash
# gloabal x and y
x=200
y=100

math(){
  # local variable x and y with passed args
  local x=$1
  local y=$2
  echo $(( $x + $y ))
}

echo "x: $x and y: $y"
echo "Calling math() with x: $x and y: $y"

# call function
math 5 10

# x and y are not modified by math()
echo "x: $x and y: $y after calling math()"
echo $(( $x + $y ))
```

Notice positional variables used to pass parameters.

Notice the use of the local command to define scope.

Notice the function call.

What do the echo commands print out?


```
#!/bin/sh
adduser()
{
  USER=$1
  PASSWD=$2
  shift ; shift
  COMMENTS=$@
  useradd -c "${COMMENTS}" $USER
  if [ "$?" -ne "0" ]; then
    echo "Useradd failed"
    return 1
  fi
  passwd $USER $PASSWD
  if [ "$?" -ne "0" ]; then
    echo "Setting password failed"
    return 2
  fi
  echo "Added user $USER ($COMMENTS) with pass $PASSWORD"
}
## Main script starts here
adduser bob letmein Bob Holness from Blockbusters
if [ "$?" -eq "1" ]; then
  echo "Something went wrong with useradd"
elif [ "$?" -eq "2" ]; then
   echo "Something went wrong with passwd"
else
  echo "Bob Holness added to the system."
fi
```

Question

Write a Bash script that asks the user for their age from the main program. It then uses a function to calculate the number of days they have been alive assuming 365 for every year (no leap years). Finally it prints the number of days from the main program.

Returning Values

- You cannot return values from a function
- When a bash function ends its return value is its status: zero for success, non-zero for failure.
- To return values:
  - you can set a global variable with the result, or
  - use command substitution, or
  - you can pass in the name of a variable to use as the result variable.

Use a global variable

```
myresult=“”

function myfunc()
{
   myresult='some value'
}

myfunc
echo $myresult
```

Use command substitution

```
function myfunc()
{
   local  myresult='some value'
   echo "$myresult"
}

result=$(myfunc)                # or result=`myfunc`
echo $result
```

Use eval

```
function myfunc()
{
   local  __result=$1
   local  myresult='some value'
   eval $__result="'$myresult'"
}

myfunc result
echo $result
```

The double underscore variable name convention is used to make it unlikely that the caller used the same variable name when they called the function.

The command eval and the $ symbol acks like a pointer referencing the thing that $1 was pointing to. This results in the value being assigned to the function call argument.

Exit Status

- Functions and Scripts terminate with an Exit Status
  - This is analogous to the exit status returned by a command.
  - The exit status may be explicitly specified by a **return** statement or the **exit** statement, otherwise it is the exit status of the last command in the function or script (0 if successful, and a non-zero error code if not).
  - This exit status may be used in the script by referencing it as $?.
  - This mechanism effectively permits script functions to have a "return value" similar to C functions but the integer value range is limited from 0 to 255.

Example with **return**

```
max2 ()                 # Returns larger of two numbers.
{                       # Note: numbers compared must be less than 254.
if [ -z "$2" ]; then
  return 255   # we are using the number 255 to mean ERROR
fi

if [ "$1" -eq "$2" ]; then
  return 254   # we are using the number 254 to mean EQUAL
else
  if [ "$1" -gt "$2" ];   then
    return $1
  else
    return $2
  fi
fi
}
max2 33 34

return_val=$?
```

Example with **exit**

```
max2 ()                 # Returns larger of two numbers.
{                       # Note: numbers compared must be less than 254.
if [ -z "$2" ]; then
  exit 255   # we are using the number 255 to mean ERROR
fi

if [ "$1" -eq "$2" ]; then
  return 254   # we are using the number 254 to mean EQUAL
else
  if [ "$1" -gt "$2" ];   then
    return $1
  else
    return $2
  fi
fi
}
max2 33 34

return_val=$?
```

The command **exit** not only return the value but it also terminates the script. The returned value is sent to the command-line or the external program that called your script.

A note about **exit**

The exit command is used to terminate a stript, therefore it is used, normally, in two ways:

- As the last instruction in your script returning the error status of your script to the command-line.
  - 0 for all is well. A positive integer number for error.
  - As the developer you determine what meaning is given to each positive integer error status value
- Anywhere within your script as part of an if-statement. The if-statement checks to see if something bad has happened requiring the termination of the script.

Part 3

Bash Developer Techniques

Techniques

- Shell memory to control script processing
- Backup and Archive scripts
- Development environment setup scripts

Shell memory to control script processing

- Methods of passing data to a script
  - Positional parameters via the command-line
  - Bash read command to prompt user for input
  - Pre-initialized within the shell memory, example:

```
Bash-prompt $ set x=10      OR      setenv x 10
Bash-prompt $ ./script
```

Within the script:

```
#! /bin/bash
if ($x .eq. 10)
   etc.
```

PATH & CLASSPATH

- PATH and CLASSPATH contain paths to folders that can be used by programs:
  - PATH is used by the shell when you type ./script or ./program
  - CLASSPATH is used by Java when you run a Java program

  (at the command-line type set to see the PATH or type echo $PATH to print it out)

- It is common for developers to setup the "run-time environment" by defining these kids of variables so that user's can define parameters without editing the code.

Shell memory to control script processing

For example:

- WORKINGDIR – the program's working directory
  - No matter what folder you are in it will default to that working directory
  - Eg:
    - Bash-prompt $ set WORKINGDIR="/home/jack/project1" OR
    - Bash-prompt $ setenv WORKINGDIR "/home/jack/project1"
    - Script: cd $WORKINGDIR
- DATAPATH – the path to important data
  - The user can change this information from the command-line without needing to open up the script to reprogram
  - Eg:
    - Bash-prompt $ set DATAPATH="/home/mary/project1/pictures" OR
    - Bash-prompt $ setenv DATAPATH "/home/mary/project1/pictures"
    - Script: files=$DATAPATH/dog.jpg

Backup and Archive Scripts

It is common that developers want to make backups on a regular basis but get tired of inputting all of the commands each time.

It is common for a developer to have:

- A master backup script
- A project specific backup script

Backup and Archive Scripts

Master backup script

- In the HOME directory
- A simple script listing all the files to backup with an optional variable destination

Project specific backup script

- Similar to the master backup script, but
- Contained in the project's top directory

Backup and Archive Scripts

```
#! /bin/bash
#default destination for backup
destination="../backup"
# check for environment variable
if (test –n "$BACKUPPATH"); then
    destination=$BACKUPPATH
fi
# check for command-line argument
if (test –n "$1"); then
    destination=$1
fi
cp file1 $destination
cp file2 $destination
#etc…
```

Commands to use:
- cp files to destination
- tar files and then mv
- zip files and then mv

Create a long list of everything you want to backup. Add to it and remove from it as things change.

Development environment setup scripts

Good practices:

- Standardizing your technique
- Automating best practices

Developers will often create a script to initiate the environment they want to work under.

- Or, the team leader will create a script that defines the standard working environment for the team members.
- This would include:
  - The project directory structure
  - Backup procedures
  - Start and end project procedures
  - Compiling procedures (we will see later)
  - Repository procedures (we will see later)

Development environment setup scripts

Six common scripts:

- NewProject - setup the OS environment for the project
- Backup - setup the way backups will happen
- EnterProject - the beginning of your work day
- ExitProject - the end of your work day
- Compile (we will see later)
- Repository (we will see later)

NewProject Script

```
#! /bin/bash
# Step 1: define the directory structure
 projectPath="projects"
 projectName="/project"
 project=$projectPath$projectName

 if (test –n $1); then
     project=$projectPath$1
 fi
 if (! test –d $projectPath); then
     mkdir $projectPath
 fi
 mkdir $project
```

```
mkdir $project/source
mkdir $project/backups
mkdir $project/docs
mkdir $project/archives
```

NewProject Script

```
# Step 2: create the extra scripts
# if they are stored somewhere then copy otherwise generate them
 if (test –r $projectPath/backup.sh); then
      cp $projectPath/backup.sh $project
      cp $projectPath/enterproject.sh $project        # assuming present
      cp $projectPath/exitproject.sh $project         # assuming present
      exit
 fi
# otherwise generate these scripts
 echo “#! /bin/bash” > backup.sh
 echo “cp *.sh $project/backup” >> backup.sh
 echo “exit 0” >> backup.sh
# create the other scripts as needed
```

Enter Project Script

Things you may want to standardize at the beginning of your workday on a project:

- Make sure you have the latest version of the source code from your team members
- Check to see if there are any special instructions from someone
- Define shell variables (if needed)
- Setup shell environment (like: alias and prompt)

Enter Project Script

Things you may want to standardize at the beginning of your workday on a project:

- Make sure you have the latest version of the source code from your team members
  - GIT commands (we will see later)
- Check to see if there are any special instructions from someone
  - chrome https://www.hotmail.com
  - cat $project/todayreadme.txt
- Define shell variables (if needed)
  - setenv workingdir "/home/jack/testing"
- Setup shell environment (like: alias and prompt)
  - alias ll ls –l
  - export PS1 "I am the greatest>"

The Alias and PS1 options

The alias command

- Lets you rename commands
- Syntax:
  - alias NEWNAME OLDEXPRESSION
- Example:
  - alias dir ls -l-a

The PS1 Shell Variable

- Lets you redefine the prompt. (Check it out, type SET)
- Different shell versions: PS1, PS2, prompt
- Syntax:
  - set prompt=STRING
  - export PS1 STRING

Exiting a project

Things you may want to standardize at the end of your workday:

- Make sure to backup everything
- Make sure to update the repository
- Clear all environment variables
- Change you working directory to HOME

Exiting a project

Things you may want to standardize at the end of your workday:

- Make sure to backup everything
  - backup.sh
- Make sure to update the repository
  - GIT (we will see later)
- Clear all environment variables
  - setenv workingdir “”
- Change your working directory to HOME
  - cd $HOME # if this is defined by your shell