Editing the Scripts We Already Have

Before we start writing new scripts, We'll take a look at some scripts we already have. These scripts were put into our home directory when our account was created, and are used to configure the behavior of our sessions on the computer. We can edit these scripts to change things.

In this lesson, we will look at a couple of these scripts and learn a few important new concepts about the shell.

During our shell session, the system is holding a number of facts about the world in its memory. This information is called the *environment*. The environment contains such things as our path, our user name, and much more. We can examine a complete list of what is in the environment with the **`set`** command.

Two types of commands are often contained in the environment. They are *aliases* and *shell functions*.

How is the Environment Established?

When we log on to the system, the bash program starts, and reads a series of configuration scripts called *startup files*. These define the default environment shared by all users. This is followed by more startup files in our home directory that define our personal environment. The exact sequence depends on the type of shell session being started. There are two kinds: a *login shell session* and a *non-login shell session*. A login shell session is one in which we are prompted for our user name and password; when we start a virtual console session, for example. A non-login shell session typically occurs when we launch a terminal session in the GUI.

Login shells read one or more startup files as shown below:

File	Contents
`/etc/profile`	A global configuration script that applies to all users.
`~/.bash_profile`	A user's personal startup file. Can be used to extend or override settings in the global configuration script.
`~/.bash_login`	If `~/.bash_profile` is not found, bash attempts to read this script.
`~/.profile`	If neither `~/.bash_profile` nor `~/.bash_login` is found, bash attempts to read this file. This is the default in

	Debian-based distributions, such as Ubuntu.

Non-login shell sessions read the following startup files:

File	Contents
/etc/bash.bashrc	A global configuration script that applies to all users.
~/.bashrc	A user's personal startup file. Can be used to extend or override settings in the global configuration script.

In addition to reading the startup files above, non-login shells also inherit the environment from their parent process, usually a login shell.

Take a look at your system and see which of these startup files you have. Remember— since most of the file names listed above start with a period (meaning that they are hidden), you will need to use the "-a" option when using ls.

The `~/.bashrc` file is probably the most important startup file from the ordinary user's point of view, since it is almost always read. Non-login shells read it by default and most startup files for login shells are written in such a way as to read the `~/.bashrc` file as well.

If we take a look inside a typical `.bash_profile` (this one taken from a CentOS system), it looks something like this:

```
# .bash_profile
# Get the aliases and functions
if [ -f ~/.bashrc ]; then
  . ~/.bashrc
fi

# User specific environment and startup programs
PATH=$PATH:$HOME/bin
export PATH
```

Lines that begin with a "#" are comments and are not read by the shell. These are there for human readability. The first interesting thing occurs on the fourth line, with the following code:

```
if [ -f ~/.bashrc ]; then
  . ~/.bashrc
fi
```

This is called an *if compound command*, which we will cover fully in a later lesson, but for now we will translate:

If the file "~/.bashrc" exists, then read the "~/.bashrc" file.

We can see that this bit of code is how a login shell gets the contents of `.bashrc`. The next thing in our startup file does is set the PATH variable to add

the ~/bin directory to the path.

Lastly, we have:

```
export PATH
```

The **export** command tells the shell to make the contents of the PATH variable available to child processes of this shell.

Aliases

An alias is an easy way to create a new command which acts as an abbreviation for a longer one. It has the following syntax:

```
alias name=value
```

where *name* is the name of the new command and *value* is the text to be executed whenever *name* is entered on the command line.

Let's create an alias called "l" and make it an abbreviation for the command "ls -l". We'll move to our home directory and using our favorite text editor, open the file .bashrc and add this line to the end of the file:

```
alias l='ls -l'
```

By adding the **alias** command to the file, we have created a new command called "l" which will perform "ls -l". To try out our new command, close the terminal session and start a new one. This will reload the .bashrc file. Using this technique, we can create any number of custom commands for ourselves. Here is another one to try:

```
alias today='date +"%A, %B %-d, %Y"'
```

This alias creates a new command called "today" that will display today's date with nice formatting.

By the way, the **alias** command is just another shell builtin. We can create our aliases directly at the command prompt; however they will only remain in effect during the current shell session. For example:

```
[me@linuxbox me]$ alias l='ls -l'
```

Shell Functions

Aliases are good for very simple commands, but to create something more complex, we need *shell functions*. Shell functions can be thought of as "scripts within scripts" or little sub-scripts. Let's try one. Open .bashrc with our text editor again and replace the alias for "today" with the following:

```
today() {
    echo -n "Today's date is: "
```

```
    echo -n "Today's date is: "
    date +"%A, %B %-d, %Y"
}
```

Believe it or not, `()` is a shell builtin too, and as with **alias**, we can enter shell functions directly at the command prompt.

```
[me@linuxbox me]$ today() {
> echo -n "Today's date is: "
> date +"%A, %B %-d, %Y"
> }
[me@linuxbox me]$
```

However, like **alias**, shell functions defined directly on the command line only last as long as the current shell session.
