Subshells

In this lesson, we'll go over why subshells are useful, how they are created, and contrasting them with group commands.

WE'LL COVER THE FOLLOWING

The concept of **subshells** is not a complicated one, but can lead to a little confusion at times, and occasionally is very useful.

How Important is this Lesson?

As with process substitution, this is another concept I came to later in my bash career. It comes in handy fairly often, and an understanding of it helps deepen your bash wisdom.

```
VAR1='the original variable'
```

Type the above code into the terminal in this lesson.

Terminal

Now create a subshell:

```
(
```

Type the above code into the terminal in this lesson.

You'll notice the prompt has changed. Now try and echo something:

```
echo Inside the subshell
```

Type the above code into the terminal in this lesson.

It's not been run. This is because the subshell's instructions aren't run until the parenthesis has been closed. Next we'll try and echo the variable we created outside.

```
echo ${VAR1}
```

Type the above code into the terminal in this lesson.

Then update that variable to another value, and echo it again:

```
VAR1='the updated variable'
echo ${VAR1}
```

Type the above code into the terminal in this lesson.

And finally create another variable, before closing the subshell out:

```
VAR2='the second variable'
)
```

Type the above code into the terminal in this lesson.

So a subshell is a shell that inherits variables from the parent shell, and whose running is deferred until the parentheses are closed.

It will pay to think about the output and review the subshell commands to grasp what you just saw. Play with the commands and experiment until you're comfortable with what a subshell can and can't do.

Subshells and Scope

What happened to the variable you updated inside the subshell?

```
echo ${VAR1}
```

Type the above code into the terminal in this lesson.

Now you now know that variables can mask their parent shell's value within the subshell. This is very handy to know if you want to do something in a slightly modified environment, as you do not have to manage variables' previous values.

What happens if we try `export`ing the variable? Will that 'export' it to the parent shell?

```
( export VAR1='the first variable exported' )
echo ${VAR1}
```

Type the above code into the terminal in this lesson.

You will see that it does not. `export` does not 'export' the variable outside the enclosing parentheses.

The `export` builtin

The `export` command can cause a lot of confusion here. Let's experiment with it here to show how it works.

> Note: The next code snippet puts quotes around the `END` delimiter. We will cover why in the next part of the course.

```
cat > echoes.sh << 'END'
#!/bin/bash
echo $EXPORTED
echo $NOTEXPORTED
END
chmod +x echoes.sh
```

Type the above code into the terminal in this lesson.

The above code creates the `echoes.sh` file and makes the file executable.

Now run this. What do you think will happen?

Now run this. What do you think will happen?

```
export EXPORTED='exported'
NOTEXPORTED='exported'
./echoes.sh
```

Type the above code into the terminal in this lesson.

The first variable (`EXPORTED` on **line 1**) is seen in the outputs of the command run in the same shell `echoes.sh` because it is preceded by the `export` builtin.

The second variables (`NOTEXPORTED` on **line 2**) is not `export`ed, so is not seen by the command run from the same shell.

If the above doesn't make sense to you, you might want to revisit Part I, where `export` was covered in 'Variables'.

Subshells and Redirection

One often useful property of subshells is the fact that the code is treated as a single unit. You can therefore redirect output from a set of commands wholesale.

You might write code that looks like this:

```
echo Some output, ls to follow >> logfile
ls >> logfile
```

Type the above code into the terminal in this lesson.

which is fine for a couple of lines, but what if you have hundreds of lines like this? It leads to quite inelegant code. Instead, you can write:

```
(
echo Some output, ls to follow
ls
) >> logfile
```

Type the above code into the terminal in this lesson.

which is much neater and easier to manage.

`()` vs `{}`

One source of confusion is the difference between these two ways of grouping commands.
First, try replacing the `(` and `)` in the above listing with curlies (`{` and `}`) and see what happens.

Then, try these:

```
( echo output ) >> logfile
{ echo output } >> logfile
```

Type the above code into the terminal in this lesson.

Hmmm. Hit `CTRL+c` to get out.

The curlies need a semicolon to indicate the end.

```
{ echo output ; } >> logfile
```

Type the above code into the terminal in this lesson.

This is because the curlies are a *grouping command* and need an indication of when the command has been completed. No subshell is created. The environment, variables, current working directory, indeed the entire context are the same as the surrounding code.

Subshells and Working Directories

Another useful feature of subshells is that if you change folder, then that folder change only applies within the subshell. When you return, you go back to where you were.

Let's see this in action. First show where you are:

```
pwd
```

Type the above code into the terminal in this lesson.

Then, in a subshell, create a folder, move into it, and show you have moved with another call to pwd.

```
( mkdir lbthw_subshells_2
```

```
cd lbthw_subshells_2
pwd )
```

Type the above code into the terminal in this lesson.

Now the subshell has run, and without doing anything other than `cd`ing to a new folder, we are returned to the folder we were in before the subshell started:

```
pwd
```

Type the above code into the terminal in this lesson.

This is a very useful feature of subshells when you're scripting, to save repeated `cd` and `cd -` commands.

Subshells Quiz

Q What are the benefits of subshells?

What You Learned

- How to create a *subshell*
- The difference between `()` and `{}`
- How variable scoping works in *subshells* and *grouping* commands
- Redirection in *subshells* and groups

What Next?

Next you will look at a common challenge when dealing with files and for loops: the **internal field separator**.

Exercises

1) Show that the grouping curly braces (`{` and `}`) affect the shell's working directory outside the enclosed code, while a subshell does not.

2) Work out what the `BASH_SUBSHELL` variable does.