Apache Spark and Scala

Module 4: Functional Programming in Scala

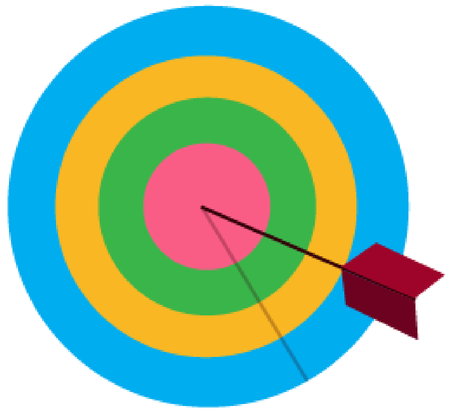
Course Topics

Module 1 Getting Started / Introduction to Scala	**Module 2** Scala – Essentials and Deep Dive	**Module 3** Introducing Traits and OOPS in Scala	**Module 4** Functional Programming in Scala
Module 5 Spark and Big Data	**Module 6** Advanced Spark Concepts	**Module 7** Understanding RDDs	**Module 8** Shark, SparkSQL and Project Discussion

Session Objectives

In this session, you will learn about

- Traits as Mixins
- Functional Programming
- Paradigms of Functional Programming
- Higher Order Functions
- Currying
- Closures
- Anonymous Blocks,
- Implicit Function Parameters
- Call by Name
- Call by Value



Why No Multiple Inheritance?

- Multiple inheritance works when parent classes have nothing in common
- Scala, like Java doesn't allow a class to inherit from multiple classes
- Problem arises if they have common functionality or fields

Example:

```
Class  Person {
     def  id: String ...
            .....
}

Class Employee{
     def id: String ...
     ...
}

//Assume we allow multiple inheritance like:

 class Analyst extends Employee, Person{
     ...
}
```

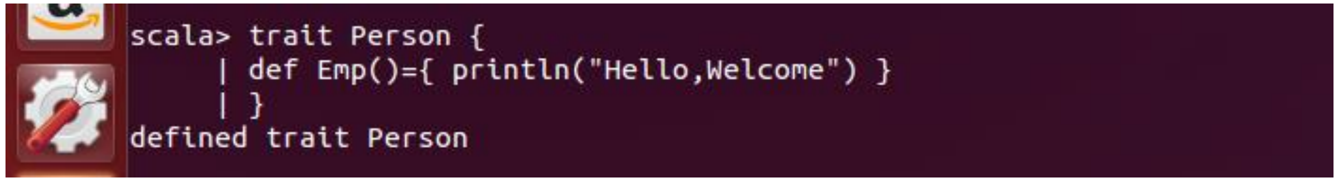
Why No Multiple Inheritance? (Cont'd)

- Now, Sample class has two id methods, which shall be used?
- The same is true for the fields
- This problem is famously called as Diamond Inheritance problem, wherein, the child class would be unable to distinguish between the common members of superclasses
- In Java, this problem is solved by the concept of interfaces, where the interfaces have only the abstract methods
- Scala has the concept of Traits instead of interfaces
- Scala traits, unlike Java interfaces could have the abstract and concrete methods

Traits as Interfaces

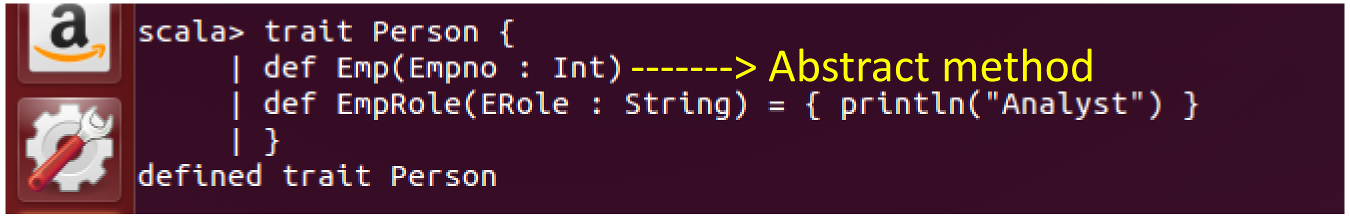
- Similar to interfaces in Java, traits are used to define object types by specifying the signature of the supported methods
- The syntax is the same syntax we use when defining a class. The only difference is using the 'trait' keyword instead of 'class'

Example:



```
scala> trait Person {
     | def Emp()={ println("Hello,Welcome") }
     | }
defined trait Person
```

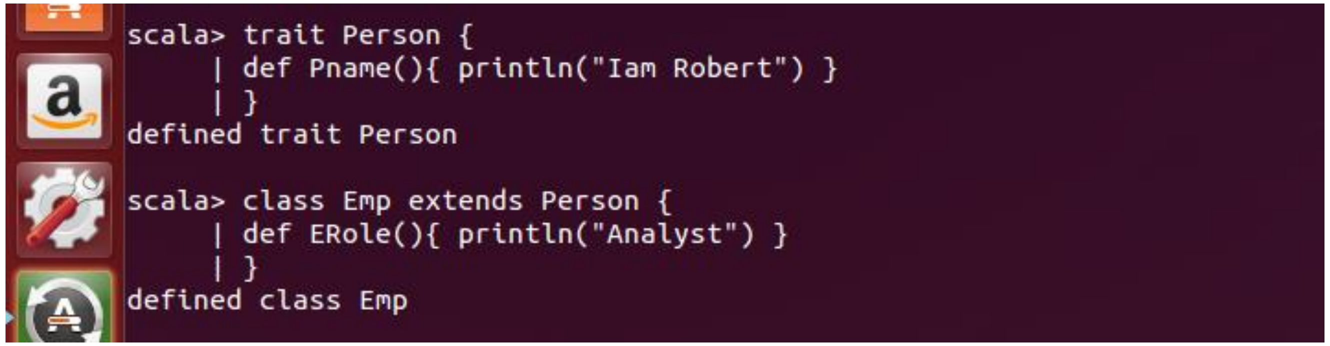
- Unlike Java, Scala allows traits to be partially implemented; i.e. it is possible to define default implementations for some methods
- Methods need not be declared as abstract
- An unimplemented method is automatically assumed as abstract method



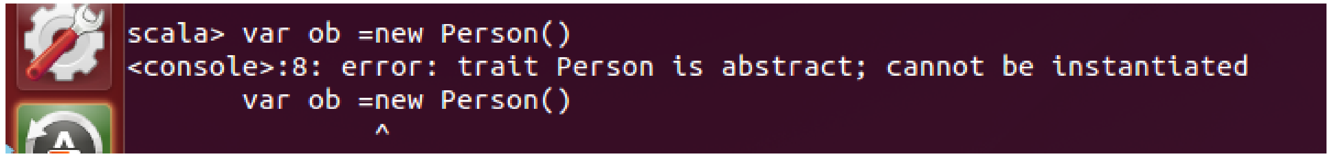
```
scala> trait Person {
     | def Emp(Empno : Int) -------> Abstract method
     | def EmpRole(ERole : String) = { println("Analyst") }
     | }
defined trait Person
```

Traits as Interfaces (Cont'd)

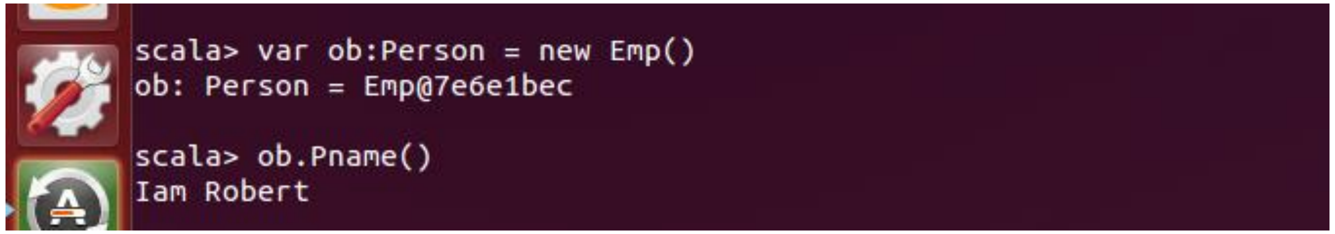
- Once a trait is defined, it can be mixed in to a class using either the keyword extends or the keyword with
- We can use methods inherited from a trait just as any method inherited from a super class
- Once a trait is defined we get a new type, similarly to defining a new class



```
scala> trait Person {
     | def Pname(){ println("Iam Robert") }
     | }
defined trait Person

scala> class Emp extends Person {
     | def ERole(){ println("Analyst") }
     | }
defined class Emp
```



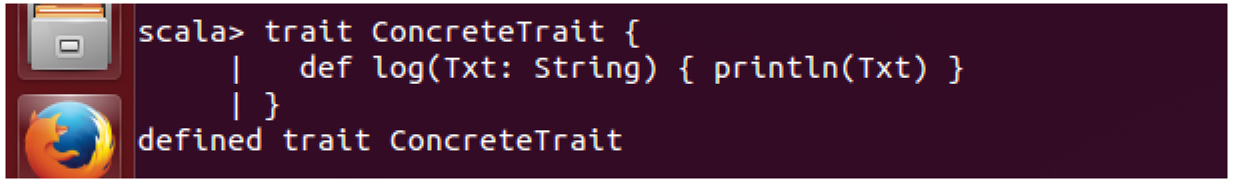
```
scala> var ob =new Person()
<console>:8: error: trait Person is abstract; cannot be instantiated
       var ob =new Person()
               ^
```



```
scala> var ob:Person = new Emp()
ob: Person = Emp@7e6e1bec

scala> ob.Pname()
Iam Robert
```

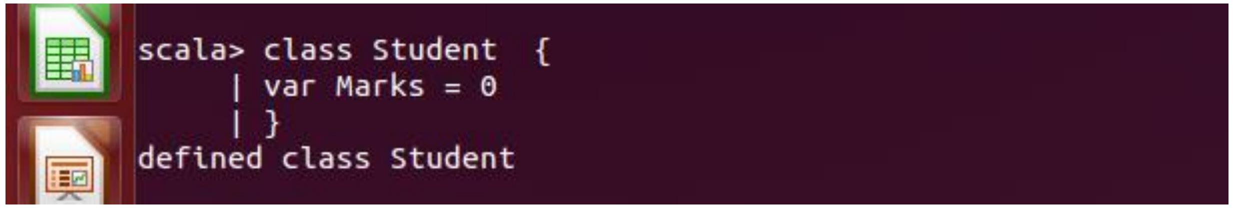
Traits with Concrete Implementation

Trait methods can be concrete also



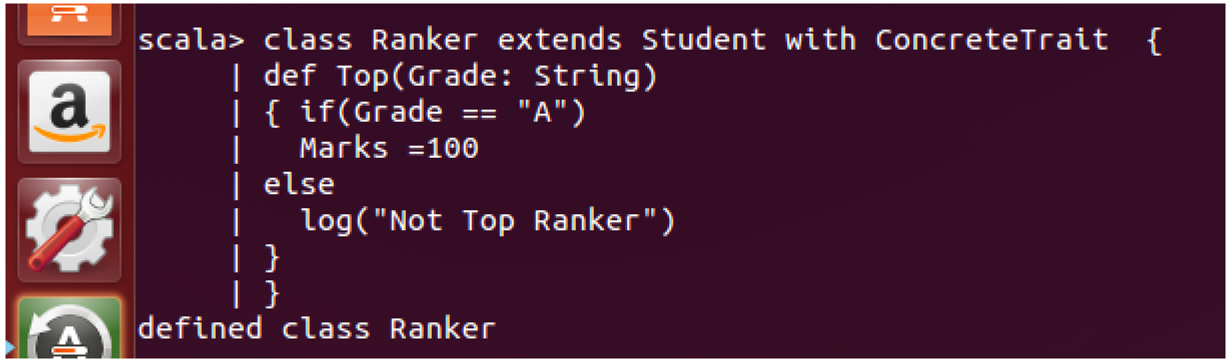
```
scala> trait ConcreteTrait {
     |   def log(Txt: String) { println(Txt) }
     | }
defined trait ConcreteTrait
```

The concrete trait method provides log method with its implementation



```
scala> class Student  {
     | var Marks = 0
     | }
defined class Student
```

Below is the example of its usage:



```
scala> class Ranker extends Student with ConcreteTrait  {
     | def Top(Grade: String)
     | { if(Grade == "A")
     |   Marks =100
     | else
     |   log("Not Top Ranker")
     | }
     | }
defined class Ranker
```

Traits with Concrete Implementation (Cont'd)

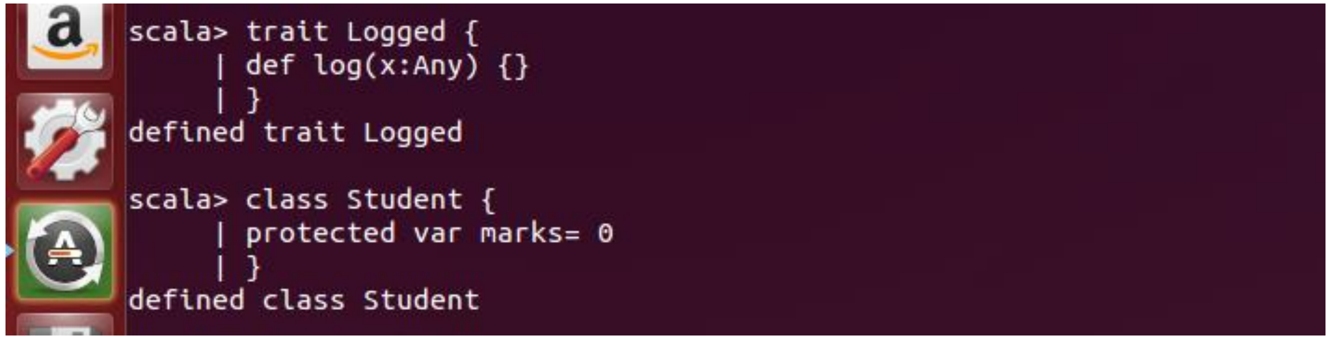
- The subclass picks up the concrete implementation from the trait
- Thus the ConcreteTrait functionality is mixed in with Ranker class
- But whenever the trait implementation changes, all the mixed in classes needs to be re-compiled

Objects with Traits

Traits could be added with individual objects while constructing them

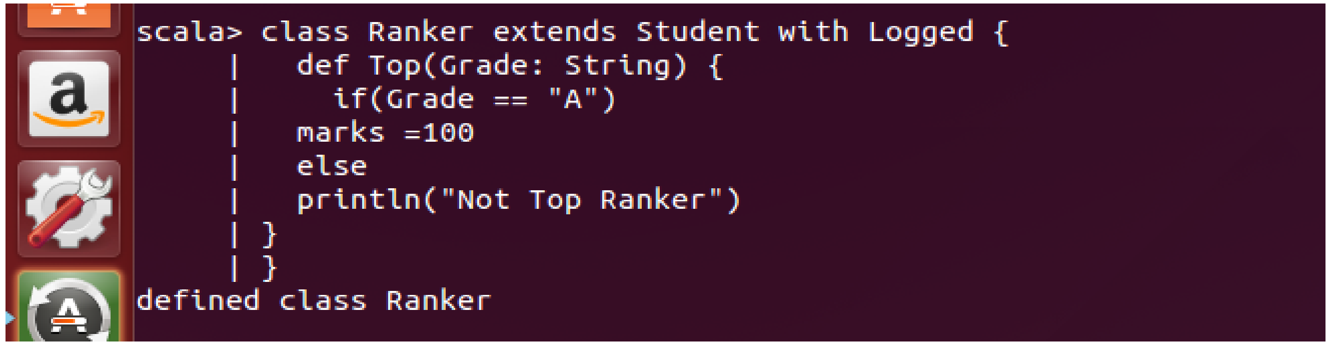
Example:

We’ll use Logged trait of standard Scala library:



```
scala> trait Logged {
     | def log(x:Any) {}
     | }
defined trait Logged

scala> class Student {
     | protected var marks= 0
     | }
defined class Student
```

What should be the expected behaviour?

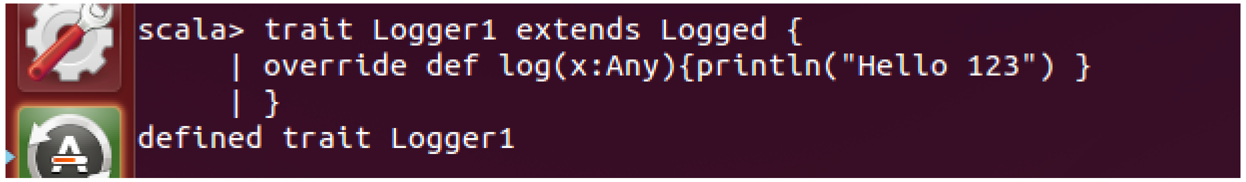


```
scala> class Ranker extends Student with Logged {
     |   def Top(Grade: String) {
     |     if(Grade == "A")
     |   marks =100
     |   else
     |   println("Not Top Ranker")
     | }
     | }
defined class Ranker
```

Objects with Traits (Cont'd)

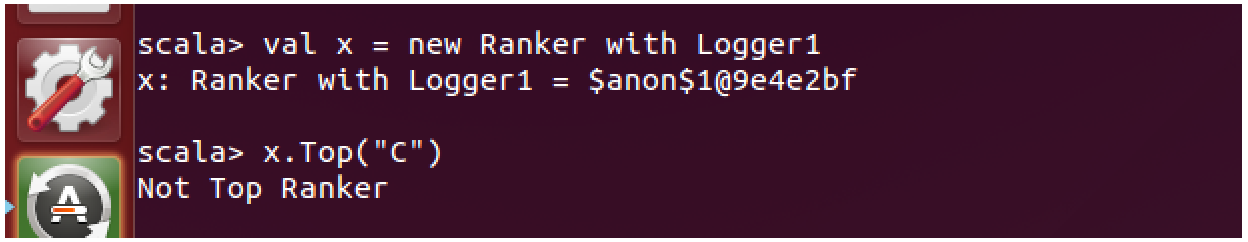
Nothing gets logged! As the extended trait didn't have any implementation

Now, let's extend the trait:



```
scala> trait Logger1 extends Logged {
     | override def log(x:Any){println("Hello 123") }
     | }
defined trait Logger1
```

Now we can add this trait while constructing a new object:



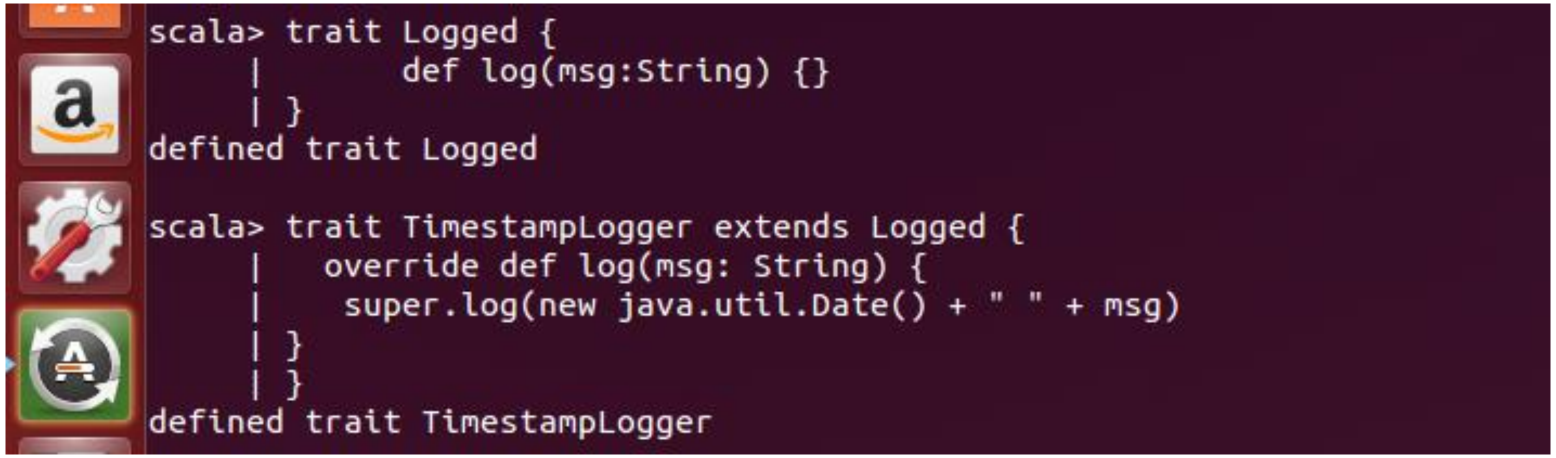
```
scala> val x = new Ranker with Logger1
x: Ranker with Logger1 = $anon$1@9e4e2bf

scala> x.Top("C")
Not Top Ranker
```

- Now when Top method is executed, the else part of the Logger1 is invoked and displays the message
- For the class-private field, private getter and setter are generated
- Thus we get the flexibility of attaching use case specific loggers!

Layered Traits

- Multiple traits can be added to a class or object in Scala
- In such case, the traits are invoked always starting from the last
- Extending multiple traits could be useful for the cases when a value needs to be transformed in stages

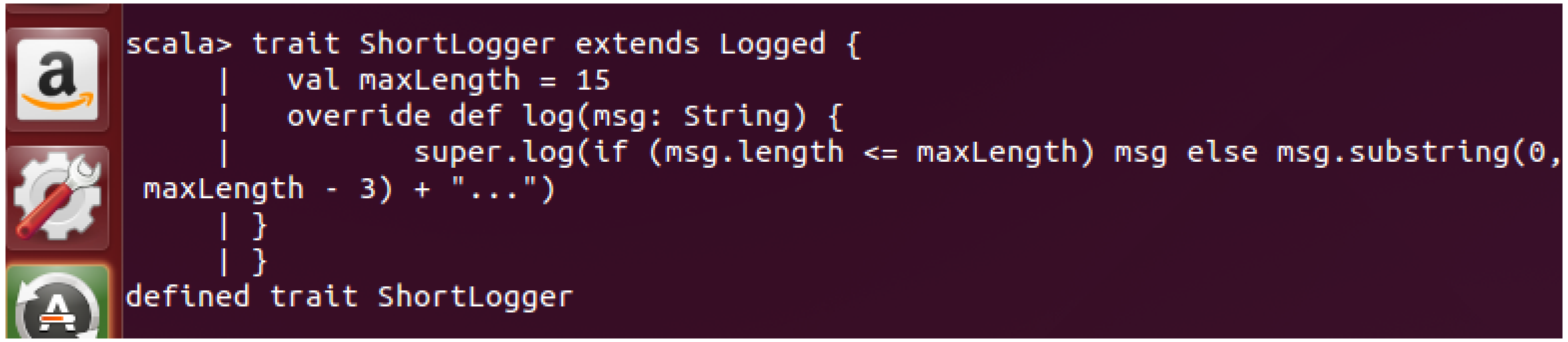
Example:



```
scala> trait Logged {
     |       def log(msg:String) {}
     | }
defined trait Logged

scala> trait TimestampLogger extends Logged {
     |   override def log(msg: String) {
     |    super.log(new java.util.Date() + " " + msg)
     | }
     | }
defined trait TimestampLogger
```

Layered Traits

Now, assume that we want to truncate long message like:



```
scala> trait ShortLogger extends Logged {
     |   val maxLength = 15
     |   override def log(msg: String) {
     |          super.log(if (msg.length <= maxLength) msg else msg.substring(0,
 maxLength - 3) + "...")
     | }
     | }
defined trait ShortLogger
```

- Note that each log method calls a super.log
- In traits, the keyword trait doesn’t have the same meaning as it does with classes
- Here super.log calls the next trait in the trait hierarchy
- Hierarchy depends upon the order in which traits are added
- Traits are processed starting with last one

Traits Construction Order

Traits are constructed in following order:

- The superclass constructor is called first
- Trait constructors are executed after the superclass constructor but before the class constructor
- Traits are constructed from left to right
- Within each trait, the parent gets constructed first
- If multiple traits share the same parent and if the parent has already been constructed, it is not re-constructed
- After all the traits are constructed, the subclass is constructed

Check your Understanding – 1

Traits construction order is decided by their they inheritance

a. True

b. False

Check your Understanding – Solution

Traits construction order is decided by their they inheritance

a. True

[✓ b. False]

False

Functional Programming

- Apart from being a pure object oriented language, Scala is also a functional programming language
- Functional programming is driven by mainly two ideas:
  - First main idea is that functions are first class values. They are treated just like any other type, say String, Integer etc. So functions can be used as arguments, could be defined in other functions
  - The second main idea of functional programming is that the operations of a program should map input values to output values rather than change data in place. This results in the immutable data structures

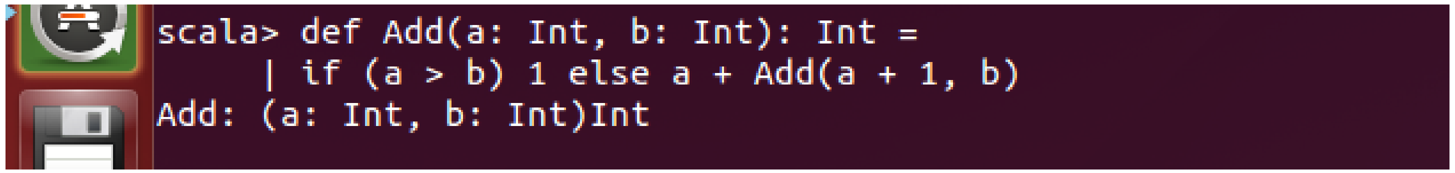
Scala supports both, immutable and mutable data structures

Higher Order Functions

- Functional languages treat functions as first-class values
- This means that, like any other value, a function can be passed as a parameter and returned as a result
- This provides a flexible way to compose programs
- Functions that take other functions as parameters or that return functions as results are called higher order functions

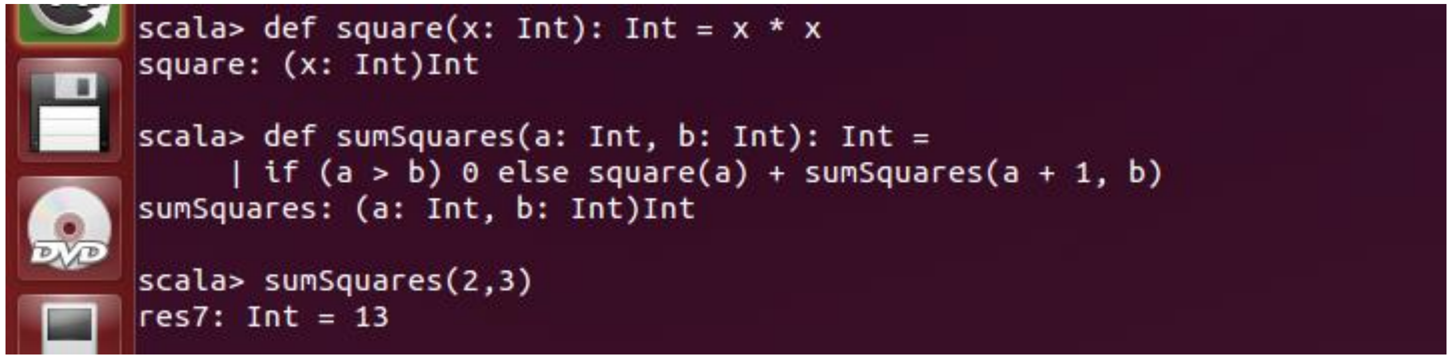
Examples

Take the sum of the integers between a and b



```
scala> def Add(a: Int, b: Int): Int =
     | if (a > b) 1 else a + Add(a + 1, b)
Add: (a: Int, b: Int)Int
```

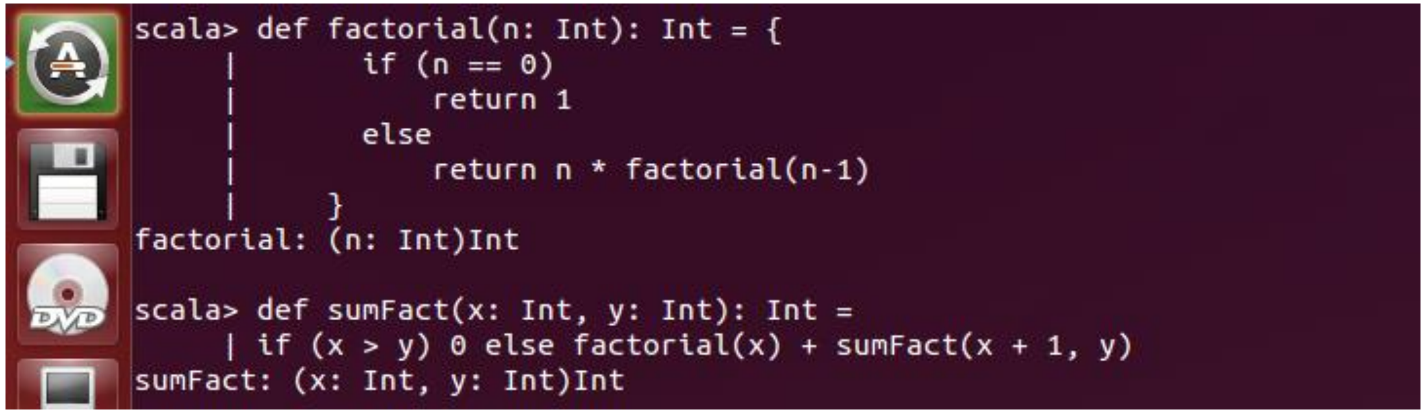
Take the sum of the squares of all the integers between a and b:



```
scala> def square(x: Int): Int = x * x
square: (x: Int)Int

scala> def sumSquares(a: Int, b: Int): Int =
     | if (a > b) 0 else square(a) + sumSquares(a + 1, b)
sumSquares: (a: Int, b: Int)Int

scala> sumSquares(2,3)
res7: Int = 13
```

Examples (Cont'd)

Take the sum of the factorials of all the integers between a and b:



```
scala> def factorial(n: Int): Int = {
     |         if (n == 0)
     |             return 1
     |         else
     |             return n * factorial(n-1)
     |     }
factorial: (n: Int)Int

scala> def sumFact(x: Int, y: Int): Int =
     | if (x > y) 0 else factorial(x) + sumFact(x + 1, y)
sumFact: (x: Int, y: Int)Int
```

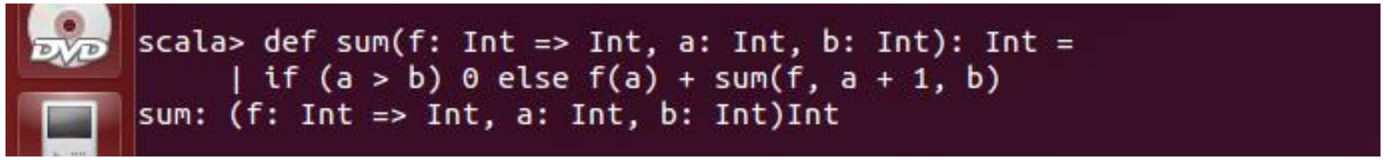
These are special cases of below for different values of f

$$\sum_{n=a}^{b} f(n)$$

Can we factor out the common pattern?

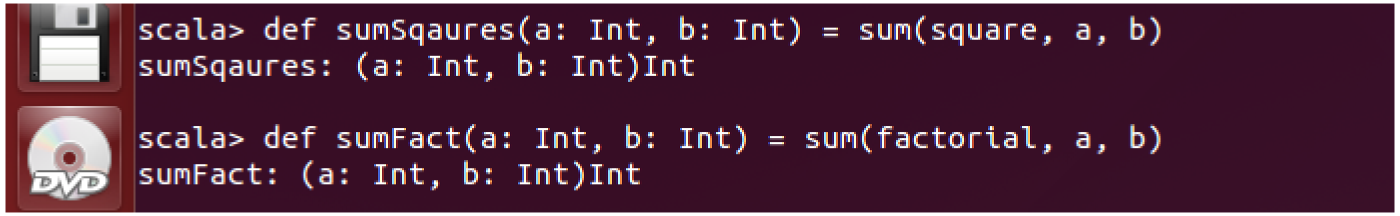
Summing with Higher Order Functions

Let's define:



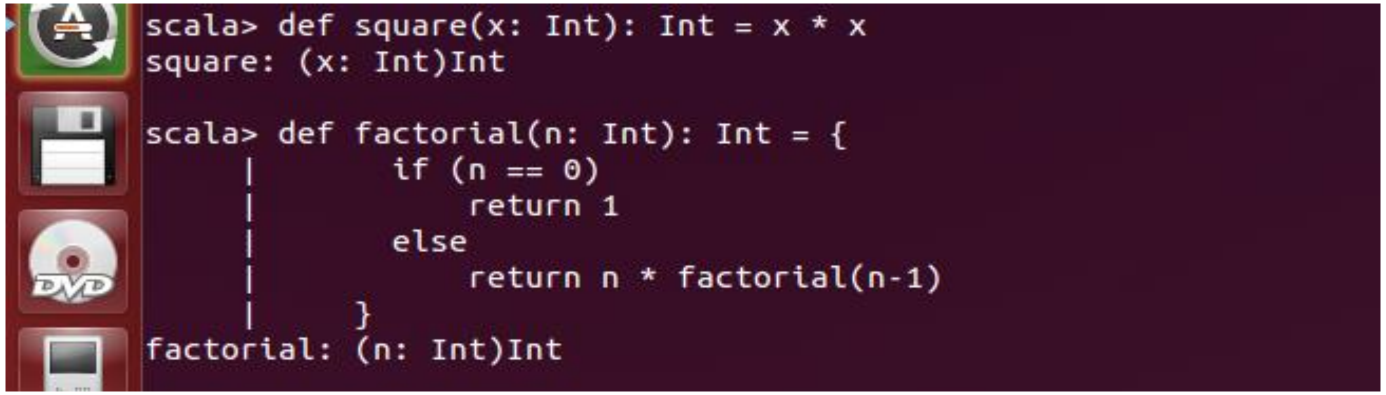
```
scala> def sum(f: Int => Int, a: Int, b: Int): Int =
     | if (a > b) 0 else f(a) + sum(f, a + 1, b)
sum: (f: Int => Int, a: Int, b: Int)Int
```

We can then write:



```
scala> def sumSqaures(a: Int, b: Int) = sum(square, a, b)
sumSqaures: (a: Int, b: Int)Int

scala> def sumFact(a: Int, b: Int) = sum(factorial, a, b)
sumFact: (a: Int, b: Int)Int
```

Where



```
scala> def square(x: Int): Int = x * x
square: (x: Int)Int

scala> def factorial(n: Int): Int = {
     |       if (n == 0)
     |           return 1
     |       else
     |           return n * factorial(n-1)
     |    }
factorial: (n: Int)Int
```

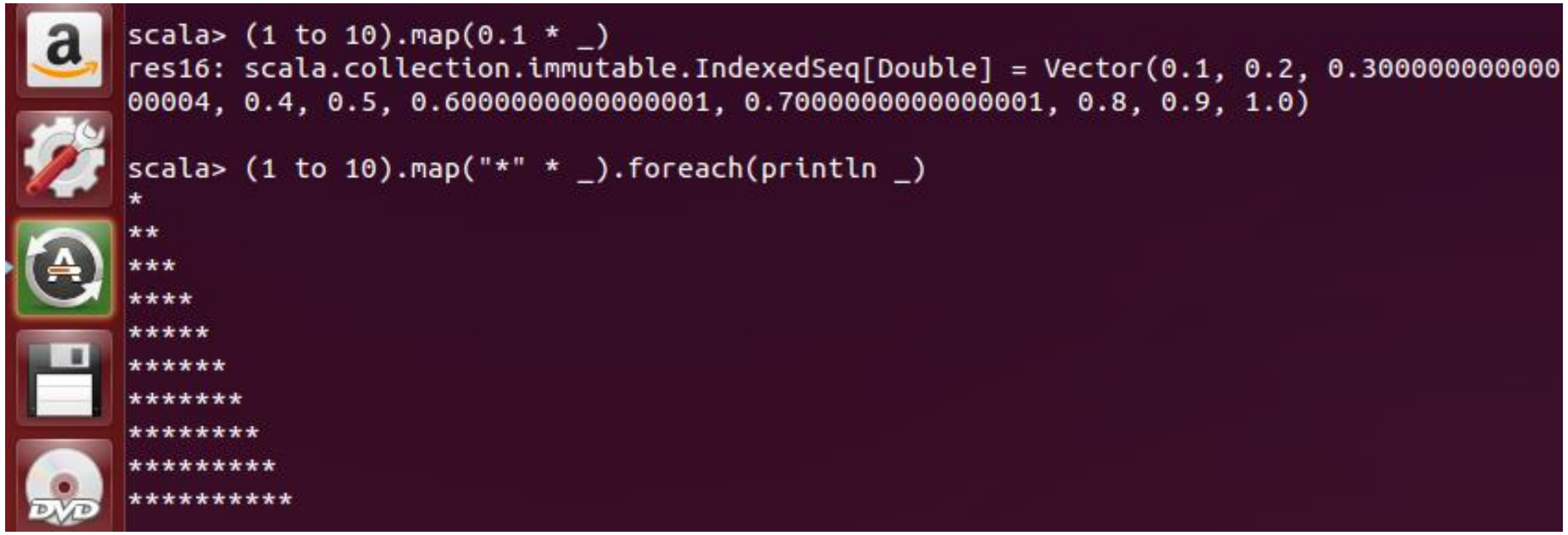
Useful Higher Order Functions

Map

```
(1 to 10).map(0.1 * _)
```

foreach

```
(1 to 10).map("*" * _).foreach(println _)
```



```
scala> (1 to 10).map(0.1 * _)
res16: scala.collection.immutable.IndexedSeq[Double] = Vector(0.1, 0.2, 0.300000000000
00004, 0.4, 0.5, 0.6000000000000001, 0.7000000000000001, 0.8, 0.9, 1.0)

scala> (1 to 10).map("*" * _).foreach(println _)
*
**
***
****
*****
******
*******
********
*********
**********
```

Useful Higher Order Functions (Cont'd)

filter

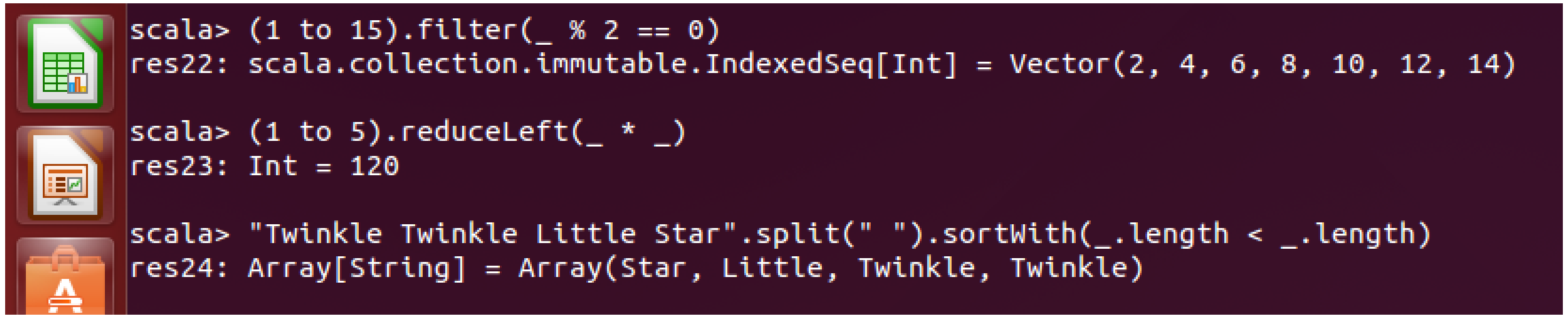
```
(1 to 15).filter(_ % 2 == 0)
```

reduceLeft

```
(1 to 5).reduceLeft(_ * _)
```

Split, sortWith

```
“Twinkle Twinkle Little Star".split(" ").sortWith(_.length < _.length)
```



```
scala> (1 to 15).filter(_ % 2 == 0)
res22: scala.collection.immutable.IndexedSeq[Int] = Vector(2, 4, 6, 8, 10, 12, 14)

scala> (1 to 5).reduceLeft(_ * _)
res23: Int = 120

scala> "Twinkle Twinkle Little Star".split(" ").sortWith(_.length < _.length)
res24: Array[String] = Array(Star, Little, Twinkle, Twinkle)
```

Anonymous Functions

- Passing functions as parameters leads to the creation of many small functions
- Sometimes it is tedious to have to define (and name) these functions using def
- Compare to strings: We do not need to define a string using def

Instead of

```
def str = ”skillspeed”; println(str)
```

We can directly write

```
println(”skillspeed”)
```

- Because strings exist as literals. Analogously we would like function literals, which let us write a function without giving it a name
- These are called anonymous functions

Anonymous Function Syntax

- Example: A function that raises its argument to a cube:

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

- Here, (x: Int) is the parameter of the function, and x * x * x is it's body
- The type of the parameter can be omitted if it can be inferred by the compiler from the context
- If there are several parameters, they are separated by commas:

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

Simplified Anonymous Functions

Anonymous Function can be made even more simplified in the following manner

```
//Explicit type declaration
val call1 = doWithOneAndTwo((x: Int, y: Int) => x + y)

//The compiler expects 2 ints so x and y types are inferred
val call2 = doWithOneAndTwo((x, y) => x + y)

//Even more concise syntax
val call3 = doWithOneAndTwo(_ + _)
```

This simplification of syntax is also more commonly and formally known as Syntactic Sugar

Check your Understanding – 2

Functions can't be passed as arguments in Scala

a. True

b. False

Check your Understanding – Solution

Functions can't be passed as arguments in Scala

a. True

[✓ b. False]

False

Check your Understanding – 3

Anonymous functions can access the out of bound variables

a. True

b. False

Check your Understanding – Solution

Anonymous functions can access the out of bound variables

[✓ a. True]

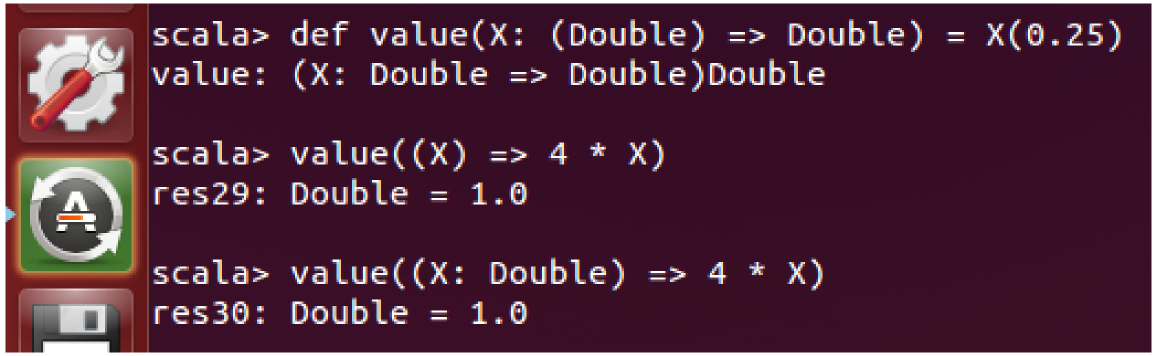
b. False

True

Parameter Inference

- If one function is passed as parameter to another function, Scala helps us in deducing types wherever possible:

Example:



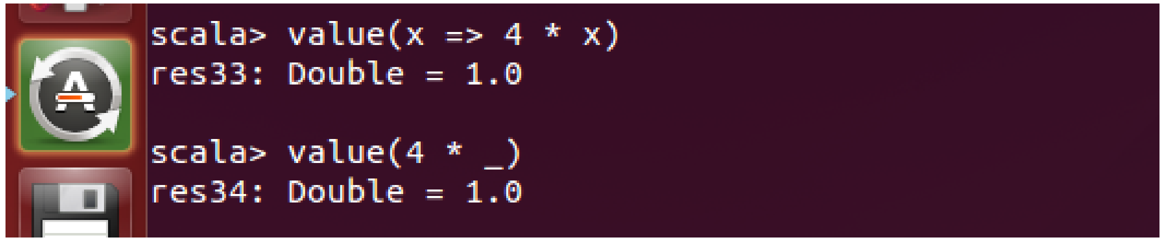
```
scala> def value(X: (Double) => Double) = X(0.25)
value: (X: Double => Double)Double

scala> value((X) => 4 * X)
res29: Double = 1.0

scala> value((X: Double) => 4 * X)
res30: Double = 1.0
```

- If a function has only one parameter, then the "()" can be omitted, hence now it can be expressed as:

```
value(x => 4 * x)
```

- If the parameter is used only ONCE on right side of =>, then it can be replaced with underscore "_" . So now

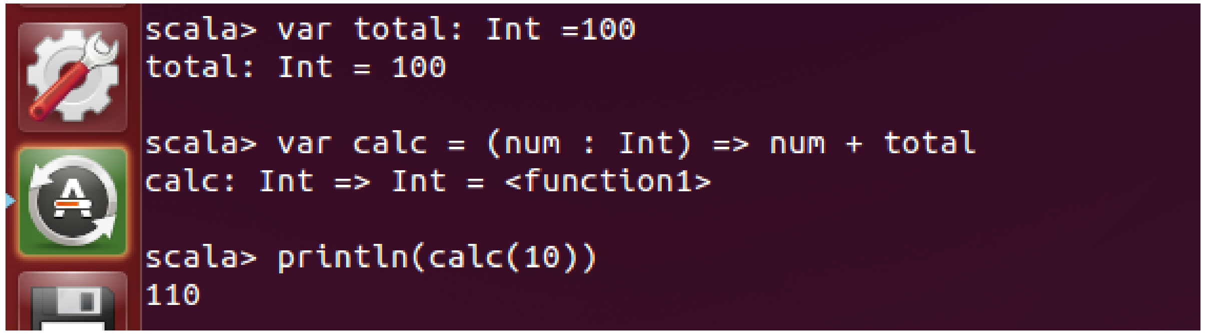
```
value(4 * _)
```



```
scala> value(x => 4 * x)
res33: Double = 1.0

scala> value(4 * _)
res34: Double = 1.0
```

Closures

- In Scala, the functions can be defined anywhere, in a package or class or inside another function or method
- We can access the variables of enclosing scope from the function
- When defining a function, at runtime we get an object. Each function call is actually an object. When we define a function that uses variables from its outer scope the object that we get is a closure
- Example:



```
scala> var total: Int =100
total: Int = 100

scala> var calc = (num : Int) => num + total
calc: Int => Int = <function1>

scala> println(calc(10))
110
```

- This is called Closure. So, a Closure comprises of code along with the definition of non-local variables used by the code

Currying

- Currying is the technique of transforming a function that takes multiple arguments into a function that takes a single argument

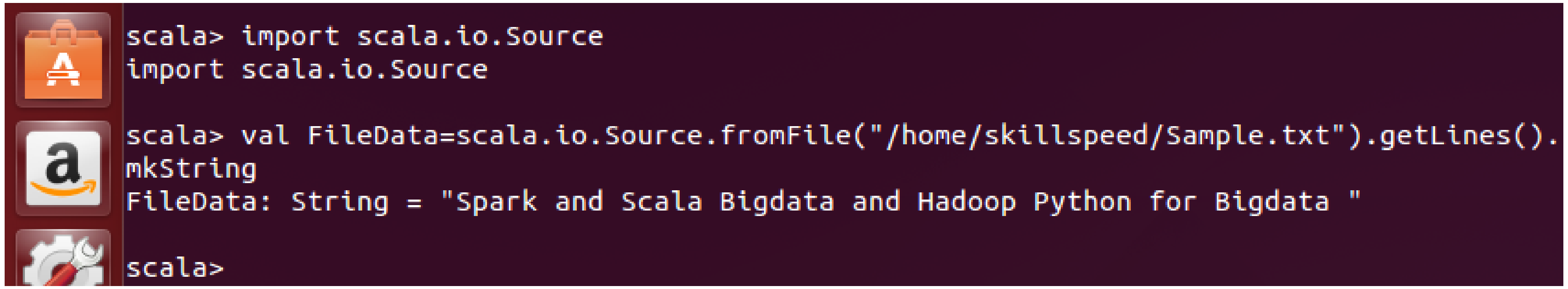
Example:

```
//before currying

def add(a:Int, b:Int) = a + b

add(1, 2)   // 3
add(7, 3)   // 10
```

```
//After Currying

def add(a:Int) = (b:Int) => a + b

add(1)(2)   // 3
add(7)(3)   // 10
```

File Processing: A Quick Look

- To read all lines from a file, use getLine method
- The result is an iterator, which then can be used to process line one at a time
- You can use fromFile also to read all lines in a file. mkString converts a collection into a flat String by each element's to String method



```
scala> import scala.io.Source
import scala.io.Source

scala> val FileData=scala.io.Source.fromFile("/home/skillspeed/Sample.txt").getLines().
mkString
FileData: String = "Spark and Scala Bigdata and Hadoop Python for Bigdata "

scala>
```

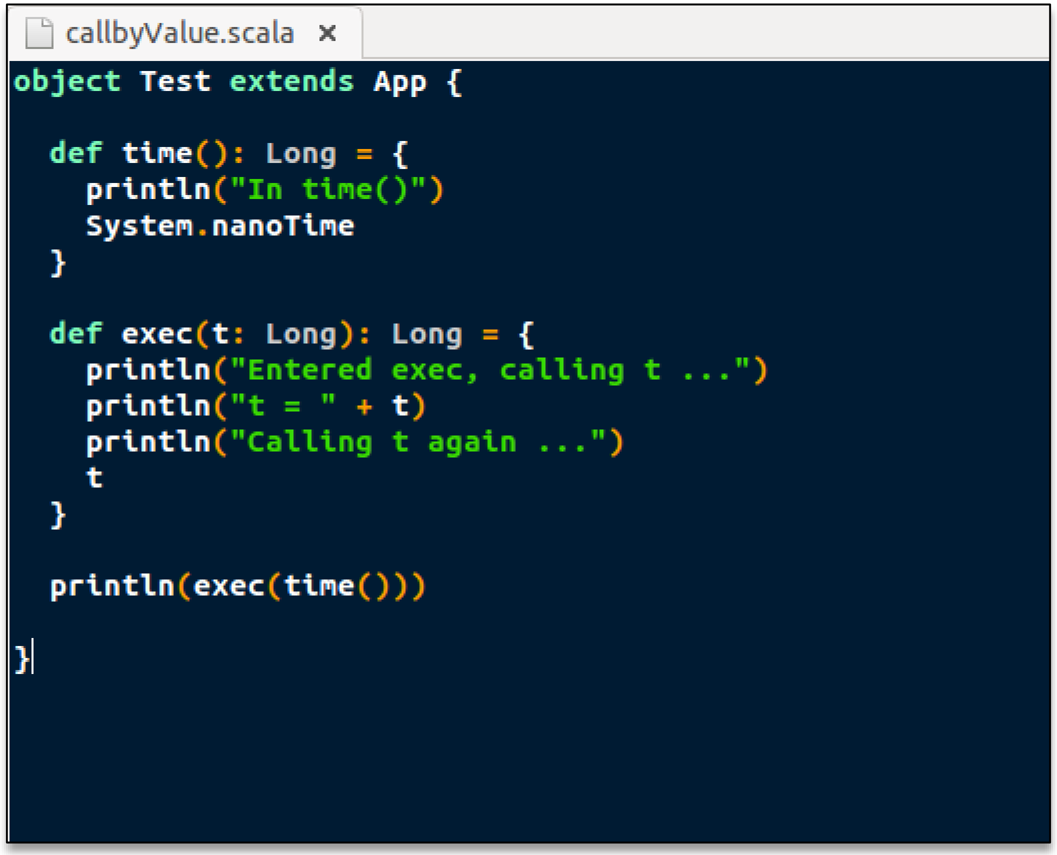
Implicit Function Parameter

- A method with implicit parameters can be applied to arguments just like a normal method
- In this case the implicit label has no effect
- However, if such a method misses arguments for its implicit parameters, such arguments will be automatically provided

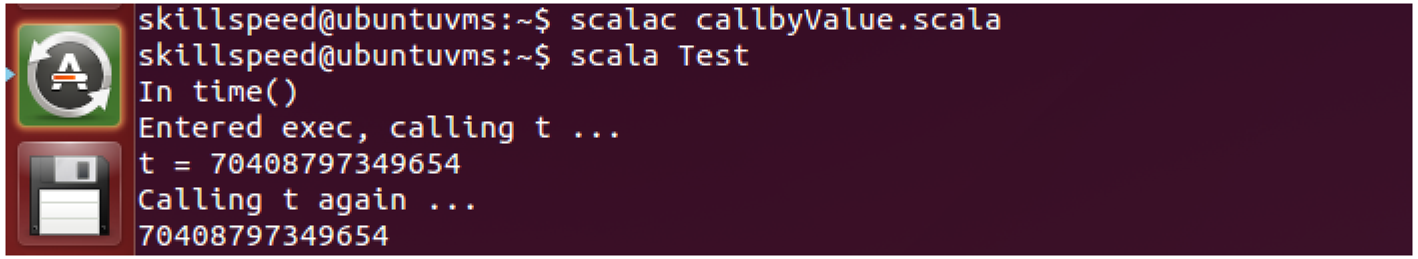
Call by Value:

- Typically, parameters to functions are by-value parameters; that is, the value of the parameter is determined before it is passed to the function
- In most circumstances, this is the behaviour we want and expect

Example



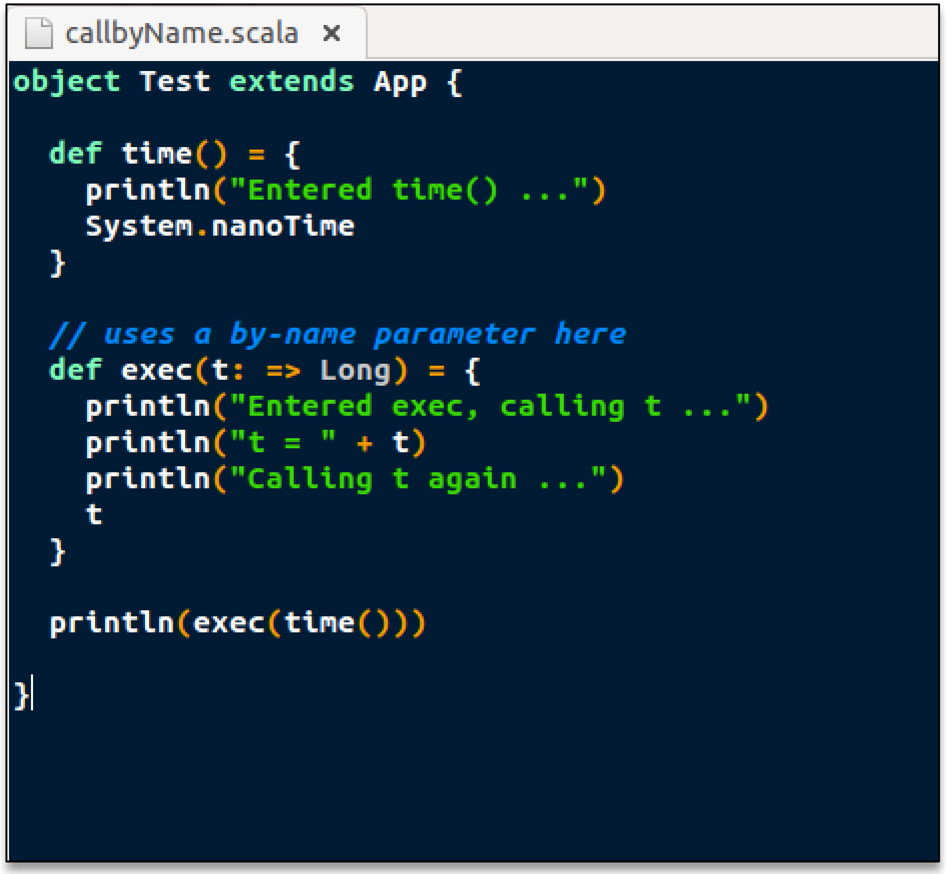
```scala
object Test extends App {

  def time(): Long = {
    println("In time()")
    System.nanoTime
  }

  def exec(t: Long): Long = {
    println("Entered exec, calling t ...")
    println("t = " + t)
    println("Calling t again ...")
    t
  }

  println(exec(time()))

}
```



```
skillspeed@ubuntuvms:~$ scalac callbyValue.scala
skillspeed@ubuntuvms:~$ scala Test
In time()
Entered exec, calling t ...
t = 70408797349654
Calling t again ...
70408797349654
```

Call by Name

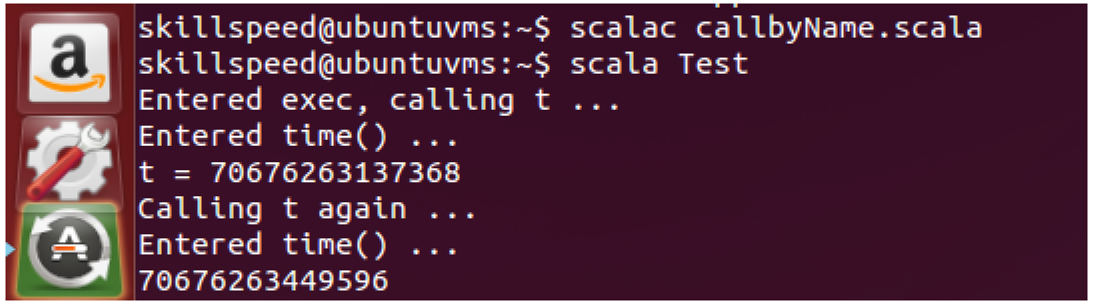
If we need to write a function that accepts as a parameter an expression that we don't want evaluated until it's called within our function. Scala offers call-by-name parameters in such cases



callbyName.scala

```scala
object Test extends App {

  def time() = {
    println("Entered time() ...")
    System.nanoTime
  }

  // uses a by-name parameter here
  def exec(t: => Long) = {
    println("Entered exec, calling t ...")
    println("t = " + t)
    println("Calling t again ...")
    t
  }

  println(exec(time()))

}
```

The output is different:



```
skillspeed@ubuntuvms:~$ scalac callbyName.scala
skillspeed@ubuntuvms:~$ scala Test
Entered exec, calling t ...
Entered time() ...
t = 70676263137368
Calling t again ...
Entered time() ...
70676263449596
```

Check your Understanding – 4

Abstract methods of a trait must be overridden in subclass

a. True

b. False

Check your Understanding – Solution

Abstract methods of a trait must be overridden in subclass

a. True

[✓ b. False]

False

Check your Understanding – 5

Closures:

a. Can't access out of scope variable

b. Can access out of scope variable

c. Essentially are anonymous functions

Check your Understanding – Solution

Closures:

a. Can't access out of scope variable

[✓ b.] Can access out of scope variable

[✓ c.] Essentially are anonymous functions

Can access out of scope variable and Essentially are anonymous functions

Questions

thank you!