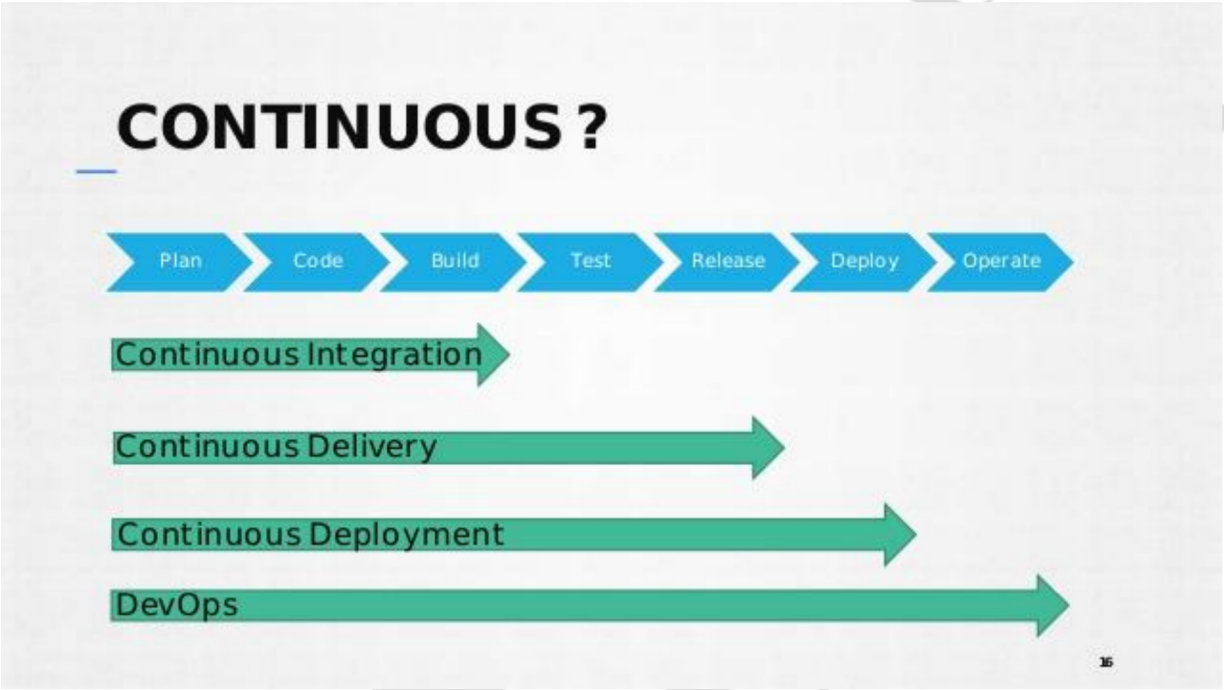
What is Continuous Deployment?

If the approval is manual process then code delivery is Continuous Delivery but if the approval process becomes automated then after staging, the code change is done directly to Production systems. This is called as Continuous Deployment.



7. DevOps and Software **Development Life Cycle**

The DevOps Lifecycle Looks Like This:

1. Check in code
2. Pull code changes for build
3. Run tests (continuous integration server to generate builds & arrange releases): Test individual models, run integration tests, and run user acceptance tests.
4. Store artefacts and build repository (repository for storing artefacts, results & releases)
5. Deploy and release (release automation product to deploy apps)
6. Configure environment
7. Update databases
8. Update apps
9. Push to users – who receive tested app updates frequently and without interruption

10. Application & Network Performance Monitoring (preventive safeguard)
11. Rinse and repeat

The above process is also called a Code Delivery Pipeline.

8. Tools for DevOps Lifecycle

We have discussed earlier that everything starts from communication and collaboration between Dev and Ops. Once we understand the culture and process of the project/product we can start working with everyone in designing Code Delivery Pipeline. We must decide what automation tools to use to create entire pipeline. We need to decide where our infrastructure would be hosted? On the cloud, virtual machines or physical machines?

In today's world we have lot of automation tools, but first we need to understand their categories what tools is used for what purpose? If we don't understand that then we won't be able to decide where to use them in our code delivery pipeline.

Version Control Systems: Is used to store the source code, a central place to keep all the code and tracks its version.

For Example:

- Git
- SVN
- Mercurial
- TFS

Build tools: Build process is where we take the raw source code, test it and build it into a software. This process is automated by build tools.

For Example

- Maven
- ANT
- MSBuild
- Gradle
- NANT

Continuous Integration Tools:

For example:

- Jenkins
- Circle CI
- Hudson
- Bamboo
- Teamcity

Configuration Management tools: Also known as automation tools, can be used to automate system related tasks like software installation, service setup, file push/pull etc. Also used to automate cloud and virtual infrastructure.

For example:

- Ansible
- Chef
- Puppet
- Saltstack

Cloud computing: Well this is not any tool but a service accessed by users through internet. A service that provides us with compute resource to create virtual servers, virtual storage, networks etc. There are few providers in the market who gives us public cloud computing services.

For example:

- AWS
- Azure
- Google Cloud
- Rackspace

Monitoring tools: Is used to monitor our infrastructure and application health. It sends us notifications and reports through email or other means.

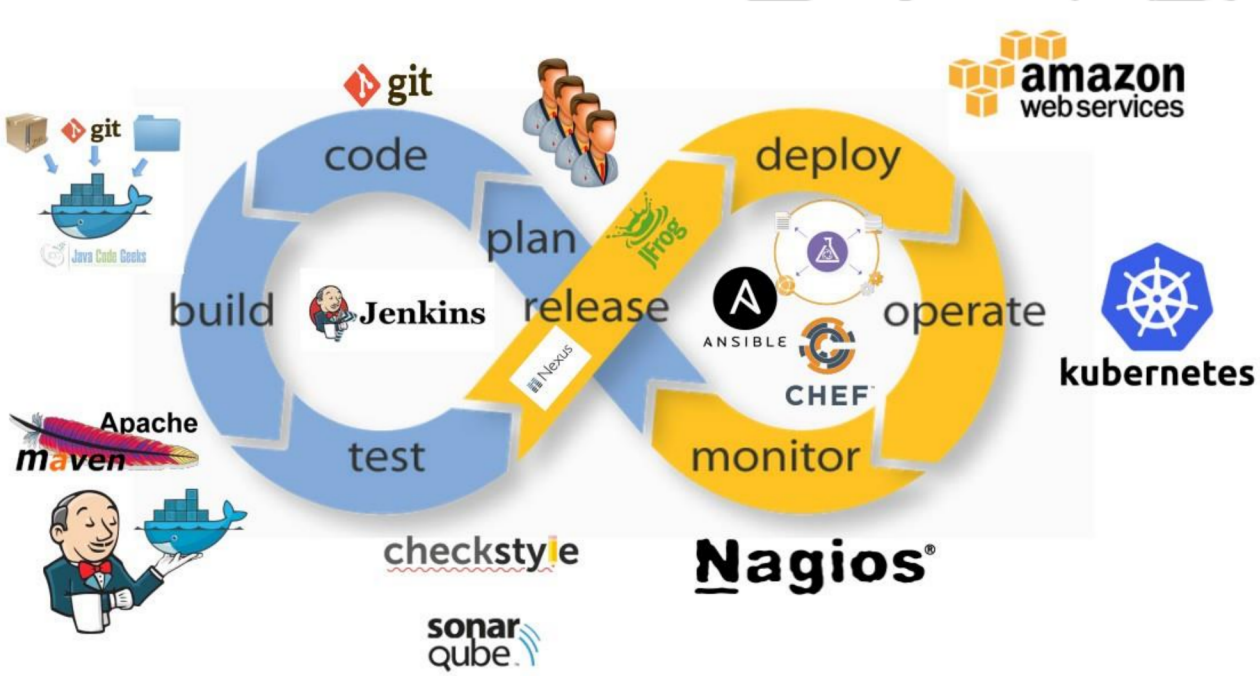
For example:

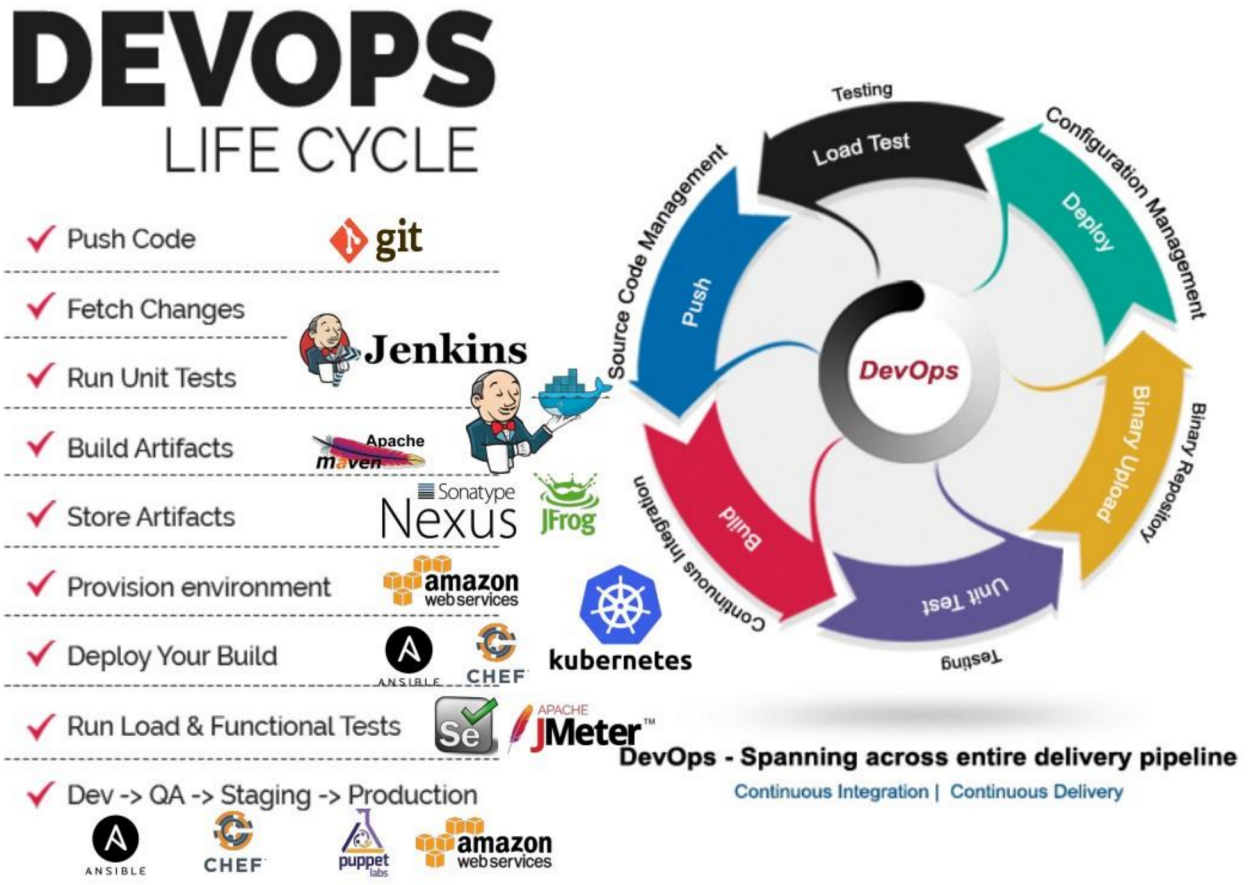
- Nagios
- Sensu
- Icinga
- Zenoss
- Monit

Containers & Microservices: Well to be very frank this cannot be described to you right now. Its described in detail in separate chapter. We need to have lot of Infra & Development knowledge to understand this category of tool.

- Docker
- RKT
- Kubernetes

9. DevOps Lifecycle with images of devops tools.





Summary:

- ✓ DevOps cannot be defined in one sentence. DevOps is the culture and also the implementation of automation tools. It depends from which area you are defining it.
- ✓ Majority of the Software development is happening with Agile model and that churns out code changes incrementally and frequently.
- ✓ Operations does not gel well with the Agile team as both have differences in their principles.
- ✓ Agile team wants quick change, Ops wants to keep system stable by not making frequent changes.
- ✓ DevOps helps in creating communication, collaboration and integration between Dev and Ops and culture, practices and tools level.
- ✓ DevOps Engineer must understand DevOps lifecycle and implement right tool of automation at right place.

Learning automation at every level in the lifecycle is highly important if you want to make a carrier in DevOps domain. You should understand Infrastructure, Development & Automation.

Later in the book we will dig more in detail into tools and learn them. We will also understand CI & CD from tools point of view.

II. Bash Scripting

1.Introduction

This tutorial will give you a solid platform in how to create bash scripts and automate day to day system admin tasks. Certainly, everything cannot be covered in this chapter but you will be equipped with right amount of knowledge to make your own scripts and excel in it if you put in your own efforts.

Bash scripting is used by many of the system admins and DevOps geeks to get things done quickly and efficiently. There are so many automation tools in the market like Ansible, Puppet, Chef etc. Which are way more sophisticated but sometimes to get things done quickly in Linux systems we use Bash scripts. Also, scripting will make you understand what automation means and then you can quickly grasp the features that is used in Configuration Management tools like Ansible or puppet.

What are scripts?

A Bash script is a plain text file which contains a series of commands. These commands are a mixture of commands we would normally type ourselves on the command line (such as **ls** or **cp** for example) and commands we could type on the command line but generally wouldn't (you'll discover these over the next few pages). A crucial point to remember though is:

Anything you can run normally on the command line can be put into a script and it will do exactly the same thing. Similarly, anything you can put into a script can also be run normally on the command line and it will do exactly the same thing.

First Script

As we discussed earlier that script is a normal text file with commands in it.
We will open a file vi editor and add some commands in it.
It is convention to give files that are Bash scripts an extension of **.sh** (print.sh for example)

```
$ vi print.sh
```

```
•#!/bin/bash
•# A sample Bash script
•echo Hello World!
```

Explanation:

Line 1 - #! is called as the SHEBANG character, it tells the script to interpret the rest of the lines with an Interpreter /bin/bash. So, if we change that to /usr/bin/python then it tells the script to use python interpreter.

Line 2- This is a comment. Anything after # is not executed. It is for our reference only. Comments is for us so or anybody who reads this script will have some reference.

Line 3- Is the command echo which will print a message to the screen. You can type this command yourself on the command line and it will behave exactly the same.

Running or Executing a script?

Running a Bash script is fairly easy. Sometimes you will hear people saying execute the script, both means same thing. Before we can execute a script, it must have the execute permission set. If you forget to grant this permission before running the script you'll just get an error message “Permission denied”. Whenever you create a file in Linux system by default it will not have an execute permission, this is for security reasons. You make your script executable and then you can run it.

```
imran@DevOps:.../bash$ ./print.sh
bash: ./print.sh: Permission denied
imran@DevOps:.../bash$ ls -l
total 4
-rw-rw-r-- 1 imran imran 53 Oct 21 17:33 print.sh
imran@DevOps:.../bash$ chmod 755 print.sh
imran@DevOps:.../bash$ ls -l
total 4
-rwxr-xr-x 1 imran imran 53 Oct 21 17:33 print.sh
imran@DevOps:.../bash$ ./print.sh
Hello World!
```

Without giving execute permission also we can run the script but then we provide a shell and ask it to run all the command in the script on that shell.

```
imran@DevOps:.../bash$ bash print.sh
Hello World!
```

2.Variables

Temporary stores of information in memory.

How do they Work?

A variable is a temporary store for a piece of information. There are two actions we may perform for variables:

Setting a value for a variable.

Reading or using the value for a variable.

To assign a variable we use = sign, VariableName=Value

To read/access the value of variable we use $VariableName

```
imran@DevOps:.../bash$ VAR1=123
imran@DevOps:.../bash$ echo $VAR1
123
```

Command line arguments

When we run a program on the command line you would be familiar with supplying arguments after it to control its behaviour.

For instance we could run the command **ls -l /tmp**. **-l** and **/tmp** are both command line arguments to the command **ls**.

We can do similar with our bash scripts. To do this we use the variables **$1** to represent the first command line argument, **$2** to represent the second command line argument and so on. These are automatically set by the system when we run our script so all we need to do is refer to them.

Let's look at an example.

copyscript.sh

```
1. #!/bin/bash
2. # A simple copy script
3. cp $1 $2
4. # Let's verify the copy worked
5. echo Details for $2
6. ls -lh $2
```

```
imran@DevOps:.../testcopy$ mkdir dir1 dir2
imran@DevOps:.../testcopy$ touch dir1/tesla
imran@DevOps:.../testcopy$ ./copyscript.sh dir1/tesla dir2/
Details for dir2/
total 0
-rw-rw-r-- 1 imran imran 0 Jun 19 23:01 tesla
```

Explanation:

Line 3 - run the command **cp** with the first command line argument as the source and the second command line argument as the destination.

Line 5 - run the command **echo** to print a message.

Line 6 - After the copy has completed, run the command **ls** for the destination just to verify it worked. We have included the options **l** to show us extra information and **h** to make the size human readable so we may verify it copied correctly.

Some System Variables

There are a few other variables that the system sets for you to use as well.

$0 - The name of the Bash script.

$1 - $9 - The first 9 arguments to the Bash script. (As mentioned above.)

$# - How many arguments were passed to the Bash script.

$@ - All the arguments supplied to the Bash script.

$? - The exit status of the most recently run process.

$$ - The process ID of the current script.

$USER - The username of the user running the script.

$HOSTNAME - The hostname of the machine the script is running on.

$SECONDS - The number of seconds since the script was started.

$RANDOM - Returns a different random number each time is it referred to.

$LINENO - Returns the current line number in the Bash script.

Setting your own variables

```
imran@DevOps:~/.../bash_scripts$ cat 1_print.sh
intA=20
floatB=20.20
stringA="first_string"
DIR_PATH="/tmp"

echo
echo "#########################"
echo "Value of integer A is $intA"
echo "#########################"
echo "Value of Float B is $intB"

echo "#########################"
echo "Value of string A is $stringA"

echo "#########################"
echo "Directory path is $DIR_PATH"

echo "#########################"
echo "Content of TMP directory."
echo "#########################"
ls $DIR_PATH
```

Quotes

Storing a single word in a variable works fine without quotes, but if we want to store a sentence and also want to store special characters like $,%,@ etc our normal variable assignment will not work.

```
imran@DevOps:.../bash$ myvar=Hello World
World: command not found
imran@DevOps:.../bash$
```

When we want variables to store more complex values however, we need to make use of quotes. This is because under normal circumstances Bash uses a space to determine separate items.

When we enclose our content in quotes we are indicating to Bash that the contents should be considered as a single item. You may use single quotes (') or double quotes (").

Single quotes will treat every character literally.

Double quotes will allow you to do substitution (that is include variables within the setting of the value).

'

```
imran@DevOps:.../bash$ myvar='Hello World'
imran@DevOps:.../bash$ echo $myvar
Hello World
imran@DevOps:.../bash$ newvar="More $myvar"
imran@DevOps:.../bash$ echo $newvar
More Hello World
imran@DevOps:.../bash$ newvar='More $myvar'
imran@DevOps:.../bash$ echo $newvar
More $myvar
imran@DevOps:.../bash$
```

Command Substitution

We have how to store a string/text into a variable but sometimes you want to store output of a command to a variable. Like you may need to store of ls command output to a variable. For this we use Command Substitution. There are two syntax for doing this.

1.

```
imran@DevOps:.../testcopy$ file=`ls`
imran@DevOps:.../testcopy$ echo $file
copyscript.sh dir1 dir2
```

2.

```
imran@DevOps:.../testcopy$ files=$(ls)
imran@DevOps:.../testcopy$ echo $files
copyscript.sh dir1 dir2
```

```
imran@DevOps:.../bash$ myvar=$( ls /etc | wc -l )
imran@DevOps:.../bash$ echo There are $myvar entries in the directory /etc
There are 249 entries in the directory /etc
```

Exporting Variables

Variable defined in the script leave with it and dies after the script dies or completes. If we want to define a variable that is accessible to all the scripts from your current shell we need to export it.

Export a variable from bash shell as mentioned below.

```
imran@DevOps:.../bash$ var1=foo
imran@DevOps:.../bash$ echo $var1
foo
imran@DevOps:.../bash$ export var1
```

Create a script which prints exported and local variable

```
imran@DevOps:.../bash$ vi script1.sh
#!/bin/bash
# demonstrate variable scope

var2=foobar

echo "Printing exported varible from bash shell"

echo $var1

echo "Printing varible defined in the script"
echo $var2
```

Execute the script to see the results

```
imran@DevOps:.../bash$ ./script1.sh
```

Printing exported variable from bash shell

foo

Printing variable defined in the script

foobar

Environment Variables or Exporting Variables Permanently

To export variables permanently, you can add the export command in any of the following start-up files :

~/.profile

~/.bashrc

/etc/profile

.profile and .bashrc lives in the users home directory so they are accessible only for that user. /etc/profile is for global access for all the variables.

For example:

```
imran@DevOps:~$ cat .bashrc | grep export

#export
GCC_COLORS='error=01;31:warning=01;35:note=01;36:caret=01;32:locus=01:quote=01'

export tesla='Alternate current'

export EDITOR=vim

export name=Cassini
```

Summary

$1, $2, ...

The first, second, etc command line arguments to the script.

variable=value

To set a value for a variable. Remember, no spaces on either side of =

Quotes " '

Double will do variable substitution, single will not.

variable=$(command)

Save the output of a command into a variable

export var1

Make the variable var1 available to child processes.

3. User Input

Interactive Scripts

Ask the User for Input

Taking input from the user while executing the script, storing it into a variable and then using that variable in our script. We would be taking inputs from user like IP addresses, usernames, passwords or confirmation Y/N to do this we use command called **read**. This command takes the input and will save it into a variable.

read var1

Let's look at a simple example:

input.sh

```
1. #!/bin/bash
2. # Ask the username
3. echo "Please enter the username"
4. read varname
5. echo "Welcome back $varname"
```

```
imran@DevOps:.../bashscripts$ ./input.sh
Please enter the username
DevTestOps
Welcome back DevTestOps
```

Explanation:

Line 3 - Print a message asking the user for input.

Line 4 - Run the command **read** and save the users response into the variable **varname**

Line 5 – echo another message just to verify the read command worked. Note: I had to put a backslash (\ ) in front of the ' so that it was escaped.

You are able to alter the behaviour of **read** with a variety of command line options. (See the man page for read to see all of them.) Two commonly used options however are **-p** which allows you to specify a prompt and **-s** which makes the input silent. This can make it easy to ask for a username and password combination like the example below:

Login.sh

```
#!/bin/bash

# Ask the user for login details

read -p 'Username: ' uservar

read -sp 'Password: ' passvar

echo

echo Thankyou $uservar we now have your login details
```

So far we have looked at a single word as input. We can do more than that however.

multiinput.sh

```
#!/bin/bash

# Demonstrate how read actually works

echo Enter your Name, Profession & Interests in same order seprated by a space?

read name profession interest

echo Your entered name is: $name

echo Your profession is: $profession

echo Your are interested in: $interest
```

```
imran@DevOps:.../bashscripts$ ./multiinput.sh

Enter your Name, Profession & Interests in same order seprated by a space?

Imran DevOps Hiking

Your entered name is: Imran

Your profession is: DevOps

Your are interested in: Hiking
```

4. If Statements

Scripts making decisions.

Basic If Statements

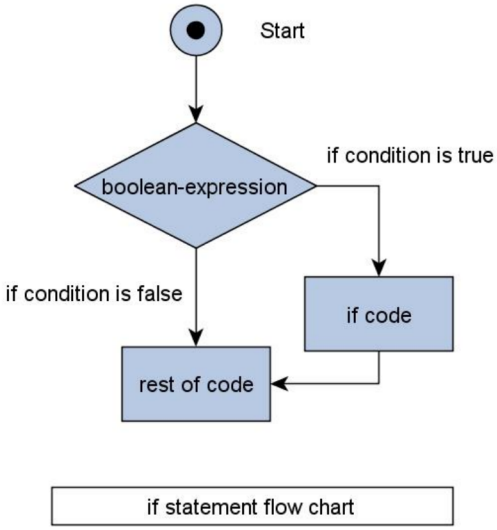
If you use bash for scripting you will undoubtedly have to use conditions a lot. Based on a condition you decide if you should execute some commands on the system or not.

A basic if statement effectively says, **if** a particular test is true, then perform a given set of actions. If it is not true then don't perform those actions. If follows the format below:

---

if [<some test>]
then
<commands>
fi

Anything between **then** and **fi** (if backwards) will be executed only if the test (between the square brackets) is true.



if statement flow chart

Let's look at a simple example:

if_example.sh

```
#!/bin/bash
# Basic if statement
if [ $1 -gt 100 ]
then
  echo Hey that's a large number.
  pwd
```

```
fi

date
```

Explanation:

Line 3 - Let's see if the first command line argument is greater than 100

Line 5 and 6 - Will only get run if the test on line 4 returns true. You can have as many commands here as you like.

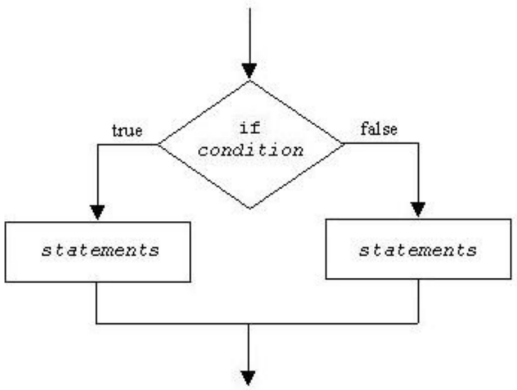
Line 7 - fi signals the end of the if statement. All commands after this will be run as normal.

Line 8 - Because this command is outside the if statement it will be run regardless of the outcome of the if statement.

```
imran@DevOps:.../bash$ ./if_example.sh 150

Hey that's a large number.

/tmp/bash

Sun Oct 30 16:19:28 IST 2016
```

Sample if/else condition script

If condition is true we execute a block of code but if its false we can execute some other block of code by using else command.



```
#!/bin/bash


intA=20

intB=30

if [ intA==intB ];

then

  echo "intA is equal to intB"

else

  echo "Not Equal"

fi


if [ -f hosts ];

then

  echo "File exists!"

else

  echo "Does not exist"

fi
```

Test

The square brackets ([ ]) in the **if** statement above are actually a reference to the command **test**. This means that all of the operators that test allows may be used here as well. Look up the man page for test to see all of the possible operators (there are quite a few) but some of the more common ones are listed below.

Operator	Description
! EXPRESSION	The EXPRESSION is false.
-n STRING	The length of STRING is greater than zero.
-z STRING	The length of STRING is zero (ie it is empty).
STRING1 = STRING2	STRING1 is equal to STRING2
STRING1 != STRING2	STRING1 is not equal to STRING2
INTEGER1 -eq INTEGER2	INTEGER1 is numerically equal to INTEGER2
INTEGER1 -gt INTEGER2	INTEGER1 is numerically greater than INTEGER2
INTEGER1 -lt INTEGER2	INTEGER1 is numerically less than INTEGER2
-d FILE	FILE exists and is a directory.
-e FILE	FILE exists.
-r FILE	FILE exists and the read permission is granted.
-s FILE	FILE exists and it's size is greater than zero (ie. it is not empty).
-w FILE	FILE exists and the write permission is granted.
-x FILE	FILE exists and the execute permission is granted.

A few points to note:

= is slightly different to **-eq**. [001 = 1] will return false as = does a string comparison (ie. character for character the same) whereas -eq does a numerical comparison meaning [001 -eq 1] will return true.

When we refer to FILE above we are actually meaning a path. Remember that a path may be absolute or relative and may refer to a file or a directory.

Because [5 is just a reference to the command **test** we may experiment and trouble shoot with test on the command line to make sure our understanding of its behaviour is correct.

5. Loops!

Execute it again and again and again....

Loops allow us to take a series of commands and keep re-running them until a particular situation is reached. They are useful for automating repetitive tasks.

For Loop

A 'for loop' is a bash programming language statement which allows code to be repeatedly executed. A for loop is classified as an iteration statement i.e. it is the repetition of a process within a bash script. For example, you can run Linux command or task 5 times or read and process list of files using a for loop. A for loop can be used at a shell prompt or within a shell script itself.

---

for var in <list>
do
<commands>
done

The for loop will take each item in the list (in order, one after the other), assign that item as the value of the variable **var**, execute the commands between do and done then go back to the top, grab the next item in the list and repeat over.

The list is defined as a series of strings, separated by spaces.

For loops iterate for as many arguments given:

`Example:`

The contents of $Variable is printed three times.

```
#!/bin/bash
for Variable in {1..3}
do
    echo "$Variable"
done
```

Or write it the "traditional for loop" way:

```
for ((a=1; a <= 3; a++))
do
    echo $a
done
```