Working with Scala - 3

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  Emp {
                def  id: String ...
        .....
}

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

Assume we allow multiple inheritance like:

 class Sample extends Employee, Student {
                ...
}
```

Why No Multiple Inheritance?

→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

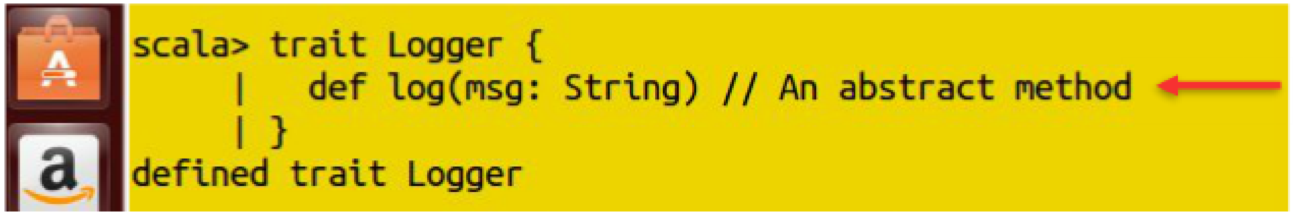
→Scala traits, unlike Java interfaces could have the abstract and concrete methods

Traits As Interfaces

→ Scala Traits work exactly like a Java interface, however they have some differences too

→ Example:

```
trait Logger {
            def log(msg: String) // An abstract method
            }
```



```
scala> trait Logger {
     |    def log(msg: String) // An abstract method
     | }
defined trait Logger
```

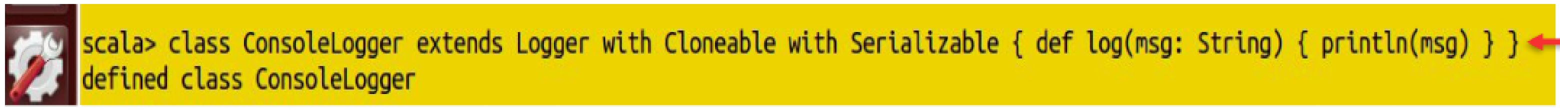
→ Methods need not be declared as abstract

→ An unimplemented method is automatically assumed as abstract method

Traits As Interfaces

→ A subclass can provide the implementation, as:

```
class ConsoleLogger extends Logger // Use extends, not implements
  with Cloneable with Serializable { // Use with to add multiple traits
            def log(msg: String) { println(msg) } // No override needed
}
```



```
scala> class ConsoleLogger extends Logger with Cloneable with Serializable { def log(msg: String) { println(msg) } }
defined class ConsoleLogger
```

→ Please note that the subclass extends the Trait, and not implements it

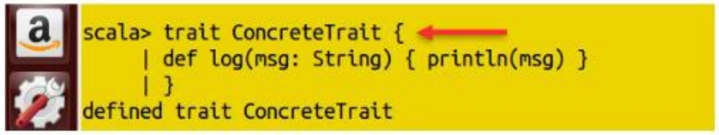
→ No need to use override for abstract methods

→ Multiple traits are extended using with keyword

Traits – Concrete Implementation

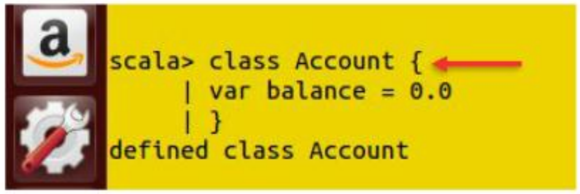
→ Trait methods can be concrete also

```
trait ConcreteTrait {
        def log(msg: String) { println(msg) }
        }
```



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

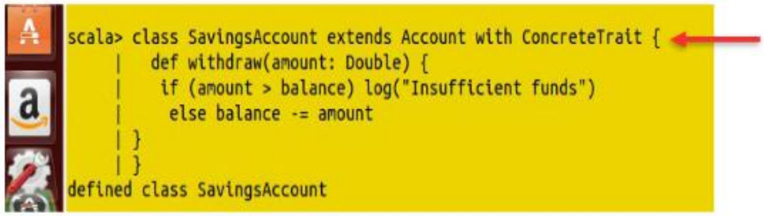
→ The concrete trait method provides log method with its implementation



```
scala> class Account {
     | var balance = 0.0
     | }
defined class Account
```

→ Below is the example of its usage:

```
class SavingsAccount extends Account with ConcreteTrait {
        def withdraw(amount: Double) {
        if (amount > balance) log("Insufficient funds")
        else balance -= amount
        }
}
```



```
scala> class SavingsAccount extends Account with ConcreteTrait {
     |   def withdraw(amount: Double) {
     |    if (amount > balance) log("Insufficient funds")
     |     else balance -= amount
     | }
     | }
defined class SavingsAccount
```

→ The subclass picks up the concrete implementation from the trait

→ Thus the ConcreteTrait functionality is "mixed in" with SavingsAccount class

→ But whenever the trait implementation changes, all the mixed in classes needs to be re-compiled

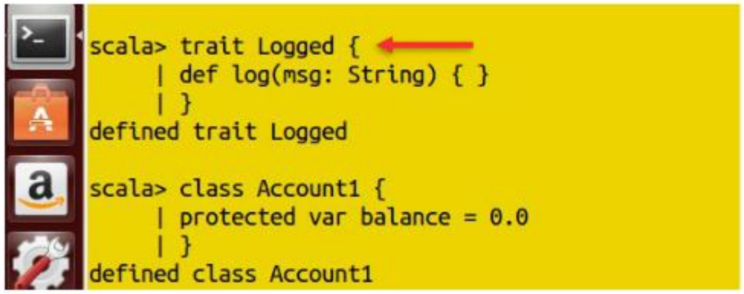
Objects with Traits

→ Traits could be added with individual objects while constructing them

→ Example:

» We'll use Logged trait of standard Scala library:

```
trait Logged {
            def log(msg: String) { }
            }

            class Account1 {
                        protected var balance = 0.0
            }

class SavingsAccount1 extends Account1 with Logged {
  def withdraw(amount: Double) {
    if (amount > balance) log("Insufficient funds")
    else ...
  }
```



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

scala> class Account1 {
     | protected var balance = 0.0
     | }
defined class Account1
```

→ What should be the expected behavior?

Objects with Traits

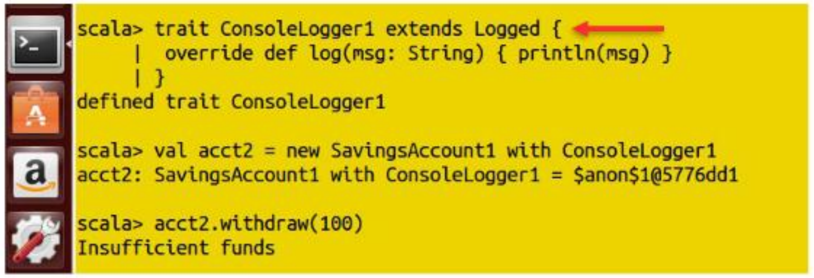
→ Nothing gets logged! As the extended trait didn't have any implementation

→ Now, let's extend the trait:

```
trait ConsoleLogger1 extends Logged {
                override def log(msg: String) { println(msg)
}
                }
```

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

```
val acct2 = new SavingsAccount1 with ConsoleLogger1
                acct2.withdraw(100)
```



```
scala> trait ConsoleLogger1 extends Logged {
     |  override def log(msg: String) { println(msg) }
     | }
defined trait ConsoleLogger1

scala> val acct2 = new SavingsAccount1 with ConsoleLogger1
acct2: SavingsAccount1 with ConsoleLogger1 = $anon$1@5776dd1

scala> acct2.withdraw(100)
Insufficient funds
```

→ Now when withdraw method is executed, the log method of the ConsoleLogger1 is invoked and the message is logged

→ For the class-private field, private getter and setter are generated

→ Thus we get the flexibility of attaching use case specific loggers!

Layered Traits

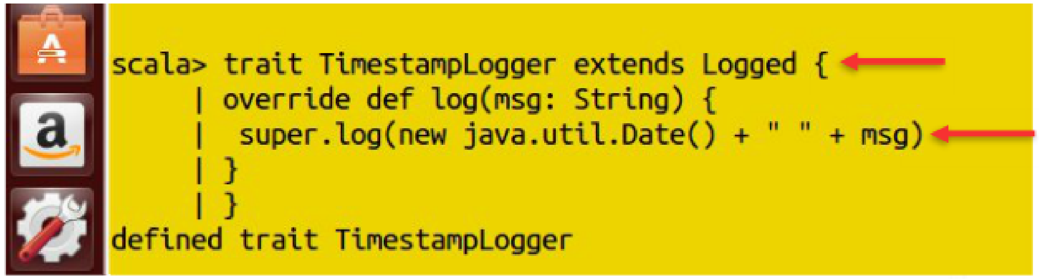
→ Multiple traits can be added to a class or object in Scala

→ 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:

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

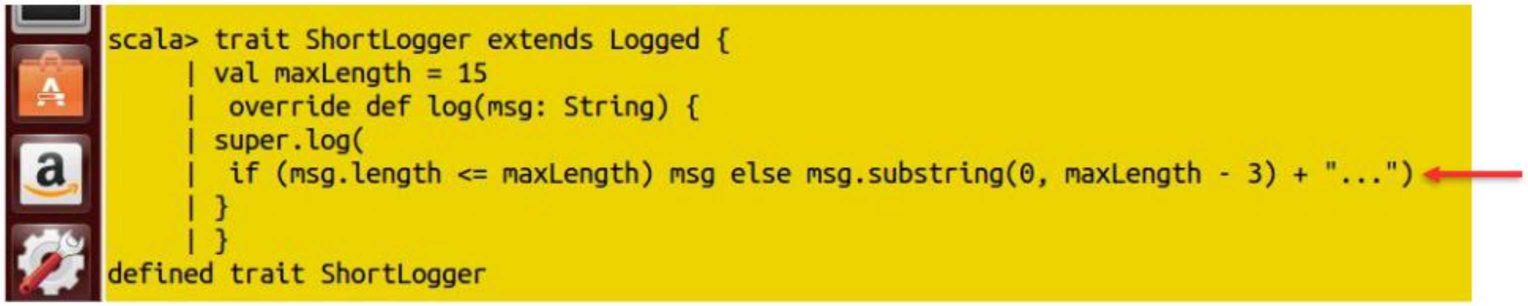


```
scala> trait TimestampLogger extends Logged {
     | override def log(msg: String) {
     |  super.log(new java.util.Date() + " " + msg)
     | }
     | }
defined trait TimestampLogger
```

Layered Traits

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

```
trait ShortLogger extends Logged {
        val maxLength = 15 // See Section 10.8 on fields in traits
        override def log(msg: String) {
          super.log(if (msg.length <= maxLength) msg else msg.substring(0, maxLength - 3) + "...")
                 }
        }
```



```
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
```

Layered Traits

→ Note that each log method calls a super.log

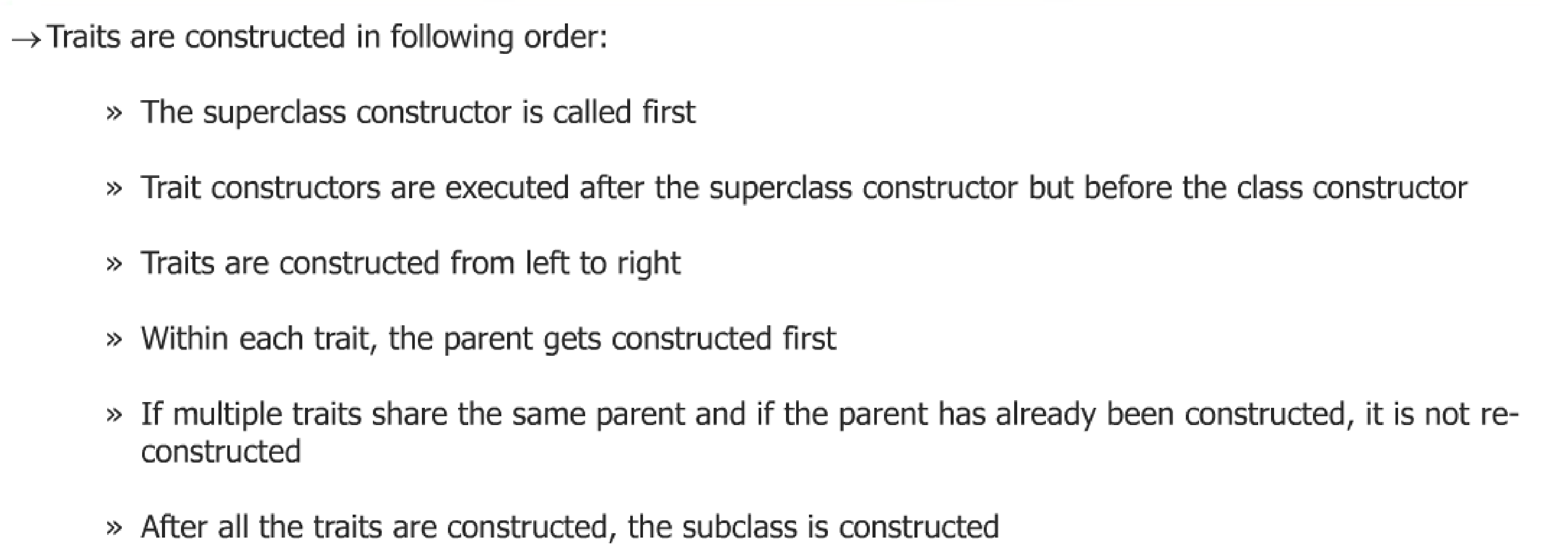
→ In traits, the keyword trait doesn’t have the same meaning as it does with classes

→ 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

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. However, the choice is for immutable ones

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

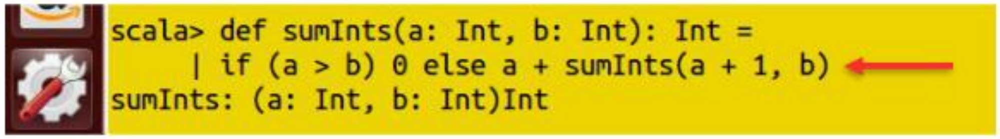
→ This provides a flexible way to compose programs

→ 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:

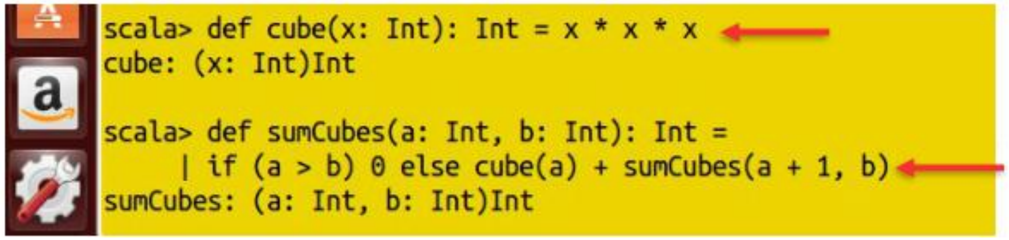
```
def sumInts(a: Int, b: Int): Int =
if (a > b) 0 else a + sumInts(a + 1, b)
```



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

→ Take the sum of the cubes of all the integers between a and b :

```
def cube(x: Int): Int = x * x * x

def sumCubes(a: Int, b: Int): Int =
if (a > b) 0 else cube(a) + sumCubes(a + 1, b)
```

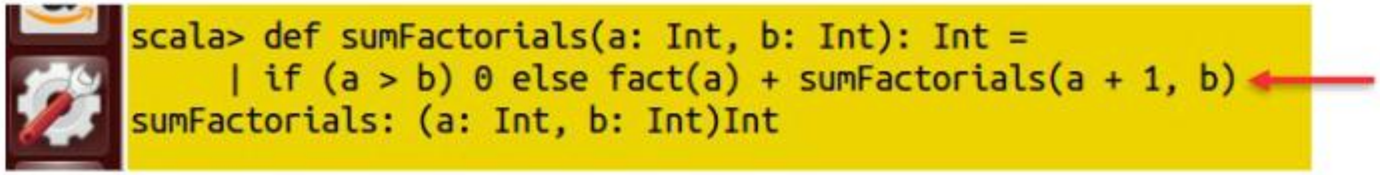


```
scala> def cube(x: Int): Int = x * x * x
cube: (x: Int)Int

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

Examples (Contd.)

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

```
def sumFactorials(a: Int, b: Int): Int =
if (a > b) 0 else fact(a) + sumFactorials(a + 1, b)
```



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

→ These are special cases of below for different values of f

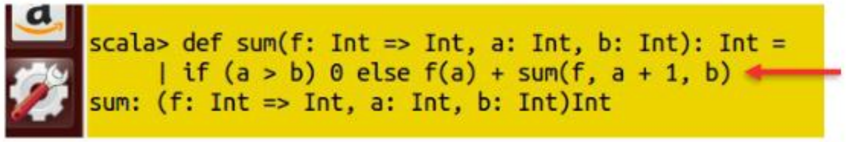
$$\sum_{n=a}^{b} f(n)$$

→ Can we factor out the common pattern?

Summing with Higher Order Functions

→ Let's define:

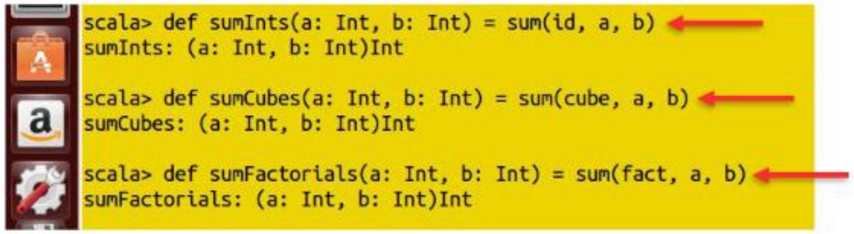
```
def sum(f: Int => Int, a: Int, b: Int): Int =
if (a > b) 0
else f(a) + sum(f, a + 1, b)
```



```
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:

```
def sumInts(a: Int, b: Int) = sum(id, a, b)
def sumCubes(a: Int, b: Int) = sum(cube, a, b)
def sumFactorials(a: Int, b: Int) = sum(fact, a, b)
```



```
scala> def sumInts(a: Int, b: Int) = sum(id, a, b)
sumInts: (a: Int, b: Int)Int

scala> def sumCubes(a: Int, b: Int) = sum(cube, a, b)
sumCubes: (a: Int, b: Int)Int

scala> def sumFactorials(a: Int, b: Int) = sum(fact, a, b)
sumFactorials: (a: Int, b: Int)Int
```

→ Where

```
def id(x: Int): Int = x
def cube(x: Int): Int = x * x * x
def fact(x: Int): Int = if (x == 0) 1 else fact(x - 1)
```

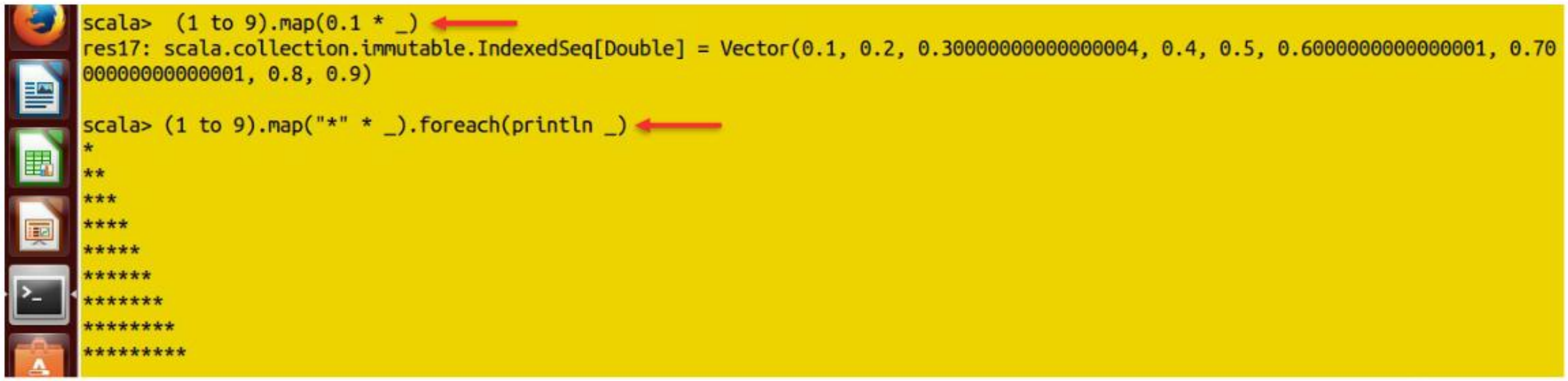
Useful Higher Order Functions

→Map

(1 to 9).map(0.1 * _)

→foreach

(1 to 9).map("*" * _).foreach(println _)



```
scala>  (1 to 9).map(0.1 * _)
res17: scala.collection.immutable.IndexedSeq[Double] = Vector(0.1, 0.2, 0.30000000000000004, 0.4, 0.5, 0.6000000000000001, 0.70
00000000000001, 0.8, 0.9)

scala> (1 to 9).map("*" * _).foreach(println _)
*
**
***
****
*****
******
*******
********
*********
```

Useful Higher Order Functions(Contd.)

→ filter

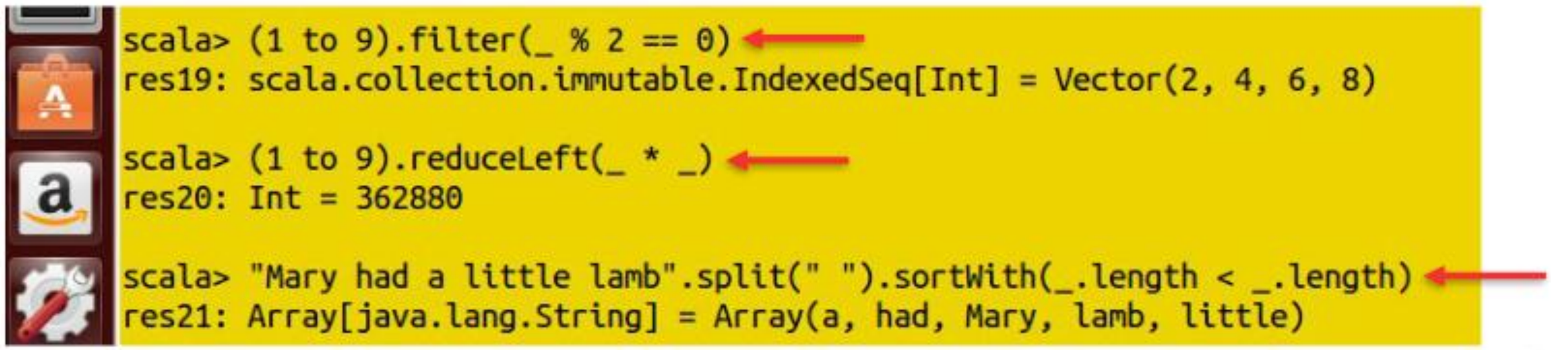
(1 to 9).filter(_ % 2 == 0)

→ reduceLeft

(1 to 9).reduceLeft(_ * _)

→ Split, sortWith

"Mary had a little lamb".split(" ").sortWith(_.length < _.length)



```
scala> (1 to 9).filter(_ % 2 == 0)
res19: scala.collection.immutable.IndexedSeq[Int] = Vector(2, 4, 6, 8)

scala> (1 to 9).reduceLeft(_ * _)
res20: Int = 362880

scala> "Mary had a little lamb".split(" ").sortWith(_.length < _.length)
res21: Array[java.lang.String] = Array(a, had, Mary, lamb, little)
```

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 = ”abc”; println(str)
```

→ We can directly write

```
println(”abc”)
```

→ 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 Functions- 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
```

Anonymous Functions- Syntax

→An anonymous function (x1 : T1; :::; xn : Tn)) E can always be expressed using def as follows:

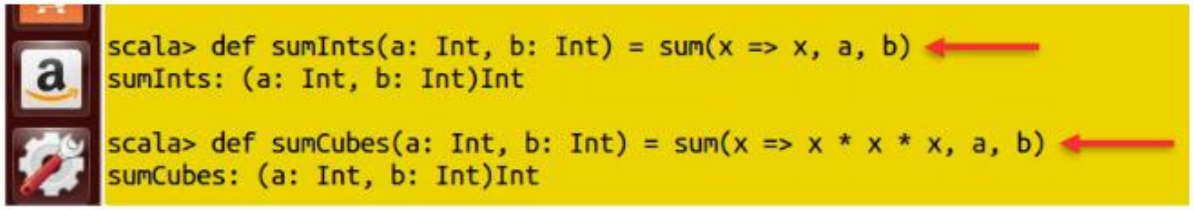
def f(x1 : T1; :::; xn : Tn) = E; f

→where f is an arbitrary, fresh name (that's not yet used in the program)

→One can therefore say that anonymous functions are syntactic sugar

→Using anonymous functions, we can write sums in a shorter way:

```
def sumInts(a: Int, b: Int) = sum(x => x, a, b)

def sumCubes(a: Int, b: Int) = sum(x => x * x * x, a, b)
```



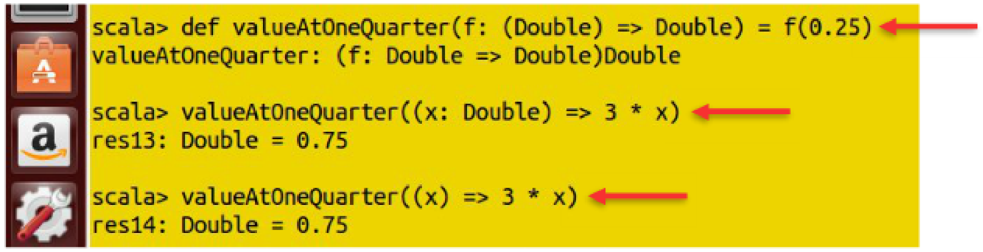
```
scala> def sumInts(a: Int, b: Int) = sum(x => x, a, b)
sumInts: (a: Int, b: Int)Int

scala> def sumCubes(a: Int, b: Int) = sum(x => x * x * x, a, b)
sumCubes: (a: Int, b: Int)Int
```

Parameter Inference

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

→ Example:

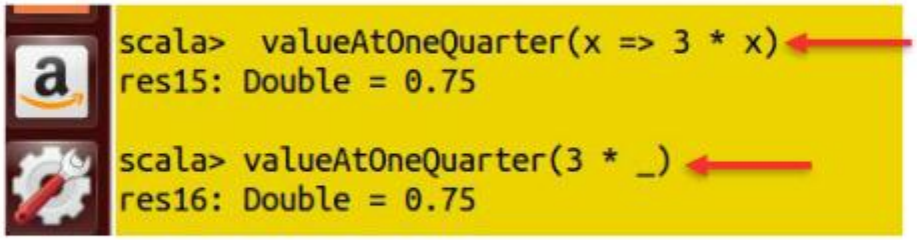
```
def valueAtOneQuarter(f: (Double) => Double) = f(0.25)
            valueAtOneQuarter((x: Double) => 3 * x)  //is same as
            valueAtOneQuarter((x) => 3 * x)
```



```
scala> def valueAtOneQuarter(f: (Double) => Double) = f(0.25)
valueAtOneQuarter: (f: Double => Double)Double

scala> valueAtOneQuarter((x: Double) => 3 * x)
res13: Double = 0.75

scala> valueAtOneQuarter((x) => 3 * x)
res14: Double = 0.75
```

→ If a function has only one parameter, then the "()" can be omitted, hence now it can be expressed as:
valueAtOneQuarter(x => 3 * x)

→ If the parameter is used only ONCE on right side of =>, then it can be replaced with underscore "_" . So now-
valueAtOneQuarter(3 * _)



```
scala>  valueAtOneQuarter(x => 3 * x)
res15: Double = 0.75

scala> valueAtOneQuarter(3 * _)
res16: Double = 0.75
```

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

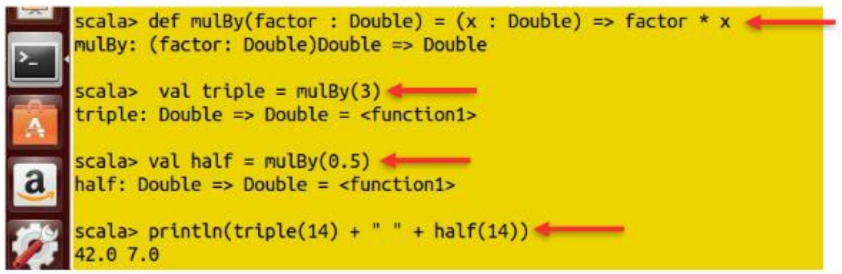
→Thus the functions which access out of bound variables (variables not in scope) are called Closures

→Example:

```
def mulBy(factor : Double) = (x : Double) => factor * x
```

→Now consider following:

```
val triple = mulBy(3)
val half = mulBy(0.5)
println(triple(14) + " " + half(14))
```



```
scala> def mulBy(factor : Double) = (x : Double) => factor * x
mulBy: (factor: Double)Double => Double

scala>  val triple = mulBy(3)
triple: Double => Double = <function1>

scala> val half = mulBy(0.5)
half: Double => Double = <function1>

scala> println(triple(14) + " " + half(14))
42.0 7.0
```

→Here, each of the returned function has its own setting for factor

→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 process of converting a function which takes two arguments to the function which takes one argument

→That function returns a function, which consumes the second argument

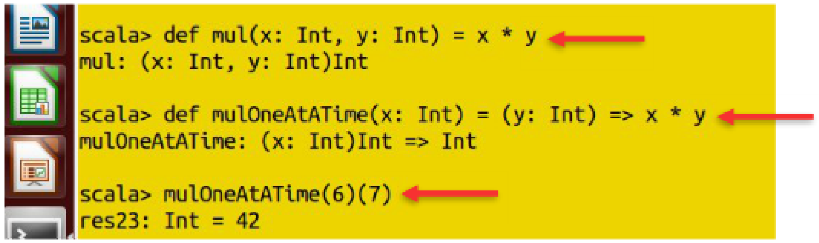
→Example:

```
def mul(x: Int, y: Int) = x * y
```

→The above function takes two arguments

→The same function can be re-written as:

```
def mulOneAtATime(x: Int) = (y: Int) => x * y
```



```
scala> def mul(x: Int, y: Int) = x * y
mul: (x: Int, y: Int)Int

scala> def mulOneAtATime(x: Int) = (y: Int) => x * y
mulOneAtATime: (x: Int)Int => Int

scala> mulOneAtATime(6)(7)
res23: Int = 42
```

→Now the modified function takes an argument, yielding a function that takes another argument

→Two multiply two numbers, we call:

```
mulOneAtATime(6)(7)
```

→Under the hood, the result of mulOneAtATime(6) is the function (y: Int) => 6*y. This function now is applied to 7, which yields 42

Currying (Contd.)

→Scala provides a shortcut for such curried functions:

```
def mulOneAtATime(x: Int)(y: Int) = x * y
```

→Currying is just a frill

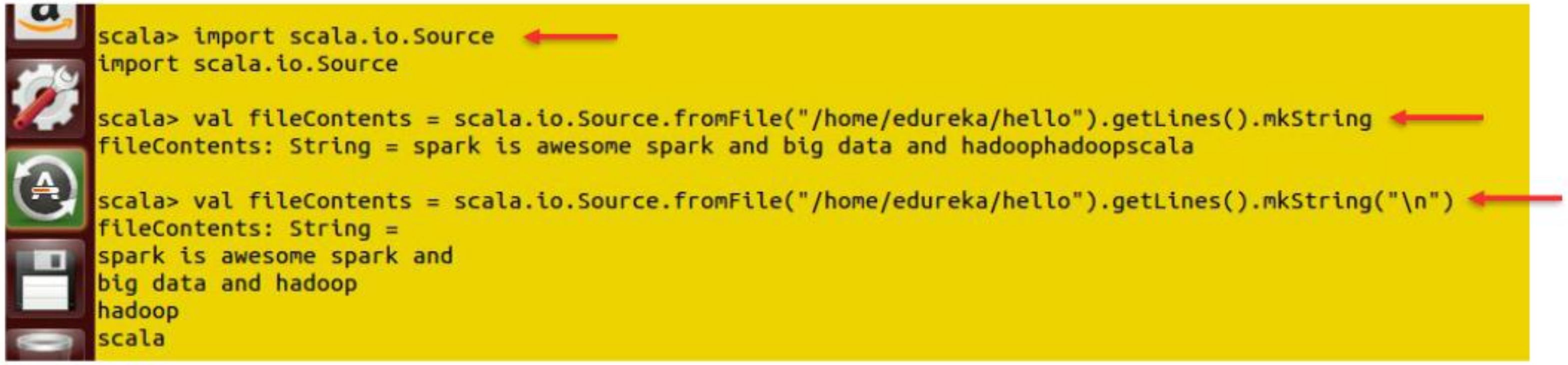
→It is NOT an essential feature of the programming language

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
  - » Demo

→ You can use fromFile also to read all lines in a file. mkString converts a collection into a flat String by each element's toString method



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

scala> val fileContents = scala.io.Source.fromFile("/home/edureka/hello").getLines().mkString
fileContents: String = spark is awesome spark and big data and hadoophadoopscala

scala> val fileContents = scala.io.Source.fromFile("/home/edureka/hello").getLines().mkString("\n")
fileContents: String =
spark is awesome spark and
big data and hadoop
hadoop
scala
```

Question

Multiple inheritance is allowed in Scala

- True
- False

Answer

- False

Question

Traits in Scala are exactly same as interfaces in Java

- True
- False

Answer

- False

Question

The super doesn't call to parent method in Traits

- True
- False

Answer

- True

Question

Traits construction order is decided by their they inheritance

- True
- False

Answer

- False

Question

Functions can’t be passed as arguments in Scala

- True
- False

Answer

- False

Thank You