Introduction to

Scala (Part – I)

Gurgaon<<April 2014

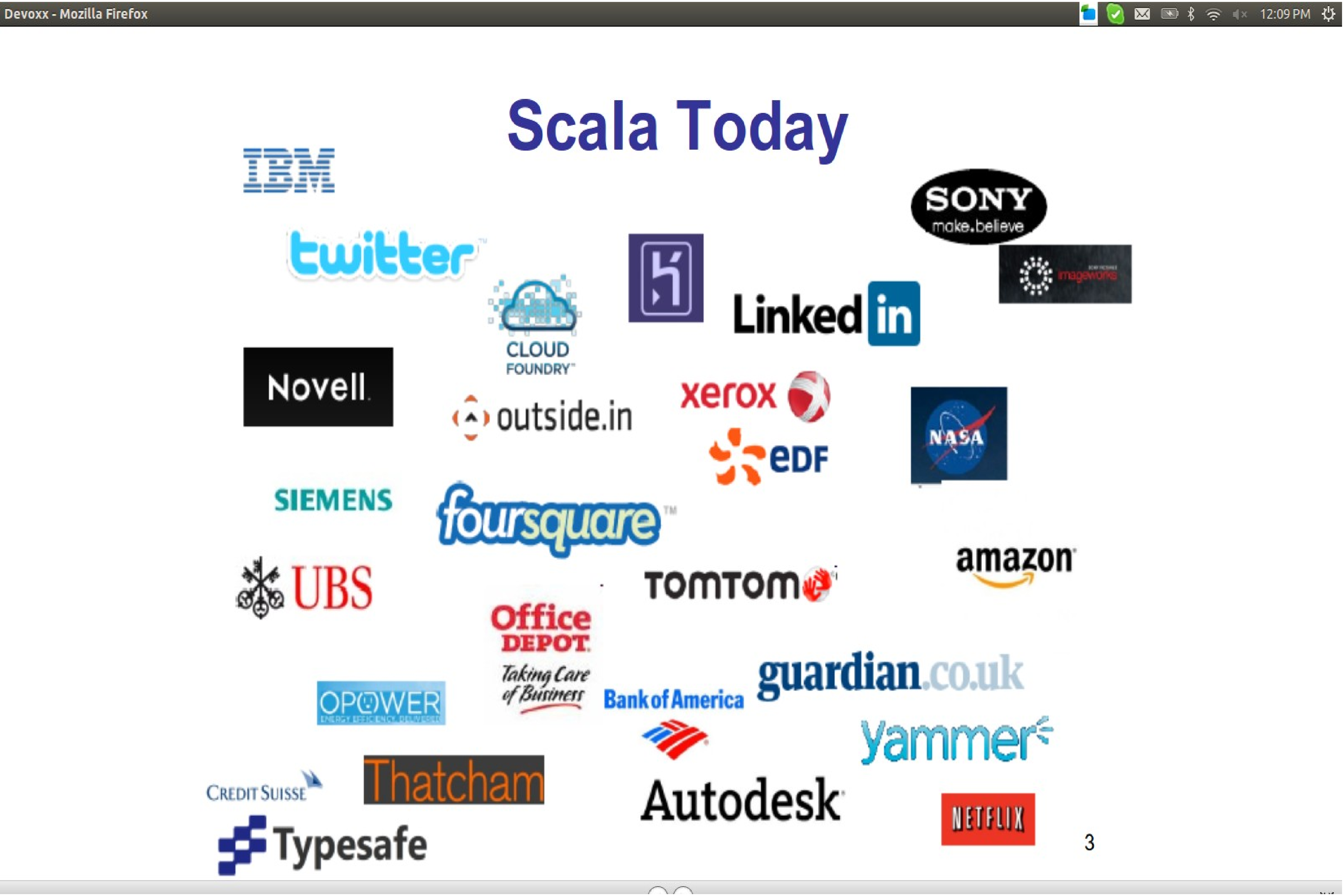
Agenda

- Scala Who & Why?
- First Steps
- Classes and Objects
- Control Structures and Functions
- Inheritance and Traits
- Testing
- Pattern Matching
- Collections
- XML Support

Agenda

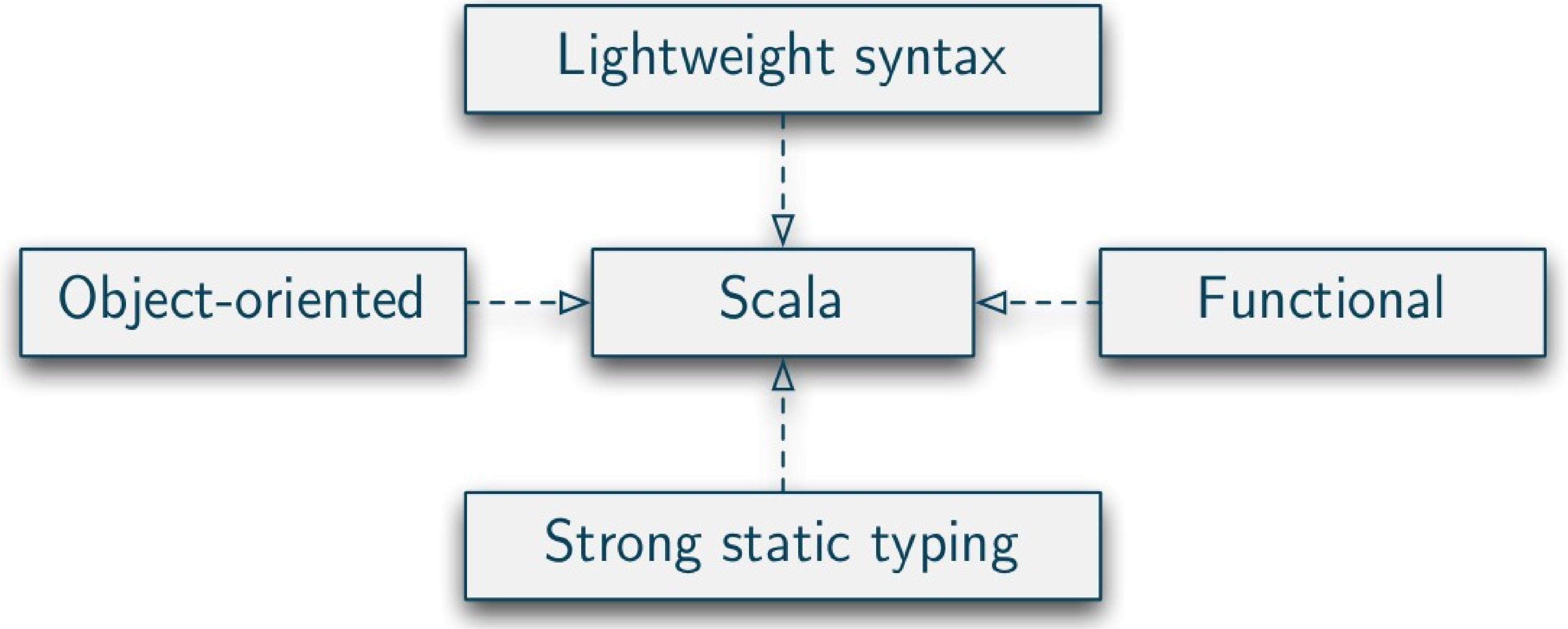
- Scala Who & Why?
- First Steps
- Classes and Objects
- Control Structures and Functions
- Inheritance and Traits
- Testing
- Pattern Matching
- Collections
- XML Support

In 2003, Martin Odersky and his team set out to create a “scalable” language that fused important concepts from functional and object-oriented programming. The language aimed to be highly interoperable with Java, have a rich type system, and to allow developers to express powerful concepts concisely.



Scala Today

Devoxx - Mozilla Firefox
12:09 PM
IBM
SONY
make.believe
twitter
imageworks
CLOUD FOUNDRY
Linked in
Novell
outside.in
xerox
eDF
NASA
SIEMENS
foursquare
amazon
UBS
TOMTOM
Office DEPOT
Taking Care of Business
guardian.co.uk
OPOWER
ENERGY EFFICIENCY. DELIVERED.
Bank of America
yammer
CREDIT SUISSE
Thatcham
Autodesk
NETFLIX
Typesafe
3


Lightweight syntax
Object-oriented
Scala
Functional
Strong static typing

Functional programming contructs make it easy to build interesting things quickly from small parts

Object oriented constructs make it easy to structure large systems and adapt to new demands

Object oriented

Everything is an object, pure OO

Not like java where we have primitives / static fields and methods which are not members of any object

1+2 is actually 1.+(2)

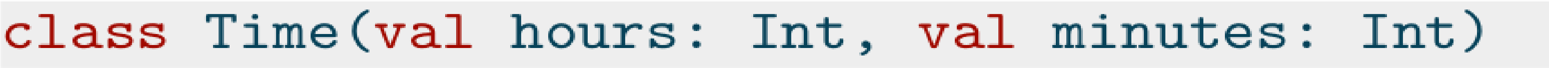
```
1 + 2
                > res1: Int(3) = 3
1 .+(2)
                > res2: Int(3) = 3
```

Functional

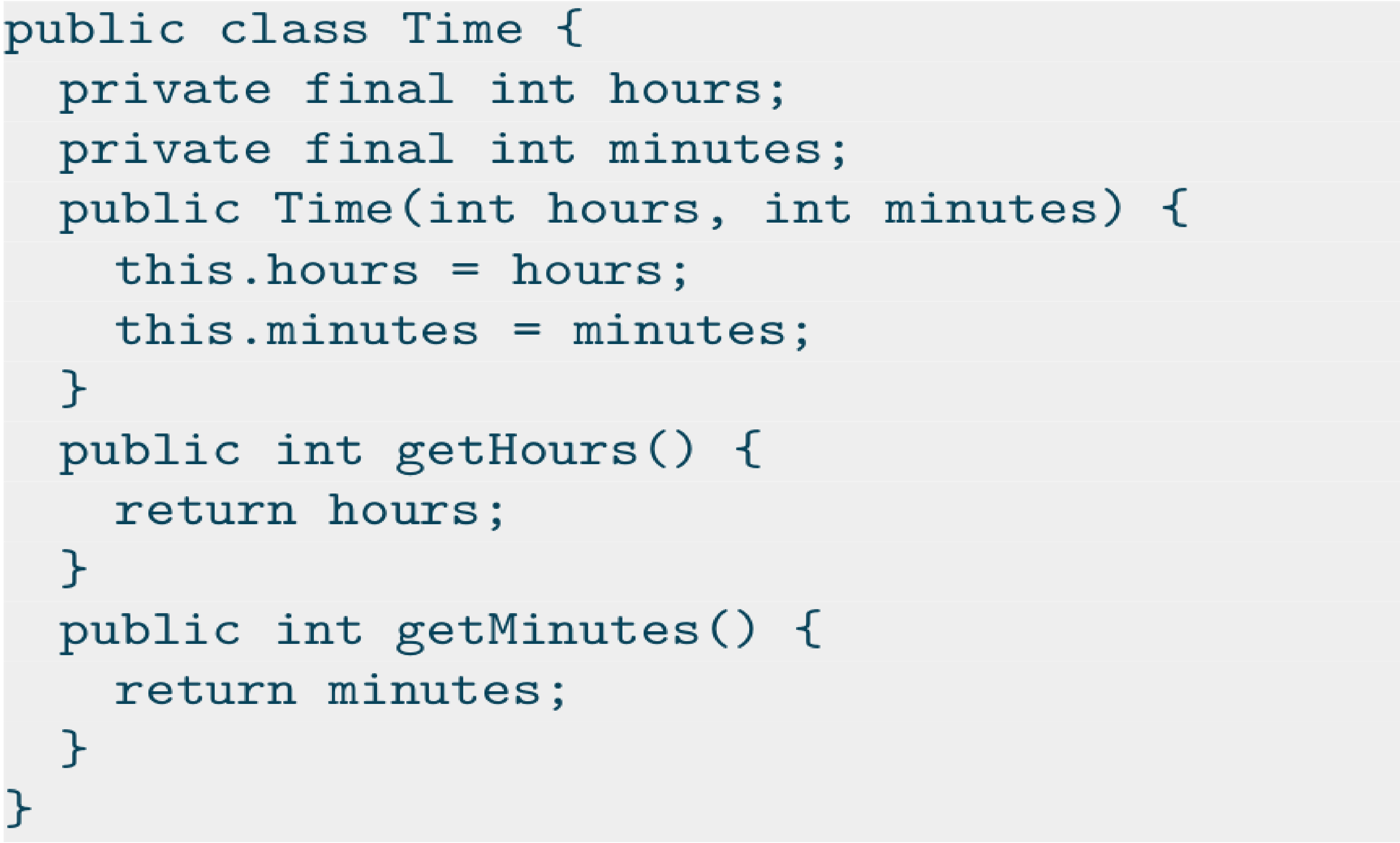
Functions are first class values

Encourages immutability where operations map input values to output rather than change data in place

Concise Code



```
class Time(val hours: Int, val minutes: Int)
```



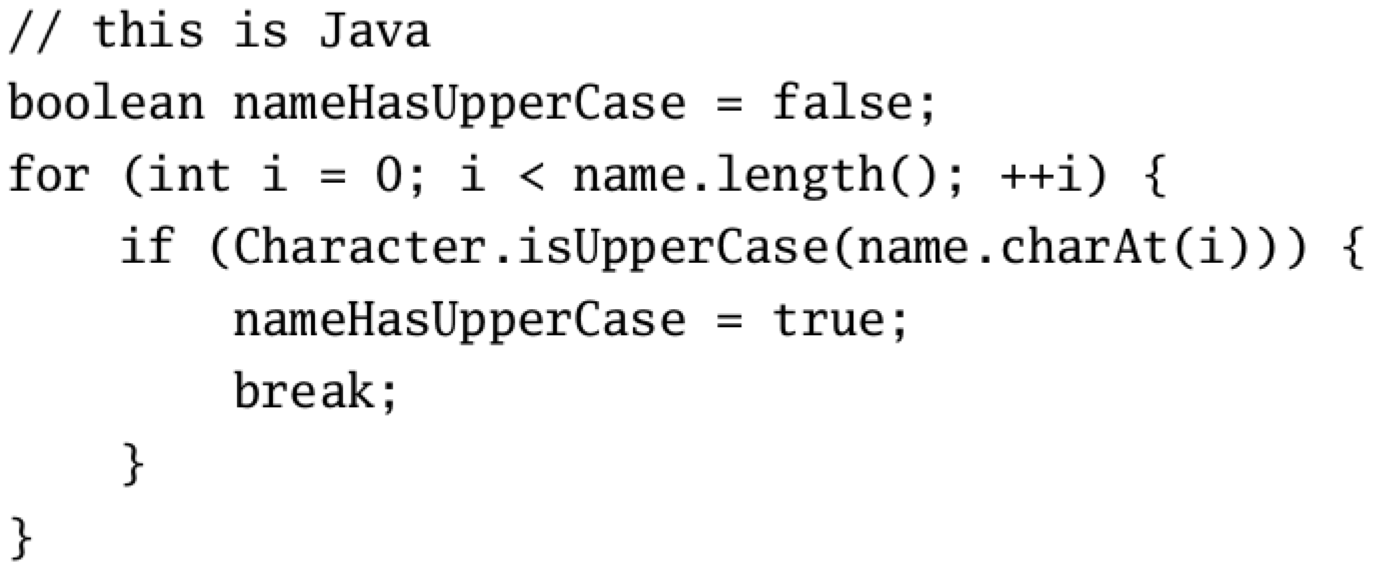
```
public class Time {
  private final int hours;
  private final int minutes;
  public Time(int hours, int minutes) {
    this.hours = hours;
    this.minutes = minutes;
  }
  public int getHours() {
    return hours;
  }
  public int getMinutes() {
    return minutes;
  }
}
```

Expressive

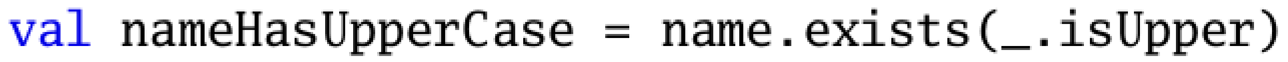
```
val numbers = 1 to 10
>Range(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)


numbers filter { _ % 2 == 0 }
> Vector(2, 4, 6, 8, 10)
```

High Level

Finding an upper case character



```java
// this is Java
boolean nameHasUpperCase = false;
for (int i = 0; i < name.length(); ++i) {
    if (Character.isUpperCase(name.charAt(i))) {
        nameHasUpperCase = true;
        break;
    }
}
```



```scala
val nameHasUpperCase = name.exists(_.isUpper)
```

Statically Typed

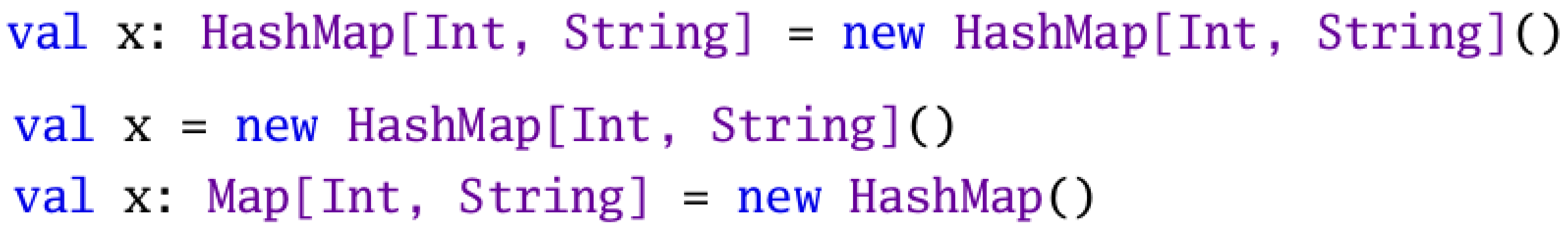
Cannot add boolean to List[Int]

Verifiable Properties

Safe Refactorings

Strong type inference

Documentation

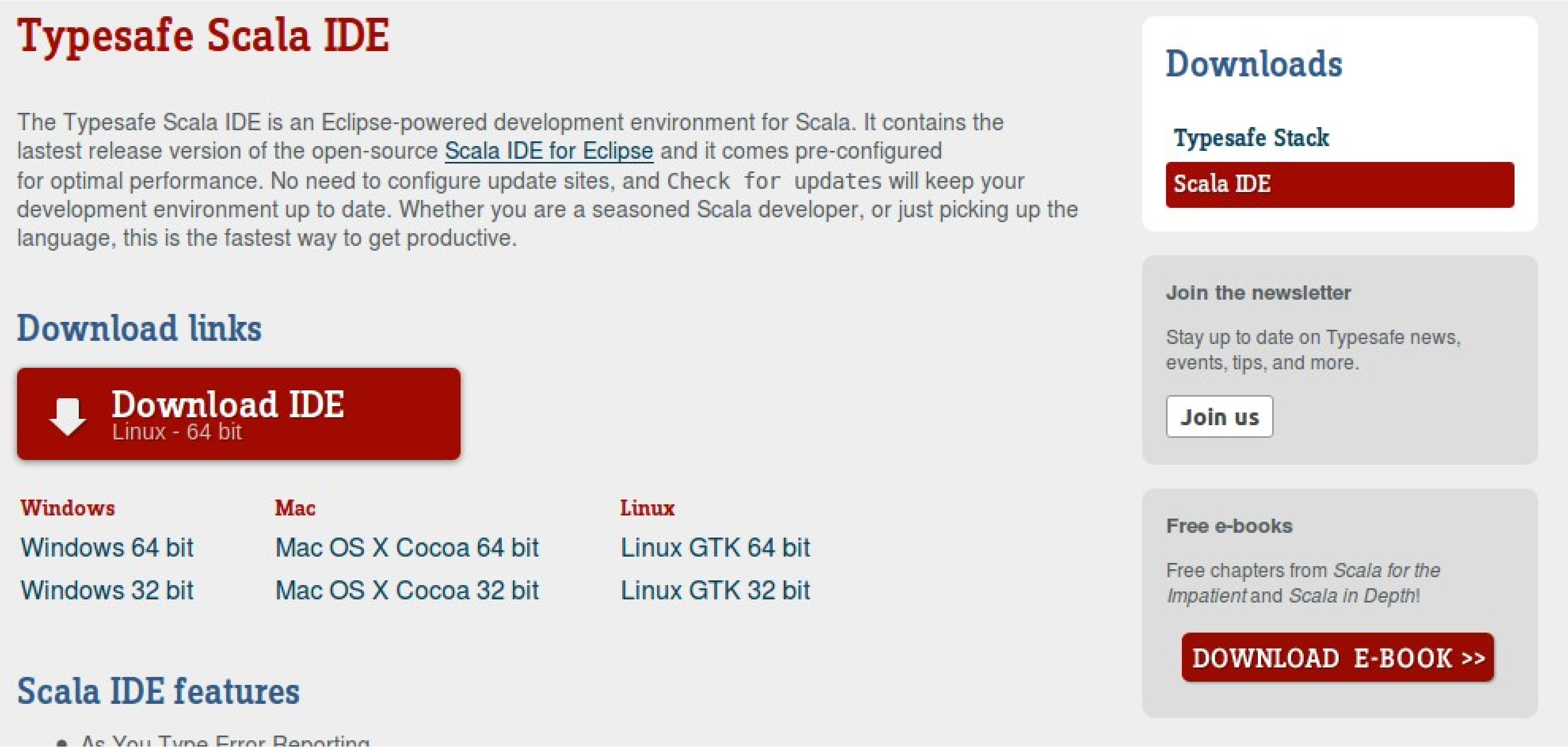


```
val x: HashMap[Int, String] = new HashMap[Int, String]()

val x = new HashMap[Int, String]()
val x: Map[Int, String] = new HashMap()
```

val x = Map(1->”somevalue”)

Setting up the development environment

SCALA IDE



Typesafe Scala IDE

The Typesafe Scala IDE is an Eclipse-powered development environment for Scala. It contains the lastest release version of the open-source Scala IDE for Eclipse and it comes pre-configured for optimal performance. No need to configure update sites, and `Check for updates` will keep your development environment up to date. Whether you are a seasoned Scala developer, or just picking up the language, this is the fastest way to get productive.

Download links

Download IDE
Linux - 64 bit

Windows	Mac	Linux
Windows 64 bit	Mac OS X Cocoa 64 bit	Linux GTK 64 bit
Windows 32 bit	Mac OS X Cocoa 32 bit	Linux GTK 32 bit

Scala IDE features

- As You Type Error Reporting

Downloads
Typesafe Stack
Scala IDE
Join the newsletter
Stay up to date on Typesafe news, events, tips, and more.
Join us
Free e-books
Free chapters from *Scala for the Impatient* and *Scala in Depth*!
DOWNLOAD E-BOOK >>

1. Create a new worksheet called Lab1
2. Print “Hello Scala World”
3. See the program execute in Worksheet

1 + 2

What do you see?

/> res1: Int(3) = 3

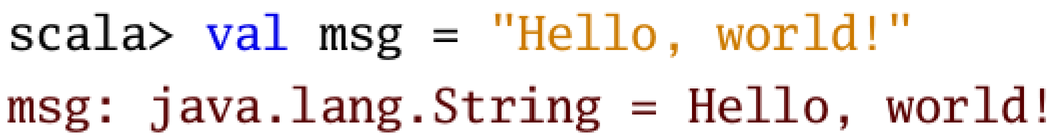
Agenda

- Scala Who & Why?
- First Steps
- Classes and Objects
- Control Structures and Functions
- Inheritance and Traits
- Testing
- Pattern Matching
- Collections
- XML Support

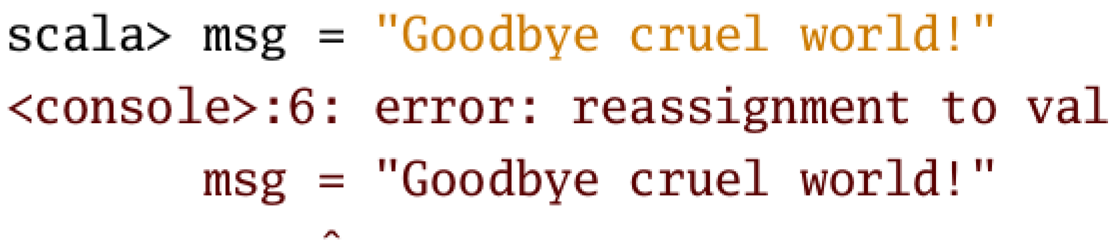
Variables

Similar to final — val

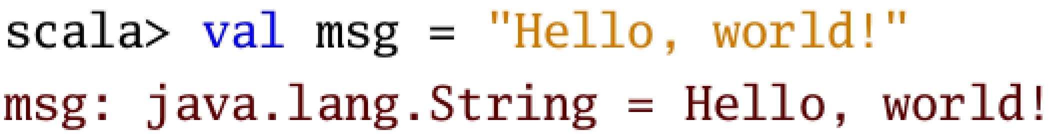
Mutable — var



```
scala> val msg = "Hello, world!"
msg: java.lang.String = Hello, world!
```



```
scala> msg = "Goodbye cruel world!"
<console>:6: error: reassignment to val
       msg = "Goodbye cruel world!"
           ^
```



```
scala> val msg = "Hello, world!"
msg: java.lang.String = Hello, world!
```

Why does this code compile?

Type Inference

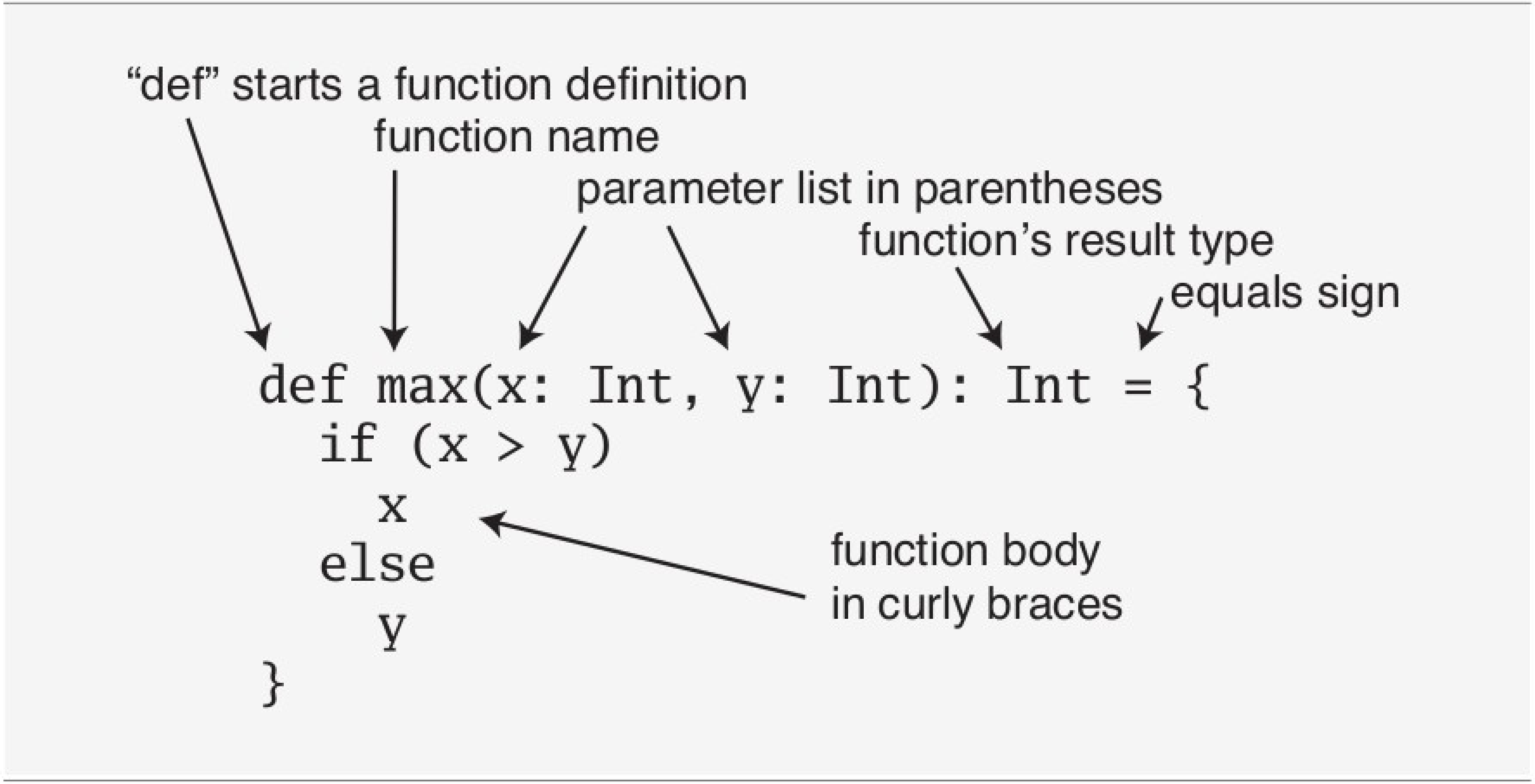
```
val msg:String = "Hello World!"          > msg  : String = Hello World!
```

(c) 2013-14, No reproduction without permission

Sometimes variables need to mutate

```
var description = "start"          > description  : java.lang.String = start

description = "end"
```

Method Definition



“def” starts a function definition

function name

parameter list in parentheses

function’s result type

equals sign

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

function body
in curly braces

Unit

A result of Unit indicates that the function returns no interesting value

```
def justGreet(name:String) = println("Hello  " + name)    >
justGreet: (name: String)Unit
```

Similar to void in Java

Everything returns a value

```
val number = {
   val x: Int = 7
   val y: Int = 9
   x + y
 }
```

```
> number  : Int = 16
```

```
if (1 < 10) 15 else 17
```

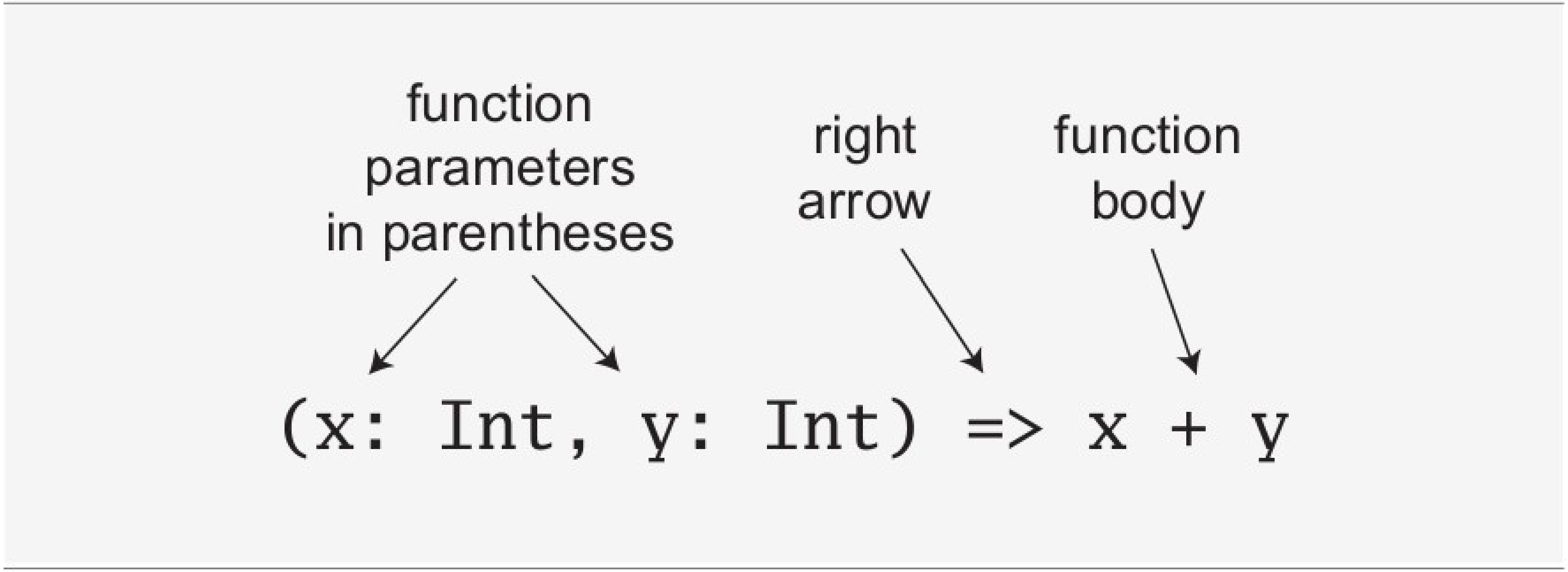
Functions, first look

```
List(1,2,3).foreach(l=>println(l))

List(1,2,3).foreach(l=>println(l))

List(1,2,3).foreach(println(_))

List(1,2,3) foreach (println(_))

List(1,2,3) foreach (println)

List(1,2,3) foreach println

  List(1,2,3) foreach ((x:Int)=>println(2*x))
```

Foreach takes a function of what needs to be done

Function Literal

What we pass inside of foreach is a function literal



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

```
List(1,2,3) foreach ((x:Int)=>println(2*x))
```

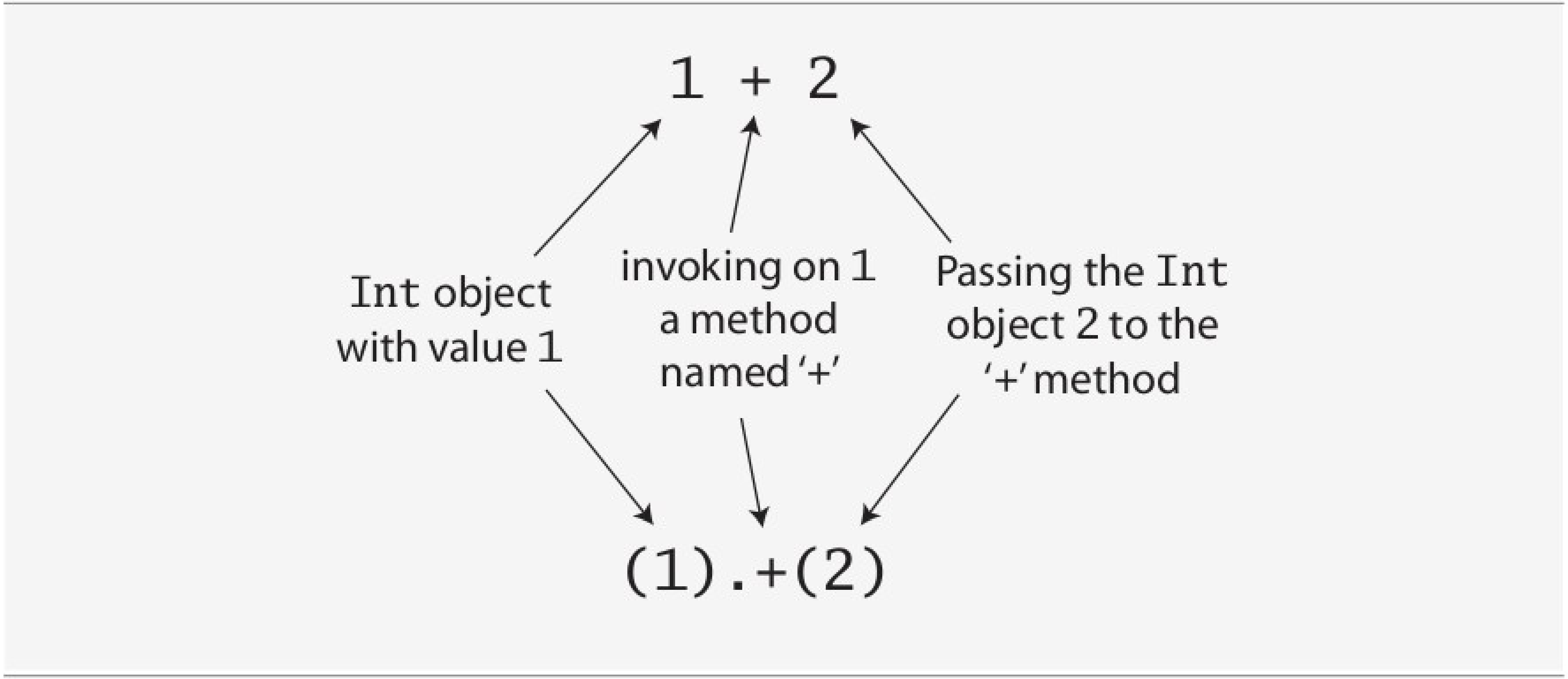
Working with Arrays

```
  val greetStrings = new Array[String](3)          > greetStrings  :
Array[String] = Array(null, null, null)
   greetStrings(0) = "Bank"
   greetStrings(1) = "of"
   greetStrings(2) = "America"
   greetStrings(0) = "BANK"
```

```
val greetStrings = List(1,2,3)
  greetStrings(0) = "Bank"
  greetStrings(1) = "of"
  greetStrings(2) = "America"
  greetStrings(0) = "BANK"
```

Notice?

No Operators



```
1 + 2
```

Int object with value 1

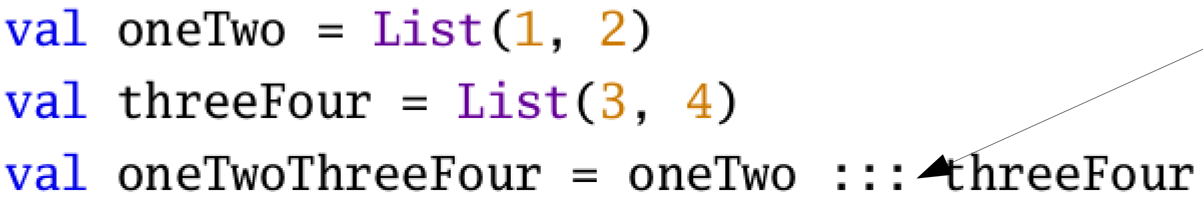
invoking on 1 a method named '+'

Passing the Int object 2 to the '+' method

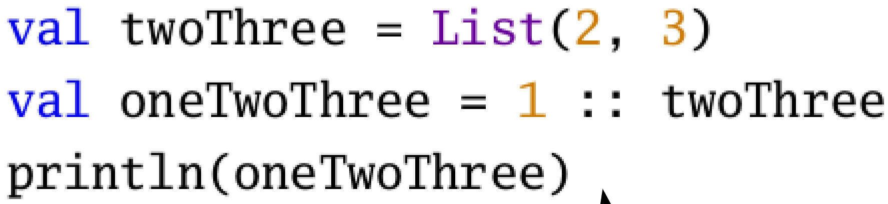
```
(1).+(2)
```

Lists

concatenation



```
val oneTwo = List(1, 2)
val threeFour = List(3, 4)
val oneTwoThreeFour = oneTwo ::: threeFour
```

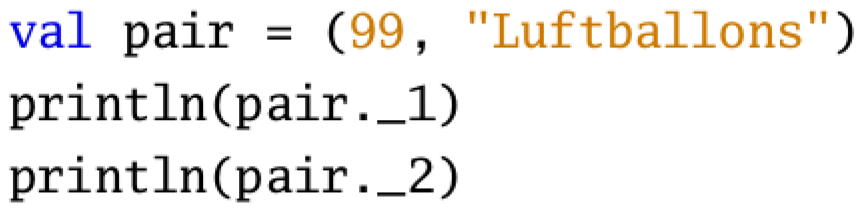


```
val twoThree = List(2, 3)
val oneTwoThree = 1 :: twoThree
println(oneTwoThree)
```

cons

Tuples

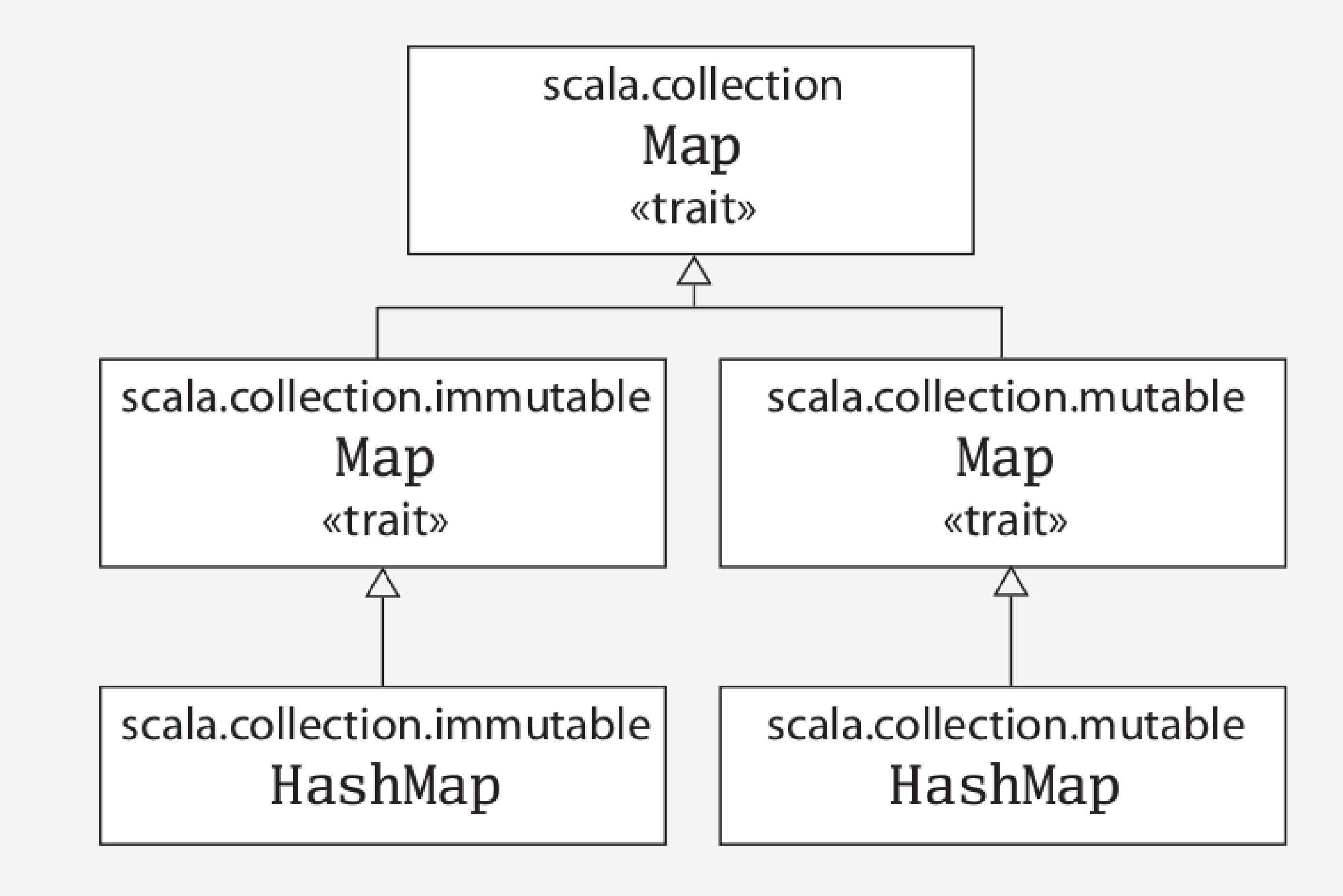
Immutable like lists, but can have different types of elements



```scala
val pair = (99, "Luftballons")
println(pair._1)
println(pair._2)
```

Why we cannot get values out of a Tuple just like list?

Sets and Maps



By default, we are provided with the immutable collections

What is happening here

```
var treasureMap = Map(1->"a", 2->"b")

treasureMap += (3->"c")
```

What happens if we change treasureMap to val?

Why?

Understanding Functional Style

No vars

No methods with side effects – Unit

No while loops

Agenda

- Scala Who & Why?
- First Steps
- Classes and Objects
- Control Structures and Functions
- Inheritance and Traits
- Testing
- Pattern Matching
- Collections
- XML Support

OO Features

Classes

- Classes are blueprints for objects
- class Bank – is a valid class definition

No Semicolon
No access modifier, public by default
No curly braces since no body

Create a new instance

new Bank

```
class Bank(id:Int, name:String)

new Bank(1,"BoA")
   > res5: lab1.Bank = 
lab1$anonfun$main$1$Bank$1@1372a7a
```

Parameters are val and cannot be accessed from outside

```
class Bank(id:Int, name:String)

val b = new Bank(1,"BoA")

b.id
```

Add another parameter to bank

Create another instance of bank

Auxilliary Constructors

```
class Bank(id: Int, name: String) {
   def this(str: String) = this(1, str)
   def this() = this(1, "")
 }
```

Must immediately call another constructor with this

Accessing fields of a class

```
class Bank(id: Int, name: String) {
  val kind = "Blue"
  var address = "Delhi"
  def this(str: String) = this(1, str)
  def this() = this(1, "")
}

val b = new Bank(1, "BoA")
b.address = "Mumbai"
b.kind = "op"
```

immutable

mutable

Making fields from class parameters

```
class Bank(val id: Int, var name: String) {
    val kind = "Blue"
    var address = "Delhi"
    def this(str: String) = this(1, str)
    def this() = this(1, "")
  }
```

See if you can access id and name outside the class now

Make a Bank class with the following immutable parameters
Id, name, address, city

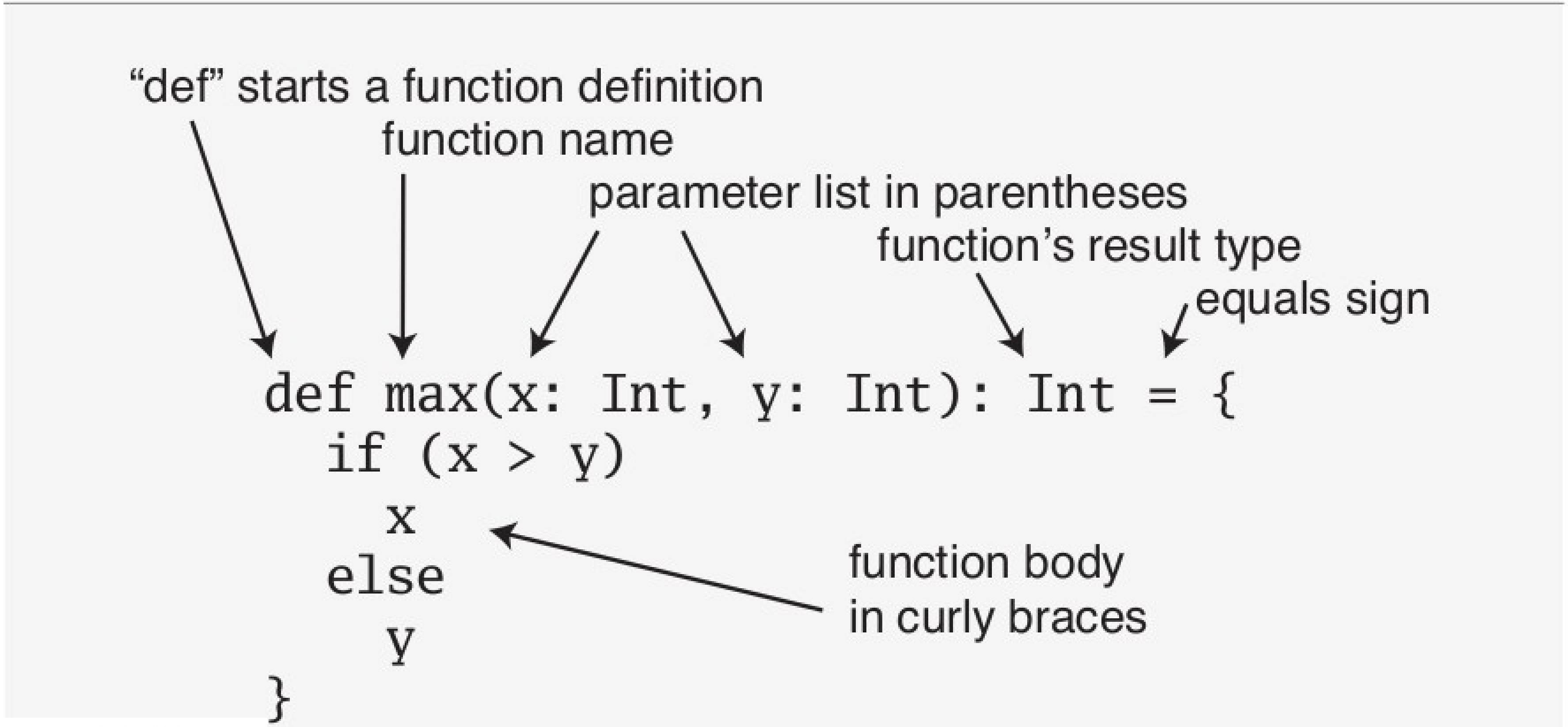
Create a bank instance and access its fields

Make a Account class with the following immutable parameters
Id, name, amount

Create a Account instance and access its fields

Define a debit method in the Account class

def debit(amount:Int):Int = {...}



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

```
class Account(id: Int, name: String, amount: Int) {
  def debit(debitAmount: Int): Int = amount - debitAmount
  def -(debitAmount: Int): Int = amount - debitAmount
 }

 new Account(1, "VikasAccount", 100) - 20
```

Default Parameters

```
class Account(id: Int, name: String, amount: Int = 0) {
    def debit(debitAmount: Int): Int = amount - debitAmount
    def -(debitAmount: Int): Int = amount - debitAmount
  }
```

With default parameters,

```
new Account(1,"Another")
```

Singleton objects

- A singleton object is like a class and its sole instance
- Use the keyword object to define a singleton object:

```
object Foo {
val bar = "Bar"
}
```

Access singleton objects like Foo.bar

Singleton Objects

- Can be thought to be like static in Java but these are real objects which can inherit and be passed around

Companion Objects (not in w/s)

- If a singleton object and a class or trait share the same name, package and file, they are called companions

```
class Bank(val id: Int, var name: String) {
   private val kind = "Blue"
    var address = "Delhi" + Bank.city
    def this(str: String) = this(1, str)
    def this() = this(1, "")
  }

  object Bank {
   private val city = "Delhi"
   val newBank = new Bank(1,"ABC")
   newBank.kind
  }
```

Class or trait can access private member of Companion object

Create a companion object for Account

Create a method in account object to get the type of account like Saving or Checking

Or

Any other method that you like

Understanding Predef

Predef object provides definitions which are accessible in all Scala compilation units without explicit qualification

Aliases for types such as Map, Set and List

Assertions – Require and Ensuring

Console IO – such as print, println, readLine, readInt. These methods are otherwise provided by scala.Console

Add a require assertion to the account that id has to be > 0

Add any other preconditions for accounts

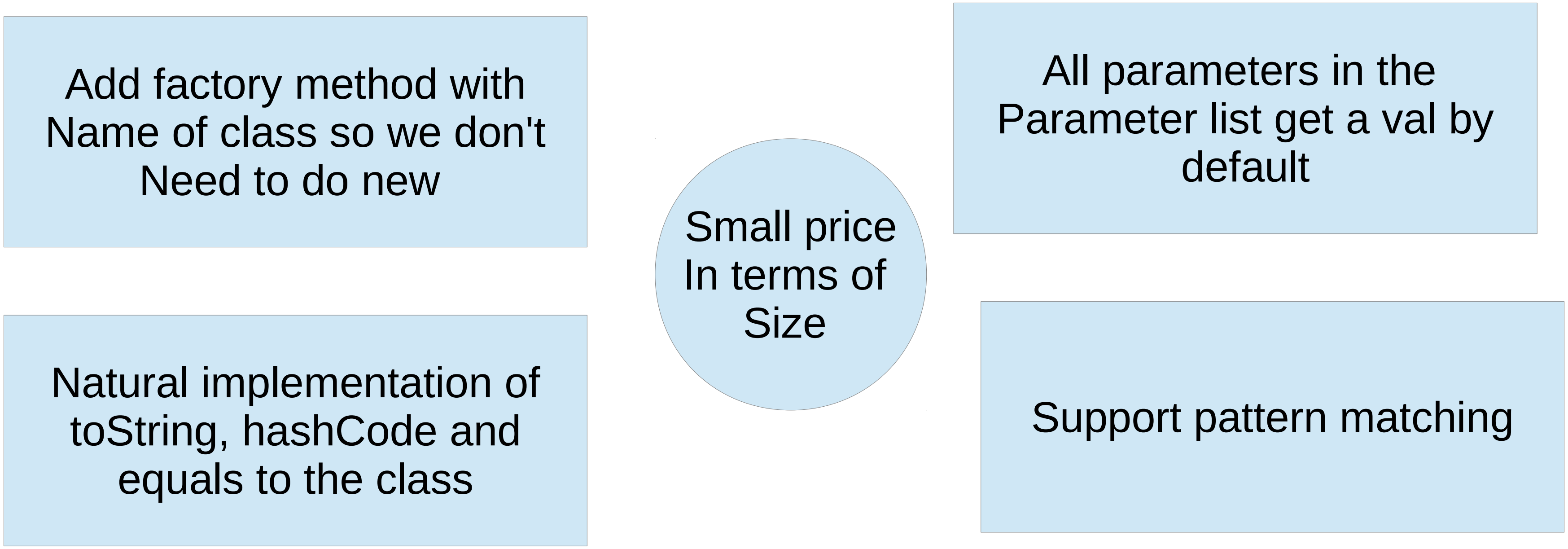
```
  class Bank(val id: Int, var name: String) {
    require(id > 0)
    private val kind = "Blue"
    var address = "Delhi"
    Bank.city
    def this(str: String) = this(1, str)
    def this() = this(1, "")
  }

  object Bank {
    val city = "Delhi"
    val newBank = new Bank(1, "ABC")
  }

  new Bank(0,"ll")                                  >
java.lang.IllegalArgumentException: requirement failed
```

Ensuring gives predicate to a value
If predicate == true return value else AssertionError

```
12 ensuring(true)

def twice(x: Int) = 2 * x ensuring(_ > 0)
twice(3)
twice(-5)
```

Case Classes

Add syntactic convenience to your class



Add factory method with Name of class so we don't Need to do new

All parameters in the Parameter list get a val by default

Small price In terms of Size

Natural implementation of toString, hashCode and equals to the class

Support pattern matching

```
case class Bird(name: String)

 val b = Bird("pigeon")   > b  : lab1.Bird = Bird(pigeon)
 val c = Bird("pigeon")   > c  : lab1.Bird = Bird(pigeon)

 b == c   > res7: Boolean = true
```

Try converting the Bank and Account classes to case classes

Remove vals, new, try out the toString, equals etc

Functional Objects

Are objects that do not have mutable state

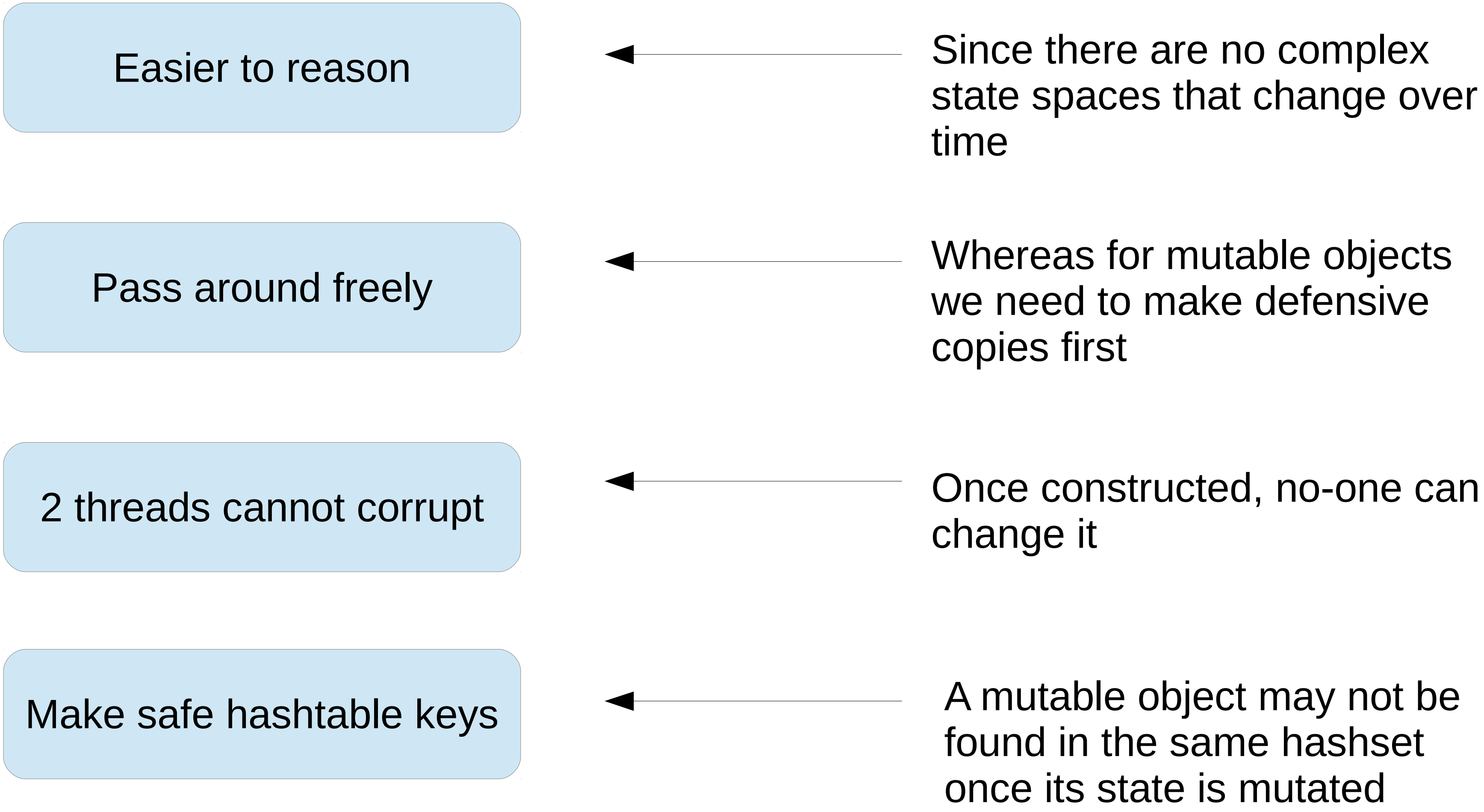
What does it mean?

$$1/2 + 5/6 = 8/6$$

Rational numbers do not have mutable state

When we add two rational numbers we get a new rational number back

Why immutability is good?



Easier to reason ← Since there are no complex state spaces that change over time

Pass around freely ← Whereas for mutable objects we need to make defensive copies first

2 threads cannot corrupt ← Once constructed, no-one can change it

Make safe hashtable keys ← A mutable object may not be found in the same hashset once its state is mutated

VulcanMoney

Since Functional objects cannot be mutated, they need to be constructed properly

All their data is required at the time of creation
The data should be checked for preconditions

```
class VulcanMoney(amount: Int) {
    val value = amount
    require(amount > 0)
    def add(that: VulcanMoney): VulcanMoney = new
VulcanMoney(this.value + that.value)
  }
```

Define methods to substract VulcanMoney
Define methods with + and -

Define method to add an integer to VulcanMoney

new VulcanMoney(10) + 10

Implicit Conversions

Implicit conversion would automatically convert an object of one kind to an object of another kind when needed

```
 implicit def intToVulcanMoney(number:Int) = new
VulcanMoney(number)

  class VulcanMoney(amount: Int) {
    val value = amount
    require(amount > 0)
    def add(that: VulcanMoney): VulcanMoney = new
VulcanMoney(this.value + that.value)
    override def toString = "VM "+value
  }


  new VulcanMoney(10)  add 10  > res8: lab1.VulcanMoney = VM 20
```

Implicit definition needs to be in scope to work. If it was defined inside VulcanMoney, it would not work

```
implicit def intToVulcanMoney(number:Int) = new VulcanMoney(number)

  class VulcanMoney(amount: Int) {
    val value = amount
    require(amount > 0)
    def add(that: VulcanMoney): VulcanMoney = this.value + that.value
    override def toString = "VM "+value
  }

  val vm1 = new VulcanMoney(10)
  val vm2 = vm1  add 10
  vm1.hashCode()                                         > res8: Int = 20037442
  vm2.hashCode()                                         > res9: Int = 17021954
```

Why does this still work?

Agenda

- Scala Who & Why?
- First Steps
- Classes and Objects
- Control Structures and Functions
- Inheritance and Traits
- Testing
- Pattern Matching
- Collections
- XML Support

Built in control structures

Almost all Scala control structures would result in a value

Without this programs must create temporary variables to hold values calculated inside control structures

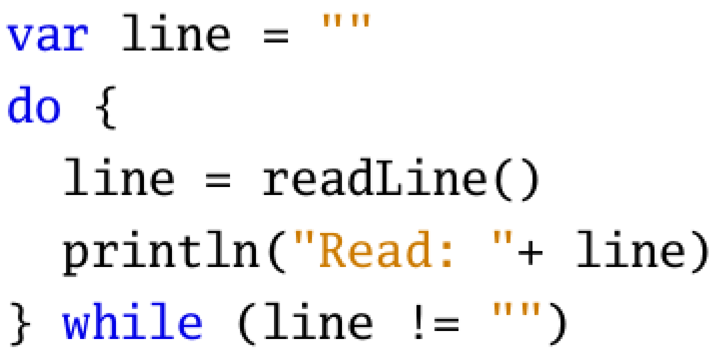
- Conventional way

```
var fileName = "default.txt"
  if (1 < 2) fileName = "newDefault.txt"
```

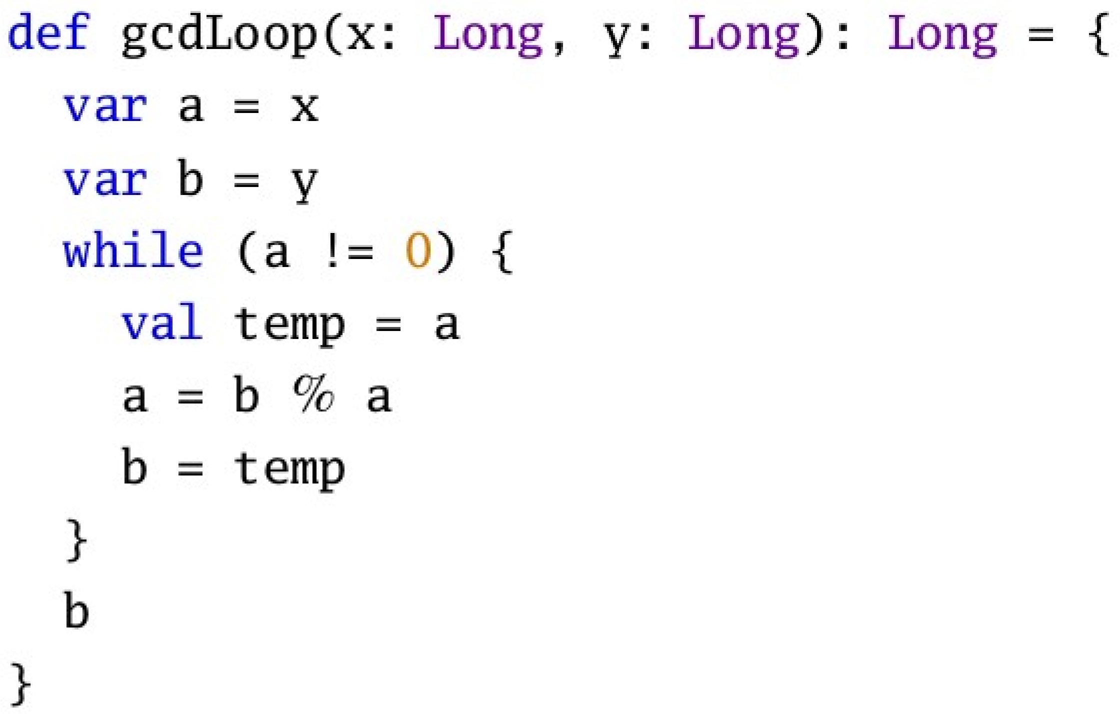
- Scala way

```
val fileNameNew = if (1<2) "newDefault.txt" else "default.txt"
```

Using val



```
var line = ""
do {
  line = readLine()
  println("Read: "+ line)
} while (line != "")
```



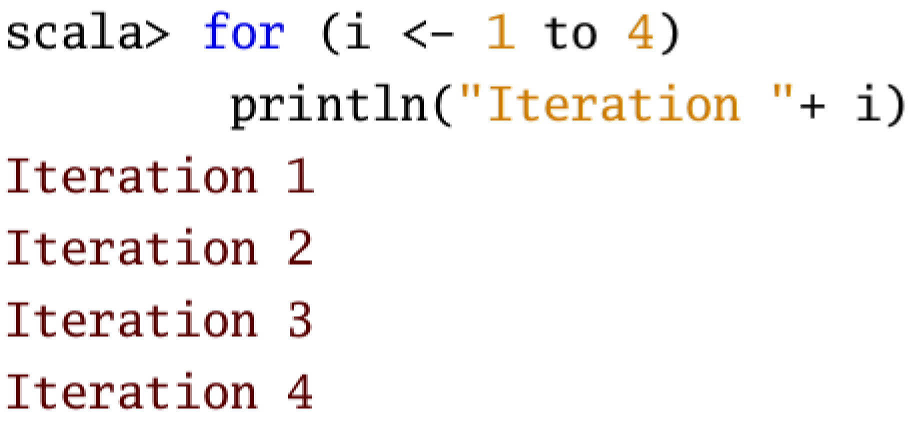
```
def gcdLoop(x: Long, y: Long): Long = {
  var a = x
  var b = y
  while (a != 0) {
    val temp = a
    a = b % a
    b = temp
  }
  b
}
```

Are called loops and
Not expressions because
They do not result in a value

Usually left out of functional
programming

For expression

Without a yield does not result in a value



```
scala> for (i <- 1 to 4)
         println("Iteration "+ i)
Iteration 1
Iteration 2
Iteration 3
Iteration 4
```

With a yield
returns a value

```
for (l <- List(1,2,3,4)) yield if (l%2==0) l else 1

>    res10: List[Int] = List(1, 2, 1, 4)
```

Try Catch

In Scala all exceptions are unchecked exceptions and we need not catch any exception explicitly.

Technically an Exception throw is of the type Nothing

```
 val n = 11
import java.io.FileNotFoundException

  try {
    val bb = if (n % 2 == 0) n / 2 else throw new Exception
  } catch {
    case ex: FileNotFoundException => // Missing file
    case ex: Exception => println("caught exception")
  }                                          > caught exception
```

Match Expression

Unlike Java's Switch statement, match expression results in a value

```
val matchValue = n match {
   case 11 => "eleven"
   case _ => "I dont know"
 }
                  > matchValue  : java.lang.String = eleven

 println(matchValue)
```

Functions

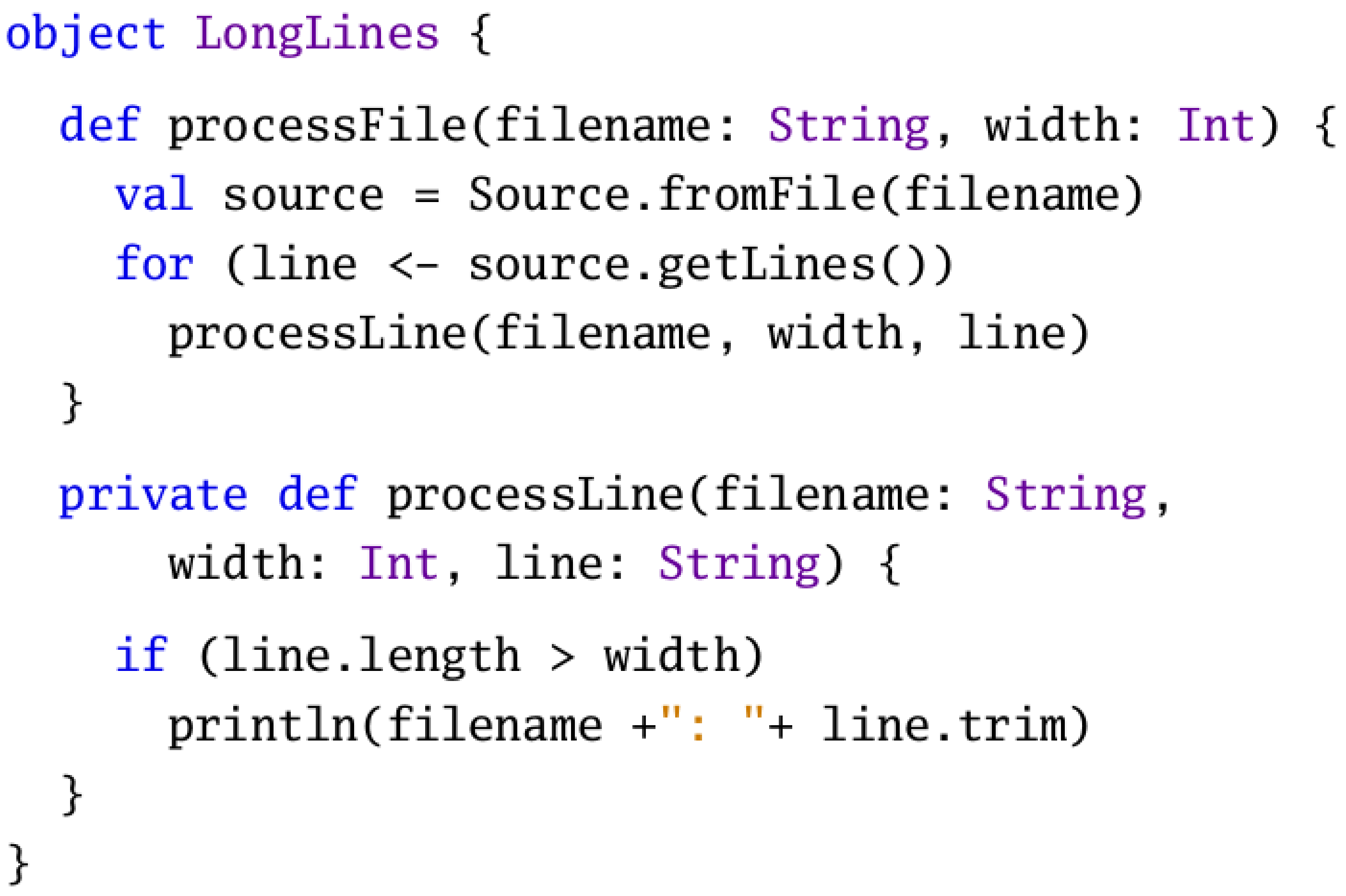
Allow us to code the functionality of the system

Most Functions are member of some object

Functions can be nested within other functions

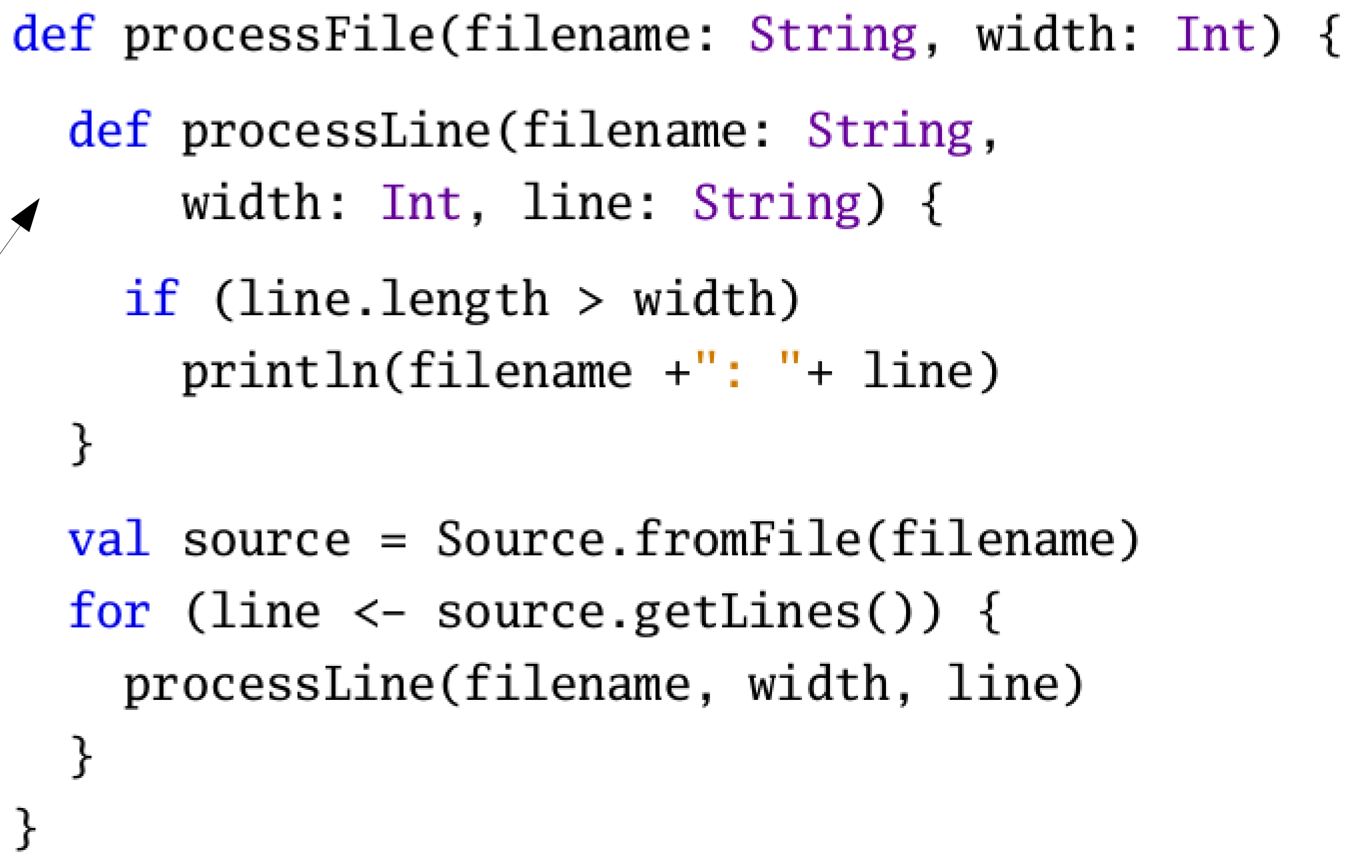
We can define function literals and function values

Regular



```scala
object LongLines {
  def processFile(filename: String, width: Int) {
    val source = Source.fromFile(filename)
    for (line <- source.getLines())
      processLine(filename, width, line)
  }
  private def processLine(filename: String,
      width: Int, line: String) {
    if (line.length > width)
      println(filename +": "+ line.trim)
  }
}
```

Local

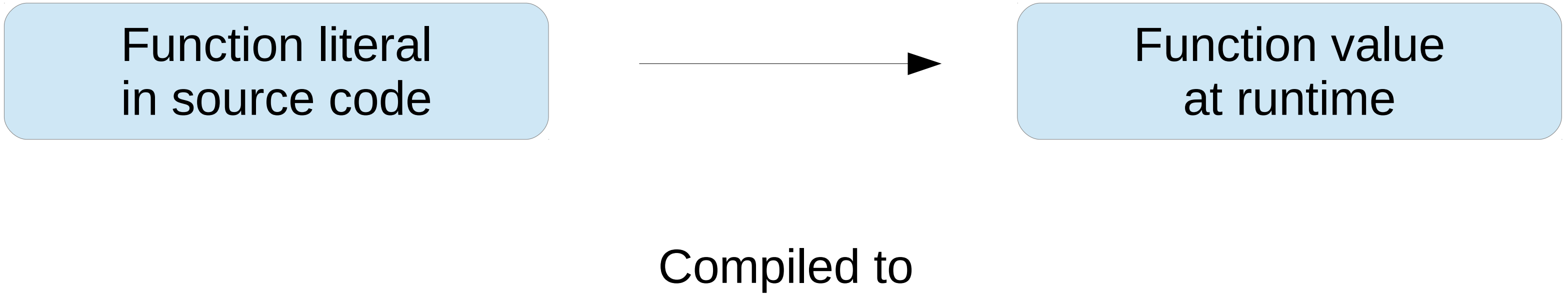


```
def processFile(filename: String, width: Int) {
  def processLine(filename: String,
      width: Int, line: String) {
    if (line.length > width)
      println(filename +": "+ line)
  }
  val source = Source.fromFile(filename)
  for (line <- source.getLines()) {
    processLine(filename, width, line)
  }
}
```

Not accessible outside

Function Literals

Functions can be written as unnamed function literals and passed as values!



Function literal in source code

Function value at runtime

Compiled to

`(x:Int)=> x * 2`

Is a valid function literal, it can be

Assigned to a variable
Passed in another function

```scala
val multiply = (x:Int) => x * 2
```

```scala
multiply (20)
multiply (25)
```

```
> res11: Int = 40
> res12: Int = 50
```

Function literals can be assigned to variables

Short forms for function literals works when the compiler can infer what it would be

```
List(1,2,3) foreach ((x)=>x*2)
```

```
val multiply = (x) => x * 2
```

This would fail

Placeholder syntax, if the parameter appears only one time within the function literal

```
List(1,2,3) foreach ((x)=>x*2)
```

```
List(1,2,3) foreach (_*2)
```

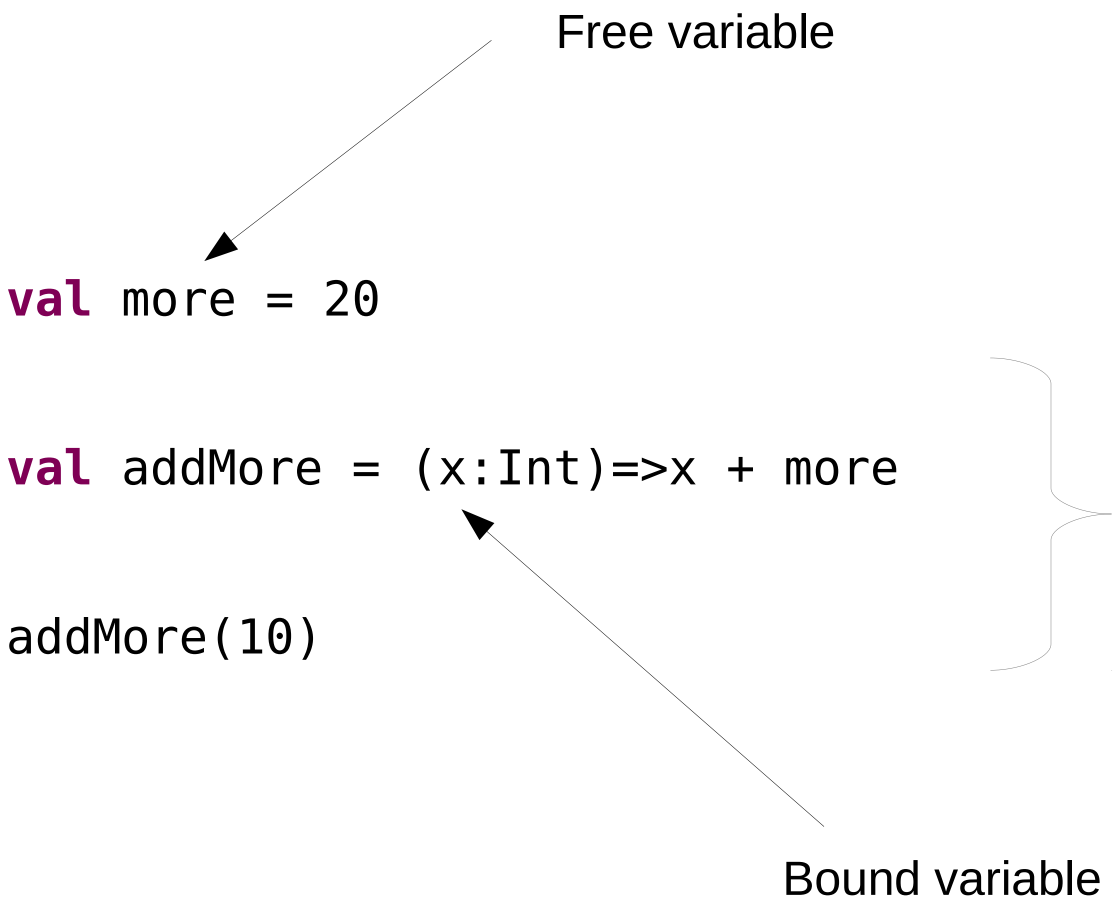
Partially applied functions

Is a function in which we do not supply all the arguments required by the function

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

  val fullyPartial = sum _
> fullyPartial  : (Int, Int, Int) => Int = <function3>
  fullyPartial(1,2,3)                                          > res0: Int = 6

  val somewhatPartial = sum(1, _:Int, 6)
> somewhatPartial  : Int => Int = <function1>
  somewhatPartial(9)                                              > res1: Int = 16
```

```
def meth(x:Int, y:Int):Int = x + y
    > meth: (x: Int, y: Int)Int

def django(f:(Int,Int)=>Int) = {f(1,2)}
    > django: (f: (Int, Int) => Int)Int

django(meth _)
    > res4: Int = 3
```

Closures



Free variable

```scala
val more = 20

val addMore = (x:Int)=>x + more

addMore(10)
```

Bound variable

```
> more  : Int = 20
```

The function value that is created
At runtime is called a closure
because it closes or captures the
bindings of its free variables

Scala captures the variables and
not its values. Hence, changes
made to more are captured

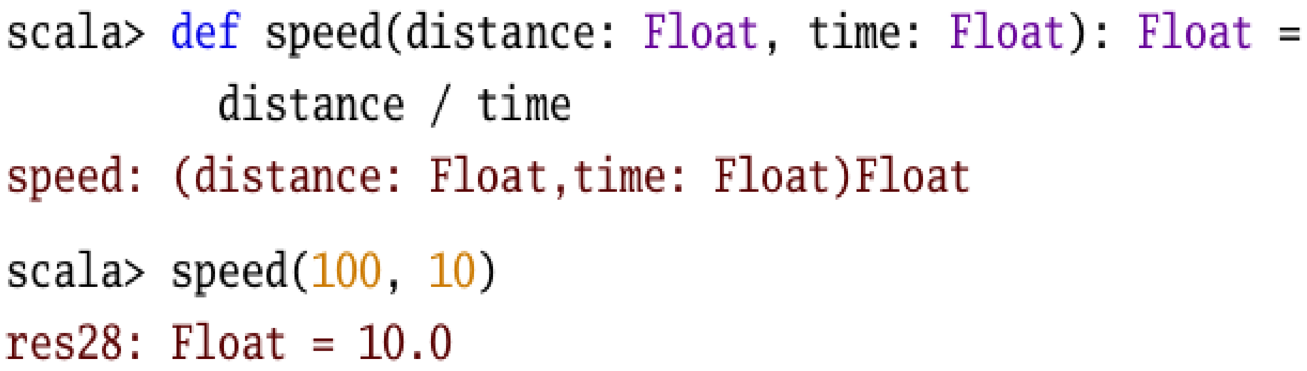
Repeated Parameters

Denoting that we can
Have any number of
strings

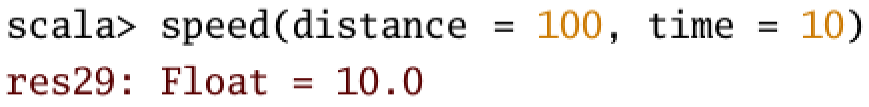
```
def sum(a:Int, b:Int, c:Int, d:String*) = a + b + c

 val a = sum _

 a(1,2,3, "Hello", "Bank", "Of","America")  > res0: Int = 6
```

Only the last parameter can be a repeated parameter

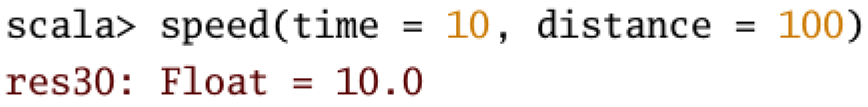
Named Arguments



```
scala> def speed(distance: Float, time: Float): Float =
          distance / time
speed: (distance: Float,time: Float)Float

scala> speed(100, 10)
res28: Float = 10.0
```



```
scala> speed(distance = 100, time = 10)
res29: Float = 10.0
```



```
scala> speed(time = 10, distance = 100)
res30: Float = 10.0
```

Tail Recursion

```
def factorial(number: Int): Int = {
    if (number == 1)
      return 1
    number * factorial(number - 1)
  }
  println(factorial(5))
```

5 * total (5 – 1)

4 * total (4 – 1) = 20

3 * total (3 – 1) = 60

2 * total (2 – 1) = 120

Tail Recursion

```
  def factorial(accumulator: Int, number: Int) : Int = {
  if(number == 1)
    return accumulator
  factorial(number * accumulator, number - 1)
}
println(factorial(1,5))
```

Tail Recursion

```
 def factorial(number: Int): Int = {
   @tailrec
    def factorialWithAccumulator(accumulator: Int, number: Int): Int
= {
      if (number == 1)
        return accumulator
      else
        factorialWithAccumulator(accumulator * number, number - 1)
    }
    factorialWithAccumulator(1, number)
  }
  println(factorial(5))
```

Exercise

```
def sum(s: Seq[Int]): BigInt = {
   if (s.isEmpty) 0 else s.head + sum(s.tail)
 }
```

Write it so that it is tail recursive, check with @tailrec

Currying

```
def add(x:Int)(y:Int) = x + y

 val addWithTwo = add(2) _
 val addWithThree = add(3) _

 addWithTwo(10)
 addWithThree(10)
```

Folding

```
def foldLeft[B](z: B)(f: (B, A) => B): B
```

```
List(1,2,3,4).foldLeft(10)((b,a) => {println(b);b+a})

List(1,2,3,4).foldRight(10)((b,a) => {println(b);b+a})
```

```
List("my","name","is","Ravi").foldRight("- ")((b,a) =>
{println(a);b+a})
```

FoldLeft and FoldRight

```
((1 + 2) + 3) + 4

1 + (2 + (3 + 4))
```

If associations do not matter

Agenda

- Scala Who & Why?
- First Steps
- Classes and Objects
- Control Structures and Functions
- Inheritance and Traits
- Testing
- Pattern Matching
- Collections
- XML Support

Class Inheritance

```
class Animal
class Dog extends Animal
```

Omitting the extends keyword means that we are extending from AnyRef

```
class Animal(name:String)
class Dog(name:String) extends Animal(name)
```

Subclasses must immediately call the superclass constructor

Calling it without the super class constructor would not work

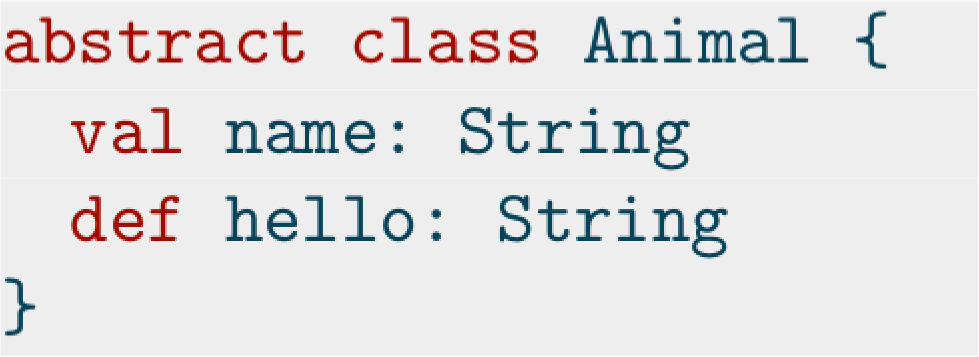
Defining the class as final restricts it from being subclassed

```
final class Animal(name:String)
class Dog(name:String) extends Animal(name)
class Animal(name: String) {
   val kind = "carnivorous"
 }
 class Dog(name: String) extends Animal(name) {
   override val kind = "vegetarian"
 }
```

```
class Dog(name: String) extends Animal(name) {
   override val kind = "vegetarian"
   override val dangerous = true
 }
```

Complains that it
overrides nothing
Hence helps us with
warning

Abstract class



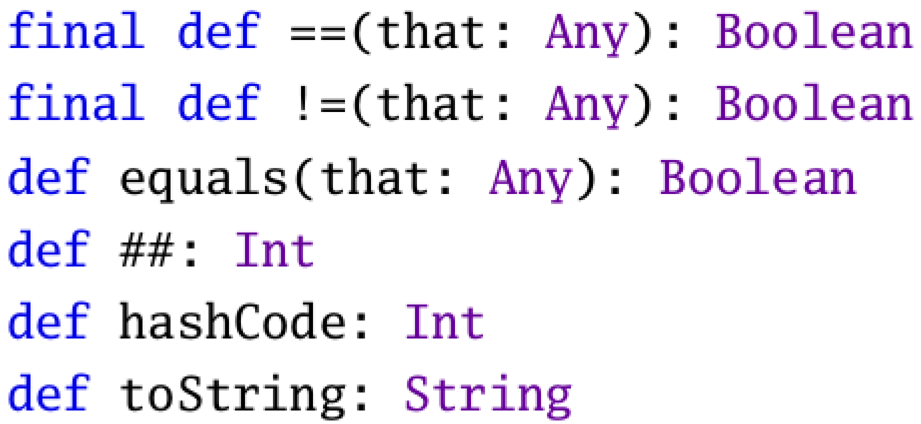
```
abstract class Animal {
  val name: String
  def hello: String
}
```

Cannot be initialized, they need to be subclassed

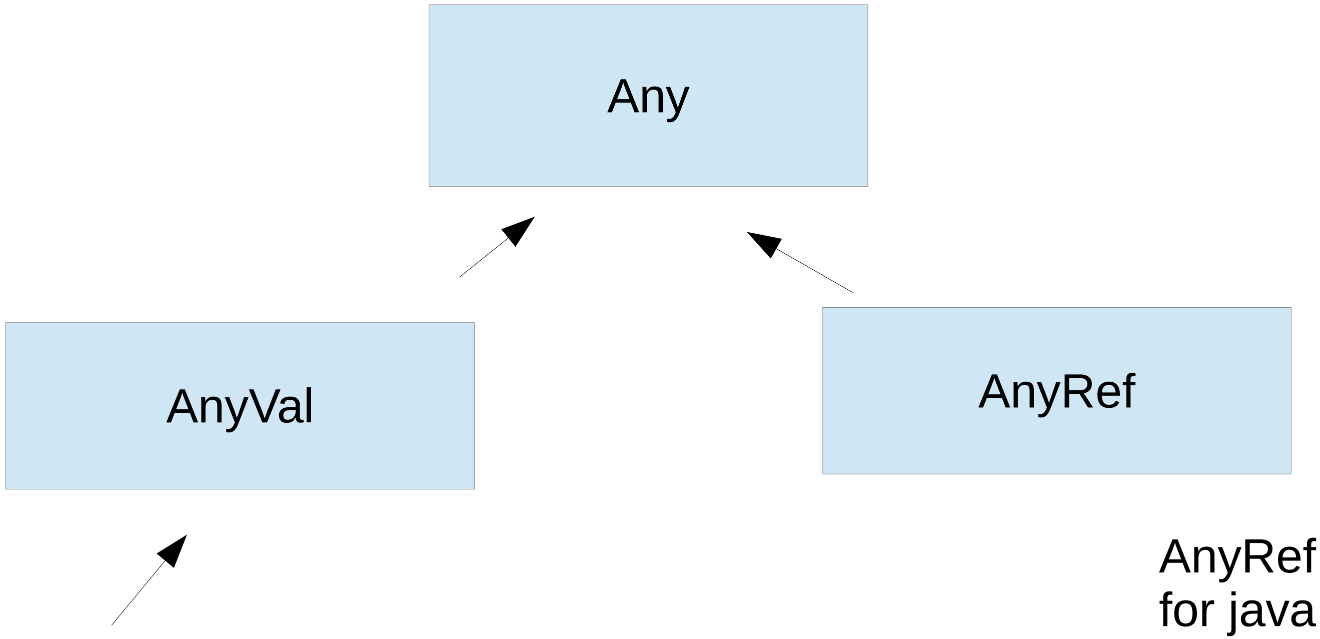
Omitting an initialization or implementation would have to make a class abstract

Scala Hierarchy

- All classes extend from a common superclass called Any
- Methods of Any are universal methods
- Null and Nothing are common subclasses at the bottom of the hierarchy



```
final def ==(that: Any): Boolean
final def !=(that: Any): Boolean
def equals(that: Any): Boolean
def ##: Int
def hashCode: Int
def toString: String
```



Any

AnyVal

AnyRef

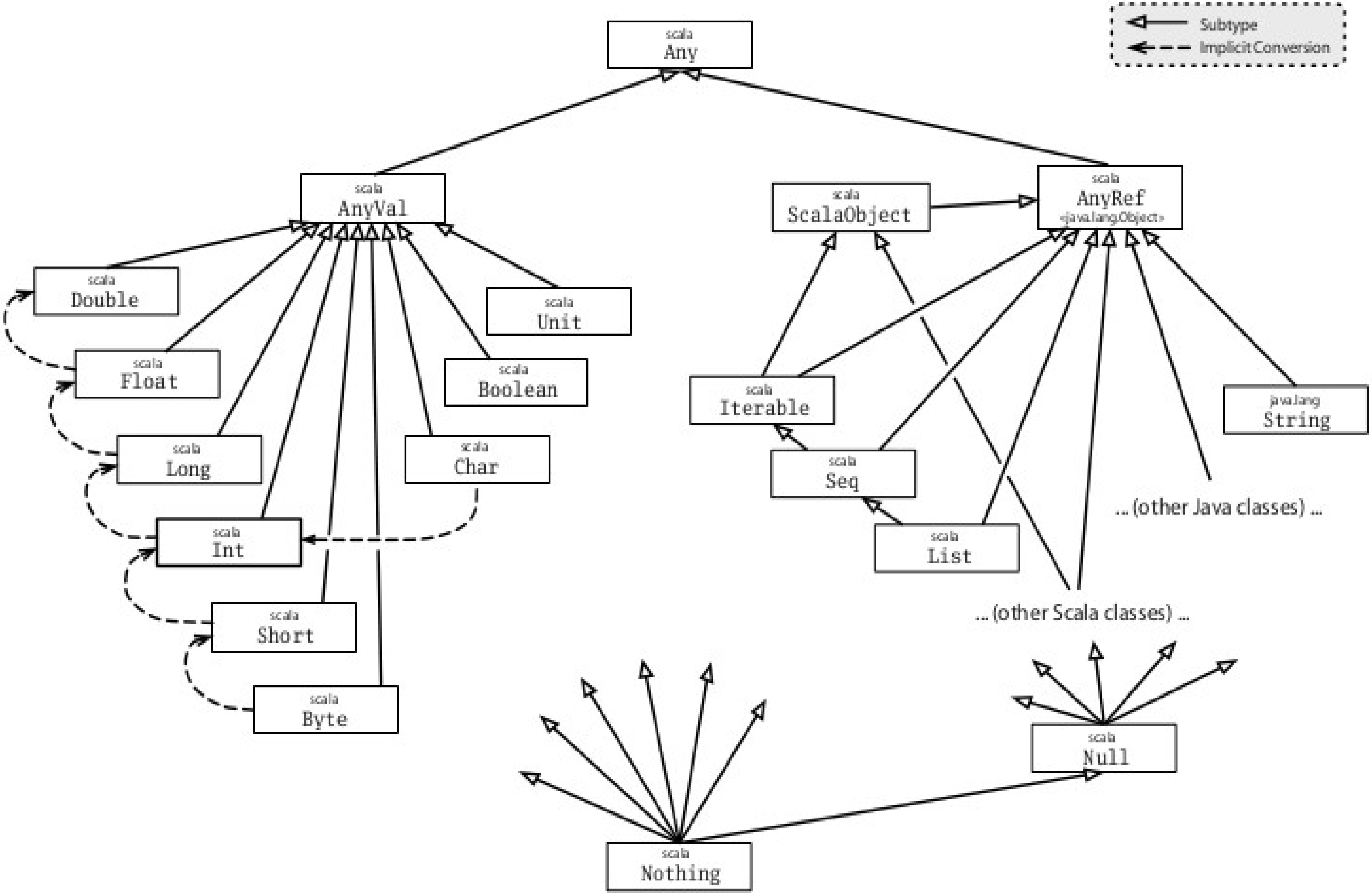
Byte
Short
Char
Int
Long
Float
Double
Boolean
Unit

Instances of these classes are written as literals in Scala

42
Not new Int

AnyRef is a an alias for java.lang.Object

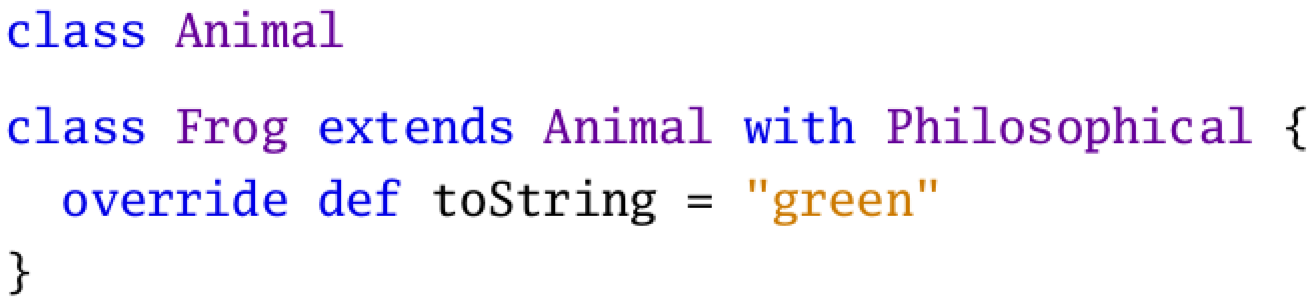
All Scala classes also inherit from special marker trait called ScalaObject which makes execution of Scala programs efficient


Subtype
Implicit Conversion
scala
Any
scala
AnyVal
scala
ScalaObject
scala
AnyRef
«java.lang.Object»
scala
Double
scala
Unit
scala
Float
scala
Boolean
scala
Iterable
java.lang
String
scala
Long
scala
Char
scala
Seq
... (other Java classes) ...
scala
Int
scala
List
... (other Scala classes) ...
scala
Short
scala
Byte
scala
Null
scala
Nothing

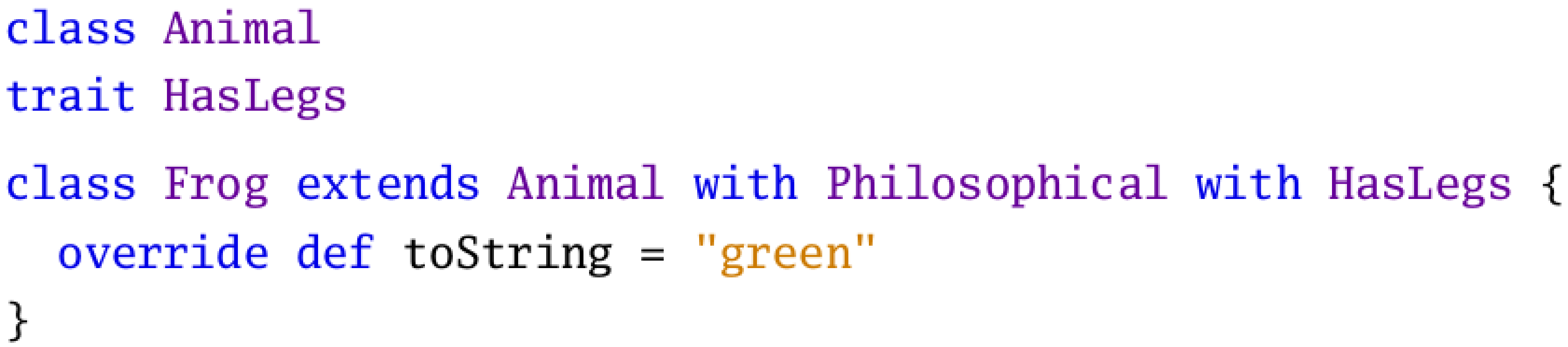
Traits

Fundamental unit of code reuse in Scala

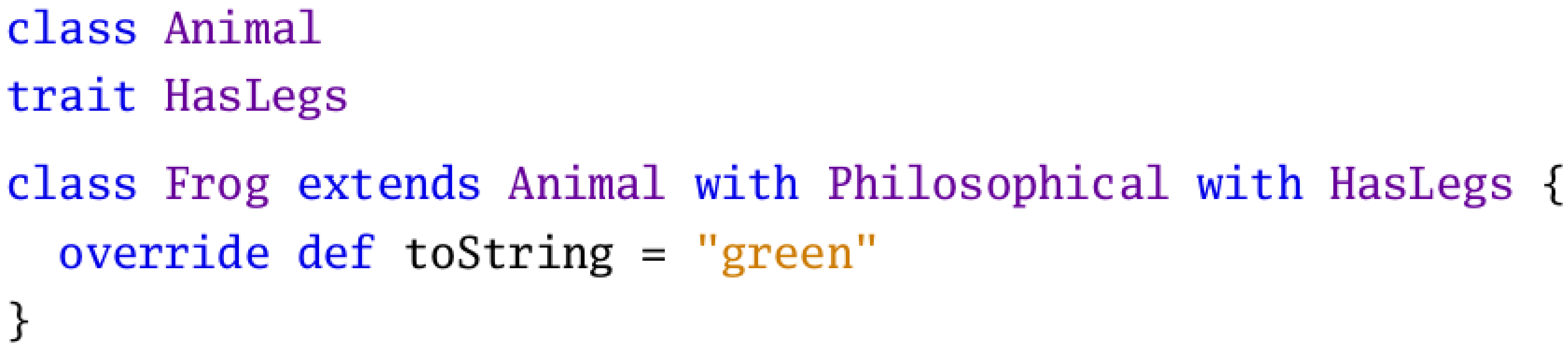
A class can mix in any number of traits but inherit from one SuperClass



```scala
class Animal

class Frog extends Animal with Philosophical {
  override def toString = "green"
}
```

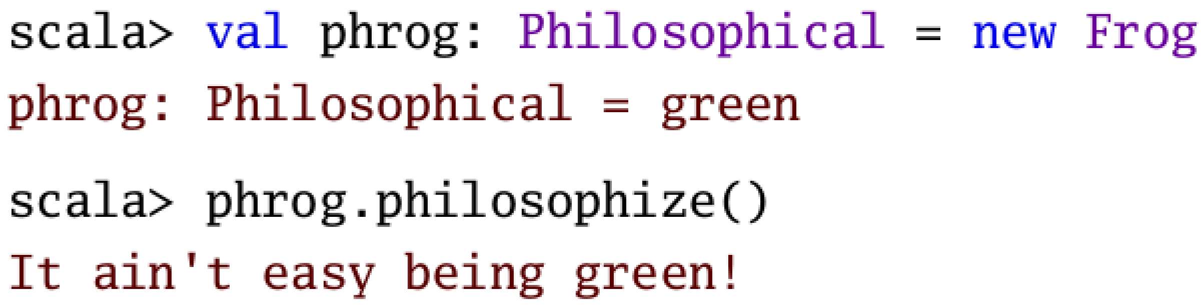


```scala
class Animal
trait HasLegs

class Frog extends Animal with Philosophical with HasLegs {
  override def toString = "green"
}
```

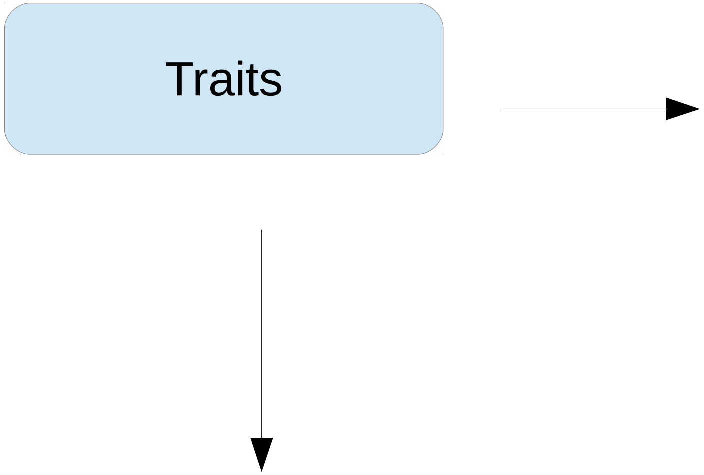


```scala
class Animal
trait HasLegs

class Frog extends Animal with Philosophical with HasLegs {
  override def toString = "green"
}
```

We can use it as variable of type Philosophical



```
scala> val phrog: Philosophical = new Frog
phrog: Philosophical = green

scala> phrog.philosophize()
It ain't easy being green!
```



Traits

Declare fields

Can have state

Define functionality

Anything that can be done in class definition can be done in trait definition as well

Cannot have class parameters (which are passed to primary constructor of class)

Super calls are dynamically bound

Rich Interface

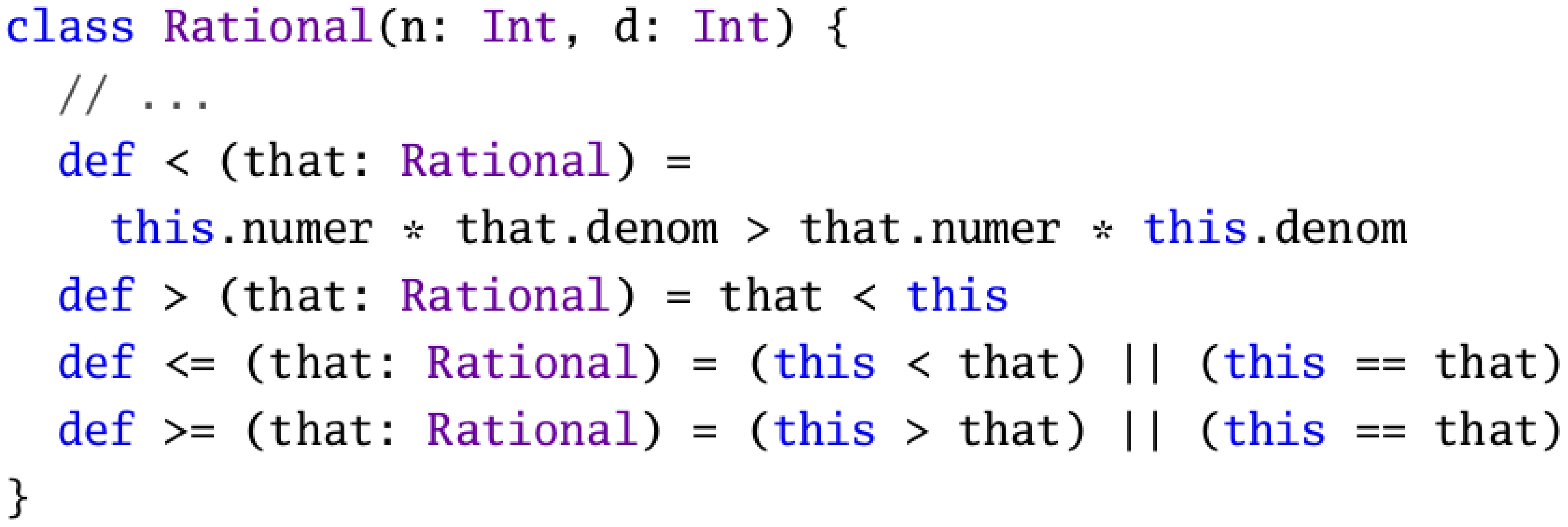
Has many methods, which makes it convenient for caller

Thin Interface

Has less methods which makes it easy for implementors

Scala allows adding concrete implementations in traits which makes it suitable for Rich interfaces + methods which need to be implemented by implementors which is like a thin interface as well

Ordered Trait



```
class Rational(n: Int, d: Int) {
  // ...
  def < (that: Rational) =
    this.numer * that.denom > that.numer * this.denom
  def > (that: Rational) = that < this
  def <= (that: Rational) = (this < that) || (this == that)
  def >= (that: Rational) = (this > that) || (this == that)
}
```

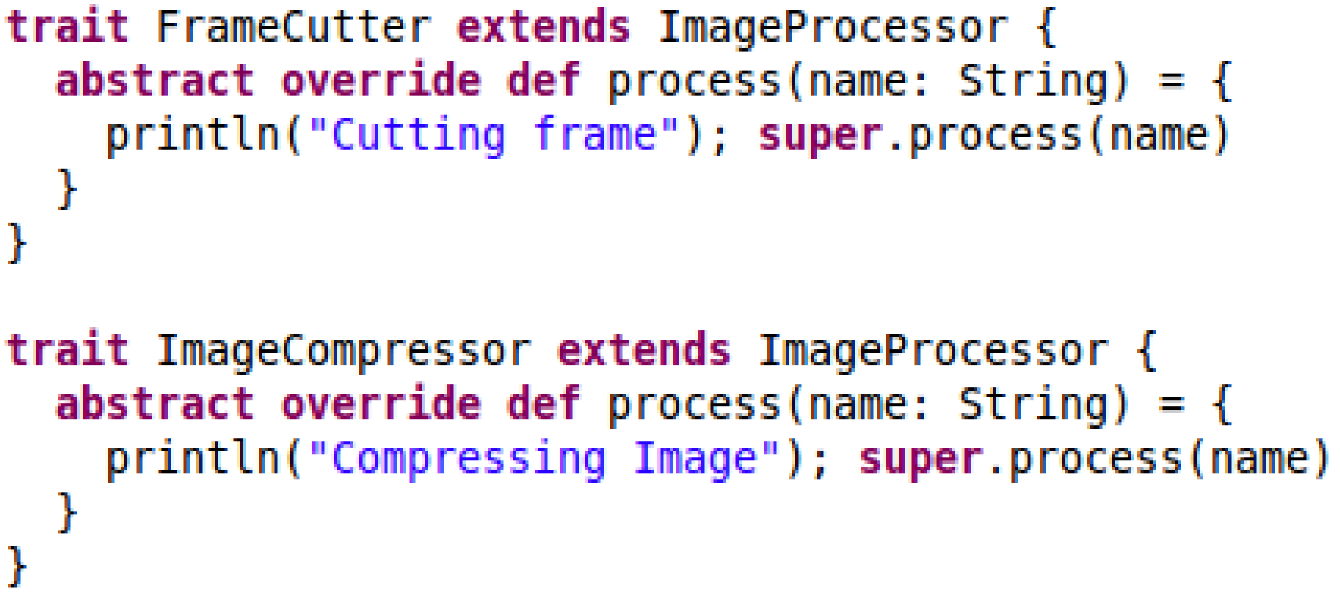
```
class Employee (id:Int, val salary:Int) extends Ordered[Employee]{
  def compare(that:Employee) = this.salary - that.salary
 }

 new Employee(1,100) < new Employee(2,200) > res3: Boolean = true
 new Employee(1,100) == new Employee(2,200)> res4: Boolean = false
 new Employee(1,100) > new Employee(2,200) > res5: Boolean = false
```

We have to define the compare method.

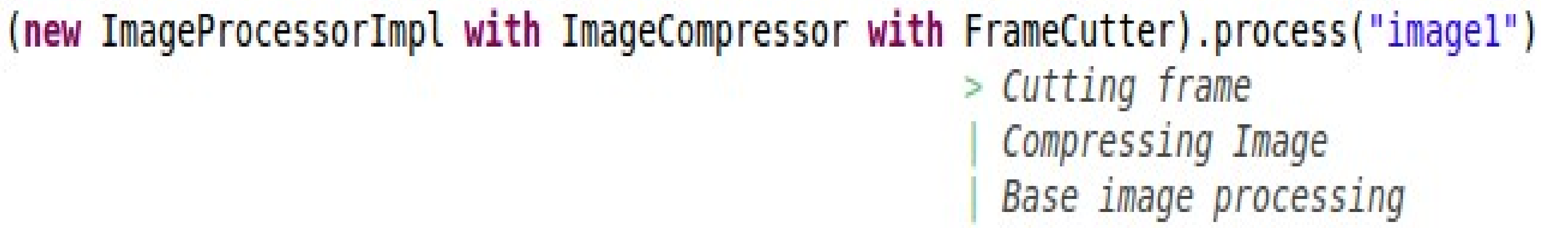
The ordered trait defines the <, >, <= and >= methods for us

Stackable Traits

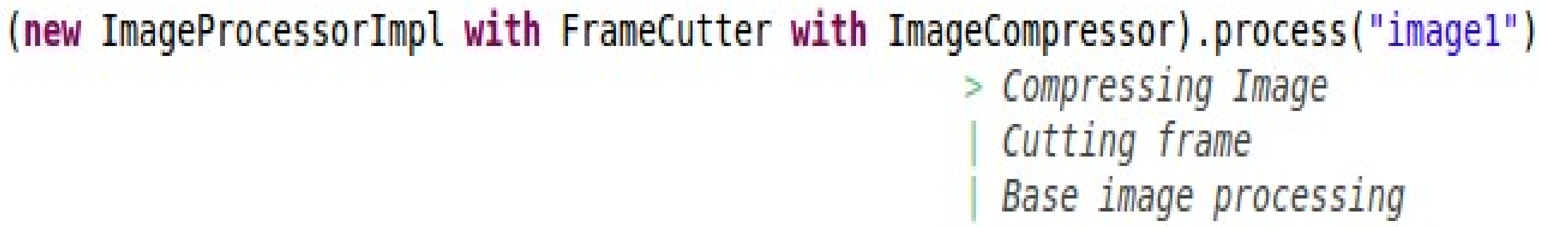
```
abstract class ImageProcessor { def process(name: String) =
println("Base image processing") }

class ImageProcessorImpl extends ImageProcessor { override def
process(name: String) = super.process("myImage") }

(new ImageProcessorImpl).process("image1")       > Base image
processing
```



```scala
trait FrameCutter extends ImageProcessor {
  abstract override def process(name: String) = {
    println("Cutting frame"); super.process(name)
  }
}

trait ImageCompressor extends ImageProcessor {
  abstract override def process(name: String) = {
    println("Compressing Image"); super.process(name)
  }
}
```



```
(new ImageProcessorImpl with ImageCompressor with FrameCutter).process("image1")
                                                          > Cutting frame
                                                          | Compressing Image
                                                          | Base image processing
```



```
(new ImageProcessorImpl with FrameCutter with ImageCompressor).process("image1")
                                                          > Compressing Image
                                                          | Cutting frame
                                                          | Base image processing
```

Stackable Traits

The key things to remember are

i) Behavior of super in traits

ii) Declaring methods with abstract override in traits

iii) Keeping in mind the order of mixin.

Agenda

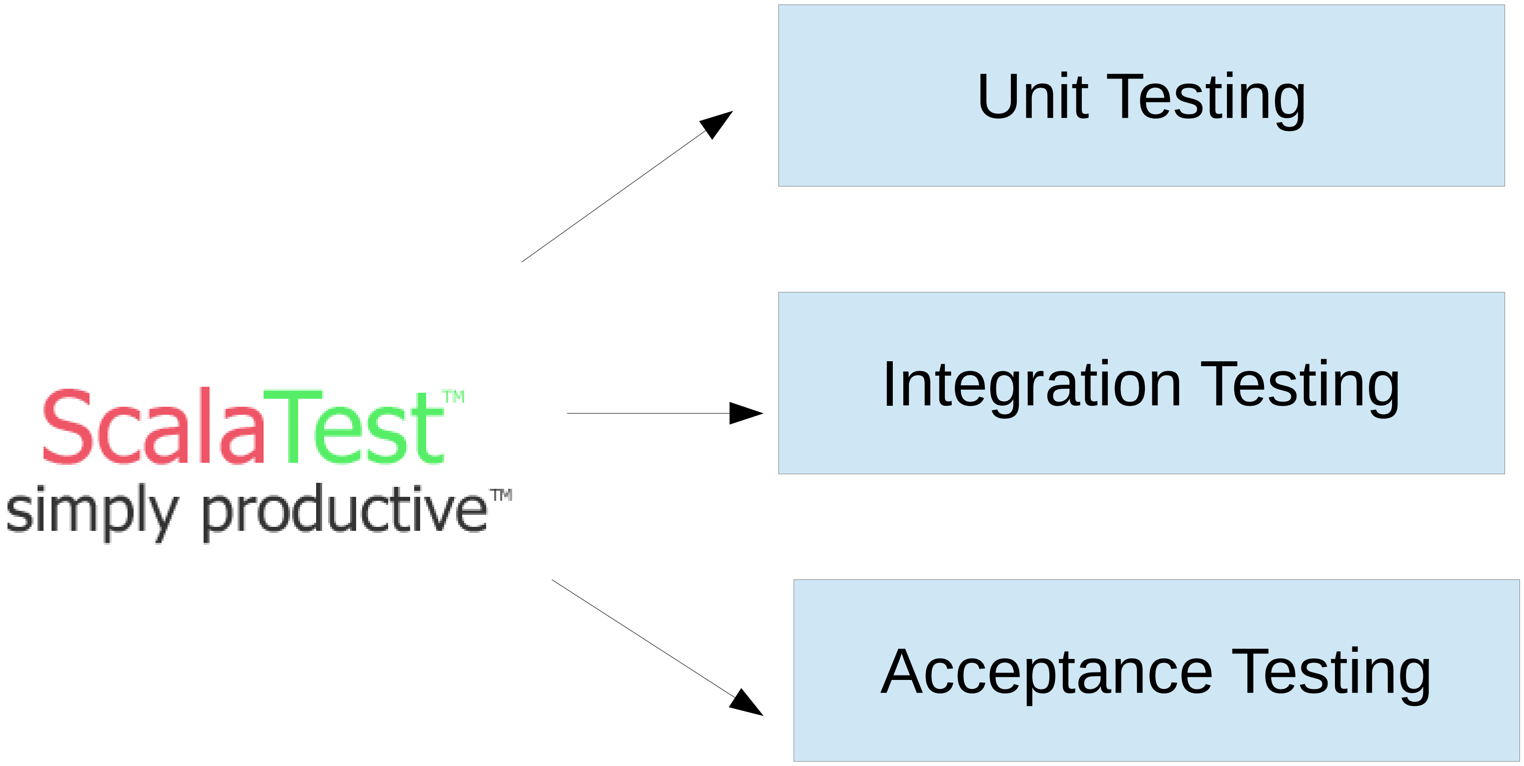
- Scala Who & Why?
- First Steps
- Classes and Objects
- Control Structures and Functions
- Inheritance and Traits
- Testing
- Pattern Matching
- Collections
- XML Support

ScalaTest™

simply productive™

ScalaTest™ is designed to increase your team's productivity through simple, clear tests and executable specifications that improve both code and communication

Just Released - ScalaTest 2.0!



ScalaTest™

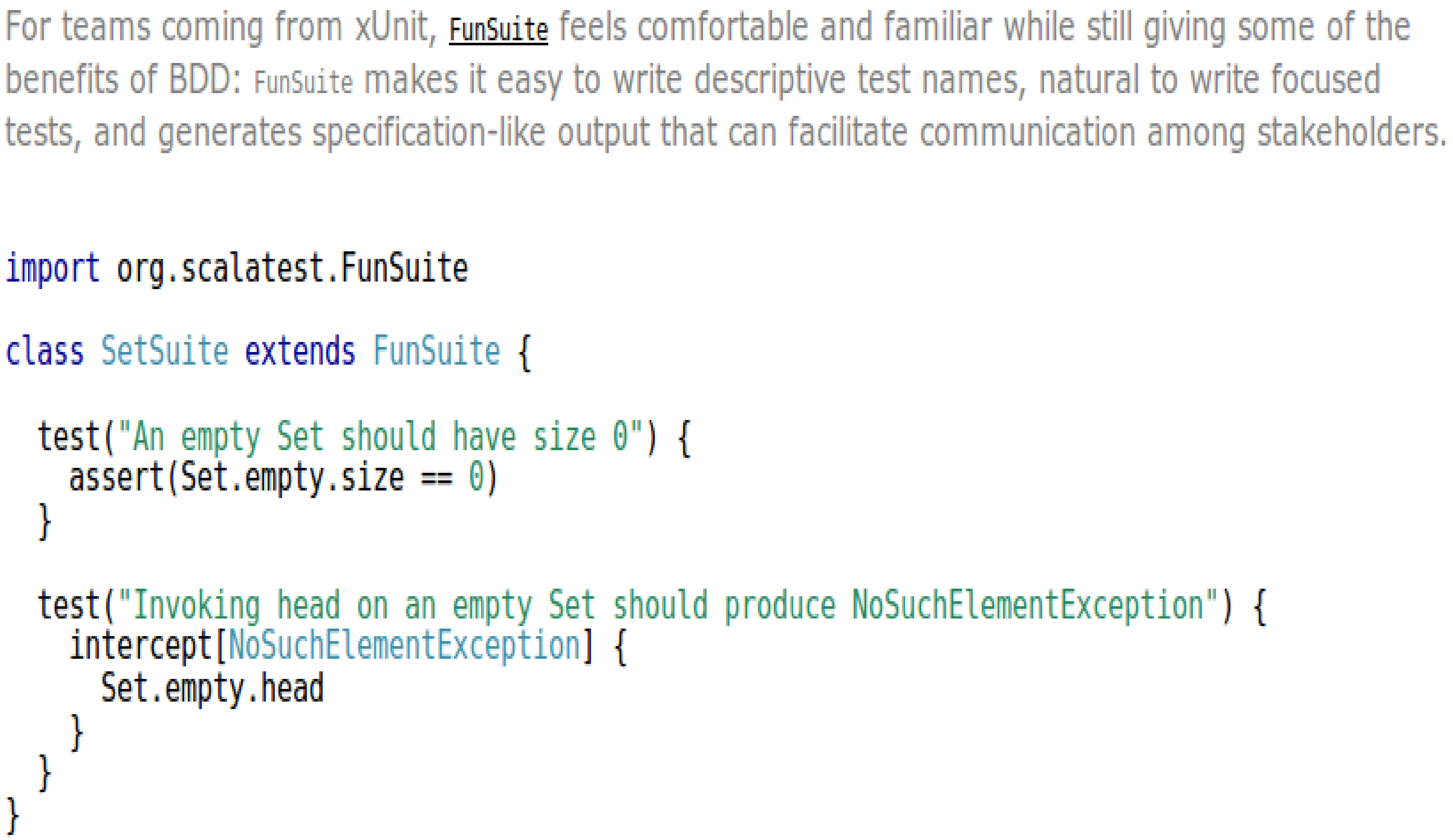
simply productive™

Unit Testing

Integration Testing

Acceptance Testing

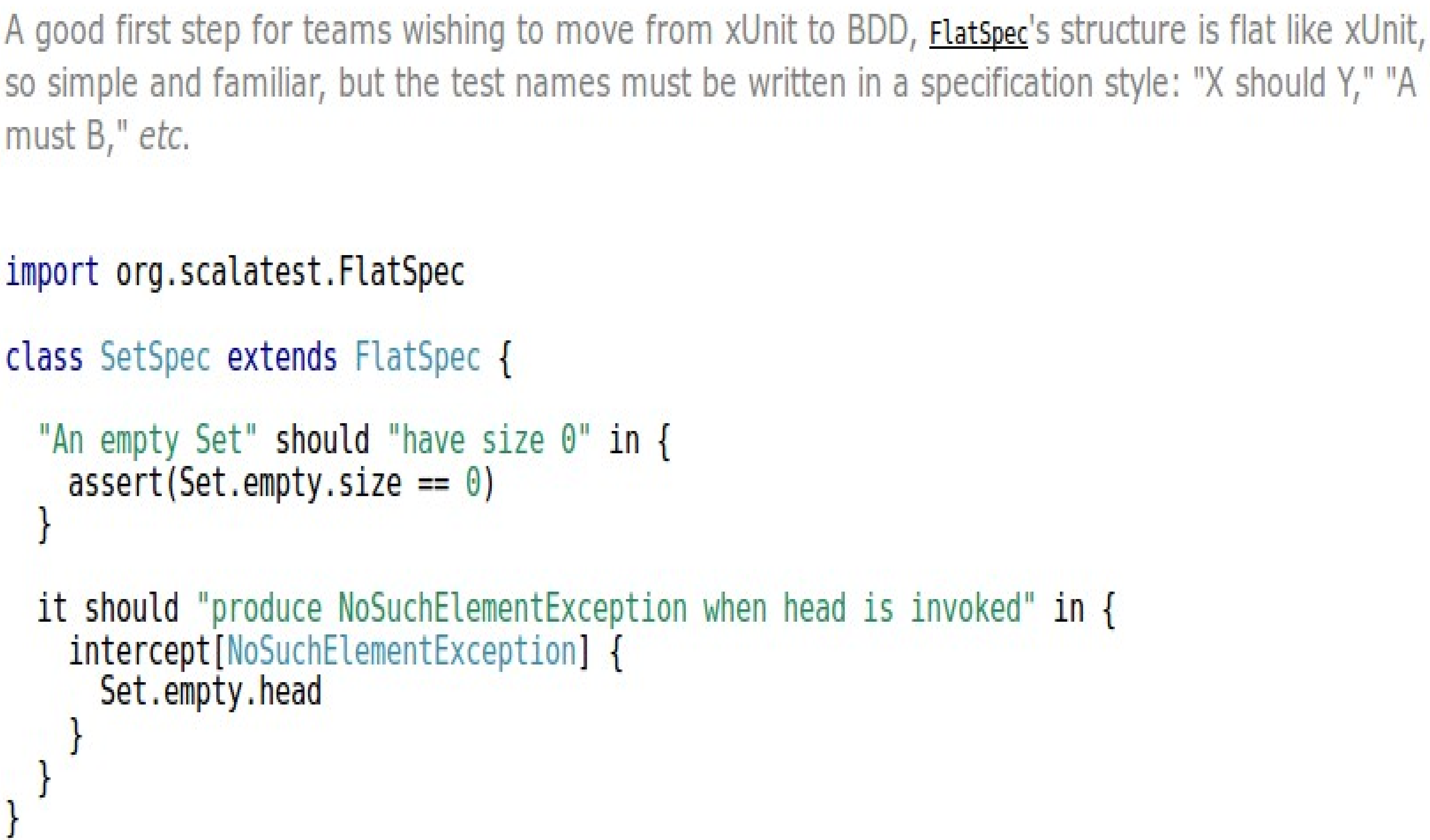
FunSuite



For teams coming from xUnit, FunSuite feels comfortable and familiar while still giving some of the benefits of BDD: FunSuite makes it easy to write descriptive test names, natural to write focused tests, and generates specification-like output that can facilitate communication among stakeholders.

```scala
import org.scalatest.FunSuite

class SetSuite extends FunSuite {

  test("An empty Set should have size 0") {
    assert(Set.empty.size == 0)
  }

  test("Invoking head on an empty Set should produce NoSuchElementException") {
    intercept[NoSuchElementException] {
      Set.empty.head
    }
  }
}
```

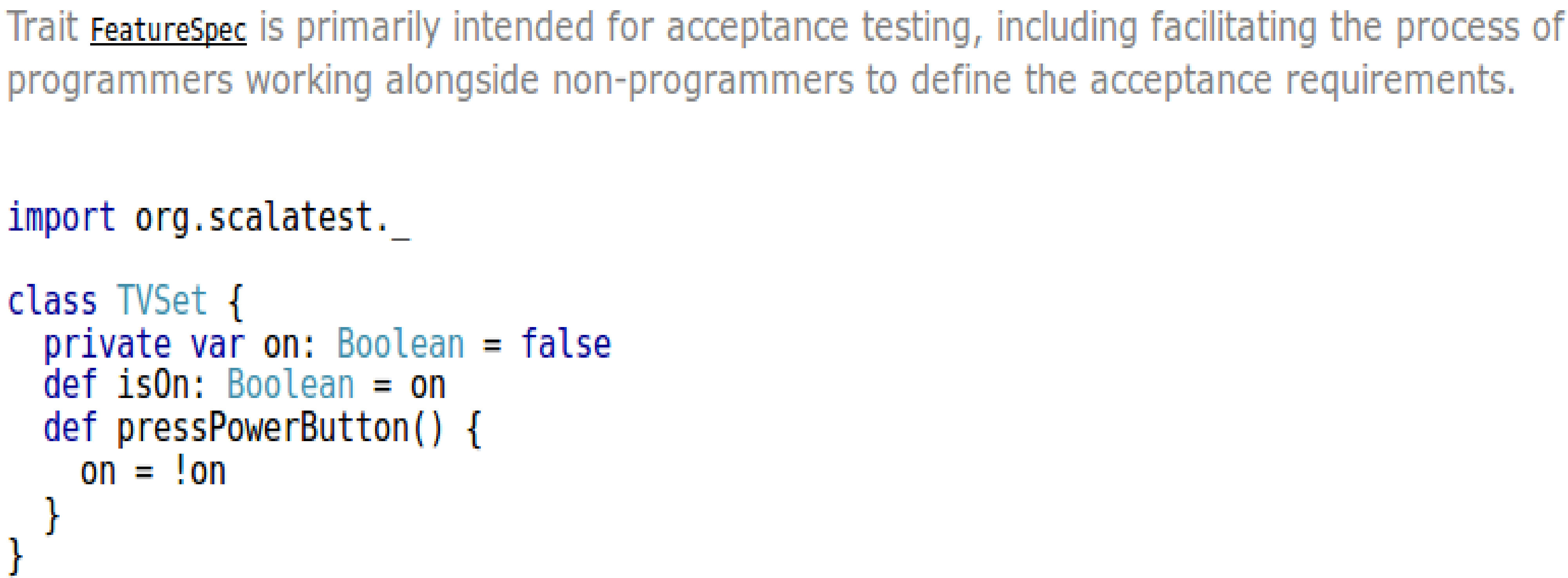
FlatSpec



A good first step for teams wishing to move from xUnit to BDD, FlatSpec's structure is flat like xUnit, so simple and familiar, but the test names must be written in a specification style: "X should Y," "A must B," *etc.*

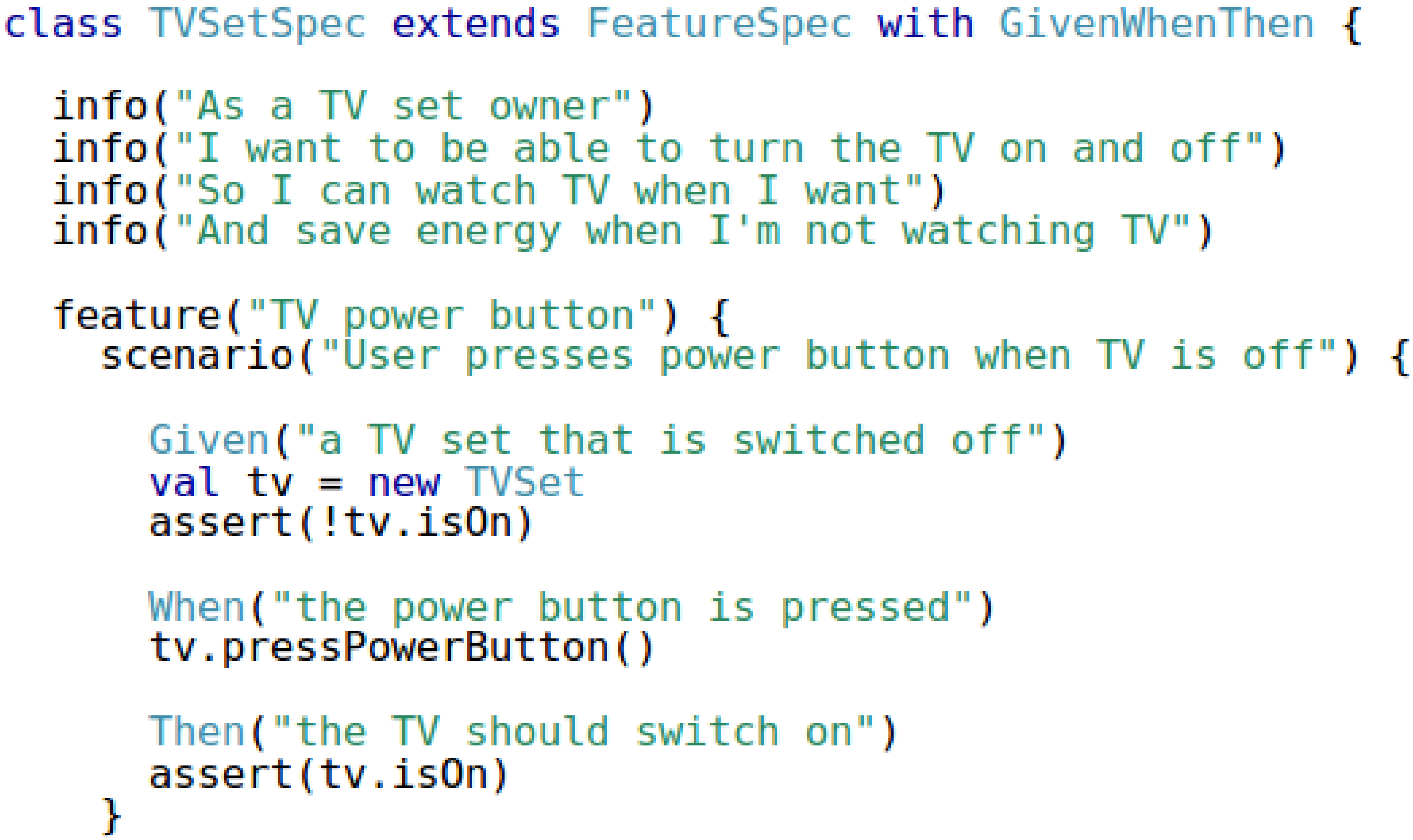
```
import org.scalatest.FlatSpec

class SetSpec extends FlatSpec {

  "An empty Set" should "have size 0" in {
    assert(Set.empty.size == 0)
  }

  it should "produce NoSuchElementException when head is invoked" in {
    intercept[NoSuchElementException] {
      Set.empty.head
    }
  }
}
```

FeatureSpec

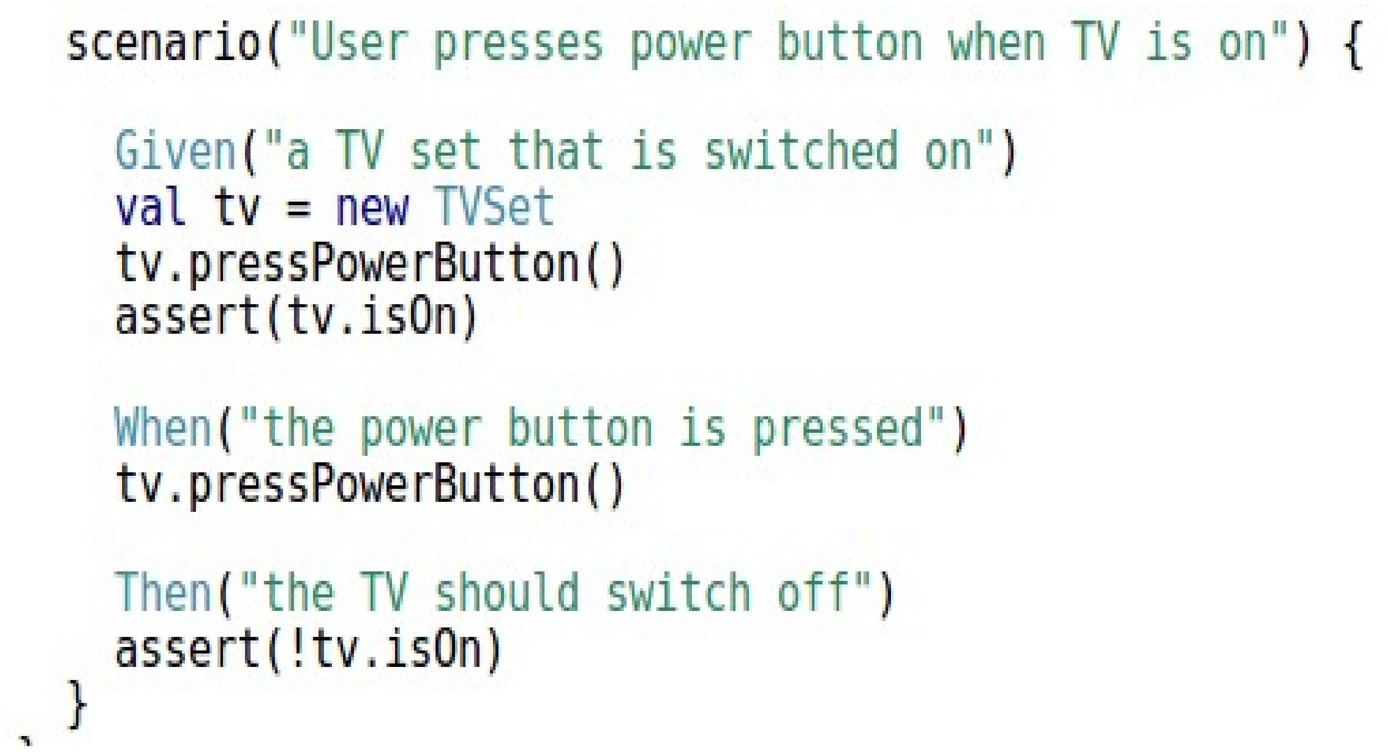


Trait FeatureSpec is primarily intended for acceptance testing, including facilitating the process of programmers working alongside non-programmers to define the acceptance requirements.

```scala
import org.scalatest._

class TVSet {
  private var on: Boolean = false
  def isOn: Boolean = on
  def pressPowerButton() {
    on = !on
  }
}
```



```
class TVSetSpec extends FeatureSpec with GivenWhenThen {

  info("As a TV set owner")
  info("I want to be able to turn the TV on and off")
  info("So I can watch TV when I want")
  info("And save energy when I'm not watching TV")

  feature("TV power button") {
    scenario("User presses power button when TV is off") {

      Given("a TV set that is switched off")
      val tv = new TVSet
      assert(!tv.isOn)

      When("the power button is pressed")
      tv.pressPowerButton()

      Then("the TV should switch on")
      assert(tv.isOn)
    }
```



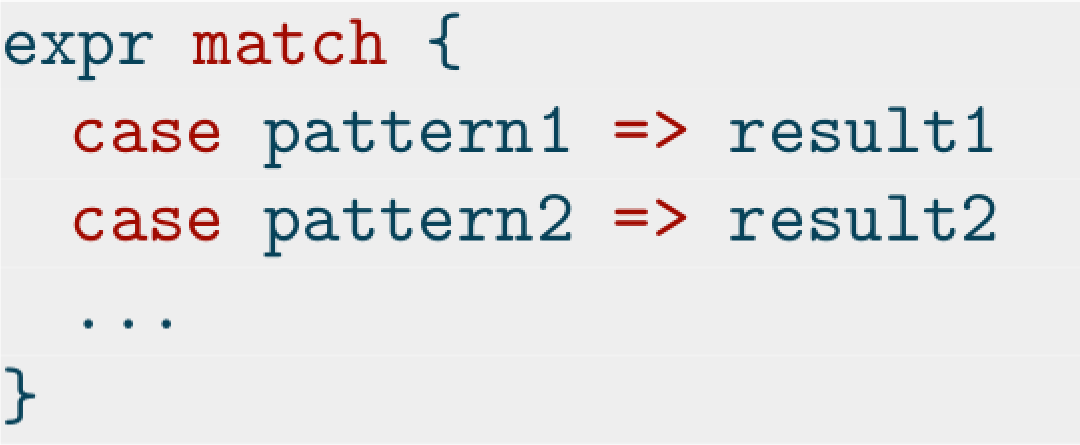
```
scenario("User presses power button when TV is on") {

  Given("a TV set that is switched on")
  val tv = new TVSet
  tv.pressPowerButton()
  assert(tv.isOn)

  When("the power button is pressed")
  tv.pressPowerButton()

  Then("the TV should switch off")
  assert(!tv.isOn)
}
```

```
class Money(amount: Int) {

  require(amount > 0)
}

import org.scalatest.FunSuite
import org.junit.runner.RunWith
import org.scalatest.junit.JUnitRunner

@RunWith(classOf[JUnitRunner])
class MoneyTest extends FunSuite{

  test("Cannot create money with a negative value")
{
    intercept[IllegalArgumentException]{new
Money(-2)}
  }

}
```

Agenda

- Scala Who & Why?
- First Steps
- Classes and Objects
- Control Structures and Functions
- Inheritance and Traits
- Testing
- Pattern Matching
- Collections
- XML Support

General Syntax



```
expr match {
  case pattern1 => result1
  case pattern2 => result2
  ...
}
```

Matching order is top to bottom

No need to give break

As soon as the first match is executed, it breaks

If a pattern matches, result is given. If no pattern matches then MatchError is thrown

Wildcard pattern

```
val name = "vikas"

 name match {
   case "Sachin" => println("Sud")
   case "Virender" => println("Virender")
   case "Dhoni" => println("Dhoni")
   case "Vikas" => println("Vikas")
   case _ => println("Nothing matched")
 }
```

Put wildcard match as the last alternative to prevent MatchError

Wildcard match

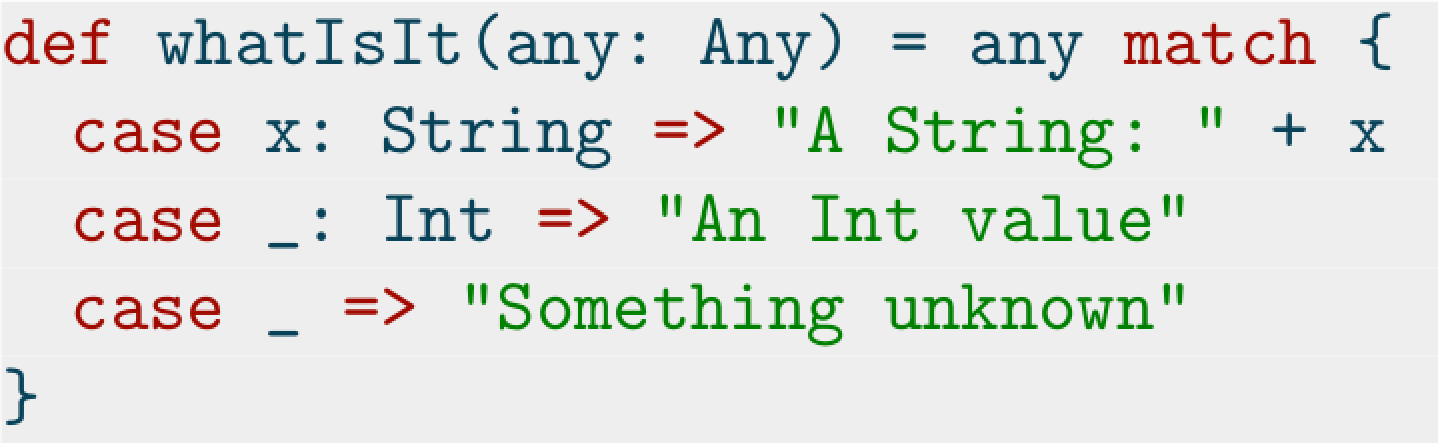
Variable pattern

```
val name = "vikas"                                > name   : java.lang.String = vikas

  name match {
    case x => println("Vikas")
  }
```

Variable given with a small name would match anything. It is used to assign the match to the variable

```
val name = "vikas"                                > name   : java.lang.String = vikas

 name match {
   case x => println("Vikas" + x)
 }
```

Typed Pattern



```
def whatIsIt(any: Any) = any match {
  case x: String => "A String: " + x
  case _: Int => "An Int value"
  case _ => "Something unknown"
}
```

Matches certain types only

Is usually combined with wildcard or variable pattern

Tuple Match

```
val value = (1-> "2")
  value   match {
  case (i:Int, "3") => 12
  case (12, x:String) => 13
  case _ => 14
  }
```

```
> value  : (Int, java.lang.String) = (1,2)

                    > res6: Int = 14
```

Constructor Pattern Match

```
case class Employee(id: Int, val salary: Int)
```

```
val value:Any = new Employee(1,100)
 value   match {
 case Employee(1, x:Int) => 12
 case (12, x:String) => 13
 case _ => 14
 }
```

```
> value  : Any = Employee(1,100)

> res6: Int = 12
```

Pattern Guards

```
x match{
        case x if(x%3==0) => println("Knol")
        case x if(x%5==0) => println("Dus")
        case x if(x%3==0&& x%5==0) => println("KnolDus")
        case x _ => printf("%d",n)
       }
```

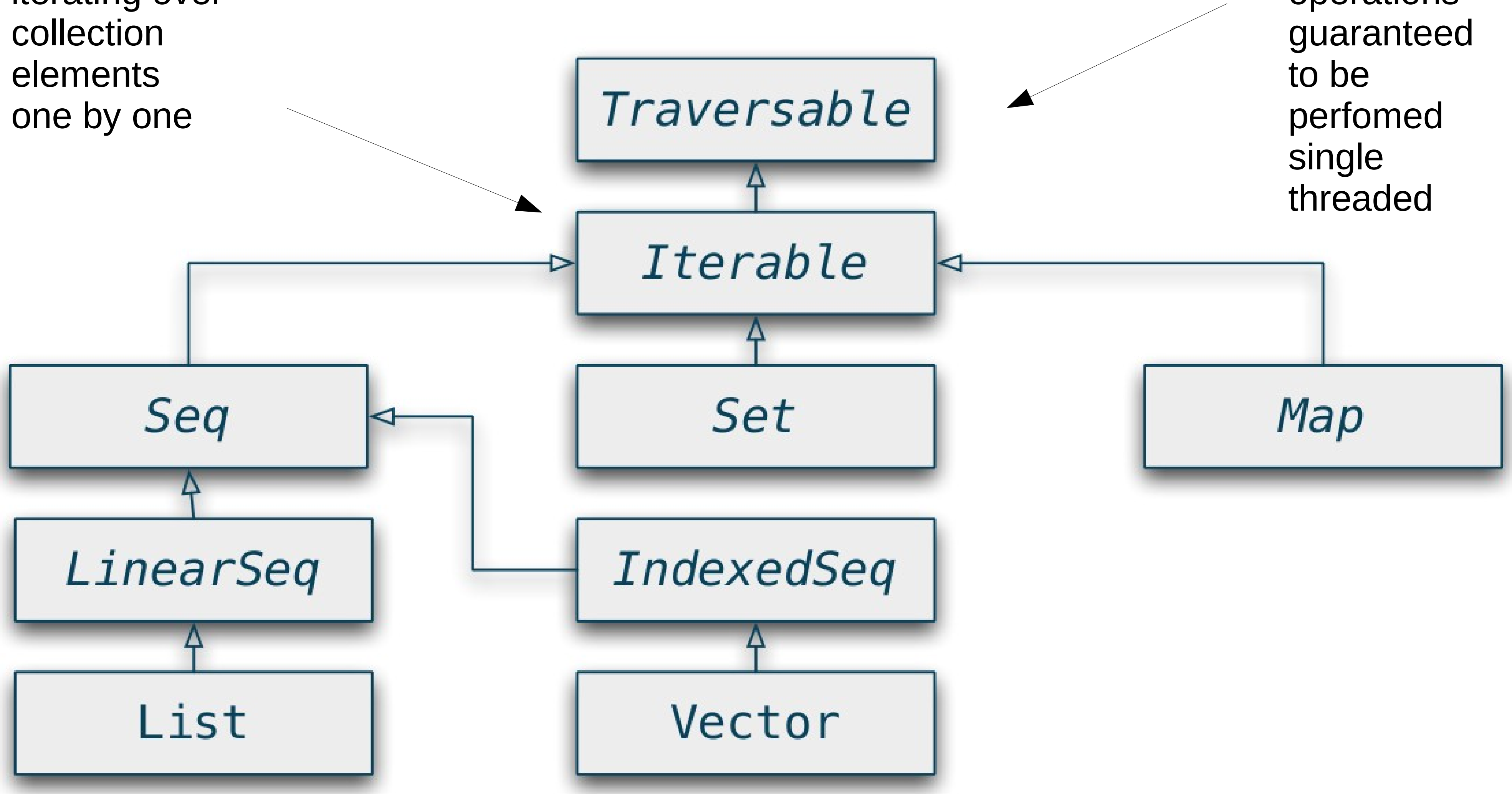
If to define a pattern guard condition

Agenda

- Scala Who & Why?
- First Steps
- Classes and Objects
- Control Structures and Functions
- Inheritance and Traits
- Testing
- Pattern Matching
- Collections
- XML Support

Collections Hierarcy

Would be iterating over collection elements one by one



All operations guaranteed to be perfomed single threaded

Similar way of creation

Class name followed by a comma separated list of items

```
List(1,2,3)                          > res7: List[Int] = List(1, 2, 3)

Seq(1,2,3)                          > res8: Seq[Int] = List(1, 2, 3)

IndexedSeq(1,2,3)                   > res9: IndexedSeq[Int] = Vector(1, 2, 3)
```

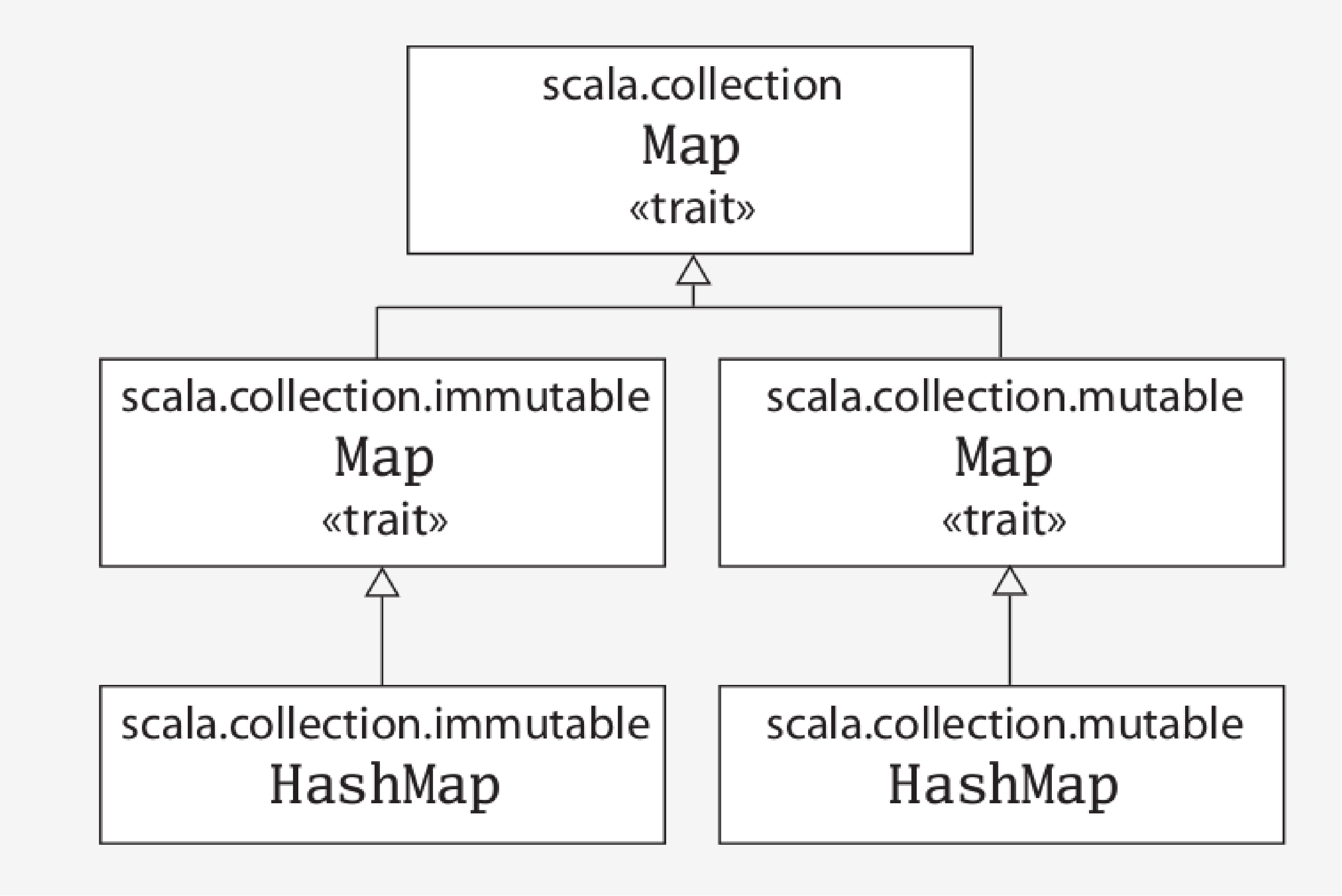
```
List(1,2,3)
```

```
> res7: List[Int] = List(1, 2, 3)
```

All collections are parameterized.
There are no raw collections

We end up calling
List.apply(1,2,3) here

Immutable and Mutable



Immutability

```
val numbers = List(1,2,3)     > numbers  : List[Int] = List(1, 2, 3)
  numbers :+ 4                > res7: List[Int] = List(1, 2, 3, 4)
  numbers                     > res8: List[Int] = List(1, 2, 3)
```

By default we get immutable collections

Some operations

$++$ appends two collections

toSeq, *toSet*, etc. turns a collection into a specific one

isEmpty and *size* for information regarding size

contains tests whether a collection contains an element

head for the first element, *last* for the last

tail for everything except for the first element, *init* for everything except for the last

take gets the first n elements, *drop* gets all elements except for the first n

groupBy partitions a collection into a *Map* of collections according to some discriminator function

For *Seqs*: +:[18] prepends an element, :+ appends one

For *Lists*: ::[19] ("Cons") prepends an element

For *Maps*: *getOrElse* returns the value for the given key or the given default

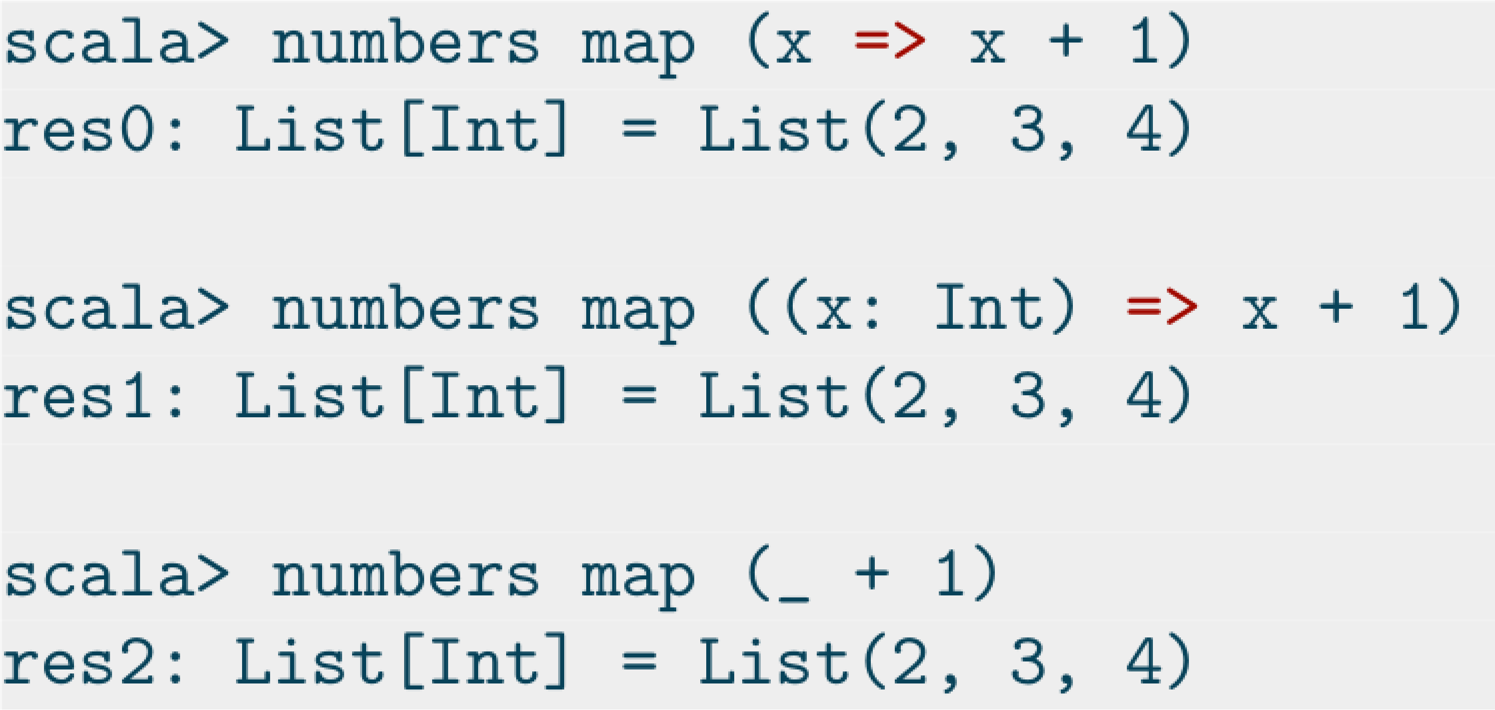
Functional Collections

Collections have a lot of methods which take functions as arguments

Eg foreach

```
val numbers = List(1,2,3)
 numbers map (_+1)
```

```
> numbers  : List[Int] = List(1, 2, 3)
> res7: List[Int] = List(2, 3, 4)
```



```
scala> numbers map (x => x + 1)
res0: List[Int] = List(2, 3, 4)

scala> numbers map ((x: Int) => x + 1)
res1: List[Int] = List(2, 3, 4)

scala> numbers map (_ + 1)
res2: List[Int] = List(2, 3, 4)
```

```
val addOne = (x:Int)=>x+1 > addOne   : Int => Int = <function1>

 val numbers = List(1,2,3)> numbers   : List[Int] = List(1, 2, 3)

 numbers map (addOne)      > res7: List[Int] = List(2, 3, 4)
```

Function Types

Since functions are objects they need to have a type as well

```
 val addOne = (x:Int)=>x+1         > addOne  : Int => Int = <function1>

 val addTwo = (x:Int, y:Int)=>x+y > addTwo  : (Int, Int) => Int =
<function2>
```

map

map transforms the existing collection into a new one, it works on every element of the collection

```
val bankList = List("Bank of America", "Citibank", "HDFC")

 bankList map (_.toLowerCase)
```

flatMap

The function maps each element to a collection which is then combined into a complete collection

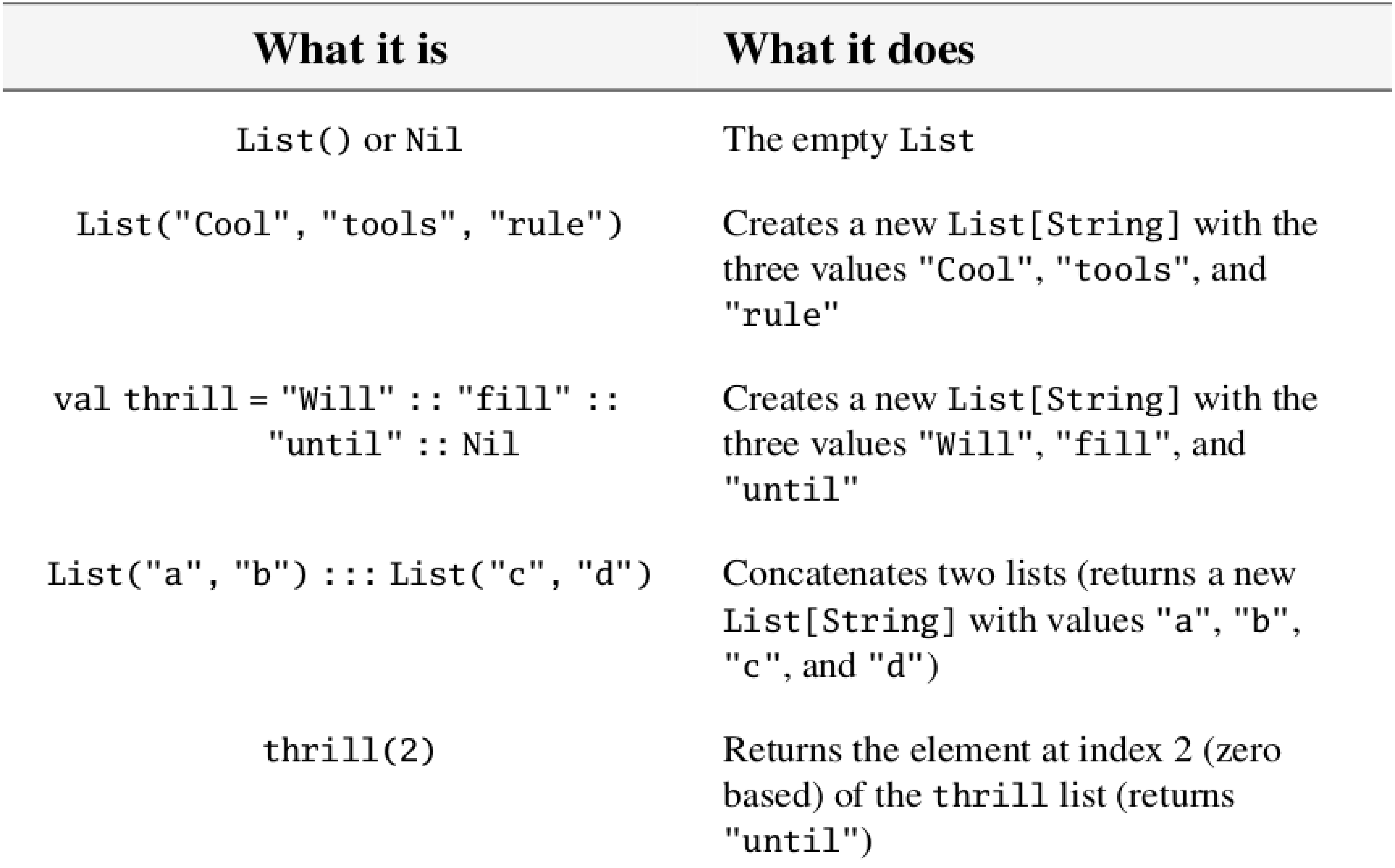
```
 val bankList = List("Bank of America",
"Citibank", "HDFC")


 bankList flatMap (_.toLowerCase)
```

filter

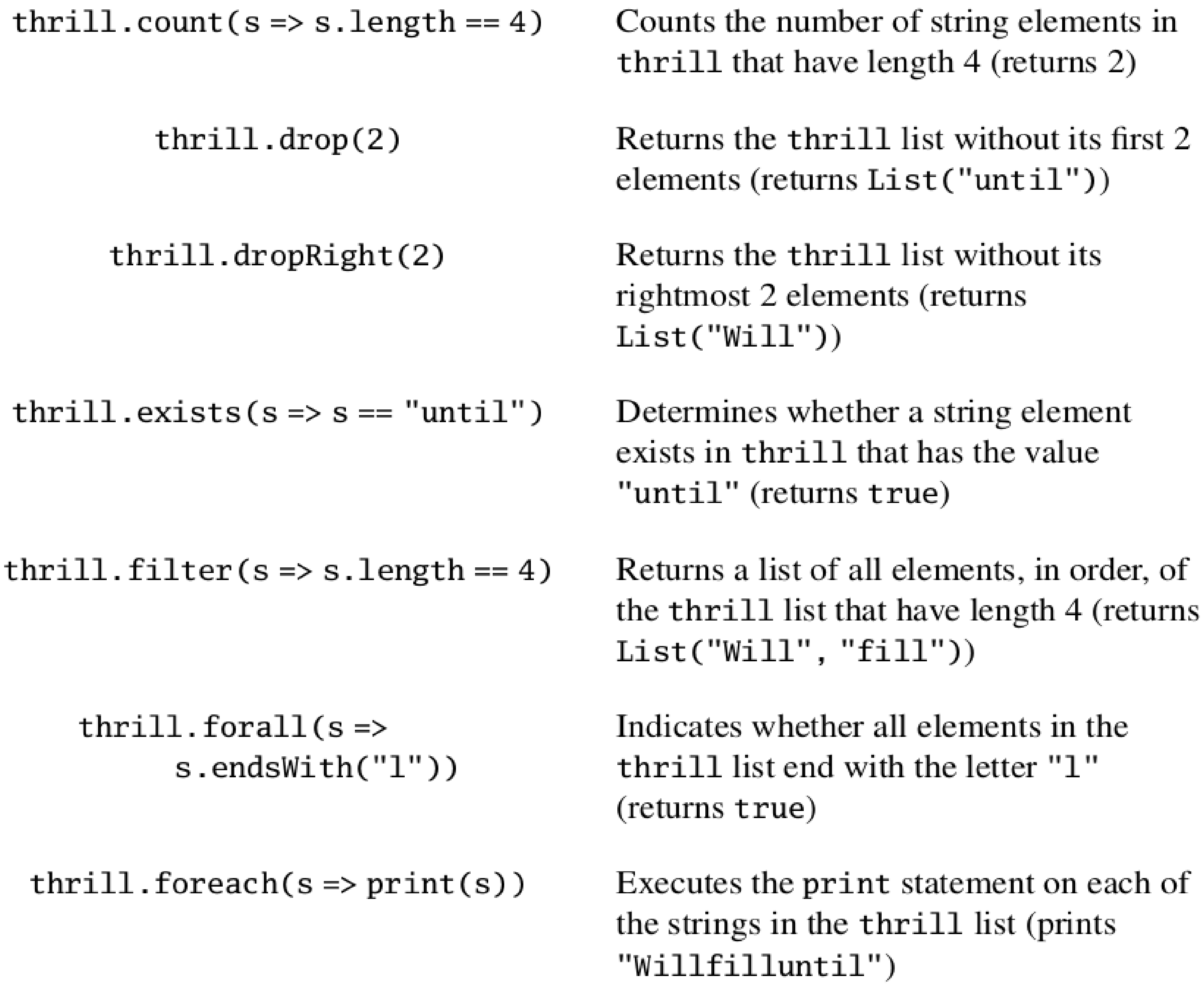
Filter copies selected elements to the resulting collection.
If the predicate is true, the element is copied

```
 val bankList = List("Bank of America", "Citibank", "HDFC")

 bankList filter (_.startsWith("B"))
> res8: List[java.lang.String] = List(Bank of America)
```

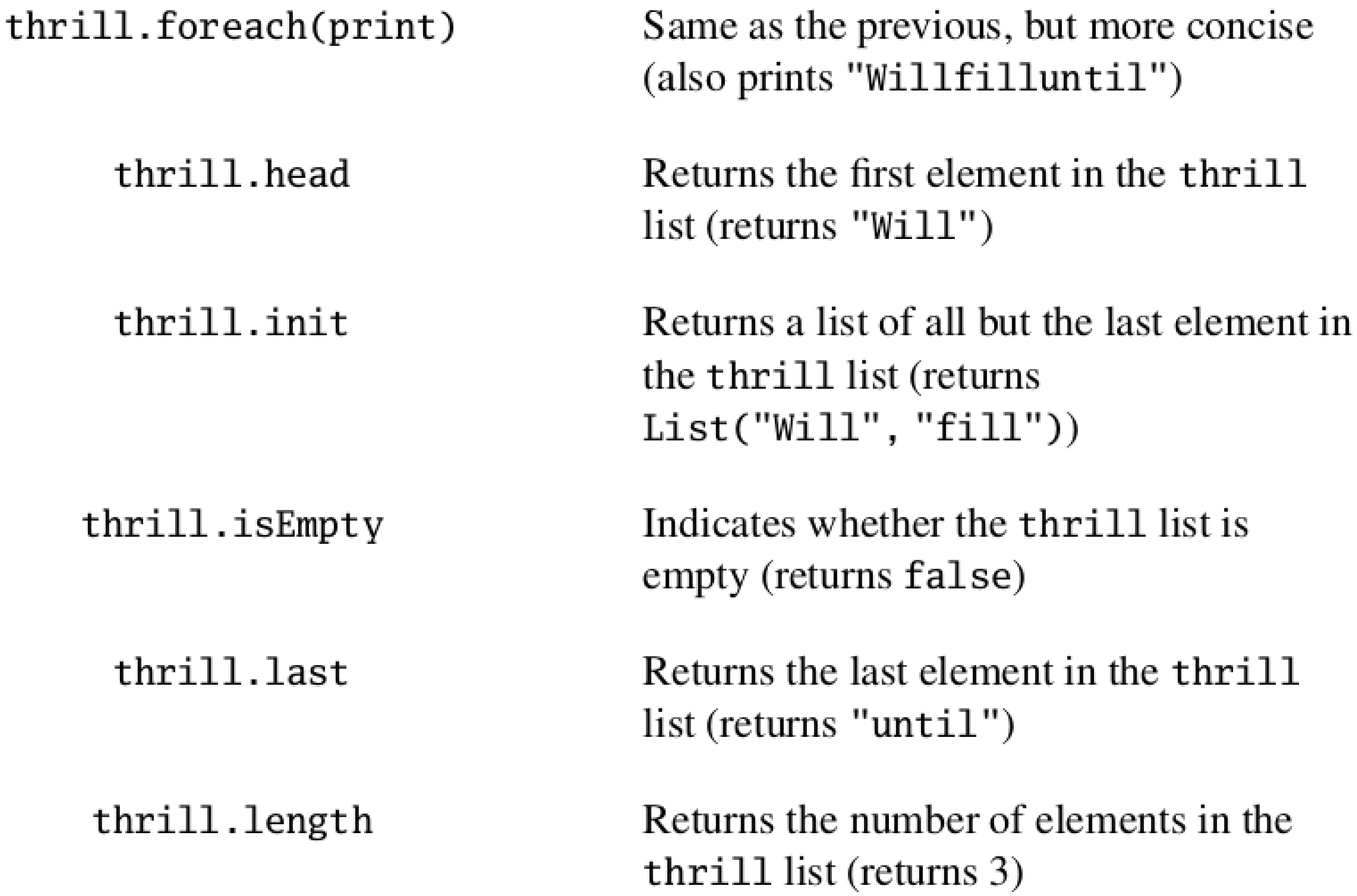
Other methods to try on List



What it is	What it does
`List()` or `Nil`	The empty `List`
`List("Cool", "tools", "rule")`	Creates a new `List[String]` with the three values `"Cool"`, `"tools"`, and `"rule"`
`val thrill = "Will" :: "fill" :: "until" :: Nil`	Creates a new `List[String]` with the three values `"Will"`, `"fill"`, and `"until"`
`List("a", "b") ::: List("c", "d")`	Concatenates two lists (returns a new `List[String]` with values `"a"`, `"b"`, `"c"`, and `"d"`)
`thrill(2)`	Returns the element at index 2 (zero based) of the `thrill` list (returns `"until"`)



`thrill.count(s => s.length == 4)`	Counts the number of string elements in `thrill` that have length 4 (returns 2)
`thrill.drop(2)`	Returns the `thrill` list without its first 2 elements (returns `List("until")`)
`thrill.dropRight(2)`	Returns the `thrill` list without its rightmost 2 elements (returns `List("Will")`)
`thrill.exists(s => s == "until")`	Determines whether a string element exists in `thrill` that has the value `"until"` (returns `true`)
`thrill.filter(s => s.length == 4)`	Returns a list of all elements, in order, of the `thrill` list that have length 4 (returns `List("Will", "fill")`)
`thrill.forall(s => s.endsWith("l"))`	Indicates whether all elements in the `thrill` list end with the letter `"l"` (returns `true`)
`thrill.foreach(s => print(s))`	Executes the `print` statement on each of the strings in the `thrill` list (prints `"Willfilluntil"`)



`thrill.foreach(print)`	Same as the previous, but more concise (also prints `"Willfilluntil"`)
`thrill.head`	Returns the first element in the `thrill` list (returns `"Will"`)
`thrill.init`	Returns a list of all but the last element in the `thrill` list (returns `List("Will", "fill")`)
`thrill.isEmpty`	Indicates whether the `thrill` list is empty (returns `false`)
`thrill.last`	Returns the last element in the `thrill` list (returns `"until"`)
`thrill.length`	Returns the number of elements in the `thrill` list (returns 3)

`thrill.map(s => s + "y")`	Returns a list resulting from adding a `"y"` to each string element in the `thrill` list (returns `List("Willy", "filly", "untily")`)
`thrill.mkString(", ")`	Makes a string with the elements of the list (returns `"Will, fill, until"`)
`thrill.remove(s => s.length == 4)`	Returns a list of all elements, in order, of the `thrill` list *except those* that have length 4 (returns `List("until")`)
`thrill.reverse`	Returns a list containing all elements of the `thrill` list in reverse order (returns `List("until", "fill", "Will")`)

`thrill.sort((s, t) =>`
`s.charAt(0).toLower <`
`t.charAt(0).toLower)`
	Returns a list containing all elements of the `thrill` list in alphabetical order of the first character lowercased (returns `List("fill", "until", "Will")`)

`thrill.tail`	Returns the `thrill` list minus its first element (returns `List("fill", "until")`)

Parallel Collections

```
(1 to 50) foreach println

(1 to 50).par foreach println
```

Agenda

- Scala Who & Why?
- First Steps
- Classes and Objects
- Control Structures and Functions
- Inheritance and Traits
- Testing
- Pattern Matching
- Collections
- XML Support

Extensive XML support

```
val v = <train station="Mumbai" time="21:00"/>

val a = <a>some value<b></a>
```

Compiler would point out malformed XML

Insert Scala code

```
case class Employee(id:Int, name:String)
  val emp1 = Employee(1,"Vikas")
> emp1  : lab3.Employee = Employee(1,Vikas)

  val e = <Employee name={emp1.name}></Employee>
> e  : scala.xml.Elem = <Employee name="Vikas"></Employee>
```

```
val xml = <a>some a text<b>some b text<c>some c text</c></b></a>
 > xml  : scala.xml.Elem = <a>some a text<b>some b text<c>some c text</c></b></a>

  xml \ "b"
> res0: scala.xml.NodeSeq = NodeSeq(<b>some b text<c>some c text</c></b>)

  xml \ "c"
> res1: scala.xml.NodeSeq = NodeSeq()
```

```
xml \\ "c"
> res2: scala.xml.NodeSeq = NodeSeq(<c>some c text</c>)
```

Querying element for a child

Query for descendents without knowing where they are in the path

Attributes

```
val xml =
<a value="10">some a text
  <b value="10">some b text
    <c>some c text
    </c>
  </b>
</a>
```

```
xml \ "@value"
```

```
xml \\ "@value"
```

```
(xml \\ "c").text
```

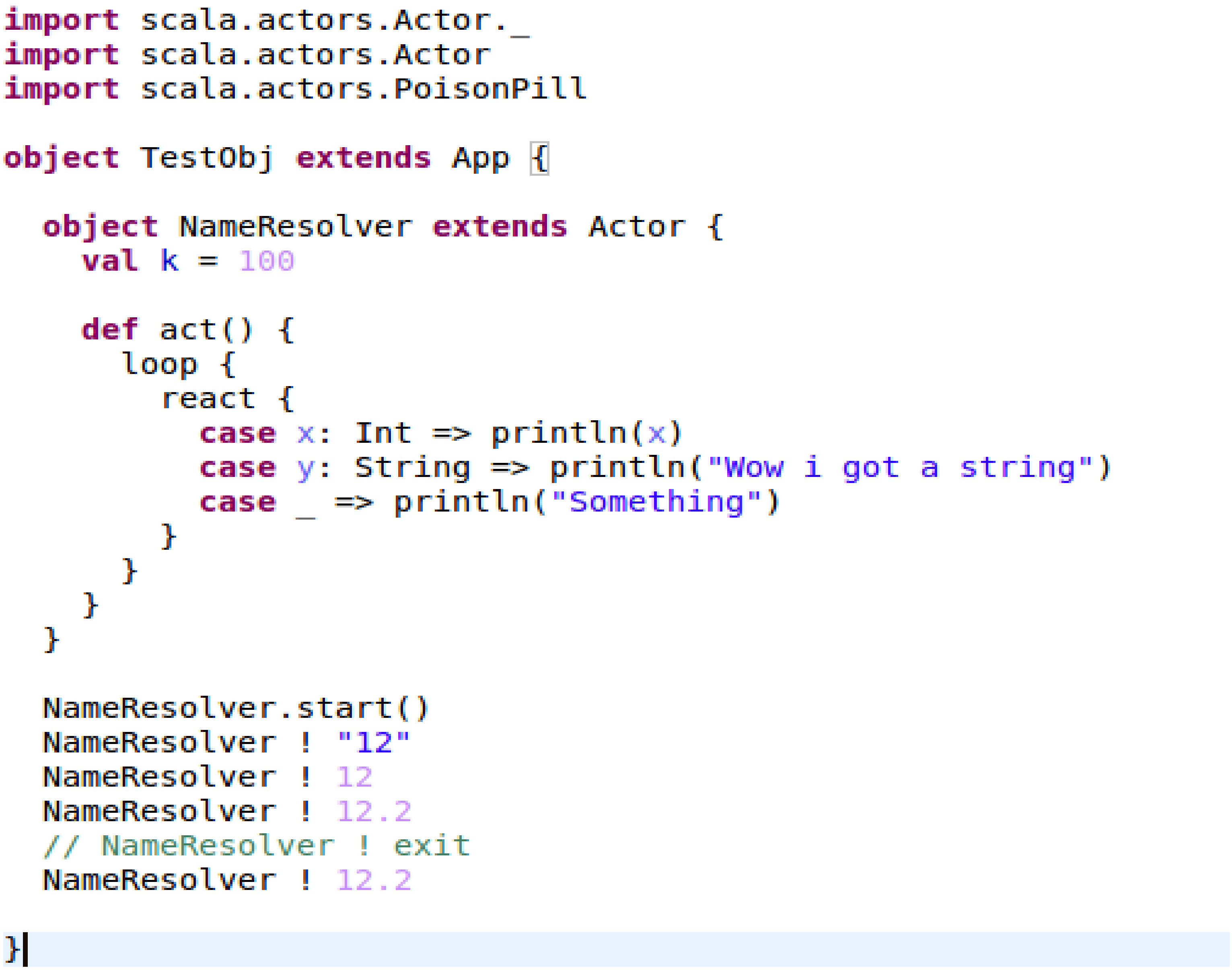
Serializing

```
case class Employee(id:Int, name:String){
  def toXML =
  <employee>
    <id>{id}</id>
    <name>{name}</name>
  </employee>
  }
  val emp1 = Employee(1,"Vikas")
  emp1.toXML
 > res0: scala.xml.Elem = <employee>
                      |      <id>1</id>
                      |      <name>Vikas</name>
                      |    </employee>
```

Deserialization

```
def fromXML (node:scala.xml.Node):Employee ={
 new Employee((node \ "id").text.toInt, (node \ "name").text)
 }
> fromXML: (node: scala.xml.Node)lab3.Employee
 val employeeXML = <employee><id>1</id><name>Vikas</name></employee>

 fromXML(employeeXML)
> res1: lab3.Employee = Employee(1,Vikas)
```

Actors



```
import scala.actors.Actor._
import scala.actors.Actor
import scala.actors.PoisonPill

object TestObj extends App {

  object NameResolver extends Actor {
    val k = 100

    def act() {
      loop {
        react {
          case x: Int => println(x)
          case y: String => println("Wow i got a string")
          case _ => println("Something")
        }
      }
    }
  }

  NameResolver.start()
  NameResolver ! "12"
  NameResolver ! 12
  NameResolver ! 12.2
  // NameResolver ! exit
  NameResolver ! 12.2

}
```