Introducing Scala

Introducing Scala

Scala is a programming language

developed at EPFL, Switzerland in 2004

Scala was developed in an attempt to build

"a better Java"

"a better Java"

Scala is object-oriented like Java

but it is also functional

Object-oriented meets functional

"a better Java"

Now, you might say that this is redundant

Java 8 added functional programming support

Object-oriented meets functional

"a better Java"

But, functional programming is as integral to Scala as Object-orientedness

Object-oriented meets functional

First class functions

Pattern matching

Closures

Currying

and many other functional programming constructs are built-in

Object-oriented
meets functional

There are other programming languages that went down the road of "a better Java"

Not many people have heard of them :)

Object-oriented meets functional

Scala has gained a following though, especially among developers in Big Data and distributed computing domains

Object-oriented meets functional

First class functions
Closures
Pattern matching
Currying

"a better Java"

Scala's focus on functional programming made it a natural choice for distributed computing applications

Object-oriented meets functional

First class functions
Closures
Pattern matching
Currying

"a better Java"

Parallel processing concepts like map and reduce are fundamentally functional programming concepts

Object-oriented meets functional

First class functions
Closures
Pattern matching
Currying

"a better Java"

First class functions and closures allow you to ship code to different nodes in a cluster without worrying too much

Object-oriented meets functional

First class functions
Closures
Pattern matching
Currying

"a better Java"

While functional programming is the main difference between Scala and Java, Scala has a few other characteristics

"a better Java"

Object-oriented meets functional

Pure Object Oriented

Syntactically concise

Operator Overloading

Compiler + Interpreter

"a better Java"

Object-oriented meets functional

Pure Object Oriented

Syntactically concise

Operator overloading

Compiler + Interpreter

Everything in Scala is an object

"a better Java"

Object-oriented meets functional

Pure Object Oriented

The main implication of this is that there are no primitives and wrapper types in Scala

"a better Java"

Object-oriented meets functional

Pure Object Oriented

Strings, Integers, Doubles etc and all the primitive data types in Java are only available as objects

"a better Java"

Object-oriented meets functional

Pure Object Oriented

Syntactically concise

Operator Overloading

Compiler + Interpreter

Functions are also objects

"a better Java"

Object-oriented meets functional

Pure Object Oriented

Syntactically concise

Operator Overloading

Compiler + Interpreter

"a better Java"

Object-oriented meets functional

Syntactically concise

Many choices in Scala were made to make it syntactically more concise than Java

"a better Java"

Object-oriented meets functional

Syntactically concise

The Scala compiler can infer many things without being explicitly specified

"a better Java"

Object-oriented meets functional

Syntactically concise

This includes semi-colon inference, type inference, specifying arguments using a placeholder syntax etc

"a better Java"

Object-oriented meets functional

Syntactically concise

The implication is that Scala needs a lot fewer lines of code than Java does to do the same things

"a better Java"

Object-oriented meets functional

Syntactically concise

The flip side is that the readability of code suffers because many things are implicit

"a better Java"

Object-oriented meets functional

Pure Object Oriented

Syntactically concise

Operator Overloading

Compiler + Interpreter

"a better Java"

Object-oriented meets functional

Operator Overloading

Scala treats operators like methods for an object

Operator Overloading

As in C++, you can define operators and even overload them in Scala

"a better Java"

Object-oriented meets functional

Pure Object Oriented

Syntactically concise

Operator Overloading

Compiler + Interpreter

"a better Java"

Object-oriented meets functional

Compiler + Interpreter

Pure object oriented

Syntactical flexibility

Operator overloading

Scala is a rare example of a programming language that uses both a compiler and an interpreter

"a better Java"

Object-oriented meets functional

Compiler + Interpreter

Pure Object Oriented

Static Typing

Operator Overloading

Scala runs using a JVM and its code is compiled to bytecode

In fact, Scala has seamless interoperability with Java

"a better Java"

Object-oriented meets functional

Compiler + Interpreter

You can use all of Java's classes and libraries in Scala directly

"a better Java"

Object-oriented meets functional

Compiler + Interpreter

Pure object oriented

Syntactic abstraction

Operator overloading

Scala also has interoperability with JavaScript

The Scala compiler can compile to JavaScript

"a better Java"

Object-oriented meets functional

Compiler + Interpreter

Pure Object Oriented

Syntactically concise

Operator Overloading

In later versions Scala added an interpreter too

"a better Java"

Object-oriented meets functional

Compiler + Interpreter

Scala's interpreter can be used in an REPL to get fast feedback on the instructions being given

Object-oriented meets functional

Pure Object Oriented

Syntactically concise

Operator Overloading

Compiler + Interpreter

These are some of the main characteristics of Scala

FUNCTIONS VERSUS METHODS

THE WORDS "FUNCTIONS" AND "METHODS" ARE SOMETIMES USED INTERCHANGEABLY

IN SCALA, "FUNCTIONS" AND "METHODS" ARE FUNDAMENTALLY DIFFERENT

"FUNCTIONS" VS "METHODS"

"METHODS" ARE USED TO DEFINE OBJECT BEHAVIOR AND ARE CREATED LIKE THIS

```
def max(x:Int,y:Int): Int ={
 if(x>y)
    x
 else
    y
}
```

"FUNCTIONS" VS "METHODS"

```
def max(x:Int,y:Int): Int ={
  if(x>y)
     x
  else
     y
}
```

ANY METHOD CREATED THIS WAY IS PART OF A CLASS/OBJECT DEFINITION

"FUNCTIONS" VS "METHODS"

```
def max(x:Int,y:Int): Int ={
 if(x>y)
    x
 else
    y
}
```

THE METHOD SIGNATURE

"FUNCTIONS" VS "METHODS"

```
def max(x:Int,y:Int): Int ={
 if(x>y)
    x
 else
    y
}
```

THE VALUE RETURNED BY THE METHOD

"FUNCTIONS" VS "METHODS"

```
def max(x:Int,y:Int): Int ={
 if(x>y)
    x
 else
    y
}
```

THE CURLY BRACES ARE USED WHEN YOU HAVE TO EVALUATE MULTIPLE STATEMENTS BEFORE RETURNING A VALUE

"FUNCTIONS" VS "METHODS"

```
object HelloWorld {

  def main (args: Array[String]) = {

    println("Hello, this is SCALA!")

  }
}
```

THE MAIN METHOD FROM WHICH A PROGRAM STARTS IS DEFINED IN THIS WAY

"FUNCTIONS" VS "METHODS"

FUNCTIONS OR FUNCTION LITERALS ON THE OTHER HAND, ARE FUNDAMENTALLY DIFFERENT

"FUNCTIONS" VS "METHODS"

```
scala> (x:Int)=>x+1
res0: Int => Int = <function1>
```

"FUNCTION LITERAL" IS A NAME FOR A FUNCTION THAT IS CREATED LIKE THIS

"FUNCTION EXPRESSION" IS WHAT THE RIGHT HAND SIDE OF THIS ASSIGNMENT IS CALLED.

"FUNCTIONS" VS "METHODS"

```
scala> (x:Int)=>x+1
res0: Int => Int = <function1>
```

"FUNCTION LITERALS" / FUNCTIONS IN SCALA ARE ACTUALLY OBJECTS

"FUNCTIONS" VS "METHODS"

```
scala> (x:Int)=>x+1
res0: Int => Int = <function1>
```

THERE ARE A SET OF FUNCTION TYPES IN SCALA THAT ALL DESCEND FROM THE FUNCTION TRAIT

Traits are like Interfaces in Scala

"FUNCTIONS" VS "METHODS"

```
scala> (x:Int)=>x+1
res0: Int => Int = <function1>
```

THE SCALA COMPILER WILL INFER WHICH FUNCTION TYPE TO PICK BASED ON THE NUMBER OF ARGUMENTS

"FUNCTIONS" VS "METHODS"

```
scala> (x:Int)=>x+1
res0: Int => Int = <function1>
```

FUNCTION1 MEANS THAT THIS IS A FUNCTION OBJECT WITH 1 ARGUMENT

"FUNCTIONS" VS "METHODS"

```
scala> (x:Int)=>x+1
res0: Int => Int = <function1>
```

FUNCTION2 REPRESENTS FUNCTIONS WITH 2 ARGUMENTS

..and so on till Function23

"FUNCTIONS" VS "METHODS"

```
scala> (x:Int)=>x+1
res0: Int => Int = <function1>
```

THIS ACTUALLY MEANS THAT FUNCTION OBJECTS CAN ONLY HAVE 23 ARGUMENTS AT THE MOST

Function1 - Function23

FUNCTION OBJECTS

IN SCALA CAN BE TREATED JUST LIKE ANY OTHER OBJECTS

FUNCTION OBJECTS IN SCALA CAN BE TREATED JUST LIKE ANY OTHER OBJECTS

YOU CAN STORE A FUNCTION IN A VARIABLE

FUNCTION OBJECTS IN SCALA CAN BE TREATED JUST LIKE ANY OTHER OBJECTS

YOU CAN STORE A FUNCTION IN A VARIABLE

YOU CAN HAVE A METHOD RETURN A FUNCTION

FUNCTION OBJECTS IN SCALA CAN BE TREATED JUST LIKE NUMBERS OR STRINGS

YOU CAN STORE A FUNCTION IN A VARIABLE

YOU CAN HAVE A METHOD RETURN A FUNCTION

YOU CAN HAVE A METHOD TAKE IN A FUNCTION AS AN ARGUMENT

FUNCTION OBJECTS IN SCALA CAN BE TREATED JUST LIKE NUMBERS OR STRINGS

1. YOU CAN STORE A FUNCTION IN A VARIABLE
2. YOU CAN HAVE A METHOD RETURN A FUNCTION
3. YOU CAN HAVE A METHOD TAKE IN A FUNCTION AS AN ARGUMENT

THESE 3 PROPERTIES COLLECTIVELY ARE CALLED "FIRST CLASS FUNCTIONS"

FUNCTION OBJECTS IN SCALA CAN BE TREATED JUST LIKE NUMBERS OR STRINGS

YOU CAN STORE A FUNCTION IN A VARIABLE

RETURN A FUNCTION

YOU CAN HAVE A METHOD TAKE IN A FUNCTION AS AN ARGUMENT

THESE 3 PROPERTIES COLLECTIVELY ARE CALLED "FIRST CLASS FUNCTIONS"

SCALA IS A PROGRAMMING LANGUAGE THAT SUPPORTS FIRST CLASS FUNCTIONS

SCALA IS A PROGRAMMING LANGUAGE
THAT SUPPORTS

FIRST CLASS FUNCTIONS

IN THIS WAY, SCALA HAS A STRONG
FUNCTIONAL PROGRAMMING FLAVOUR TO IT.

FUNCTION OBJECTS IN SCALA CAN BE TREATED JUST LIKE NUMBERS OR STRINGS

1. YOU CAN STORE A FUNCTION IN A VARIABLE
2. YOU CAN HAVE A METHOD RETURN A FUNCTION
3. YOU CAN HAVE A METHOD TAKE IN A FUNCTION AS AN ARGUMENT

THESE 3 PROPERTIES COLLECTIVELY ARE CALLED "FIRST CLASS FUNCTIONS"

1 YOU CAN STORE A FUNCTION IN A VARIABLE

```
scala> var increase = (x:Int)=>x+1
increase: Int => Int = <function1>
```

You can create a variable to represent a function

1

YOU CAN STORE A FUNCTION IN A VARIABLE

```
scala> var increase = (x:Int)=>x+1
increase: Int => Int = <function1>
```

This is a function that adds 1 to it's input

1

YOU CAN STORE A FUNCTION IN A VARIABLE

```
scala> var increase = (x:Int)=>x+1
increase: Int => Int = <function1>
```

The input to the function

1 YOU CAN STORE A FUNCTION IN A VARIABLE

```
scala> var increase = (x:Int)=>x+1
increase: Int => Int = <function1>
```

The => designates that this function converts the thing on the left (any integer x) to the thing on the right (x + 1).

1 YOU CAN STORE A FUNCTION IN A VARIABLE

```
scala> var increase = (x:Int)=>x+1
increase: Int => Int = <function1>
```

```
scala> increase(10)
res1: Int = 11
```

1

YOU CAN STORE A FUNCTION IN A VARIABLE

```
scala> var increase=(x:Int)=>{
     |   println("hi")
     |   x+3
     | }
increase: Int => Int = <function1>
```

```
scala> increase(4)
hi
res0: Int = 7
```

If the function needs multiple statements, place them inside curly braces

1

YOU CAN STORE A FUNCTION IN A VARIABLE

Storing methods as variables

1

YOU CAN STORE A FUNCTION IN A VARIABLE

Storing methods as variables

We've already said that functions and methods are different

YOU CAN STORE A FUNCTION IN A VARIABLE 1

Storing methods as variables

But you can create a function object using a method

```
val maxfnobject = (x: Int ,y: Int) => max(x,y)
```

YOU CAN STORE A FUNCTION IN A VARIABLE

1

Storing methods as variables

```
val maxfnobject = (x: Int ,y: Int) => max(x,y)
```

Calls the method max for the function inputs x,y

YOU CAN STORE A FUNCTION IN A VARIABLE 1

Storing methods as variables

```
val maxfnobject = max(_: Int,_ :Int)
```

Scala provides a more concise way of saying this using the placeholder syntax

1

YOU CAN STORE A FUNCTION IN A VARIABLE

Storing methods as variables

```
val maxfnobject = max(_: Int,_ :Int)
```

The _ act as blanks that will be filled in by the method's arguments

YOU CAN STORE A FUNCTION IN A VARIABLE 1

Storing methods as variables

```
val maxfnobject = max(_: Int,_ :Int)
```

Each _ represents a different argument

YOU CAN STORE A FUNCTION IN A VARIABLE 1

Storing methods as variables

```
val maxfnobject = max(_: Int,_ :Int)
```

This syntax makes it seem as if we can store a method as a variable

YOU CAN STORE A FUNCTION IN A VARIABLE 1

Storing methods as variables

```
val maxfnobject = max(_: Int,_ :Int)
```

But the Scala compiler understands that this is a function object created using a method

```
(x: Int ,y: Int) => max(x,y)
```

Storing methods as variables

1

YOU CAN STORE A FUNCTION IN A VARIABLE

```
val maxfnobject = max(_: Int,_ :Int)
```

It will pick the right function type to represent it (Function2)

1

YOU CAN STORE A FUNCTION IN A VARIABLE

Partially applied functions

YOU CAN STORE A FUNCTION IN A VARIABLE 1

Partially applied functions

Say you have a method with 3 arguments

```
scala> def sum(a: Int, b: Int, c: Int) = a + b + c
sum: (a: Int, b: Int, c: Int)Int

scala> sum(1, 2, 3)
res8: Int = 6
```

YOU CAN STORE A FUNCTION IN A VARIABLE

1

Partially applied functions

```
scala> def sum(a: Int, b: Int, c: Int) = a + b + c
sum: (a: Int, b: Int, c: Int)Int

scala>  val b = sum(1, _: Int, 3)
b: Int => Int = <function1>
```

You can create a function variable with some arguments applied

Partially applied functions

1 YOU CAN STORE A FUNCTION IN A VARIABLE

```
scala> def sum(a: Int, b: Int, c: Int) = a + b + c
sum: (a: Int, b: Int, c: Int)Int

scala>  val b = sum(1, _: Int, 3)
b: Int => Int = <function1>

scala> b(3)
res10: Int = 7
```

Since only one argument is missing, the Scala compiler generates a new function object that takes one argument

FUNCTION OBJECTS IN SCALA CAN BE TREATED JUST LIKE NUMBERS OR STRINGS

1. YOU CAN STORE A FUNCTION IN A VARIABLE
2. YOU CAN HAVE A METHOD RETURN A FUNCTION
3. YOU CAN HAVE A METHOD TAKE IN A FUNCTION AS AN ARGUMENT

THESE 3 PROPERTIES COLLECTIVELY ARE CALLED "FIRST CLASS FUNCTIONS"

SUCH A METHOD IS CALLED "A HIGHER ORDER METHOD"

1

YOU CAN STORE A FUNCTION IN A VARIABLE

2

YOU CAN HAVE A METHOD RETURN A FUNCTION

3

YOU CAN HAVE A METHOD TAKE IN A FUNCTION AS AN ARGUMENT

THESE 3 PROPERTIES COLLECTIVELY ARE CALLED "FIRST CLASS FUNCTIONS"

YOU CAN HAVE A METHOD RETURN A FUNCTION

SAY YOU WANTED TO PRINT A GREETING TO A USER

"HELLO SWETHA"

"NAMASTE JANANI"

"BONJOUR VITTHAL"

YOU CAN HAVE A METHOD RETURN A FUNCTION

YOU WANT THE GREETING TO CHANGE BASED ON THE LANGUAGE OF THE USER

"HELLO SWETHA"	ENGLISH
"NAMASTE JANANI"	HINDI
"BONJOUR VITTHAL"	FRENCH

YOU CAN HAVE A METHOD RETURN A FUNCTION

HAVE A METHOD RETURN A FUNCTION BASED ON THE LANGUAGE!

```
def greeting(lang: String)= {
  lang match {
     case "English" => (x: String) => println("Hello "+x)
     case "Hindi" => (x: String) => println("Namaste "+x)
     case "French" => (x: String) => println("Bonjour "+x)
     case "Spanish" => (x: String) => println("Hola "+x)
  }
```

YOU CAN HAVE A METHOD RETURN A FUNCTION

MATCH IS SIMILAR TO SWITCH IN JAVA

```
def greeting(lang: String)= {
  lang match {
     case "English" => (x: String) => println("Hello "+x)
     case "Hindi" => (x: String) => println("Namaste "+x)
     case "French" => (x: String) => println("Bonjour "+x)
     case "Spanish" => (x: String) => println("Hola "+x)
  }
```

YOU CAN HAVE A METHOD RETURN A FUNCTION

THIS IS A FUNCTION OBJECT

```
def greeting(lang: String)= {

  lang match {

    case "English" => (x: String) => println("Hello "+x)

    case "Hindi" => (x: String) => println("Namaste "+x)

    case "French" => (x: String) => println("Bonjour "+x)

    case "Spanish" => (x: String) => println("Hola "+x)

  }
```

YOU CAN HAVE A METHOD RETURN A FUNCTION

IT TAKES A STRING AND PRINTS A GREETING TO SCREEN

```
def greeting(lang: String)= {
  lang match {
    case "English" => (x: String) => println("Hello "+x)
    case "Hindi" => (x: String) => println("Namaste "+x)
    case "French" => (x: String) => println("Bonjour "+x)
    case "Spanish" => (x: String) => println("Hola "+x)
  }
```

YOU CAN HAVE A METHOD RETURN A FUNCTION

FOR EACH LANGUAGE, RETURN A FUNCTION THAT PRINTS THE APPROPRIATE GREETING

```
def greeting(lang: String)= {

  lang match {

    case "English" => (x: String) => println("Hello "+x)

    case "Hindi" => (x: String) => println("Namaste "+x)

    case "French" => (x: String) => println("Bonjour "+x)

    case "Spanish" => (x: String) => println("Hola "+x)

  }
```

YOU CAN HAVE A METHOD RETURN A FUNCTION

```
def greeting(lang: String)= {
  lang match {
    case "English" => (x: String) => println("Hello "+x)
    case "Hindi" => (x: String) => println("Namaste "+x)
    case "French" => (x: String) => println("Bonjour "+x)
    case "Spanish" => (x: String) => println("Hola "+x)
```

```
def main (args: Array[String]){

  val greetEnglish = greeting("English")
  greetEnglish("Swetha")

  val greetSpanish = greeting("Spanish")
  greetSpanish("Janani")

}
```

YOU CAN GET THE APPROPRIATE FUNCTION BASED ON THE LANGUAGE OF THE USER

YOU CAN HAVE A METHOD RETURN A FUNCTION

```
def greeting(lang: String)= {
  lang match {
    case "English" => (x: String) => println("Hello "+x)
    case "Hindi" => (x: String) => println("Namaste "+x)
    case "French" => (x: String) => println("Bonjour "+x)
    case "Spanish" => (x: String) => println("Hola "+x)

def main (args: Array[String]){
    val greetEnglish = greeting("English")
    greetEnglish("Swetha")

    val greetSpanish = greeting("Spanish")
    greetSpanish("Janani")

}
```

GREETENGLISH IS A FUNCTION OBJECT FOR GREETING ENGLISH SPEAKING USERS

YOU CAN HAVE A METHOD RETURN A FUNCTION

```
def greeting(lang: String)= {
  lang match {
    case "English" => (x: String) => println("Hello "+x)
    case "Hindi" => (x: String) => println("Namaste "+x)
    case "French" => (x: String) => println("Bonjour "+x)
    case "Spanish" => (x: String) => println("Hola "+x)
 def main (args: Array[String]){
    val greetEnglish = greeting("English")
    greetEnglish("Swetha")
    val greetSpanish = greeting("Spanish")
    greetSpanish("Janani")
 }
```

CALL THE GREETING METHOD AND STORE THE VALUE RETURNED IN GREETENGLISH

YOU CAN HAVE A METHOD RETURN A FUNCTION

```
def greeting(lang: String)= {
  lang match {
    case "English" => (x: String) => println("Hello "+x)
    case "Hindi" => (x: String) => println("Namaste "+x)
    case "French" => (x: String) => println("Bonjour "+x)
    case "Spanish" => (x: String) => println("Hola "+x)
```

```
def main (args: Array[String]){
  val greetEnglish = greeting("English")
  greetEnglish("Swetha")
  val greetSpanish = greeting("Spanish")
  greetSpanish("Janani")
}
```

USE IT TO PRINT THE APPROPRIATE GREETING FOR ALL ENGLISH SPEAKING USERS

YOU CAN HAVE A METHOD RETURN A FUNCTION

```
def greeting(lang: String)= {
  lang match {
    case "English" => (x: String) => println("Hello "+x)
    case "Hindi" => (x: String) => println("Namaste "+x)
    case "French" => (x: String) => println("Bonjour "+x)
    case "Spanish" => (x: String) => println("Hola "+x)
  }

def main (args: Array[String]){

  val greetEnglish = greeting("English")
  greetEnglish("Swetha")

  val greetSpanish = greeting("Spanish")
  greetSpanish("Janani")

}
```

IF THE USER IS SPANISH - GET A DIFFERENT FUNCTION AND USE IT

SUCH A METHOD IS CALLED "A HIGHER ORDER METHOD"

1

YOU CAN STORE A FUNCTION IN A VARIABLE

2

YOU CAN HAVE A METHOD RETURN A FUNCTION

3

YOU CAN HAVE A METHOD TAKE IN A FUNCTION AS AN ARGUMENT

THESE 3 PROPERTIES COLLECTIVELY ARE CALLED "FIRST CLASS FUNCTIONS"

YOU CAN HAVE A METHOD TAKE IN A FUNCTION AS AN ARGUMENT

LET'S LOOK AT COLLECTIONS IN SCALA

Collections

Collections

Scala has quite a few built-in collection data types

List

Set

Queue

Array

Stack etc

Tuple

Map

Collections

Array

Tuple

For now, let's concentrate on these 2 commonly used collection data types

Collections

Array

An Array is a homogenous collection of objects

Array[String], Array[Int] etc

Collections

Array

Array[String],
Array[Int] etc

Arrays are normally used for collections whose size is known

```
scala> val fiveInts = new Array[Int](5)
fiveInts: Array[Int] = Array(0, 0, 0, 0, 0)
```

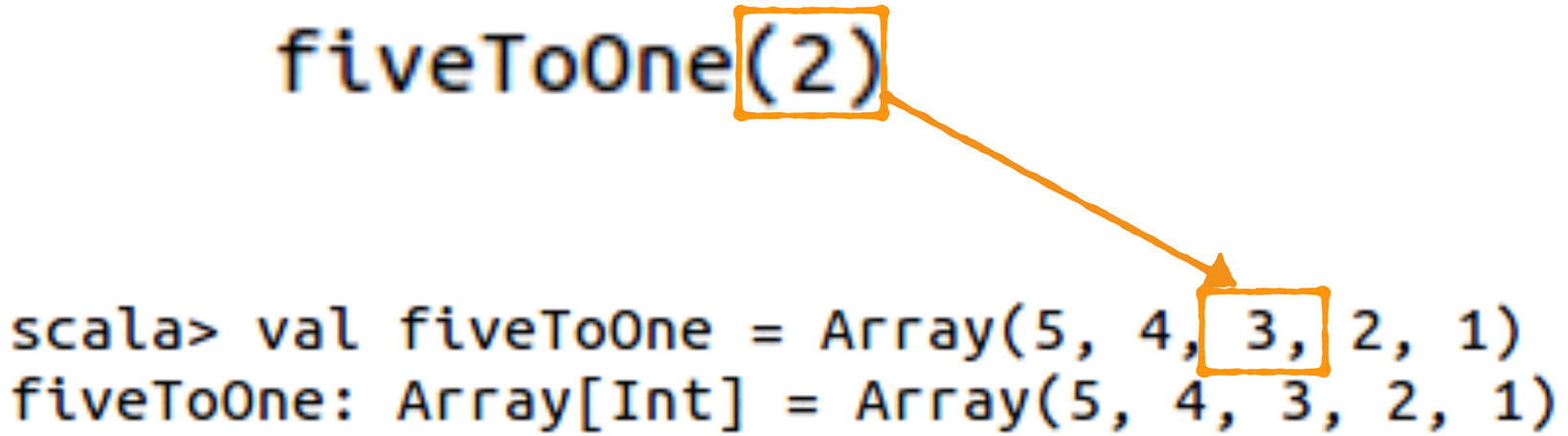
```
scala> val fiveToOne = Array(5, 4, 3, 2, 1)
fiveToOne: Array[Int] = Array(5, 4, 3, 2, 1)
```

Collections

Array

Array[String], Array[Int] etc

Arrays have a zero based index and can be indexed using the () operator



Collections

Array

```
scala> val fiveInts = new Array[Int](5)
fiveInts: Array[Int] = Array(0, 0, 0, 0, 0)

scala> fiveInts(0) = fiveToOne(2)

scala> fiveInts
res16: Array[Int] = Array(3, 0, 0, 0, 0)
```

Array[String],
Array[Int] etc

Elements of an array
can be updated

Collections

Array

```
scala> val fiveInts = new Array[Int](5)
fiveInts: Array[Int] = Array(0, 0, 0, 0, 0)
```

Array[String], Array[Int] etc

While this variable is immutable ie. it cannot be reassigned to a new Array

It's contents are mutable

Collections

Array

```
scala> val fiveInts = new Array[Int](5)
fiveInts: Array[Int] = Array(0, 0, 0, 0, 0)
```

Array[String],
Array[Int] etc

Lists are very similar to Arrays

except that **their contents are immutable**

Collections

Array 

Tuple

Collections

Tuple

A Tuple is heterogeneous collection of objects

Tuples can have any number of elements, but most commonly they have 2 elements

Collections

Tuple

```
object Demo{
   def main(args:Array[String]):Unit = {
      val Tuple=(13,"EHLA")
print(Tuple._1);
print(" ");
print(Tuple._2);
println();
   }
}
```

A Tuple with an integer and a String

Collections

Tuple

```
object Demo{
   def main(args:Array[String]):Unit = {
      val Tuple=(13,"EHLA")
print(Tuple._1);
print(" ");
print(Tuple._2);
println();
   }
}
```

Tuples are initialized using ()

Collections

Tuple

```
object Demo{
   def main(args:Array[String]):Unit = {
      val Tuple=(13,"EHLA")
print(Tuple._1);
print(" ");
print(Tuple._2);
println();
   }
}
```

Scala infers the type of the tuple to be Tuple2[Int, String]

Collections

Tuple

```
object Demo{
   def main(args:Array[String]):Unit = {
      val Tuple=(13,"EHLA")
 print(Tuple._1);
 print( " " );
 print(Tuple._2);
 println();
   }
 }
```

Indexing in Tuples starts with 1

Collections

Array ✓

Tuple ✓

What do collections have to do with functional programming and higher order methods?

YOU CAN HAVE A METHOD TAKE IN A FUNCTION AS AN ARGUMENT

Collections

Let's say you wanted to "do something" with each element of a collection

YOU CAN HAVE A METHOD TAKE IN A FUNCTION AS AN ARGUMENT

Collections

"do something" with each element of a collection

In an imperative paradigm, you would use for loops to achieve this

YOU CAN HAVE A METHOD TAKE IN A FUNCTION AS AN ARGUMENT

Collections

"do something" with each element of a collection

In an functional paradigm, use higher order methods

filter, map, foreach etc

YOU CAN HAVE A METHOD TAKE IN A FUNCTION AS AN ARGUMENT

Collections

filter, map, foreach etc

All of these methods take a function as an argument

YOU CAN HAVE A METHOD TAKE IN A FUNCTION AS AN ARGUMENT

Collections

filter, map, foreach etc

Use these methods in place of a loop that iterates through each element of the collection

YOU CAN HAVE A METHOD TAKE IN A FUNCTION AS AN ARGUMENT

Collections

Example: Computing a value for each element of a collection

map

Collections

map

```
scala>  val words = List("the", "quick", "brown", "fox")
words: List[String] = List(the, quick, brown, fox)
```

Compute the lengths of all the words

```
scala> words map (_.length)
res53: List[Int] = List(3, 5, 5, 3)
```

Collections

map

```
scala>  val words = List("the", "quick", "brown", "fox")
words: List[String] = List(the, quick, brown, fox)
```

```
scala> words map (_.length)
res53: List[Int] = List(3, 5, 5, 3)
```

Map will apply a function to each element of the List and return another List with the results

Collections

map

```
scala>  val words = List("the", "quick", "brown", "fox")
words: List[String] = List(the, quick, brown, fox)
```

```
scala> words map (_.length)
res53: List[Int] = List(3, 5, 5, 3)
```

Map will **apply a function** to each element of the List and return another List with the results

Collections

map

```
scala>  val words = List("the", "quick", "brown", "fox")
words: List[String] = List(the, quick, brown, fox)
```

```
scala> words map (_.length)
res53: List[Int] = List(3, 5, 5, 3)
```

Map will apply a function to each element of the List and **return another List with the results**

Collections

map

```
scala>  val words = List("the", "quick", "brown", "fox")
words: List[String] = List(the, quick, brown, fox)
```

```
scala> words map (_.length)
res53: List[Int] = List(3, 5, 5, 3)
```

_.length is a function object

Collections

map

```
scala>  val words = List("the", "quick", "brown", "fox")
words: List[String] = List(the, quick, brown, fox)
```

```
scala> words map (_.length)
res53: List[Int] = List(3, 5, 5, 3)
```

It's a more concise way of saying

```
(x: String) => x.length
```

Collections

map

```
scala>  val words = List("the", "quick", "brown", "fox")
words: List[String] = List(the, quick, brown, fox)

scala> words map (_.length)
res53: List[Int] = List(3, 5, 5, 3)
```

Here's another example

```
scala> words map (_.reverse)
res56: List[String] = List(eht, kciuq, nworb, xof)
```

Each element of the list has been reversed

Collections

Example: Computing a collection of values from each element of a collection

flatMap

Collections

flatMap

flatMap is similar to map

It applies a function to each element of a collection

It returns a collection which is a concatenation of all the results

Collections

flatMap

```
scala> words map (_.toList)
res57: List[List[Char]] = List(List(t, h, e), List(q, u, i, c, k),
 List(b, r, o, w, n), List(f, o, x))

scala> words flatMap (_.toList)
res58: List[Char] = List(t, h, e, q, u, i, c, k, b, r, o, w, n, f,
 o, x)
```

A function that converts a String to a List of Characters

flatMap

Collections

```
scala> words map (_.toList)
res57: List[List[Char]] = List(List(t, h, e), List(q, u, i, c, k),
 List(b, r, o, w, n), List(f, o, x))

scala> words flatMap (_.toList)
res58: List[Char] = List(t, h, e, q, u, i, c, k, b, r, o, w, n, f,
 o, x)
```

map returns a list of lists

flatMap

Collections

```
scala> words map (_.toList)
res57: List[List[Char]] = List(List(t, h, e), List(q, u, i, c, k),
 List(b, r, o, w, n), List(f, o, x))

scala> words flatMap (_.toList)
res58: List[Char] = List(t, h, e, q, u, i, c, k, b, r, o, w, n, f,
 o, x)
```

flatMap returns a single list in which all element lists are concatenated

Collections

Example: Applying a condition on each element of a collection

filter

Collections

filter

```
scala> List(1, 2, 3, 4, 5) filter (_ % 2 == 0)
res63: List[Int] = List(2, 4)
```

Select only even numbers from a list of integers

Collections

filter

```
scala> List(1, 2, 3, 4, 5) filter (_ % 2 == 0)
res63: List[Int] = List(2, 4)
```

Check this condition for each element of the List

Collections

filter

```
scala> List(1, 2, 3, 4, 5) filter (_ % 2 == 0)
res63: List[Int] = List(2, 4)
```

Return only those elements that satisfy the condition

Collections

Example: Performing an operation using each element of a collection

foreach

Collections

foreach

```
scala> var sum = 0
sum: Int = 0

scala> List(1, 2, 3, 4, 5) foreach (sum += _)

scala> sum
res60: Int = 15
```

Increment a variable using each element from a List

Collections

foreach

```
scala> var sum = 0
sum: Int = 0

scala> List(1, 2, 3, 4, 5) foreach (sum += _)

scala> sum
res60: Int = 15
```

Initialize a mutable variable

Collections

foreach

```
scala> var sum = 0
sum: Int = 0

scala> List(1, 2, 3, 4, 5) foreach (sum += _)

scala> sum
res60: Int = 15
```

Increment it's value using each element of the list

Collections

foreach

```
scala> var sum = 0
sum: Int = 0

scala> List(1, 2, 3, 4, 5) foreach (sum += _)

scala> sum
res60: Int = 15
```

This is a placeholder for each element of the list

Collections

foreach

```
scala> var sum = 0
sum: Int = 0

scala> List(1, 2, 3, 4, 5) foreach (sum += _)

scala> sum
res60: Int = 15
```

```
scala> for (value <- List(1,2,3,4,5)) sum+=value
```

This is equivalent to using a loop that iterates through the list

Classes

Classes

Classes in Scala are pretty similar to Java

There are a few important differences though

Classes (Scala vs Java)

1. Primary vs Auxiliary constructors
2. Mutable and Immutable member variables
3. Inferred access modifiers
4. Operator Overloading
5. Singleton Objects
6. Inheritance through Traits

1. Primary vs Auxiliary constructors

Let's set up a class to create Rational Numbers

```
val q = new Rational(2,3)
```

A rational number is a number that can be expressed as a ratio n/d

1. Primary vs Auxiliary constructors

Let's set up a class to create Rational Numbers

```
val q = new Rational(2,3)
```

Examples of rational numbers:
1/2, 2/3, 112/239, and 2/1

1. Primary vs Auxiliary constructors

```
val q = new Rational(2,3)
```

```
class Rational(n: Int, d: Int) {

  require(d != 0)

  def this(n: Int) = this(n, 1)

  override def toString = numer +"/"+ denom

  private val g = gcd(n.abs, d.abs)

  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)

  val numer = n / g
  val denom = d / g

  def + (that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )

  def / (that: Rational): Rational =
    new Rational(numer * that.denom, denom * that.numer)


  def + (i: Int): Rational =
    new Rational(numer + i * denom, denom)


  def / (i: Int): Rational =
    new Rational(numer, denom * i)

}
```

A class for Rational Numbers is like a blueprint for how Rational number objects should look and behave

1. Primary vs Auxiliary constructors

```
val q = new Rational(2,3)
```

```
class Rational(n: Int, d: Int) {

  require(d != 0)

  def this(n: Int) = this(n, 1)

  override def toString = numer +"/"+ denom

  private val g = gcd(n.abs, d.abs)

  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)

  val numer = n / g
  val denom = d / g

  def + (that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )

  def / (that: Rational): Rational =
    new Rational(numer * that.denom, denom * that.numer)


  def + (i: Int): Rational =
    new Rational(numer + i * denom, denom)


  def / (i: Int): Rational =
    new Rational(numer, denom * i)

}
```

A class should have :

A way/ways to create a new object of the class

Constructor

1. Primary vs Auxiliary constructors

```
val q = new Rational(2,3)
```

```
class Rational(n: Int, d: Int) {
  require(d != 0)

  def this(n: Int) = this(n, 1)

  override def toString = numer +"/"+ denom

  private val g = gcd(n.abs, d.abs)

  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)

  val numer = n / g
  val denom = d / g

  def + (that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )
```

Constructor

In Scala, the primary constructor of a class is defined along with the name of the class

1. Primary vs Auxiliary constructors

```
val q = new Rational(2,3)
```

Constructor

```
class Rational(n: Int, d: Int) {
  require(d != 0)

  def this(n: Int) = this(n, 1)

  override def toString = numer +"/"+ denom

  private val g = gcd(n.abs, d.abs)

  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)

  val numer = n / g
  val denom = d / g

  def + (that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )
```

This constructor takes in 2 parameters which are required to define the object

1. Primary vs Auxiliary constructors

```
val q = new Rational(2,3)
```

Constructor

```
class Rational(n: Int, d: Int) {
  require(d != 0)

  def this(n: Int) = this(n, 1)

  override def toString = numer +"/"+ denom

  private val g = gcd(n.abs, d.abs)

  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)

  val numer = n / g
  val denom = d / g

  def + (that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )
```

A class can have multiple constructors

1. Primary vs Auxiliary constructors

```
val q = new Rational(2,3)
```

Constructor

```
class Rational(n: Int, d: Int) {

  require(d != 0)

  def this(n: Int) = this(n, 1)

  override def toString = numer +"/"+ denom

  private val g = gcd(n.abs, d.abs)

  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)

  val numer = n / g
  val denom = d / g

  def + (that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )
```

Use the this keyword to define auxiliary constructors

1. Primary vs Auxiliary constructors

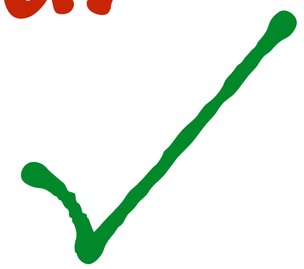
```
val q = new Rational(2,3)
```

Constructor

```
class Rational(n: Int, d: Int) {

  require(d != 0)

  def this(n: Int) = this(n, 1)

  override def toString = numer +"/"+ denom

  private val g = gcd(n.abs, d.abs)

  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)

  val numer = n / g
  val denom = d / g

  def + (that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )
```

You can define some constraints on the parameters passed to the primary constructor

Classes (Scala vs Java)

1. Primary vs Auxiliary constructors 
2. Mutable and Immutable member variables
3. Type inference
4. Operator Overloading
5. Singleton Objects
6. Inheritance through Traits

2. Mutable and Immutable member variables

```
class Rational(n: Int, d: Int)
{
  require(d != 0)

  def this(n: Int) = this(n, 1)

  override def toString = numer +"/"+ denom

  private val g = gcd(n.abs, d.abs)

  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)

  val numer = n / g
  val denom = d / g

  def + (that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )

  def / (that: Rational): Rational =
    new Rational(numer * that.denom, denom *
that.numer)


  def + (i: Int): Rational =
    new Rational(numer + i * denom, denom)
```

If the Constructor takes parameters, these are normally used to set up the member variables of the class

2. Mutable and Immutable member variables

```
class Rational(n: Int, d: Int) {

  require(d != 0)

  def this(n: Int) = this(n, 1)

  override def toString = numer +"/"+ denom

  private val g = gcd(n.abs, d.abs)

  private def gcd(a: Int, b: Int): Int =
   if (b == 0) a else gcd(b, a % b)

  val numer = n / g
  val denom = d / g

  def + (that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )

  def / (that: Rational): Rational =
    new Rational(numer * that.denom, denom * that.numer)


  def + (i: Int): Rational =
    new Rational(numer + i * denom, denom)
```

Use the integer parameters to compute the numerator and denominator for our Rational number

2. Mutable and Immutable member variables

```
class Rational(n: Int, d: Int) {

  require(d != 0)

  def this(n: Int) = this(n, 1)

  override def toString = numer +"/"+ denom

  private val g = gcd(n.abs, d.abs)


  private def gcd(a: Int, b: Int): Int =
   if (b == 0) a else gcd(b, a % b)

  val numer = n / g
  val denom = d / g

  def + (that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )

  def / (that: Rational): Rational =
    new Rational(numer * that.denom, denom * that.numer)


  def + (i: Int): Rational =
    new Rational(numer + i * denom, denom)
```

Variables are defined using val or var

val is similar to the final keyword in Java

2. Mutable and Immutable member variables

val — Designates the variable as **immutable**

var — Designates the variable as **mutable**

2. Mutable and Immutable member variables

val	immutable	The variable cannot be reassigned to a new value
var	mutable	The variable can be reassigned

Every variable must be declared using one of these keywords

2. Mutable and Immutable member variables

var or val

```
var myVar : Char = 'a'
myVar: Char = a
```

A Mutable variable

```
val m : Int = 3
m: Int = 3
```

An Immutable variable

var or val

```
val m : Int = 3
m: Int = 3
```

An Immutable variable

cannot be reassigned to a new value

```
scala> m= 4
<console>:12: error: reassignment to val
        m= 4
          ^
```

2. Mutable and Immutable member variables

var or val

Immutability is an important concept in Scala

Scala encourages the use of immutable variables over mutable ones because of its emphasis on functional programming

var or val

The philosophy is that Objects passed to functions should essentially remain unchanged

2. Mutable and Immutable member variables

var or val

If you know that a variable's state should not change once it is assigned

declare it immutable

2. Mutable and Immutable member variables

var or val

This makes sure that there are no unintentional state changes to the variable

It also makes debugging code easier

Classes (Scala vs Java)

1. Primary vs Auxiliary constructors ✓
2. Mutable and Immutable member variables ✓
3. Type Inference
4. Operator Overloading
5. Singleton Objects
6. Inheritance through Abstract classes and Traits

```
var myVar : Char = 'a'
```

```
val m : Int = 3
```

Scala is statically typed

The compiler needs to know the type of every value that's used

Scala tends towards writing concise code

So, the compiler can also infer the type of a variable even if it's not explicitly specified

```
val myVar2 = 4
```

```
myVar2: Int = 4
```

Classes (Scala vs Java)

1. Primary vs Auxiliary constructors ✓
2. Mutable and Immutable member variables ✓
3. Type Inference ✓
4. Operator Overloading
5. Singleton Objects
6. Inheritance through Abstract classes and Traits

Operators

```
val q = new Rational(2,3)
```

If you are familiar with operators in C++, Scala has a similar way to define behavior

Operators

```
val q = new Rational(2,3)
```

For instance, define arithmetic operators for Rationals

```
scala> val oneHalf = new Rational(1, 2)
oneHalf: Rational = 1/2

scala> val twoThirds = new Rational(2, 3)
twoThirds: Rational = 2/3

scala> (oneHalf / 7) + (1 - twoThirds)
res0: Rational = 17/42
```

Operators

```
val q = new Rational(2,3)
```

```
class Rational(n: Int, d: Int) {
  require(d != 0)

  def this(n: Int) = this(n, 1)

  override def toString = numer +"/"+ denom

  private val g = gcd(n.abs, d.abs)

  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)

  val numer = n / g
  val denom = d / g

  def + (that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )

  def / (that: Rational): Rational =
    new Rational(numer * that.denom, denom * that.numer)

  def + (i: Int): Rational =
    new Rational(numer + i * denom, denom)

  def / (i: Int): Rational =
    new Rational(numer, denom * i)
}
```

A plus operator to add 2 Rationals

Operators

```
val q = new Rational(2,3)
```

```
q + r
```

```
class Rational(n: Int, d: Int) {
  require(d != 0)

  def this(n: Int) = this(n, 1)

  override def toString = numer +"/"+ denom

  private val g = gcd(n.abs, d.abs)

  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)

  val numer = n / g
  val denom = d / g

  def + (that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )

  def / (that: Rational): Rational =
    new Rational(numer * that.denom, denom * that.numer)

  def + (i: Int): Rational =
    new Rational(numer + i * denom, denom)

  def / (i: Int): Rational =
    new Rational(numer, denom * i)
}
```

A plus operator to add 2 Rationals

Operators

```
val q = new Rational(2,3)
```

```
q + r
```

```
class Rational(n: Int, d: Int) {
  require(d != 0)

  def this(n: Int) = this(n, 1)

  override def toString = numer +"/"+ denom

  private val g = gcd(n.abs, d.abs)

  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)

  val numer = n / g
  val denom = d / g

  def + (that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )

  def / (that: Rational): Rational =
    new Rational(numer * that.denom, denom * that.numer)


  def + (i: Int): Rational =
    new Rational(numer + i * denom, denom)


  def / (i: Int): Rational =
    new Rational(numer, denom * i)

}
```

Adding 2 Rationals returns another Rational

Operators

```
class Rational(n: Int, d: Int) {
  require(d != 0)

  def this(n: Int) = this(n, 1)

  override def toString = numer +"/"+ denom

  private val g = gcd(n.abs, d.abs)

  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)

  val numer = n / g
  val denom = d / g

  def + (that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )

  def / (that: Rational): Rational =
    new Rational(numer * that.denom, denom * that.numer)


  def + (i: Int): Rational =
    new Rational(numer + i * denom, denom)


  def / (i: Int): Rational =
    new Rational(numer, denom * i)

}
```

```
val q = new Rational(2,3)
```

q + r

this

Operators

```
class Rational(n: Int, d: Int) {
  require(d != 0)

  def this(n: Int) = this(n, 1)

  override def toString = numer +"/"+ denom

  private val g = gcd(n.abs, d.abs)

  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)

  val numer = n / g
  val denom = d / g

  def + (that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )

  def / (that: Rational): Rational =
    new Rational(numer * that.denom, denom * that.numer)


  def + (i: Int): Rational =
    new Rational(numer + i * denom, denom)


  def / (i: Int): Rational =
    new Rational(numer, denom * i)

}
```

```
val q = new Rational(2,3)
```

```
q + r
```

this that

Operators

```
val q = new Rational(2,3)
```

q + r

this that

```
def + (that: Rational): Rational =
  new Rational(
    numer * that.denom + that.numer * denom,
    denom * that.denom
  )
```

Compute the parameters of the new Rational using the parameters of this and that

Operators

```
val q = new Rational(2,3)
```

```
q + r
```

this that

```
class Rational(n: Int, d: Int) {
  require(d != 0)

  def this(n: Int) = this(n, 1)

  override def toString = numer +"/"+ denom

  private val g = gcd(n.abs, d.abs)

  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)

  val numer = n / g
  val denom = d / g

  def + (that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )

  def / (that: Rational): Rational =
    new Rational(numer * that.denom, denom * that.numer)

  def + (i: Int): Rational =
    new Rational(numer + i * denom, denom)

  def / (i: Int): Rational =
    new Rational(numer, denom * i)

}
```

Sometimes we might want to add a Rational and an Integer

Operators

```
val q = new Rational(2,3)
```

```
q + r
```

this that

```
class Rational(n: Int, d: Int) {
  require(d != 0)

  def this(n: Int) = this(n, 1)

  override def toString = numer +"/"+ denom

  private val g = gcd(n.abs, d.abs)

  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)

  val numer = n / g
  val denom = d / g

  def + (that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )

  def / (that: Rational): Rational =
    new Rational(numer * that.denom, denom * that.numer)


  def + (i: Int): Rational =
    new Rational(numer + i * denom, denom)


  def / (i: Int): Rational =
    new Rational(numer, denom * i)

}
```

Overload the + operator

Operators

```
val q = new Rational(2,3)
```

q + r

this that

```
class Rational(n: Int, d: Int) {
  require(d != 0)
  def this(n: Int) = this(n, 1)
  override def toString = numer +"/"+ denom
  private val g = gcd(n.abs, d.abs)
  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)
  val numer = n / g
  val denom = d / g

  def + (that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )

  def / (that: Rational): Rational =
    new Rational(numer * that.denom, denom * that.numer)

  def + (i: Int): Rational =
    new Rational(numer + i * denom, denom)

  def / (i: Int): Rational =
    new Rational(numer, denom * i)
}
```

Overload the + operator

Classes (Scala vs Java)

1. Primary vs Auxiliary constructors ✓
2. Mutable and Immutable member variables ✓
3. Type Inference ✓
4. Operator Overloading ✓
5. Singleton Objects
6. Inheritance through Abstract classes and Traits

In Scala, Static member variables and methods cannot be defined inside a class

Scala provides a special construct for defining static variables/behaviour

Singleton Objects

Singleton Objects

Singleton objects are a way to

1. Define a class that's only going to be used to create 1 object

2. Define static behavior and variables

Singleton Objects

```
object Rational{

  val inf = "infinity"

  def divideByZero(x:Int): Unit ={
    println("Not defined")
  }

}
```

Use the object keyword to create a Singleton object

Singleton Objects

```
object Rational{
  val inf = "infinity"

  def divideByZero(x:Int): Unit ={
    println("Not defined")
  }

}
```

Use these methods and variables as if they are static members of Rational

```
println(Rational.inf);

Rational.divideByZero(5);
```

Singleton Objects

The main method that's used to run a program must be static i.e. enclosed in a Singleton object

```
object HelloWorld {

   def main (args: Array[String]){

     println("Hello, this is SCALA!")

   }

}
```

Classes (Scala vs Java)

1. Primary vs Auxiliary constructors ✓
2. Mutable and Immutable member variables ✓
3. Type Inference ✓
4. Operator Overloading ✓
5. Singleton Objects ✓
6. Inheritance through Traits

Traits

Inheritance in Scala is driven using

1. Classes
2. Abstract Classes
3. Traits

1. Classes

2. Abstract Classes

3. Traits

Abstract Classes are classes with at least 1 unimplemented method

1. Classes

2. Abstract Classes

3. Traits

Abstract Classes cannot be instantiated directly

1. Classes
2. Abstract Classes
3. Traits

Scala does not have Interfaces

1. Classes

2. Abstract Classes

Traits are the closest thing to Interfaces in Scala

3. Traits

1. Classes

2. Abstract Classes

3. Traits

Traits can have implemented methods (like interface default methods in Java 8)

1. Classes

2. Abstract Classes

Like interfaces, Traits can't be instantiated directly

3. Traits

1. Classes

2. Abstract Classes

3. Traits

Classes can inherit from only 1 Class/ Abstract Class

Classes can inherit from any number of traits

This is the main difference between Classes and Traits

1. Classes

2. Abstract Classes

Classes can inherit from only 1 Class/ Abstract Class

3. Traits

Classes can inherit from any number of traits

This is the main difference between Classes and Traits

```
abstract class Animal {
  def speak
}
```

An abstract class

```
abstract class Animal {
  def speak
}
```

```
trait FourLeggedAnimal {
  def walk
  def run
}
```

A trait with abstract methods

```scala
abstract class Animal {
  def speak
}

trait FourLeggedAnimal {
  def walk
  def run
}


trait WaggingTail {
  def startTail { println("tail started") }
  def stopTail { println("tail stopped") }
}
```

A trait with implemented methods

```scala
abstract class Animal {
  def speak
}

trait FourLeggedAnimal {
  def walk
  def run
}

trait WaggingTail {
  def startTail { println("tail started") }
  def stopTail { println("tail stopped") }
}

class Dog extends Animal with WaggingTail with FourLeggedAnimal {
  def speak { println("Dog says 'woof'") }
  def walk { println("Dog is walking") }
  def run { println("Dog is running") }
}
```

A class that inherits from 1 abstract class and 2 traits

```scala
abstract class Animal {
  def speak
}

trait FourLeggedAnimal {
  def walk
  def run
}
```

```scala
trait WaggingTail {
  def startTail { println("tail started") }
  def stopTail { println("tail stopped") }
}
```

```scala
class Dog extends Animal with WaggingTail with FourLeggedAnimal {
  def speak { println("Dog says 'woof'") }
  def walk { println("Dog is walking") }
  def run { println("Dog is running") }
}
```

Use the extends keyword with the first inheritance (either trait/class)

```scala
abstract class Animal {
  def speak
}
```

```scala
trait WaggingTail {
  def startTail { println("tail started") }
  def stopTail { println("tail stopped") }
}
```

```scala
trait FourLeggedAnimal {
  def walk
  def run
}
```

```scala
class Dog extends Animal with WaggingTail with FourLeggedAnimal {
  def speak { println("Dog says 'woof'") }
  def walk { println("Dog is walking") }
  def run { println("Dog is running") }
}
```

Use the with keyword for any inheritances that follow

```scala
abstract class Animal {
  def speak
}

trait FourLeggedAnimal {
  def walk
  def run
}

trait WaggingTail {
  def startTail { println("tail started") }
  def stopTail { println("tail stopped") }
}

class Dog extends Animal with WaggingTail with FourLeggedAnimal {
  def speak { println("Dog says 'woof'") }
  def walk { println("Dog is walking") }
  def run { println("Dog is running") }
}
```

If a class inherits from both another class and traits, use extends with the class and with for all the subsequent traits

Classes (Scala vs Java)

1. Primary vs Auxiliary constructors ✓
2. Mutable and Immutable member variables ✓
3. Type Inference ✓
4. Operator Overloading ✓
5. Singleton Objects ✓
6. Inheritance through Traits ✓