# Vim class notes

Chris Wallace

Week 4

Recap: What did we talk about last week?

- conditionals

## Other Comparators

In our previous classes, we have checked for *equality* using `-eq`. However, there are other comparison operatiors too:

| **english** | **math** | **bash** |
|---|---|---|
| equals | $=$ | `-eq` |
| not equals | $\neq$ | `-ne` |
| less than | $<$ | `-lt` |
| greater than | $>$ | `-gt` |
| less than or equal to | $\leq$ | `-le` |
| greater than or equal to | $\geq$ | `-ge` |

## Loops

Just like conditionals, all languages have loops. It's something that allows us to run the same code multiple times.

```
i=1
while [ $i -lt 5 ]
do
    echo "$i"
    (( i++ ))
done
```

You can also use loops to do something over and over until a human stops it.

```
times=1
go=1
while [ $go -eq 1 ]
do
    echo "we went $times times"
    (( times++ ))
    read -p "should we continue? press 1: " go
done
```

## Data Types

What are data types? Just the types of data we have! We discussed that, in this class, we've primarily dealth with three: Booleans, Integers, and Strings.

### Booleans

Booleans have two possible values: **True** or **False**. Each one of our **if** statements dealt with booleans.

### Integers

If you remember from math class, integers are positive or negative whole numbers, including zero. Some examples are -1, 0, 1, 2, etc…

For the shell/bash programming language, the comparisons we've learned (`-eq`, `-lt`, etc are all operations on integers. You couldn't use these to compare strings, which we'll discuss next.

### Strings

Strings are the types of data we keep in the " (quotation) marks. `"hi"` is a string in bash.

### Examples

Some programming languages make you specify the data type of a variable you even name it! However, bash is not one of those languages- it's up to you to look at the variable and think about the type of data it is.

```
# integers
x=4
y=7
z=-1

# strings
name="chris"
greeting="Hello there, human"
stringnum="5"

# convert x ond y to strings
x="$x"
y="$y"
```

## File Permissions

Finally, we discussed that you can run your program directly by doing:

```
./program_name.sh
```

However, we noted that there are two prerequisites:

1. Your first line of the file is `#!/bin/bash`
    - This says that "Hey, this file is meant to be run by this program"
    - `/bin/bash` is where `bash` lives.
2. You have *execute permissions* to this script.
    - We used the command `chmod u+x ./program_name.sh`
    - This means "give **U**ser the e**X**ecute permissions to this file."

Here's example output of the command `ls -l`

```
total 12
--w------- 1 ubuntu ubuntu  48 Jan 21 00:48 no_read.txt
---x--x--x 1 ubuntu ubuntu   9 Jan 21 00:51 no_read_no_write.txt*
-r--r--r-- 1 ubuntu ubuntu 177 Jan 28 23:25 read_no_write.txt
```

Looking at the left, we explained that there are 10 "spots". Each one of these spots is either "occupied" with a letter, or is a `-`. To phrase it differently, each file has 10 *boolean* spots about their permissions.

- The first spot is `d`, which will be set if this is a directory.
- The next 3 spots are for the **user's** read, write, and execute permissions.

- The next 3 spots are for the **group's** read, write, and execute permissions.
- The next 3 spots are for everyone else, or **other's** read, write, and execute permissions.

This command also lists the owner (ubuntu), the owning group (ubuntu), and the last modified time of the file in UTC time.

We can use the **chmod** command to add or remove permissions.