Bash is a name of the unix shell, which was also distributed as the shell for the GNU operating system and as default shell on Linux and Mac OS X. Nearly all examples below can be a part of a shell script or executed directly in the shell.

Read more here.

```bash
#!/bin/bash
# First line of the script is shebang which tells the system how to execute
# the script: http://en.wikipedia.org/wiki/Shebang_(Unix)
# As you already figured, comments start with #. Shebang is also a comment.

# Simple hello world example:
echo Hello world!

# Each command starts on a new line, or after semicolon:
echo 'This is the first line'; echo 'This is the second line'

# Declaring a variable looks like this:
Variable="Some string"

# But not like this:
Variable = "Some string"
# Bash will decide that Variable is a command it must execute and give an error
# because it can't be found.

# Or like this:
Variable= 'Some string'
# Bash will decide that 'Some string' is a command it must execute and give an
# error because it can't be found. (In this case the 'Variable=' part is seen
# as a variable assignment valid only for the scope of the 'Some string'
# command.)

# Using the variable:
echo $Variable
echo "$Variable"
echo '$Variable'
# When you use the variable itself - assign it, export it, or else - you write
# its name without $. If you want to use the variable's value, you should use $.
# Note that ' (single quote) won't expand the variables!

# String substitution in variables
echo ${Variable/Some/A}
# This will substitute the first occurrence of "Some" with "A"

# Substring from a variable
Length=7
echo ${Variable:0:Length}
# This will return only the first 7 characters of the value

# Default value for variable
echo ${Foo:-"DefaultValueIfFooIsMissingOrEmpty"}
# This works for null (Foo=) and empty string (Foo=""); zero (Foo=0) returns 0.
# Note that it only returns default value and doesn't change variable value.

# Builtin variables:
```

```
# There are some useful builtin variables, like
echo "Last program's return value: $?"
echo "Script's PID: $$"
echo "Number of arguments passed to script: $#"
echo "All arguments passed to script: $@"
echo "Script's arguments separated into different variables: $1 $2..."

# Reading a value from input:
echo "What's your name?"
read Name # Note that we didn't need to declare a new variable
echo Hello, $Name!

# We have the usual if structure:
# use 'man test' for more info about conditionals
if [ $Name != $USER ]
then
    echo "Your name isn't your username"
else
    echo "Your name is your username"
fi

# NOTE: if $Name is empty, bash sees the above condition as:
if [ != $USER ]
# which is invalid syntax
# so the "safe" way to use potentially empty variables in bash is:
if [ "$Name" != $USER ] ...
# which, when $Name is empty, is seen by bash as:
if [ "" != $USER ] ...
# which works as expected

# There is also conditional execution
echo "Always executed" || echo "Only executed if first command fails"
echo "Always executed" && echo "Only executed if first command does NOT fail"

# To use && and || with if statements, you need multiple pairs of square brackets:
if [ "$Name" == "Steve" ] && [ "$Age" -eq 15 ]
then
    echo "This will run if $Name is Steve AND $Age is 15."
fi

if [ "$Name" == "Daniya" ] || [ "$Name" == "Zach" ]
then
    echo "This will run if $Name is Daniya OR Zach."
fi

# Expressions are denoted with the following format:
echo $(( 10 + 5 ))

# Unlike other programming languages, bash is a shell so it works in the context
# of a current directory. You can list files and directories in the current
# directory with the ls command:
ls

# These commands have options that control their execution:
```

```
ls -l # Lists every file and directory on a separate line

# Results of the previous command can be passed to the next command as input.
# grep command filters the input with provided patterns. That's how we can list
# .txt files in the current directory:
ls -l | grep "\.txt"

# You can redirect command input and output (stdin, stdout, and stderr).
# Read from stdin until ^EOF$ and overwrite hello.py with the lines
# between "EOF":
cat > hello.py << EOF
#!/usr/bin/env python
from __future__ import print_function
import sys
print("#stdout", file=sys.stdout)
print("#stderr", file=sys.stderr)
for line in sys.stdin:
    print(line, file=sys.stdout)
EOF

# Run hello.py with various stdin, stdout, and stderr redirections:
python hello.py < "input.in"
python hello.py > "output.out"
python hello.py 2> "error.err"
python hello.py > "output-and-error.log" 2>&1
python hello.py > /dev/null 2>&1
# The output error will overwrite the file if it exists,
# if you want to append instead, use ">>":
python hello.py >> "output.out" 2>> "error.err"

# Overwrite output.out, append to error.err, and count lines:
info bash 'Basic Shell Features' 'Redirections' > output.out 2>> error.err
wc -l output.out error.err

# Run a command and print its file descriptor (e.g. /dev/fd/123)
# see: man fd
echo <(echo "#helloworld")

# Overwrite output.out with "#helloworld":
cat > output.out <(echo "#helloworld")
echo "#helloworld" > output.out
echo "#helloworld" | cat > output.out
echo "#helloworld" | tee output.out >/dev/null

# Cleanup temporary files verbosely (add '-i' for interactive)
rm -v output.out error.err output-and-error.log

# Commands can be substituted within other commands using $( ):
# The following command displays the number of files and directories in the
# current directory.
echo "There are $(ls | wc -l) items here."

# The same can be done using backticks `` but they can't be nested - the preferred way
# is to use $( ).
```

```
echo "There are `ls | wc -l` items here."

# Bash uses a case statement that works similarly to switch in Java and C++:
case "$Variable" in
    #List patterns for the conditions you want to meet
    0) echo "There is a zero.";;
    1) echo "There is a one.";;
    *) echo "It is not null.";;
esac

# for loops iterate for as many arguments given:
# The contents of $Variable is printed three times.
for Variable in {1..3}
do
    echo "$Variable"
done

# Or write it the "traditional for loop" way:
for ((a=1; a <= 3; a++))
do
    echo $a
done

# They can also be used to act on files..
# This will run the command 'cat' on file1 and file2
for Variable in file1 file2
do
    cat "$Variable"
done

# ..or the output from a command
# This will cat the output from ls.
for Output in $(ls)
do
    cat "$Output"
done

# while loop:
while [ true ]
do
    echo "loop body here..."
    break
done

# You can also define functions
# Definition:
function foo ()
{
    echo "Arguments work just like script arguments: $@"
    echo "And: $1 $2..."
    echo "This is a function"
    return 0
}
```

```
# or simply
bar ()
{
    echo "Another way to declare functions!"
    return 0
}

# Calling your function
foo "My name is" $Name

# There are a lot of useful commands you should learn:
# prints last 10 lines of file.txt
tail -n 10 file.txt
# prints first 10 lines of file.txt
head -n 10 file.txt
# sort file.txt's lines
sort file.txt
# report or omit repeated lines, with -d it reports them
uniq -d file.txt
# prints only the first column before the ',' character
cut -d ',' -f 1 file.txt
# replaces every occurrence of 'okay' with 'great' in file.txt, (regex compatible)
sed -i 's/okay/great/g' file.txt
# print to stdout all lines of file.txt which match some regex
# The example prints lines which begin with "foo" and end in "bar"
grep "^foo.*bar$" file.txt
# pass the option "-c" to instead print the number of lines matching the regex
grep -c "^foo.*bar$" file.txt
# if you literally want to search for the string,
# and not the regex, use fgrep (or grep -F)
fgrep "^foo.*bar$" file.txt


# Read Bash shell builtins documentation with the bash 'help' builtin:
help
help help
help for
help return
help source
help .

# Read Bash manpage documentation with man
apropos bash
man 1 bash
man bash

# Read info documentation with info (? for help)
apropos info | grep '^info.*('
man info
info info
info 5 info

# Read bash info documentation:
info bash
info bash 'Bash Features'
info bash 6
info --apropos bash
```