{ Introduction to programming with Scala }

About this course

Prerequisites

Have a genuine interest in programming.
Have a laptop preferably with Scala and IntelliJ or equivalent installed.

Objective

Get people with little/no experience programming started with Scala.
BUILD A GAME!!! :D

Format

We'll introduce a topic/feature and have a discussion then have a quick recap and give you some time to experiment and try different things.

Please ask questions!!

Overview

Programming

Storing values

Types

Classes, Objects, Inheritance and Traits

Operators

Conditions (if/else)

Methods

Scope

Unit Testing/Test Driven Development (TDD)

Collections

Programming

Programming languages
Software to write programs in
Compiling/running a program
Scala

Programming - languages

We write computer programs/instructions using programming language(s).

Examples of programming languages: Scala, Java, C, C#, Python, JavaScript

Like the languages we use for communication, different programming languages use different keywords and support different styles of programming.

They also have different syntax (similar to grammar) that dictates how symbols should be used to build valid files.

Scala	`println("Hello, world!") or println "Hello, world!"`
Java	`System.out.println("Hello, world!");`
C	`printf("Hello, world!");`
C#	`Console.WriteLine("Hello, world!");`
JavaScript	`document.write("Hello, world!");`
Python	`print "Hello, world!"`

Programming - writing code

You can use (almost) anything to write a program, even notepad or a command-line text editor.

As long as the code and file extension are correct it will work.

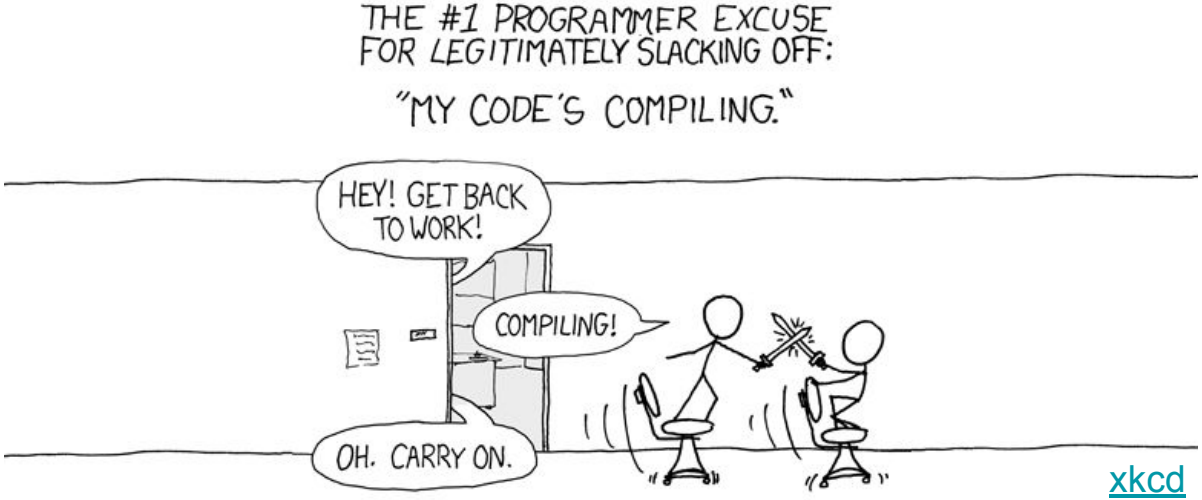
However, it's much easier to write code using an IDE (Integrated Development Environment).

Typically, an IDE will check your code while you're writing it as well as providing syntax highlighting and suggestions making it much faster and more efficient to write your code in.

Programming - compiling/running

To run the code we write it needs to be compiled/translated into something that can be read by a machine.

When Scala/Java programs are compiled, the code that is created is not machine code, like most compiled languages. Instead, Scala/Java compiles to files with a .class or .jar extension, and these files contain "bytecode".



xkcd

Programming - compiling/running

The main difference between machine code and bytecode is that bytecode instructions can't be run directly by the processor in your computer.

The reason for not compiling direct to machine code is that machine code is specific to one particular type of hardware and operating system whereas bytecode can be run on any type of computer (so long as a Java Virtual Machine (JVM) exists for that computer).

A JVM reads .class and .jar files, does some initial checks and then runs them. Because bytecode is so close to machine code, running each instruction takes only a few instructions within the JVM, and this makes it fast...

Programming - compiling/running

...very fast!

An introduction to programming - Scala

Now we've covered some of the foundations of programming we can talk about Scala - the main programming language used in the Digital Delivery Centres.

Scala was designed by Martin Odersky to improve on Java. It is very **Scala**ble in the sense it can be run as a small script or as a large system.

One of the things we've touched on already is that Scala compiles to Java bytecode and runs on the Java virtual machine.
This means it has all the portability of Java as well as being interoperable with Java (write Scala and Java in the same application). Being interoperable with Java is important because it means that Scala can make use of the thousands of tried and tested Java libraries out there.


Kahoot!

Storing values

Vals
Vars
Mutability
Immutability

Storing values

Suzie went to the shops and bought 5 melons at 50p, 2 apples at 10p and 6 sodas priced at £1.20.

What was the total cost in **pence**? Show your working.

Storing values

Suzie went to the shops and bought 5 melons at 50p, 2 apples at 10p and 6 sodas priced at £1.20.

What was the total cost in **pence**? Show your working.

Melons = 5 x 50 = 250

Apples = 2 x 10 = 20

Sodas = 6 x 120 = 720

Total = 250 + 20 + 720 = 990

Storing values

In Scala we can store things in vals (values) and then reference them by name later to get the value.

It can also read numbers and knows how to multiply and add them together.

Our maths problem can easily be represented in code by using a value for each calculation we make.

Melons = 5 x 50 = 250

Apples = 2 x 10 = 20

Sodas = 6 x 120 = 720

Total = 250 + 20 + 720 = 990

```
val melons = 5 * 50

val apples = 2 * 10

val sodas = 6 * 120

val total = melons + apples + sodas
```

Storing values

Now lets try the same exercise again but this time keep a running total.

```
val total = 0

val melons = 5 x 50

total = total + melons
```

Compile

->

```
error: reassignment to val

     total = total + melons
                     ^
one error found
```

This fails because a `val` is immutable and therefore cannot be changed.

Storing values

If we need to store something that will change we use a var (variable) because they are mutable and therefore can be changed as many times as we want.

```
var total = 0

val melons = 5 * 50
total = total + melons

val apples = 2 * 10
total = total + apples

val sodas = 6 * 120
total = total + sodas
```

Storing values - best practice

While vars might seem useful because of their flexibility, there are a few implications that come with it.

- You can't always be sure what the value is at a given time because it can be modified by any part of the program that has access to it.
- It can cause issues if two parts of the same program are trying to access/manipulate it at the same time.
- It can make it easier to write less efficient code as we saw previously.

For these reasons if you program with vals and immutability in mind you can have a lot more trust that your code will always work as expected.

Despite this important difference between values and variables, code programmers will often use the term variable when referring to either a value or variable. It's best to assume they are referring to a value.


Kahoot!

Scala worksheets

Worksheets are self-contained scala files that can be easily run on their own inside IntelliJ. They are a useful way to test/experiment with small pieces of code without having to worry about project structures or using sbt.

After typing your code you can click the play button to run your worksheet and IntelliJ will put compiled information in the panel on the right.

Alternatively you can check Interactive mode where IntelliJ will listen for changes you make and run your worksheet whenever you make a change.

Group practical

Use a scala worksheet to play with vals/vars:

Store a val
Re-assign a val
Store a var
Re-assign a var

Practical

Using a scala worksheet can you represent this maths question:

John went to the shops and bought 5 lemons at 30p each, 2 bags of flour at 90p and 6 ciders priced at £2.50.

What was the total cost in **pence**? Show your working.

Types

What is a type?
Integer
String
Booleans
Doubles
BigDecimal
Dates

Types

Types are like a label to describe what data you're storing (think film genres).

Once defined a value/variable cannot store something that is not of the same type.

Scala offers a variety of data types to model different types of information relevant to the problem that you as a developer may want to solve.

Types

The type of a value/variable can be defined explicitly or inferred implicitly by Scala.

A variable named 'one' is defined as an Integer (Int) and contains the value 1.

A variable named 'uno' has the value 1 stored in it and because 1 is a whole number Scala will default this to an Integer.

The process of allowing Scala to fill in the type for you is called type inference. Scala can do this because it is statically typed - meaning that at compile time Scala will check/calculate the type of every variable.

```
val one: Int = 1

val uno = 1
```

Types

In the remainder of this section all declarations will be defined explicitly (with the data type).

In practice you will often leave it to Scala to infer the data type unless you want to declare what the type is to improve readability.

Types - Integer

An `'Integer'` is used to store a whole number in the range of: -2,147,483,648 to 2,147,483,647.

Integers can be defined in the following way:

```
val x: Int = -20  // This will store a value of -20

val y: Int = 3    // This will store a value of 3
```

Types - String

A `String` is used to store text, numbers and symbols.

Strings can be defined in the following way:

```
val x: String = "This is a string"

val y: String = "String 2"

val gibberish: String = "$Q^HQ%  $T  %Y%GREVE$$"
```

Types - Booleans

A `'Boolean'` is a value that can store two values and they are 'true' or 'false'. These types are useful when storing the result of a check e.g. Is Pepsi better than Coke?

Booleans can be defined in the following way:

```
val x: Boolean = true   // This will store a value of true

val y: Boolean = false  // This will store a value of false
```


Kahoot!

Group practical

Create a new scala worksheet

Store the following by explicitly declaring their type:

Catch-22, catdog, 7, true, Yippee ki-yay, false, a11y, 314159265359

Practical

Create a new scala worksheet

Store the following by explicitly declaring their type:

maybe, dog, -11, true, They may take our lives, but they'll never take our freedom!, false, 4815162342

Classes, Objects, Inheritance and Traits

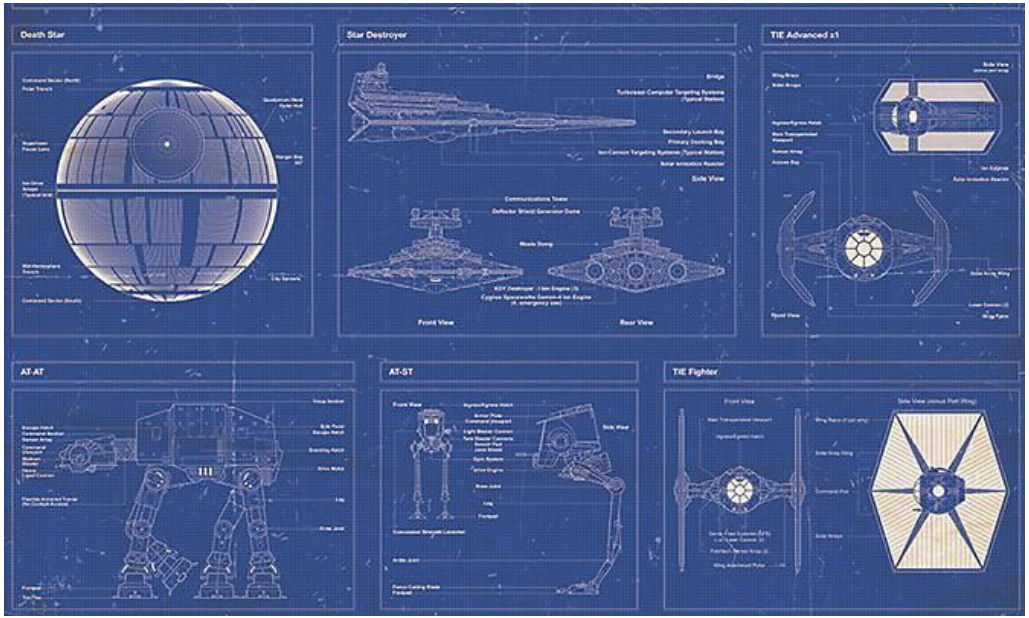
Classes
Objects
Inheritance
Traits

Classes

A class is essentially a blueprint/pattern that can be used to make something.

Engineers and construction/factory workers use blueprints to build cars, buildings, bridges, virtually anything.

Tailors, seamsters, printers use patterns to make the clothes we wear, books we read.



Classes

In Scala, classes are created like the example below.

```
class SpaceStation {
     val shape: String = "Sphere"
     val hasSignificantWeaknessThatCanBeExploitedByHero: Boolean = true
}
```

We specify we're creating a class, we give it a name (SpaceStation). Similar to our names the best practice is to start class names with a capital letter because they represent the name of something. This also makes it easier to know when we're referencing a class in other places in our code.

We mark the start and the end of our class using curly braces {}. Everything inside these curly braces belong to that class. We can have simple values in there, we can have functions. These are referred to as members of the class.

Instantiating classes

Blueprints and patterns aren't practically useful in their raw form. We can't drive/live in blueprints, we could wear blueprints for clothes... but definitely wouldn't recommend it.

What we really want to do is use the blueprint/pattern to make something and use that.
It's the same in programming, you can't use a class in its raw form you have to use it to create an object.

```
val deathStar: SpaceStation = new SpaceStation()
```

Once we've created our object we have full access to all of our class' members.
We can use our class to create as many instances of a class as we like.

So Disney can now use this to create as many sequels/prequels as they want!!!

Parameters

Parameters are what we call the argument(s) passed into a function or class.

We can pass in raw values or we can pass in values we already stored.

```
class Apple(val brand: String, val colour: String)
val green: String = "Green"
val grannySmith: Apple = new Apple("Smith", green)
```

Here we've used brackets to list the parameters we need to build objects using our class.
We've used the **`val`** keyword to tell Scala to make the parameter a member of the class for us (as opposed to having to write the code to do this ourselves - Java).
We also need to give the parameter a meaningful name and specify the type.

Then when we want to make our *grannySmith* object we say we want a new object and provide/pass it the values it requires to create an object.

Objects

If we don't need to make the object again and only need 1 copy of it then we can just create the object and use it immediately, which saves us having to make a class and creating a new object from that class.

```
object MathematicalConstants {
     val pi: Double = 3.1415926535
     val e: Double = 2.7182818284
     val goldenRatio: Double = 1.6180339
     val euler: Double = 0.57721
}
```

`MathematicalConstants.`*`goldenRatio`*

`MathematicalConstants.`*`euler`*

Differences between classes and objects

Objects have no params

Objects are globally available to be imported and used
(you don't need to do use `new`)

Inheritance

Similar to how when we're born we inherit genetic traits from our parents - in computer programming we can inherit the members of another class.

We do this using the extends keyword when declaring our class. The class we extend from is referred to as a parent/superclass of the class extending it.

We can also be somewhat picky and override members we inherit to customise them for our new class.

```
class ChocolateBar {
     val colour: String = “Brown”
     val hasNuts: Boolean = false
     val isNice: Boolean = true
}
```

```
class Bounty extends ChocolateBar {
     //colour: String = “Brown”
     override val hasNuts: Boolean = true
     //isNice : Boolean = true
}
```

Traits

Traits can be used in a similar way to inheritance. Where instead of extending a class we 'mix-in' the members of a trait using the with keyword. Then similar to inheriting from a class we now have access to the members of our trait.

Something we can do with traits that we can't do with classes is have unimplemented members. That way when they are mixed-in or extended an implementation must be provided.

```
class ChocolateBar

trait Filling {
     val filling: String
}

trait Celebration {
     val hasMiniature: Boolean = true
}
```

```
class Bounty extends ChocolateBar with
Filling with Celebration {
     val filling = “Coconut”
     //hasMiniature: Boolean = true
     //hasNuts: Boolean= false
     //colour: String = “Brown”
     override val isNice : Boolean = false
}
```

Traits vs Classes

- A class can only extend one class, but many traits
- Class inheritance is typically hierarchical
- Class inheritance asks the question ‘is x a y?’
- You can create an instance of a class, you can’t create an instance of a trait
- A class can be implemented on its own, a trait is generally designed to be extended/implemented by something else

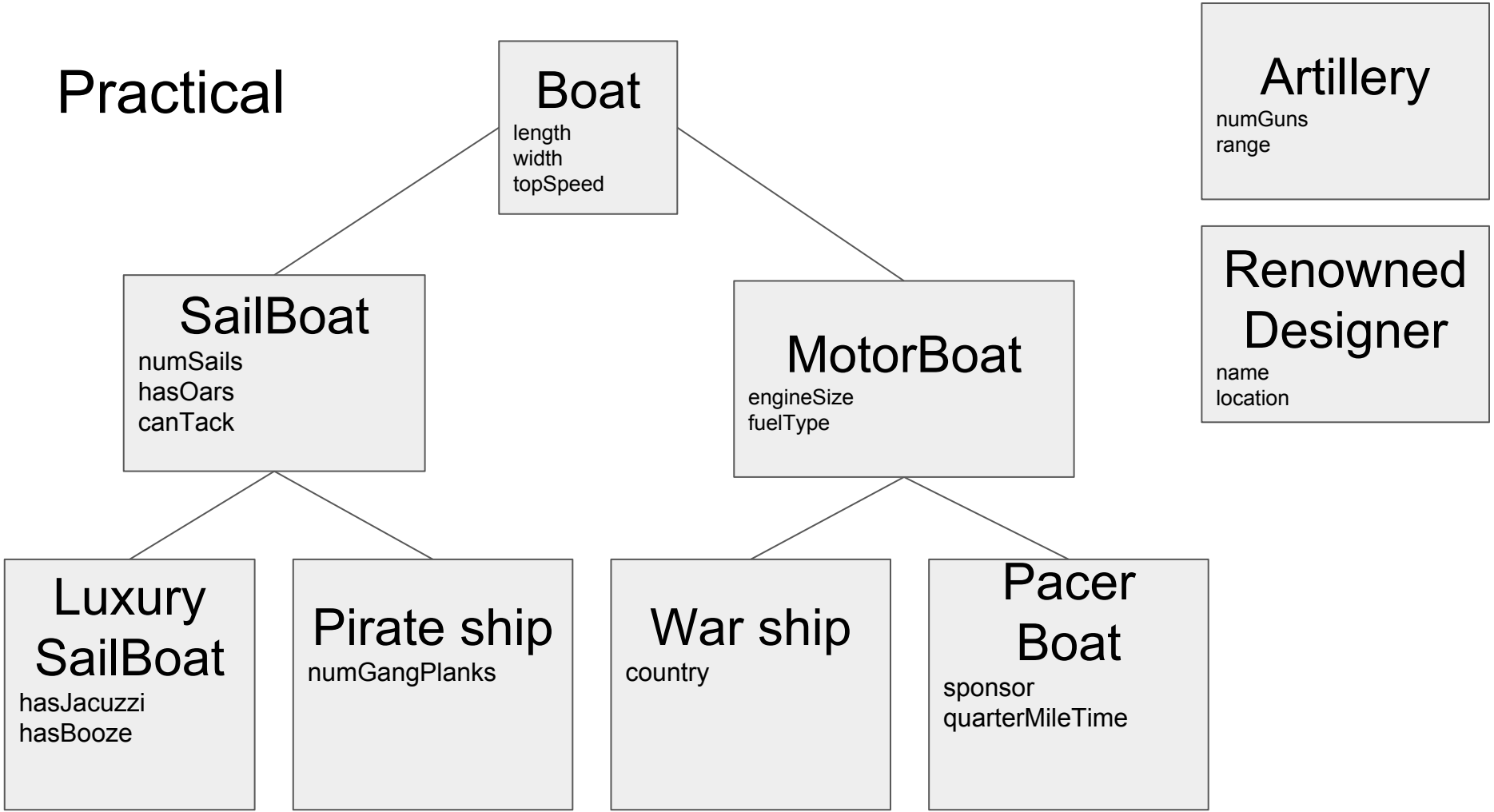

Kahoot!

Group practical

We’re going to do the classic inheritance example: Animals! We need to construct an inheritance hierarchy of the animal kingdom. We’ll need to consider the following rules and groupings:

- Are there any properties common to all animals?
- Animals will need to be grouped into their Class (as in ‘mammal’ etc., not like a scala class), then maybe into a species
- Look for common attributes that *all* animals in a class/species share
- If there are attributes that *some*, but not all animals in a class/species share then maybe consider a trait that can be used as a mixin (e.g. Domesticated)

Practical


Boat
length
width
topSpeed
SailBoat
numSails
hasOars
canTack
MotorBoat
engineSize
fuelType
Luxury SailBoat
hasJacuzzi
hasBooze
Pirate ship
numGangPlanks
War ship
country
Pacer Boat
sponsor
quarterMileTime
Artillery
numGuns
range
Renowned Designer
name
location

Operators

Arithmetic operators
Relational operators
Logical operators

Operators - Arithmetic

Arithmetic operators in scala are pretty much just standard mathematical operators. You can apply these operators to numbers themselves or to vals that hold the numbers.

We'll cover the following:

- **+** - The **addition** operator (a + b)
- **-** - The **subtraction** operator (a - b)
- **/** - The **division** operator (a / b)
- ***** - The **multiplication** operator (a * b)
- **%** - The **modulus** operator (a % b) - Finds the remainder after dividing two numbers

Operators - Arithmetic

For these examples: `val a = 10, val b = 5`

- `+`

```
val x = a + b
x: Int = 15
val y = 2 + 3
y: Int = 5
```

- `-`

```
val x = a - b
x: Int = 5
val y = 8 - 4
y: Int = 4
```

- `*`

```
val x = a * b
x: Int = 50
val y = 2 * 3
y: Int = 6
```

- `/`

```
val x = a / b
x: Int = 2
val y = 8 / 4
y: Int = 2
```

- `%`

```
val x = a % b
x: Int = 0
val y = 10 % 3
y: Int = 1
```

Operators - Relational

Relational operators compare two values, returning a Boolean result.

We'll discuss the following relational operators, all examples are those that would evaluate to **true**:

- `==` (Equality): `5 == 5` `"Hello" == "Hello"`
- `!=` (Inequality): `5 != 3` `"Hello" != "hello"`
- `<` (Less than): `3 < 5` `"a" < "c"`
- `<=` **(**Less than or equal to) `5 <= 5` `"b" <= "b"`
- `>` **(**Greater than**)** `5 > 3` `"c" > "a"`
- `>=` **(**Greater than or equal to) `5 >= 5` `"c" >= "c"`

Operators - Relational

- ==

```
val x = 1 == 1
x: Boolean = true

val y = “word” == “differentword”
y: Boolean = false
```

- !=

```
val x = 1 != 1
x: Boolean = false

val y = “word” != “differentword”
y: Boolean = true
```

Operators - Relational

- `>`

```
val x = 5 > 2
x: Boolean = true

val y = “alphabetical” > “lexicographical”
y: Boolean = false
```

- `<`

```
val x = 5 < 2
x: Boolean = false

val y = “alphabetical” < “lexicographical”
y: Boolean = true
```

Operators - Logical

Logical operators can be used to combine Booleans.

We'll cover the operators **AND, OR** and **NOT**. These may be familiar from any other kind of logic you've seen. You can perform these operations on straight up boolean values (true, false) or on Boolean expressions, like those covered in the relational operators section.

Operators - Logical - **AND**

The AND operator requires both Booleans provided to be true and uses the symbol **&&**

```
val x = true && true
x: Boolean = true

val y = true && false
y: Boolean = false

val z = (1 == 2) && (3 > 4)
z: Boolean = false
```

A	B	A && B
true	true	true
true	false	false
false	true	false
false	false	false

Operators - Logical - **OR**

The OR operator requires one or both Booleans provided to be true and uses the symbol **||**

```
val x = true || false
x: Boolean = true

val y = true || true
y: Boolean = true

val z = (1 == 2) || (3 > 4)
z: Boolean = false
```

A	B	A || B
true	true	true
true	false	true
false	true	true
false	false	false

Operators - Logical - **NOT**

The NOT operator, unlike the other two logical operators, takes a single Boolean and negates the result. It uses the symbol `!`.

```
val x = !true
x: Boolean = false

val y = !(false && false)
y: Boolean = true

val z = !(1==2)
z: Boolean = true
```

A	!A
true	false
false	true


Kahoot!

Group practical

Use operators to store values relating to the following:

- Four plus three times fifty-five is less than three-hundred
- ninety-six divided by twelve is greater than eight or exactly eight
- Hamster is less than Hippo (strings)
- Eight is even (modulus)
- One-hundred and fifty-six divided by eight is less than twenty and fifty-four is less than 20 and fifty-four divided by four is less than fourteen

Practical

Use operators to store values relating to the following:

- Three plus four times fifty-seven is less than three-hundred
- One-hundred and forty-four divided by 12 is greater than twelve or equal to twelve
- Cat is less than Dog (strings)
- Seventeen is odd (modulus)
- Seventy-five divided by nine is less than thirty and eighty-nine divided by six is less than twenty

Conditional if/else

if

else

else if

Conditional if/else

When programming we sometimes want to go make a decision on what to do next based on the value of something else.

Real-world example, you're about to leave the house - you'll only take a coat if it's cold. If it's not cold then you don't need a coat.

```
val weather: String = "Cold"

if(weather == "Cold"){
   //take a coat
} else {
   //don't take a coat
}
```

Here, we're using the equals function which will do an exact case-sensitive match.
If the condition is met it executes the code inside the curly braces.
Else will catch any scenarios that don't match the if condition.

Conditional if/else

Sometimes things are a little more complicated so we can add extra conditions using else if

```
val weather: String = "Cold"

if(weather == "Cold"){
     //take a coat
} else if(weather == "Raining"){
     //take an umbrella
} else {
     //take nothing
}
```

Conditional if/else

Unfortunately, this also means that roughly per each additional condition we add we'll need ~ 3 lines of extra code and it can become difficult to read and difficult to maintain.

There is a better way to do this with Scala using something called pattern matching that we'll cover later

```
val weather: String = "Cold"

if(weather == "Cold"){
    //take a coat
} else if(weather == "Raining"){
    //take an umbrella
} else if(weather == "Hailing"){
    //wait
} else if(weather == "Hurricane"){
    //go to basement
} else if(weather == "Apocalypse"){
    //YOLO
} else {
    //take nothing
}
```


Kahoot!

Group practical

In a new scala worksheet use conditions to calculate somebody's grade given the percentage they scored.

90%+ = "A"
80%+ = "B"
70%+ = "C"
60%+ = "D"
50%+ = "E"
49%- = “F”

```
if(weather == "Cold"){
     //take a coat
} else if(weather == "Raining"){
     //take an umbrella
} else {
     //take nothing
}
```

Practical

In a new scala worksheet use conditions to work out for a given age, what film ratings they are able to see. Make sure your conditions are exhaustive and any age will get a response (what if they're younger than 4, what if they're exactly 8).

4+ = "U"
8+ = "PG"
12+ = "12A"
15+ = "15"
18+ = "18"

```
if(weather == "Cold"){
     //take a coat
} else if(weather == "Raining"){
     //take an umbrella
} else {
     //take nothing
}
```

Stretch task: Instead of just returning 1 film rating, can you return all of them? (hint: you'll need to look online to find out how to store/return multiple values)

Scala Functions

What are functions?
Functions Vs Methods
Methods
Functions
Where's the `return` statement!?!
Best Practices

What are functions?

Functions, sometimes called methods, allow you to define specific steps once so they can be reused in multiple places by referring to a name you have assigned to these set of instructions (similar to how we reference store values).

This avoids constantly writing instructions every time we need to perform an action.

What are functions?

A good analogy is your friend/partner/relative/nemesis (delete as appropriate) asks you to make them a cheese sandwich. Each time they ask, do they say "Walk into the kitchen, take the bread out of the bread bin, take the cheese out of the fridge, take 2 slices of bread from the bread packaging, cut a slice of che...." you get the idea **or** do they simply say "Oi! Make me a cheese sandwich"? They can talk to you like that, they've known you for ages and/or they're your nemesis.

In the above analogy "make me a cheese sandwich" is the name of the function and the individual steps (Walk into the kitchen...) are the defined steps.

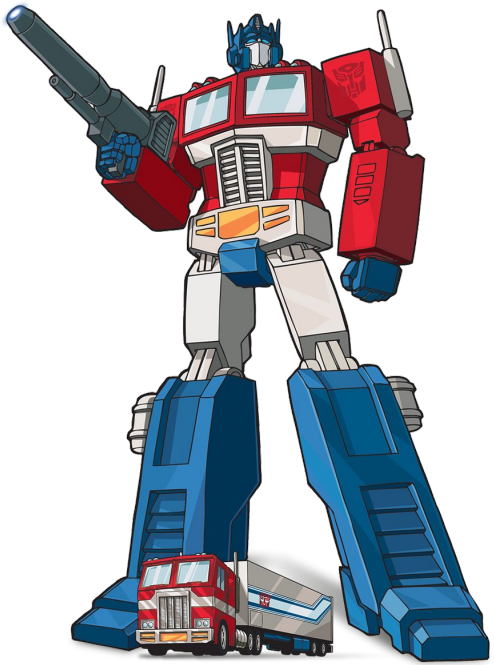
Operators (functions in disguise)

All of the operators that we've seen are actually just **functions**. For example:

- `val x = 1.+(2)`
  `x: Int = 3`

- `val y = true.&&(true)`
  `y: Boolean = true`

- `val z = “hello”.==(“hi”)`
  `z: Boolean = false`



Operators

We can reproduce this by defining the following class **Number** with the function **add**.

Now if we define a number we can perform the function **add** either with the dot notation we just saw or using spaces as we used when we first saw operators.

```
class Number(num: Int) {

  def add(otherNum: Int): Int = {

    num + otherNum

  }

}
```

```
val x = new Number(5)
val y = x.add(2)
y: Int = 7

val z = x add 2
z: Int = 7
```

Functions vs Methods

In Scala a function is no different to any other type such as **`Int`** or **`String`**, it can be declared and assigned to a **`val`** or passed to another function or be returned from a function or... you catch my drift (*hopefully*).

Methods are like functions but they are defined using the **`def`** keyword, are not passed around and are referred to by the name associated to the method.

Methods

A method is defined like so:

```
def nameOfMethod(<parameters, but this can be empty>): ReturnType = {

  <expression 1...N>

  <last expression results in a value that is of the same type as the ReturnType>

}
```

Expressions can be read as instructions or steps.
Each expression returns a result though you may choose to disregard it's output e.g. **`println("bob")`** returns a result of **`Unit`** and what are we supposed to do with that? Nothing, that's why you never see **`val x = println("bob")`**

Methods

An example of a method could be one that gives you the price of an item inc. VAT e.g.

```
def priceIncludingVat(price: BigDecimal): BigDecimal = {

    val vatRate = BigDecimal(1.2) // expression

    price * vatRate // last expression results in a BigDecimal

}
```

Methods

Because the below method only has one expression, we can re-write the VAT method like this:

```
def priceIncludingVat(price: BigDecimal): BigDecimal =

  price * BigDecimal(1.2)
```

Methods

We could, if we wanted to, omit the return type in the shortened version of the method above as it can be inferred by the compiler. I left it in because it makes the code more readable and readability trumps *succinct* code every time!

The VAT method can be called simply by typing:

```
val computerExclVat = BigDecimal(1000)

val computerIncVat = priceIncludingVat(computerExclVat) // results in BigDecimal(1200)
```

Best practices - pure functions

A pure function is one that has no side effects. That's just a fancy way of saying it takes inputs and returns a value based on those inputs and whilst processing the inputs does not change the state of the application.

Best practices - pure functions

An example of an impure function:

```
var sentence: String = ""

def append(word: String): String = {
  sentence = sentence + " " + word
  sentence.trim()
}

val a = append("Allo") // "Allo"
val b = append("Allo") // "Allo Allo"
```

Best practices - pure functions

An example of a pure function:

```
val sentence: String = ""

def append(sentence:String, word: String): String =
  sentence + " " + word.trim()

val a = append(sentence, "Allo") // "Allo"
val b = append(sentence, "Allo") // "Allo"
```

Best practices - Referential transparency

Referential transparency simply means a function's return value is based solely on the inputs. For example, given inputs **1** and **1**, the **add** function will always return a result of **2**. This means every reference to **`add(1, 1)`** can be replaced with **2** and the application would continue to function as it did prior to the the function to result replacement.


Kahoot!

Recap

Simple functions that only take one input can be called without the dot to look more like natural statement e.g. `1 + 1`

Functions are just grouped steps accessible by a meaningful name e.g. `makeMeSomeFood()`

Pure Functions: Given the same input(s) the output will **always** be the same

Group practical

Let's create a method called **`getBigVal`** that takes in two **`Int`**egers called **`input1`** and **`input2`** and returns a **`String`**.

The method should return the value **"`first`"** if **`input1`** is greater (**`>`**) than **`intput2`**.

The method should return the value **"`second`"** if **`input1`** is smaller (**`<`**) than **`input2`**.

The method should return the value **"`same`"** if **`input1`** is equal to (**`==`**) than **`input2`**.

Practical

Create a method called **`nameLength`** that takes in two **`String`**s called **`firstName`** and **`surname`** and returns an **`Int`**eger.

The result of the method should return the length/**`size`** of the name which is greater than the other. If both names are the same length return **`0`**.

Test Data

First Name	Surname	Result
Arnold	Schwarzenegger	14
Bruce	Lee	5
Ethan	Hawke	0

Scope

What is scope

Examples of how scope works

Scope - introduction

Scope is the term used for what information is available to a class, object, trait, function etc.

Scopes are hierarchical, and any scope can contain many other scopes within it. If the compiler can’t find the information it is looking for in the current scope, it will look in its parent scope for that information. The compiler will always take information from the closest scope it can find.

Scope - example

Numbers.scala

```
package maths

trait Numbers {
  val one = 1
  val two = 2
  val three = 3

  def addOneAndThree: Int = {
    one + three
  }

  def addNumAndTwo(num: Int): Int = {
    num + two
  }
}
```

Subtractions.scala

```
package maths

object Subtractions extends Numbers {
  def subtractNumFromFour(num: Int) = {
    addOneAndThree - num
  }
}
```

Multiplications.scala

```
package maths

object Multiplications {
  val three = 3
  def multiplyNumAndThree(num: Int) = {
    num * Subtractions.three
  }
  def multiplyTwoAndThree = {
    import Subtractions._
    two * three
  }
}
```

Scope - example

Numbers.scala

```
package maths

trait Numbers {
  val one = 1
  val two = 2
  val three = 3

  def addOneAndThree: Int = {
    one + three
  }

  def addNumAndTwo(num: Int): Int = {
    num + two
  }
}
```

Subtractions.scala

```
package maths

object Subtractions extends Numbers {
  def subtractNumFromFour(num: Int) = {
    addOneAndThree - num
  }
}
```

Multiplications.scala

```
package maths

object Multiplications {
  val three = 3
  def multiplyNumAndThree(num: Int) = {
    num * Subtractions.three
  }
  def multiplyTwoAndThree = {
    import Subtractions._
    two * three
  }
}
```

Scope - example

Numbers.scala

```
package maths

trait Numbers {
  val one = 1
  val two = 2
  val three = 3

  def addOneAndThree: Int = {
    one + three
  }

  def addNumAndTwo(num: Int): Int = {
    num + two
  }
}
```

Subtractions.scala

```
package maths

object Subtractions extends Numbers {
  def subtractNumFromFour(num: Int) = {
    addOneAndThree - num
  }
}
```

Multiplications.scala

```
package maths

object Multiplications {
  val three = 3
  def multiplyNumAndThree(num: Int) = {
    num * Subtractions.three
  }
  def multiplyTwoAndThree = {
    import Subtractions._
    two * three
  }
}
```

Scope - example

Numbers.scala

```
package maths

trait Numbers {
  val one = 1
  val two = 2
  val three = 3

  def addOneAndThree: Int = {
    one + three
  }

  def addNumAndTwo(num: Int): Int = {
    num + two
  }
}
```

Subtractions.scala

```
package maths

object Subtractions extends Numbers {
  def subtractNumFromFour(num: Int) = {
    addOneAndThree - num
  }
}
```

Multiplications.scala

```
package maths

object Multiplications {
  val three = 3
  def multiplyNumAndThree(num: Int) = {
    num * Subtractions.three
  }
  def multiplyTwoAndThree = {
    import Subtractions._
    two * three
  }
}
```

Scope - example

Numbers.scala

```
package maths

trait Numbers {
  val one = 1
  val two = 2
  val three = 3

  def addOneAndThree: Int = {
    one + three
  }

  def addNumAndTwo(num: Int): Int = {
    num + two
  }
}
```

Subtractions.scala

```
package maths

object Subtractions extends Numbers {
  def subtractNumFromFour(num: Int) = {
    addOneAndThree - num
  }
}
```

Multiplications.scala

```
package maths

object Multiplications {
  val three = 3
  def multiplyNumAndThree(num: Int) = {
    num * Subtractions.three
  }
  def multiplyTwoAndThree = {
    import Subtractions._
    two * three
  }
}
```

Scope - example

Numbers.scala

```
package maths

trait Numbers {
  val one = 1
  val two = 2
  val three = 3

  def addOneAndThree: Int = {
    one + three
  }

  def addNumAndTwo(num: Int): Int = {
    num + two
  }
}
```

Subtractions.scala

```
package maths

object Subtractions extends Numbers {
  def subtractNumFromFour(num: Int) = {
    addOneAndThree - num
  }
}
```

Multiplications.scala

```
package maths

object Multiplications {
  val three = 3
  def multiplyNumAndThree(num: Int) = {
    num * Subtractions.three
  }
  def multiplyTwoAndThree = {
    import Subtractions._
    two * three
  }
}
```

Recap

Scopes can be nested

If not in the immediate scope scala will look for a val/function/etc in the parent scope or it’s parent scope etc etc.

Declaring a val/function/etc in the current scope that exists higher up in the hierarchy will take precedence though this can sometimes cause confusion and you may see warning relating to shadowing.

Group practical

```
package maths

object Calculator {
  val one = 1
  val two = 2
  val three = 3

  println(one + three)
}
```

What number is printed?

Group practical

```
package maths

object Numbers {
  val x = 10
  val y = 20
  val z = 30
}
```

```
package maths

object Calculator {

  import Numbers._

  println(x + y)
}
```

What number is printed?

Group practical

```
package maths

object Numbers {
  val x = 10
  val y = 20
  val z = 30
}
```

```
package maths

object Calculator {

  import Numbers._

  def addXAndY(x: Int, y: Int) = {
    x + Numbers.y
  }

  println(addXAndY(1, 2))
}
```

What number is printed?

Group practical

```
package maths

object Numbers {
  val x = 10
  val y = 20
  val z = 30
}
```

```
package maths

object Calculator {

  def addXAndY(x: Int, y: Int) = {
    import Numbers._
    x + y
  }

  println(addXAndY(1, 2))
}
```

What number is printed?

Group practical

```
package maths

object Numbers {
  val x = 10
  val y = 20
  val z = 30
}
```

```
package maths

object Calculator {

  def addXAndY(x: Int, y: Int) = {
    import Numbers._
    x + y
  }

  println(addXAndY(1, 2))
}
```

This actually won't compile!
There are two variables x in the same scope
Look out for '*reference to x is ambiguous*'

Group practical

```
package maths

object Numbers {
  val x = 10
  val y = 20
  val z = 30
}
```

```
package maths

object Calculator {

  import Numbers._

  def addXAndY(x: Int, y: Int) = {
    val x = 100
    x + Numbers.y
  }

  println(addXAndY(1, z))
}
```

What number is printed?

Practical

1. Go back to your boats exercise.
2. Can you repurpose your racing function:
    a. to sit inside the Boat class
    b. rename it to be something appropriate like 'isFasterThan'
    c. for a parameter it will only need another boat to compare it against
    d. it should return a boolean
3. Add a function to your pirate ships to make them fight eg 'canOutgun'
4. Get creative!! What other useful functions can you add to your traits and classes? It doesn't need to be sensible

Unit testing

What is unit testing?
ScalaTest
TDD

Unit testing

Unit testing is the process in which the smallest testable components of a program are individually tested for correctness.

The tests only include checking characteristics of the particular piece of code, or "Unit", being tested.

Unit testing

For any examples we'll be using the ScalaTest library. In ScalaTest, tests are written as a sentence that specifies a piece of expected behaviour, and a block of code that performs the test.

Unit testing

```
import org.scalatest._

class TwoNumbers(x: Int, y: Int) {

     def add(): Int = {
          x + y
     }

}
```

```
import org.scalatest._

class TwoNumbersSpec extends FlatSpec {

     "add" should "add numbers" in {
          val nums = new TwoNumbers(1, 2)
          assert(nums.add === 3)
     }

}
```

In the example we have a subject, "TwoNumbers" followed by the verb "should", then the tests to perform on the subject.

Unit testing

Naming convention for test classes is the same name as the file you're testing followed by "Spec".

We're extending FlatSpec, in most projects you'll have your own base "Spec" to extend but for here FlatSpec is suitable.

assert is the part necessary for a test to pass/fail. It takes one parameter, a boolean that has to evaluate to **`true`** for the test to pass.

Other keywords can be used in place of assert, commonly "Should Matchers" or "Must Matchers" which can offer further functionality and make it more readable.

```
import org.scalatest._

class TwoNumbersSpec extends FlatSpec {

     "add" should "add numbers" in {
          val nums = new TwoNumbers(1, 2)
          assert(nums.add === 3)
     }

}
```

Unit testing

```
"add" should "add numbers" in {...}
```

In this line we're specifying the subject to be "add", then we're describing a piece of functionality for that subject "add numbers".

Anything in the block of code between the curly brackets will be run for this test and as mentioned, the assert line will determine the pass/failure status of this test. We have this same style for the remaining functions, defining the subject and the expected functionality.

```
import org.scalatest._

class TwoNumbersSpec extends FlatSpec {

    "add" should "add numbers" in {
        val nums = new TwoNumbers(1, 2)
        assert(nums.add === 3)
    }

}
```

Test Driven Development (TDD)

Test driven development is a development style that uses short cycles of development.

1. Produce unit test cases for desired functionality, this will fail as the implementation doesn't exist.

2. Write code to pass the test cases.

3. Refactor the code to the quality desired.

This process is also called **red, green, refactor**. When the test fails the text is highlighted in red, when you write the code to pass the test it is green and finally, you refactor your code.

TDD ensures that tests are written to specification, meaning code is developed to meet expectations, rather than potentially tests being developed to work for your code allowing bugs to creep in.

Test Driven Development (TDD)

Scala offers dummy implementations to help with TDD. When writing functions you don't to supply an implementation and can leave the signature such that it can be tested and will compile whilst failing the tests.

```
def addTwo(x: Int, y: Int): Int = ???
```

When your test executes the above code it will throw a NotImplementedError. This means you can provide a function signature, then write your test case to get failed tests, then progressively implement the function until your tests all pass.


Kahoot!

Recap

Unit tests test a unit of code only

A unit of code should be no larger than a class/object/trait

Unit tests typically only test public functions

TDD is the process by which tests code is only written to pass a failing test.

Code developed using TDD must be as simple as possible and would evolve into something more complex as more tests are added.

Refactoring is a step often missed but one of the most important

Group practical

1. Create a function called **`fizzBuzz`** that takes an **`Int`** and returns the same value as a **`String`**
2. If the input is divisible by **3** (with no remainder) return the word **“`Fizz`”**
3. If the input is divisible by **5** (with no remainder) return the word **“`Buzz`”**
4. If the input is divisible by **3 and 5** (with no remainder) return the word **“`FizzBuzz`”**

Flow Reminder

1. Write a breaking test(s)
    - A test that doesn’t compile because the subject doesn't yet exist is a breaking test
2. Write just enough code to pass the test
    - Code doesn't have to be beautiful code
3. Refactor code
    - make it prettier
    - break it up **if required**

Practical

1. Create a function called **`calculateTax`** that takes an **`Int`** and returns 10% of the total value
2. If the input greater than 10,000, instead return 15% of the total value
3. If the input is greater than 50,000, instead return 20% of the total value
4. If the input is greater than 100,000, instead return 40% of the total value

Flow Reminder

1. Write a breaking test(s)
    - A test that doesn’t compile because the subject doesn't yet exist is a breaking test
2. Write just enough code to pass the test
    - Code doesn't have to be beautiful code
3. Refactor code
    - make it prettier
    - break it up **if required**

Collections

What are collections

How to access data in a collection

Manipulating collections with map, filter, exists

Collections

More than one item of data can be stored in one of a number of different collections.

Examples of collections:

- `Seq: (1, 2, 3, 4, 5)`
- `List: (1, 2, 3, 4, 5)`
- `Map: (“bananas” -> 200, “apples” -> 100, “oranges” -> 400)`

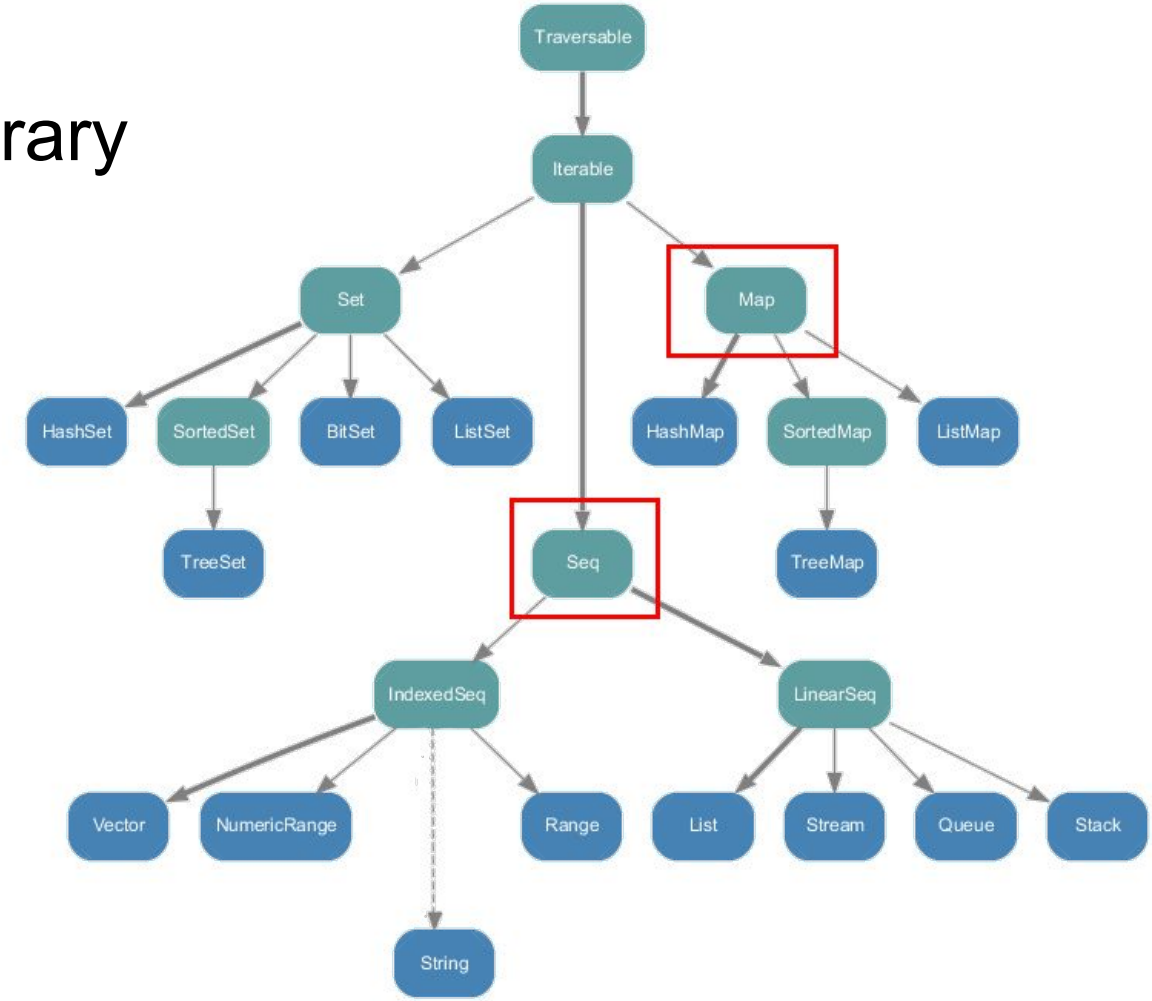
Creating collections:

```
val x: List[Int] = List(1, 2, 3)

val y: Seq[Int] = Seq(1, 2, 3)

val z: Map[String, Int] = Map(“num1” -> 3, “num2”
-> 4, “num43” -> 30123)
```

Collections Library



Accessing data in collections

```
// Seq/List

val testSeq: Seq[String] = Seq("a", "b", "c")

val letter = testSeq(0)
letter: String = "a"

val failedLetter = testSeq(10)
java.util.NoSuchElementException

// Map

val testMap: Map[Int, String] = Map(10 -> "dog", 11 -> "cat", 12 -> "leopard")

val animal = testMap(10)
animal: String = "dog"

val failedAnimal = testMap(13)
java.util.NoSuchElementException

val animalOpt = testMap.get(13)
animalOpt: Option[String] = None
```

Things to do on rainy days with collections - map

map: iterate through every item in a collection and perform a function on it

```
val testSeq = Seq(1, 2, 3)
val doubledSeq = testSeq.map {
  num => num * 2
}
doubledSeq: Seq[Int] = Seq(2, 4, 6)
```

In the example above, mapping through testSeq takes every value in testSeq and doubles it. ‘num’ is simply a name that we have chosen by which to reference each member of testSeq.

Note: Don’t get confused between Map and map! A ‘Map’ is a Collection with key/value pairs and ‘map’ is an operator to move through a Collection and transform its members.

Yes, you can map a Map...

Things to do on rainy days with collections - filter

filter: iterate through every item in a collection and remove it if a condition is not met

```
val testSeq = Seq(1, 2, 3, 4, 5)
val filteredSeq = testSeq.filter {
  num => num > 3
}
filteredSeq: Seq[Int] = Seq(4, 5)
```

Here we perform the same process as when using map, but the function on each member must evaluate to *true* or *false.*

Only the values for which the function evaluates to *true* are kept. The rest are discarded.

Things to do on rainy days with collections - exists

exists: iterate through every item in a collection and return *true* if a condition is met at least once

```
val testSeq = Seq(1, 3, 5)
val outcome = testSeq.exists {
  num => num > 3
}
outcome: Boolean = true
```

Here we perform the same iterating process again with a Boolean outcome function as in *filter.* However, as soon as the function evaluates to *true* with one of the members of testSeq, *exists* evaluates to *true*.

If none of the members of testSeq were to evaluate to *true* through the function, *exists* would evaluate to *false.*


Kahoot!

Group practical

1. Create a Seq of the names of everybody on the course
2. Create a Map of 1-“red”, 2-“yellow”, 3-“blue”, 4-“refrigerator” and play with accessing the elements
3. write a function to add 1 to all numbers in a Seq
4. write a function to remove all even numbers from a collection
5. write a function that returns true if a Seq has a String that contains the letter “t” (to check if a string contains a character use .contains("t"))

Practical

1. Create a Seq of the numbers 1 to 11
2. Create a Map of “rose”-“red”, “sunflower”-“yellow”, “violet”-“blue”, and play with accessing the elements
3. write a function to double the values of all numbers in a Seq
4. write a function to strip all Strings from a collection except those that contain the letter “y” (to check if a string contains a character use .contains("y"))
5. write a function that returns true if a Seq has a number greater than 5 in it

Final practical

We're going to build a small application to model the 'Guess who?' board game.

It's just the backend we're building, there will be no frontend user interface. Instead we'll exercise our application through tests to gain confidence it works correctly.

...GO!!

Final practical

1. Where the f*** to start??? - list all the components/parts of a Guess Who game and find one that can be built/modelled without anything else.
2. What's the minimum we can put in our code to model it?
3. Now that we have our first bit of code and we've tested it and it is working, what can we do next?
4. What's the minimum we need to do that?
5. Now we've done that what's next etc. etc.

Eventually we should have an M.V.P, it's a very basic game, but it can be played. Now we have that, lets add some more functionality and test that, then add more etc. etc.

Final practical

Remember some of the key lessons we've learnt about developing something:

- Never move too far away from a working solution.
- Test frequently! TDD is recommended practice.
- Writing code is collaborative, talk to your neighbours, discuss ideas.
- Have fun with it!! Use your favourite TV/Film characters in your game.