Overview

Learners will be able to...

- **Display Bash Environmental Variables to the terminal and describe their output.**

- **Describe the difference between relative and absolute paths.**

- **Direct information between files using the following elements:**

  - **| Pipes**

  - **> Overwrite Redirection**

  - **>> Append Redirection**

  - **< Input Redirection**

  - **<< Here Documents**

- **Perform the following Bash Expansions**

  - **~ Tilde Expansion**

  - **{ } Brace Expansion**

  - **${ } Parameter Expansions**

  - **$(( )) Arithmetic Expansion**

- **Substitute characters and commands using Wild Card Characters, Character Sets, and Command Substitution**

- **Locate files using basic file globbing.**

info

Make Sure You Know

What do we expect the students to already know

Limitations

What might not be covered, or design decisions made by Codio

What is Bash?

Bash is a **shell**, a program that lets us interact with the computer. The name *"Bash"* is short for **B.ourne A.gain SH.ell**.

Many times, we'll type Bash commands one-by-one in the terminal. We can also create and run **bash scripts**, which are files that contain several lines of bash commands and can be run like any other program.

Bash scripts allow us to build logic, text formatting, and other helpful tools into files that we can distribute or use to automate command execution.

Let's make sure we have Bash installed on our system before we can use its tools. It's pre-installed in most Linux systems, but it's always good to check and be sure that it's available.

Copy and paste the following command in the terminal to see which Bash version is available to you:

```
bash --version
```

For our exercises, we'll be using Bash version 4.4.20.

```
GNU bash, version 4.4.20(1)-release (x86_64-pc-linux-gnu)
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>

This is free software; you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```
bash version

When a user opens a terminal, the default shell opens in the window. **The `$SHELL` environmental variable shows us exactly what shell is open.**

Copy the line into the terminal and press `return` to see which shell we're using:

```
echo $SHELL
```

You should see that we're using a Bash shell `/bin/bash`.

Checkpoint

Which of the following best describes the `$SHELL` variable?

Environment Variables

Environment variables are defined for us, by the system. They provide information about our scripting environment.

A Linux environment variable can be **global** or **local**.

Global

A **globally** scoped environment variable can be accessible from anywhere in that terminal's environment. It may be used in any script, application, or process connected to that terminal.

We can set new global commands using the syntax below.

```
export NAME=value
```

Let's create a global variable named `IRONMAN` and set the value of the variable to `Tony`.

```
export IRONMAN=Tony
```

We can recall the value of this variable using `echo` and the dollar sign `$`.

```
echo $IRONMAN
```

The `set` command will show us global and local environmental variables that are defined in the shell.

Click the button below to run the `set` command and see that `IRONMAN` is included in this list.

We can unset a global environment variable using the `unset` command.

```
unset IRONMAN
```

Check to see that the variable `IRONMAN` is, indeed, empty.

```
echo $IRONMAN
```

Local

Locally scoped environment variables are not accessible to every application or process running in the terminal. It can only be accessed by the terminal that defined it.

We can set a new local environment variable using the following syntax.

```
NAME=value
```

We can also unset a local variable.

```
NAME=''
```

We can use the `printenv` command to display all of the local environment variables that are currently defined.

```
printenv
```

Let's explore a few environmental variables below.

Variable	Description	Try It!
$PATH	Directories searched for commands	
$HOME	User's home directory	
$USER	User of the session	
$PWD	Path of the present working directory	
$HOME	User's home directory.	

▼ **A Few More Common Variables**

Here are a few more variables that you may encounter. Feel free to retrieve their values in the terminal using `echo $`.

- `$TERM` - type of command-line terminal being used

- `$EDITOR` - user's default editor

- `$HOSTNAME` - hostname of the machine
- `$SHELL` - the shell currently in use

Checkpoint

Match the environmental variable with it's description.

File Paths

A **file path** is a text string that represents a file or directory. We use paths to refer to files and directories on the file system.

Slashes / in a path separate directory names, leading to the directory or file.

There are two types of file paths: **absolute paths** and **relative paths**.

Absolute Path

An **absolute path** defines a location from the highest level, or root, of the file system, indicated by a slash /, to a certain directory or file. These are useful when we want to prevent confusion about the location of a file or directory.

Click the button below print the absolute path to our current working directory in the terminal with the `pwd` command.

We can also specify the absolute path to a file within the current directory. For example, the absolute path from the root directory to the file `champs.txt` would be:

```
/home/codio/workspace/champs.txt
```

info

Notice

Both of these paths begin with \, meaning they being at the root directory.

The user's **home directory** can be referred to with the tilde character ~. This allows us to define absolute file paths without knowing the exact name of the user's home directory.

```
~/workspace/champs.txt
```

Relative Filename

A **relative path** is a portion of a path that starts from the user's current working directory rather than the root directory.

A command will always execute in the current working directory unless we tell it to use files with a path leading to a different location.

These paths are considered **relative** because if the working directory changes, the path leading to a specific file location changes as well.

Relative paths do not start at the root directory, so they do not start with a slash `/`.

Consider our current working directory.

`/home/codio/workspace`

We can refer to a folder's location based on it's relationship to this directory. If we want to refer to the `ExampleFolder` directory from here, the following would be its relative path in relation to our current directory.

```
ExampleFolder
```

We can use `..` to refer to the current directory's **parent directory**, or the directory just above this one in the system hierarchy.

Let's change directories to the parent of our current working directory with the following command.

```
cd ..
```

Now display your new working directory.

```
pwd
```

If we were to change our working directory, the relative path to the same folder would change in relation to our new folder. The relative path to `ExampleFolder` from our new directory would be:

```
workspace/ExampleFolder
```

Checkpoint

Fill in the blanks to complete the statement below.

Built-In Bash Commands

Using a shell, like bash, involves running commands like:
- `ls`
- `rm`
- `grep`
- `arc`

These aren't bash commands. They're installed programs. Bash does, however, have certain **built-in commands** that we can use.

These built-ins can be bash-specific or share names with other commands on the system.

`echo`

The `echo` command displays text to the terminal. Use the button below to run this command with the text `Hey, all you people` in the terminal. Because `echo` ends a line with a new line character, it can be used to output text to the user without issue.

```
echo Hey, all you people!
```

`printf`

The `printf` built-in also displays text, but without the new line character at the end. Use the button below to run `printf` with the string input `"Won't you listen to me?"`.

```
printf "Won't you listen to me?"
```

`source`

`source` reads and executes the contents of a file (generally a set of commands) passed as an argument. The command takes the contents of the provided files and passes them to the command line interpreter as a text script.

When `filename` is performed, any arguments become positional parameters. The other settings stay unchanged. If `$PATH` does not include `FILENAME`, it searches the current directory for it.

The `source` command takes only one parameter, the file.

Check out the contents of the file `commands.txt`

```
cat commands.txt
```

Let's perform the commands within the file `commands.txt` using the `source` command.

```
source /home/codio/workspace/commands.txt
```

These examples barely scratch the surface of all of Bash's built-in commands. You can see a full list of built-ins on it's `man` pages.

```
man bash
```

Checkpoint

Fill in the blanks to complete the statement.

Challenge 1

In the file to the left, `challenge1.sh`, complete the following:

Write a command to display `Hello from user: _____`
1. - To fill in the blank, use the environmental variable containing the user's name.
2. **Write a command to display `Our home directory is: _____`**
1. - To fill in the blank, use the environmental variable containing the home directory.
3. **Write a command to display `The shell we're using is: _____`**
- To fill in the blank, use the environmental variable containing the name of the shell currently in use.
- 4. **Write a command to display `Our current working directory is: _____`**
- To fill in the blank, use the environmental variable containing the current working directory.
-
5. **Run the contents of the file `commands2.txt` using the `source` command and the file's absolute filename.**

1. Click the `Try It!` button to run your script in the terminal and check your output. It should look similar to the output below.
2.

```
Hello from user: codio
Our home directory is: /home/codio
The shell we're using is: /bin/bash
Our current working directory is: /home/codio/workspace

Congratulations!! You've run all of the commands in the file!
```
Challenge Expected Output

When you are finished writing your script, click the `Check It!` button to check your work.

Redirection

##

Redirection and **Piping** are two ways we can control command input and output in Bash.

Redirection sends streams of information either to, or from files.

Output Redirection >

Output Redirection > allows us to send the results of a command to a file. It sends the standard output `stdout` by default. If we use > to redirect to an existing file, the contents of the file will be overwritten.

.guides/img/outputRedirect

Let's take the results of the `pwd` command and redirect it to the file `directory.txt` using the > character.

```
pwd > directory.txt
```

Use `cat` to display the contents of `directory.txt`.

```
cat directory.txt
```

Output Redirection (Append) >>

We can redirect information onto the end of an existing file by using >>. This appends the results of a command to the end of the file's contents.

Let's append the results of `ls` to the end of `directory.txt`.

```
ls >> directory.txt
```

Now, let's display the new contents of `directory.txt` to display the appended information.

```
cat directory.txt
```

Redirection works using 3 data streams available in Bash:
- **Standard Input** `stdin` - Receives text data as input.
- **Standard Output** `stdout` - Stores the text output of a command.
- **Standard Error** `stderr` - Records the error message when a command fails.

These data streams are given unique **file descriptor numbers**. We can access the data by referring to its file descriptor number as described by the table below.

Data Stream	FD Number
stdin	0
stdout	1
stderr	2

Let's try to list a non-existent directory in the terminal.

```
ls /fakedir
```

Here, we get an error message:

```
ls: cannot access '/fakedir': No such file or directory
```

Let's redirect the following:

- The standard output, using 1, to a file `output.txt`, and
- The standard output, using 2, to a file `error.txt`

```
ls /fakedir 1>output.txt 2>error.txt
```

Now, we can check the contents of each file.

```
cat output.txt
cat error.txt
```

Input Redirection <

Input redirection < can also be used to extract data from a file and seem as if it were entered as standard input.

.guides/img/inputRedirect

In this example, we'll use input redirection to use the output of `champs.txt` as input to the `cat` command.

```
cat < champs.txt
```

Here Document <<

A **here document** << is a variation of redirection that lets us input text freely until the command encounters a specified string **delimiter** to represent the end of input.

When using a **here document**, make sure that the specified string is unique so it isn't confused for any other line in the input.

We'll specify `ThisIsTheEnd` as our unique string and enter a series of lines as input.

```
cat << ThisIsTheEnd
Is this the real life?
Is this just fantasy?
Caught in a landslide,
No escape from reality
ThisIsTheEnd
```

The command above displays the lines of input up to the termination string.

Checkpoint

Complete the following redirections in the terminal.

1. Redirect the results of `echo "Disk Space Usage Report"` into a file called `diskspace.txt`.
2. Append the results of the `df` command onto `diskspace.txt`
3. Redirect the output of `diskspace.txt` into the terminal display using `cat`
4. Redirect the following lines into the terminal display using `cat` and the delimiter `EndOfReport`.
   ○ `I have reported the usage of disk space in KBs.`
   ○ `If you'd like, I can also run the report in megabytes`
   ○ `It would be as easy as using the df -m command`

Piping

Piping

Piping connects the output of two processes by sending the first process' output to the second process as input, without being displayed on the screen.

As long as the output of one command can be used as input for another, **pipes** (`|`) allow you to chain commands. They are used in Bash scripting to create custom flows.

.guides/img/pipe

We can use the `cat` command to look at the contents of a file `champs.txt` that holds every winning NFL Super Bowl team in history.

```
cat champs.txt
```

This is a lot of information. We can pipe (`|`) the output of the previous command into the less command to display this information page by page.

```
cat champs.txt | less
```

You can use the UP or DOWN arrows to move up and down the file line by line.

To move through the file page by page:
- Use `fn + UP` or `fn + DOWN` if you're on a MAC.
- Use `PgUp` or `PgDown` if you're on a Windows machine.

Press the `q` key to quit.

We can use the `wc` command to count how many lines are in this file by piping the output of a `cat` command to the input of the `wc` command.

```
cat champs.txt | wc
```

You can string many pipes together, as long as the output of a command makes sense as input for the next command.

Piping and redirection seem similar, but we use them differently. Mainly, piping sends information from one **process** to another, while redirection sends information to and from **files**.

For reference, you can find the different characters we use for redirection and piping below.

- > - **Output redirection (overwrite)**
- >> - **Output redirection (append)**
- < - **Input redirection**
- << - **Here document**
- | - **Pipe**

Checkpoint

Fill in the blank to complete the statement.

Challenge 2

In the file to the left, `challenge2.sh`, complete the following:

1. **Write a command to redirect the message `Reporting for user: _____` into a file called `introduction.txt`.**

   - To fill in the blank, use the environmental variable containing the user's name.

2. **Write a command to append the message `The home directory is: _____` into the file `introduction.txt`**

   - To fill in the blank, use the environmental variable containing the home directory.

3. **Write a command to list the contents of the imaginary directory `/notReally` and:**

   - Redirect the standard output of the command to a file called `output.txt`
   - Redirect the standard error to a file called `error.txt`.

4. **Write a command to:**

   - Redirect the contents of a file called `champs.txt` to be used as input into the command `sort`
   - Redirect the results of the input redirection into a file called `alpha-winners.txt`

5. **Using the `cat` command, pipe the contents of the file `alpha-winners.txt` into the `less` command**

6. Click the `Try It!` button to run your script in the terminal and check your output. It should look similar to the output below.

When you are finished writing your script, click the `Check It!` button to check your work.

Tilde Expansion

It's common to need to use unknown values when we work with Bash. For example, we might need to use the path to a user's home folder, the user's own data, or the result of a calculation.

In Bash, you can use **expansions** and **substitutions** to show these values. When they are run, they are read and replaced with a value or group of values.

Tilde Expansion ~

The **tilde expansion** is used to represent a user's home directory. Type the following in the terminal to display your home directory.

```
echo ~
```

We can see that our home directory is:

```
/home/codio
```

We can use ~ and the name of a user if we want to retrieve the home directory of a specific user.

```
echo ~codio
```

Tilde expansion lets us work in a user's home directory even if we don't know their user name.

We can use **tilde** + **plus** ~+ to represent the PWD, or to present the current directory that we're working in.

We can use **tilde** + **dash** ~- to represent the OLDPWD, or the previous directory you were in if you've changed directories.

Checkpoint

We can use expansions and substitutions to...

Brace Expansion

Brace expansion `{}` lets us substitute elements from a list of values separated by commas, or ranges of numbers or letters separated by two periods.

Brace expansion is commonly used to preserve part of a path but replace a small portion of it.

```
echo /tmp/{one,two,three}/file.txt
```

We can use it to present a set of values to use in the same part of a string.

```
echo r{i,o,u}se
```

We can also use brace expansion to create a sequence of numbers or letters.

```
echo {1..10}
```

```
echo {a..z}
```

We can also add intervals to sequences with a third argument in our brace expansion. The following command displays numbers $0 - 20$ in intervals of 2.

```
echo {0..20..2}
```

We can create a few files neatly labeled filed with sequential numbering and lettering.

```
touch file_{1..3}{a..c}
```

We can also remove these files sequentially.

```
rm file_{1..3}{a..c}
```

Brace expansion can also be used to work with lists of things.

```
echo {earth,wind,fire}_{1..3}
```

Checkpoint

Complete the following in the terminal.

1. **Use the `touch` command and brace expansion to create files named `image_N.extension` where:**

   - `N` is a number ranging from 1 to 5
   - `extension` is a set of file types including:
     - `.gif`
     - `.png`, and
     - `.jpeg`
   - File names should look similar to:
     - `image_1.gif`
     - `image_2.jpg`
     - `image_1.png`

2. Use the `rm` command to remove every file you just created with a single command.

Parameter Expansion

Parameter expansion allows us to retrieve and modify stored values. The symbol for this is ${...}.

The most common usage of parameter expansion is setting and retrieving values, like with variables in scripting.

To do this, we:
- set a parameter to a value, then
- use the dollar sign with the parameter name to get it later.

In parameter expansion, braces are often used to indicate which parameter is being used and avoid the shell confusing it with neighboring words or characters.

Let's set a parameter in the terminal:

```
book="Where The Sidewalk Ends"
```

We can retrieve the value of the parameter with a dollar sign $ and the name of the parameter.

```
echo $book
```

Parameter expansion can be used to change the stored value when it is used by using only part of it or replacing parts of it before use.

For example, `echo ${book:6}` returns the parameter's value starting with the sixth character in the string.

```
echo ${book:6}
```

Or `echo ${book:6:3}` will retrieve the value starting at the sixth character and going for three characters.

The first character in the string is in position 0.

Parameter expansion can be used in many ways. To explore more, check out the bash man pages by typing `man bash` in the terminal. Type `/` and press `return` to search the "**Parameter Expansion**" header.

Pattern Substitution

Pattern substitution is a parameter expansion feature that lets us obtain a value and change it depending on a pattern. This is commonly used for replacing a word or character using the pattern below.

`echo ${variableName/oldWord/newWord}`

Below is a complete pattern substitution example.

```
echo ${book/Ends/Begins}
```

In Pattern Substitution, one slash after the parameter replaces the first instance of a match, and two slashes replaces them all.

Let's replace all e's with ampersands & using two forward slashes `//`.

```
echo ${book//e/&}
```

Now, let's do the same thing with only one slash.

```
echo ${book/e/&}
```

Be sure to use the braces for most parameter expansions. If we left them off, the shell will read the part after the parameter name as characters and show them instead of using them to achieve what we wanted.

```
echo $book/e/$
```

Checkpoint

Complete the following in the terminal.

1. **Set a parameter called `title` and assign it the value `Avatar: The Last Airbender`.**

2. **Use `echo` and parameter expansion to replace all of the a's with an underscore `_`.**

Arithmetic Expansion

Bash can perform arithmetic expansion and then use the result in a command. This is represented by a dollar sign and two parentheses containing a mathematical expression.

Bash previously used a dollar sign surrounded by single brackets, but this is now obsolete, though may be present in older scripts.

Let's do some calculations in the terminal.

Let's start by adding four and five.

```
echo $(( 4 + 5 ))
```

Subtract five from ten.

```
echo $(( 10 - 5 ))
```

Then, multiply six by three.

```
echo $(( 6 * 3 ))
```

Finally, divide nine by ten.

```
echo $(( 9 / 10 ))
```

Integers Only!

Bash can only work with integers. $9 / 10 = 0.9$, which is less than 1. Since Bash can only handle integers, this operation gives us a result of zero.

Checkpoint

Use the arithmetic expansion to perform the following calculations in the terminal.

1. 221 plus 858 multiplied by 24.

2. 343 minus 162 divided by 47.

3. 447 multiplied by 255.

4. 890 divided by 161.

Challenge 3

In the file to the left, `challenge3.sh`, complete the following:

1. **Write a command to display the current user's working directory using tilde expansion.**

2. **Using brace expansion, write a command to display numbers** 1 **through** 100 **in intervals of** 5

3. **Using the `mkdir` command and brace expansion to create directories named `TYPE_filesN` where:**

   ○ `TYPE` is an image file type including:
     - `png`
     - `jpg`, and
     - `pdf`
   ○ `N` is a number ranging from 1 to 3.

4. **Use the `touch` command and brace expansion to create files inside each directory named `image_N` where:**

   ○ `N` is a number ranging from 1 to 5

   ○ File names should look similar to:

     - m

5. Use the `rm` command to remove every file you just created with a single command.

6. Now use the `rmdir` command to remove every directory we've created with a single command.

7. **Write a command to display `6 times 3 is equal to: _____`**

   ○ To fill in the blank, use arithmetic expansion to perform the calculation and display the correct answer.

8. **Set a parameter called `message` and assign it the value `impossible work!`.**

9. **Using parameter expansion, write a command to display the value of `message` starting from the character in position `2`.**

10. **Click the `Try It!` button to run your script in the terminal and check your output. It should look similar to the output below.**

When you are finished writing your script, click the `Check It!` button to check your work.

Wild Card Characters

In Linux, a wild card is a symbol or group of symbols that replace other characters. It can replace any character in a string. These are helpful for finding texts that are similar in some way, but not the same.

Asterisk `*`: used to match any number of characters (zero or more)

```
ls *.txt
```

```
ls d*
```

Question Mark `?`: used to find exactly one character of any type. If we want to find two characters, we need to use two question marks `??`

Square Brackets `[ ]`: used to match a set of characters inside.

Exclamation Point `!` excludes characters from a list defined inside square brackets.

Named Character Classes

Named character classes are used to print values that have been given a name. Their value is determined by the `LC_CTYPE` locale. A few examples include:
- `[:alnum:]`: Prints all files that include alphabets and numbers. Both lower and uppercase letters are taken into account.
- `[:alpha:]`: Prints all files that exclusively contain alphabets. Both lower and uppercase letters are taken into account.
- `[:digit:]`: prints all files that include digits.
- `[:lower:]`: prints all files with lower-case letters.
- `[:punct:]`: prints all files that include punctuation characters, including:
- `! " # $ % &' () * +, -. /: ; = >? @ [] '|.`
- `[:space:]`: prints all files that contain space characters.
- `[:upper:]`: prints all files with lower-case letters.

Checkpoint

Select all of the options that would be returned using the following wildcard command:

```
ls i*.p*
```

Command Substitution

Command substitution lets us use the output of one command inside another. This is represented as a dollar sign followed by a command enclosed in parentheses `$(...)`.

Bash uses a sub shell to run the specified command and the output is returned to the current command.

It's typically used with string manipulation tools to extract data from a command's output that has to be passed back up to the parent command.

.guides/img/commandSub

Let's explore command substitution:

We could use `uname -r` to find the kernel release version and use command substitution in an echo statement to display this information to the user.

```
echo "The kernel is $(uname -r)."
```

This is typically used in scripts to retrieve the user's installed version of something.

If we have a complex set of commands that rely on the results of other commands, we can nest command substitutes within one another.

```
echo "Today is $(date)."
```

```
echo "Computer name is $(hostname)."
```

Checkpoint

Command substitution works by...

File Globbing

File globbing detects wild card patterns and expands them into a file path.

Globs and regular expressions are very similar. Globs specifically match file names, where regular expressions match text.

Let's explore a few patterns. Click the button below to begin.

Asterisk * matches any number of any characters in a filename.

```
ls *.txt
```

```
ls c*.txt
```

Question Mark ? is used to locate one single character of any type.

```
ls doc1?.txt
```

```
ls doc??.txt
```

[] matches to one character within a range.

```
ls doc[1-5].png
```

```
ls doc[1-5][a-e].png
```

A caret ^ along with square brackets describes a globbing pattern with more detail.

We can also match a character against certain character classes.

```
ls doc[[:digit:]]a.txt
```

```
ls doc[[:digit:]][[:lower:]].txt
```

We use wildcards with file globbing to determine which types of a single character can be present in a filename.

If we want to match against whole words, we would use brace expansion.

```
ls {*.txt,*.png}
```

When you're finished exploring, click the button below to clean up our file tree.

Checkpoint

Fill in the blanks to complete the statement.

Challenge 4

In the file to the left, `challenge4.sh`, complete the following:

1. **Write a command to display `The current network host name is:`**
   ____

   - Fill in the blank using command substitution and the `hostname` command to return the network host name.

2. **Write a command using wildcard characters to list all of the `.txt` files that begin with `c`.**

3. **Write a command using a named character class to list all of the files that contain a number.**

4. **Write a command using the `?` wildcard character to list all of the documents with filenames similar to `file2_.txt`.**

   - The blank may contain only one of any type of character.

5. **Click the `Try It!` button to run your script in the terminal and check your output. It should look similar to the output below.**

When you are finished writing your script, click the `Check It!` button to check your work.