Contents

Exercise 1: Setting up the Environment

Step1: Installing Vmware from Training Bundle.

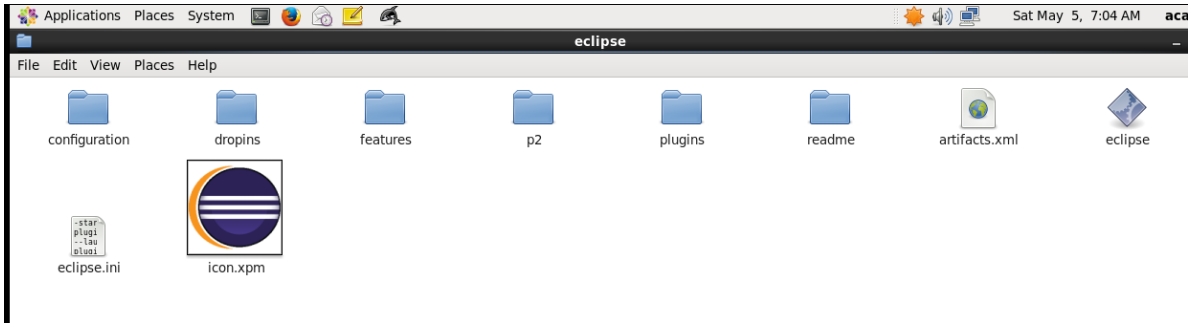
- Under 'Training Bundle' folder, navigate to "Software" folder.
- Find the executable file and complete the installation.

Step2: Extracting the Image

- From the same training bundle, locate the folder named VM image> 'Acadgild_64bit'
- Load the image to get started with the machine.

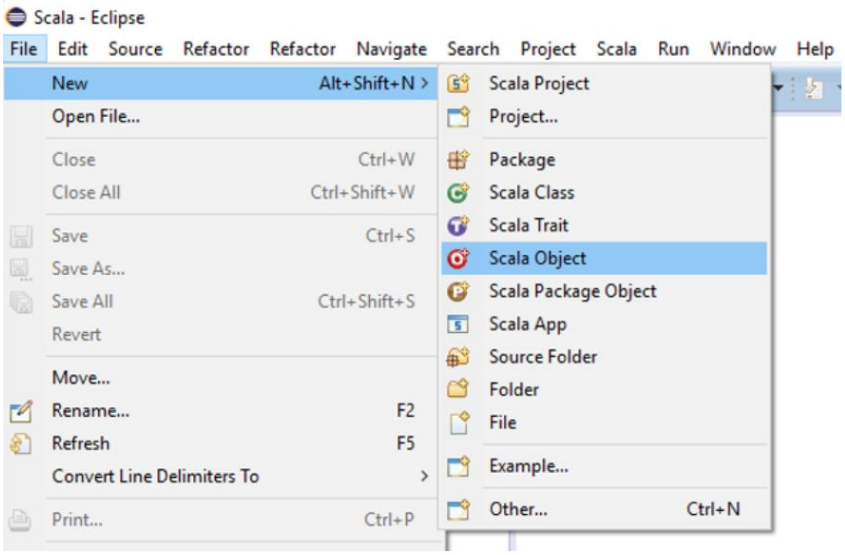
Exercise 2: Hello world Program

Open eclipse from /home/acadgild/eclipse. Click on Eclipse Icon.

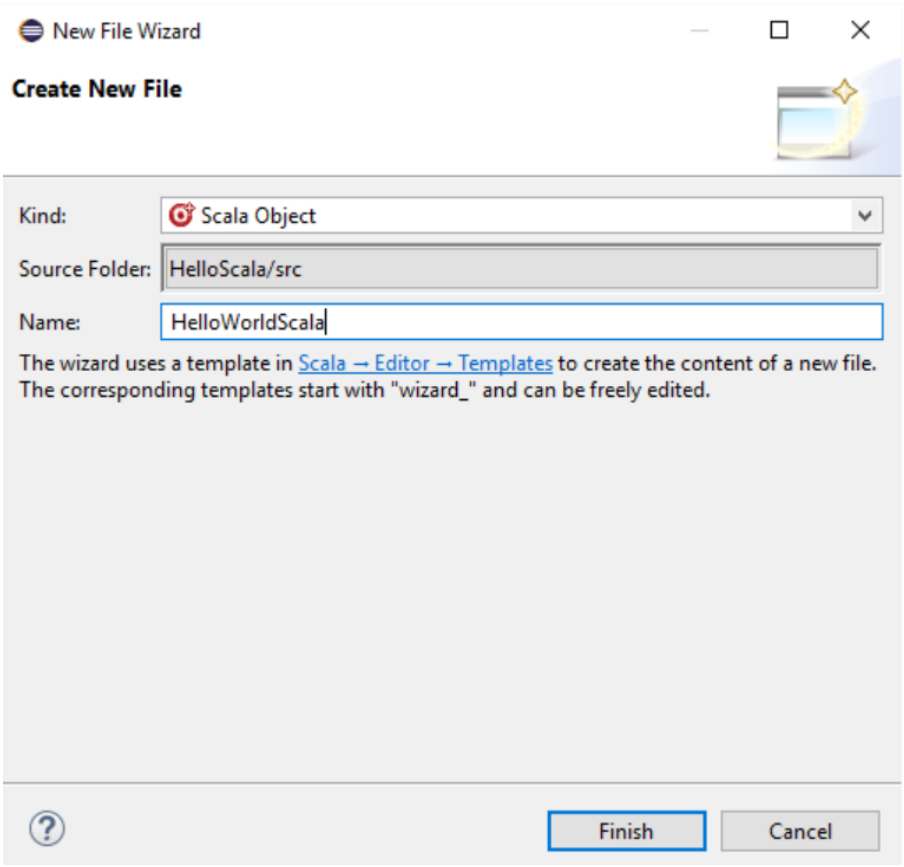


Let us create the first eclipse project for our first Scala program.

1. Now add a new Scala object file to project TrainingDay1,



2. Right click on project > New > Scala Object > HelloWorld.



Add code that prints Hello Scala.

```
object HelloWorld {
  def main(args: Array[String]) {
    println("Hello Scala !!")
  }
}
```

Run the Scala program
Click on Run button or Run menu to run this as a Scala application.

Set the Run configuration if needed.

Output :

```
Hello Scala !!
```

Exercise 3: Mutable and Immutable variables

Declare an immutable value before with val. Let's explore this a bit more:

val immutableValue = ...: Once initialized, we can't assign a different value to immutableValue.
var mutableVariable = ...: We can assign new values to mutableVariable as often as we want.

We will add another Scala Object as VarDataDemo

```
// Import File. Unlike Java, the semicolon ';' is not required.
import java.io.File

object Demo {
  def main(args: Array[String]) {
    var myVar :Int = 10;
    val myVal :String = "Hello Scala with datatype declaration.";
    var myVar1 = 20;
    val myVal1 = "Hello Scala new without datatype declaration.";

    println(myVar); println(myVal); println(myVar1);
    println(myVal1);
  }
}
```

This program declares four variables — two variables are defined with variable declaration and remaining two are without variable declaration.

Output

```
10
Hello Scala with datatype declaration.
20
Hello Scala without datatype declaration.
```

Now, in the same example

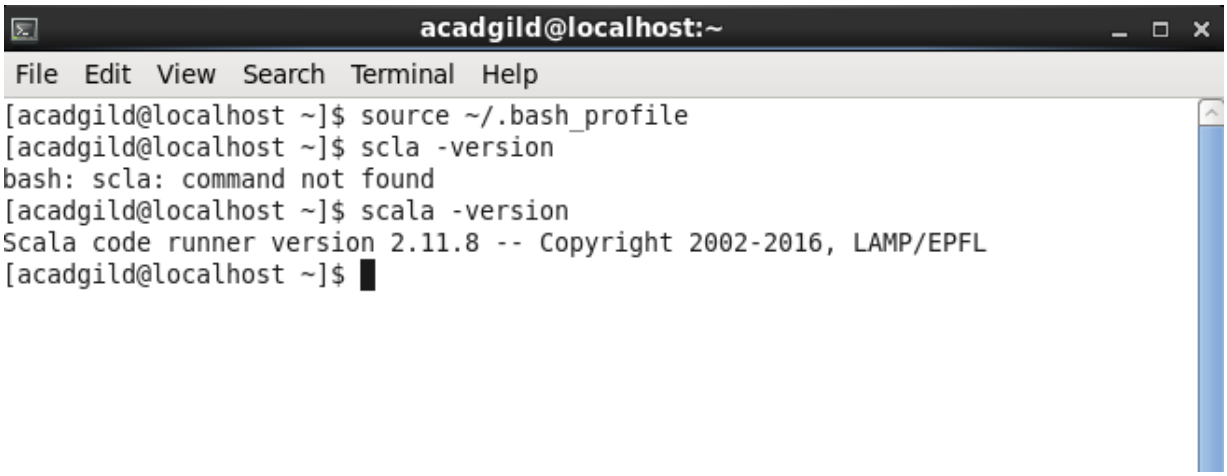
Change the value of variable myVal1

```
myVal1 = "Hello Scala reassignment not allowed.";
```

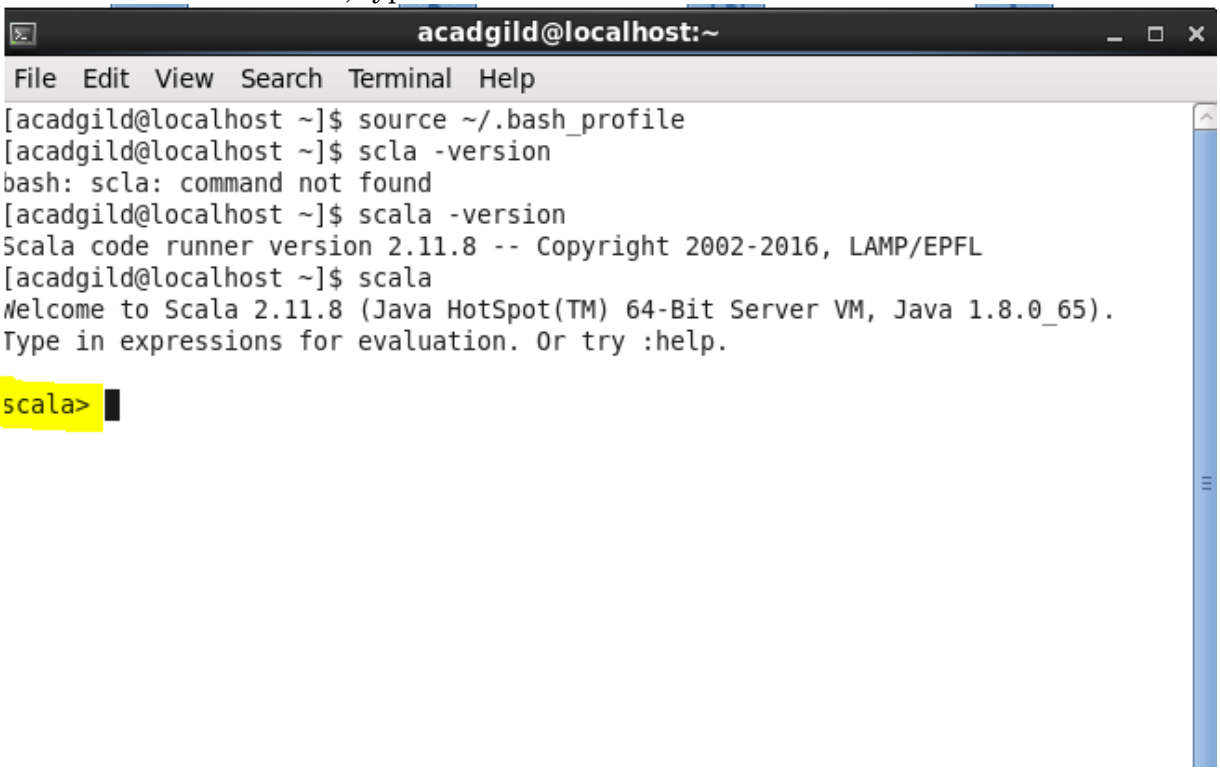
When you execute the Scala object, You will see the message "**Reassignment to val**". The message tells us that once a value has been assigned, we can no longer change it!

Exercise 4: String operations

Open command prompt, and type “source ~/.bash_profile”.
Type scala -version



In the command terminal, type scala.



It will open Scala REPL where we can write scripts.

These are commands followed with the Output

```
scala> val s = "Hello, world"
```

You can get the length of a string:

```
scala> s.length  // 12
```

You can concatenate strings:

```
scala> val s = "Hello" + " world"
```

you can iterate over every character in the string using the foreach method:

```
scala> "hello".foreach(println)
```

You can treat a String as a sequence of characters in a for loop:

```
scala> for (c <- "hello") println(c)
```

You can also treat it as a sequence of bytes:

```
scala> s.getBytes.foreach(println)
```

```
scala> "scala".drop(2)
```

res0: String = ala

```
scala> "scala".drop(2).take(2)
```

res1: String = al

```
scala> "scala".drop(2).take(2).capitalize
```

res2: String = Al

In Scala, you compare two String instances with the == operator. Given these strings:

```
scala> val s1 = "Hello"
```

s1: String = Hello

```
scala> val s2 = "Hello"
```

s2: String = Hello

```
scala> val s3 = "H" + "ello"
```

s3: String = Hello

You can test their equality like this:

```
scala> s1 == s2
```

res0: Boolean = true

```
scala> s1 == s3
```

res1: Boolean = true

However, be aware that calling a method on a null string can throw a NullPointerException:

```
scala> val s1: String = null
```

s1: String = null

```
scala> val s2: String = null
```
s2: String = null

```
scala> s1.toUpperCase == s2.toUpperCase
```
java.lang.NullPointerException // more output here ...

```
scala> val s = "eggs, milk, butter, Coco Puffs"
```
s: java.lang.String = eggs, milk, butter, Coco Puffs

// 1st attempt
```
scala> s.split(",")
```
res0: Array[java.lang.String] = Array(eggs, " milk", " butter", " Coco Puffs")

- Create a Regex object by invoking the .r method on a String, and then use that pattern with findFirstIn when you're looking for one match, and findAllIn when looking for all matches.

```
scala> val numPattern = "[0-9]+".r
```
numPattern: scala.util.matching.Regex = [0-9]+
Next, create a sample String you can search:

```
scala> val address = "123 Main Street Suite 101"
```
address: java.lang.String = 123 Main Street Suite 101
The findFirstIn method finds the first match:

```
scala> val match1 = numPattern.findFirstIn(address)
```
match1: Option[String] = Some(123)

When looking for multiple matches, use the findAllIn method:

```
scala> val matches = numPattern.findAllIn(address)
```
matches: scala.util.matching.Regex.MatchIterator = non-empty iterator
As you can see, findAllIn returns an iterator, which lets you loop over the results:

```
scala> matches.foreach(println)
```
123
101

If you'd rather have the results as an Array, add the toArray method after the findAllIn call:

```
scala> val matches = numPattern.findAllIn(address).toArray
```
matches: Array[String] = Array(123, 101)
If there are no matches, this approach yields an empty Array. Other methods like toList, toSeq, and toVector are also available.

Note: We will discuss on the Collection about List, Seq and Vectors etc in later sections.

```
scala> "hello"(0)
```

res1: Char = h

```
scala> "hello"(1)
```

res2: Char = e

Using Eclipse

Create a Scala Object in Eclipse, with name as OperationsString and let us create some Strings and see the functionality of it in Scala.

```
object Demo {
  def main(args: Array[String]) {
val name = "Alvin"
val fullName = name + " Alexander"  // Alvin Alexander
val fullName = s"$name Alexander"   // Alvin Alexander

name.length             // 5
name.foreach(print)      // Alvin
name.take(2)            // Al
name.take(2).toLowerCase   // al
}
}
```

Multiline strings

Triple-quoted string: """..."""". Useful for embedding newlines, as in the below example. (We'll see another benefit later.)

String interpolation: Invoked by putting s before the string, e.g., s"..." or s"""..."""". Lets us embed variable references and expressions, where the string conversion will be inserted automatically. For example:

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

val foo = """This is
   a multiline
   String"""

val speech = """Four score and
           |seven years ago""".stripMargin
}
}
```

Examples of String substitution/interpolation:

```
object Demo {
  def main(args: Array[String]) {
val name = "Joe"
val age = 42
val weight = 180.5

// prints "Hello, Joe"
println(s"Hello, $name")

// prints "Joe is 42 years old, and weighs 180.5 pounds."
```

```
println(f"$name is $age years old, and weighs $weight%.1f pounds.")

// 'raw' interpolator
println(raw"foo\nbar")

val speech = """Four score and
               |seven years ago
               |our fathers""".stripMargin.replaceAll("\n", " ")
println(speech)
}
}
```

Run the above program and check the output.

Exercise 5: Creating a Range, List, or Array of Numbers

Try the below commands in Scala REPL.

- **Range**

```
scala> val r = 1 to 10
```

r: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

You can set the step with the by method:

```
scala> val r = 1 to 10 by 2
```

r: scala.collection.immutable.Range = Range(1, 3, 5, 7, 9)

```
scala> val r = 1 to 10 by 3
```

r: scala.collection.immutable.Range = Range(1, 4, 7, 10)

Ranges are commonly used in for loops:

```
scala> for (i <- 1 to 5) println(i)
```

1
2
3
4
5

When creating a Range, you can also use until instead of to:

```
scala> for (i <- 1 until 5) println(i)
```

1
2
3
4

- You can easily convert a Range to other sequences, such as an Array or List, like this:

```
scala> val x = (1 to 10).toList
```

x: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

```
scala> val x = (1 to 10).toArray
```

x: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

By using a range with the for/yield construct, you don't have to limit your ranges to sequential numbers:

```
scala> for (i <- 1 to 5) yield i * 2
```

res0: scala.collection.immutable.IndexedSeq[Int] = Vector(2, 4, 6, 8, 10)

You also don't have to limit your ranges to just integers:

```
scala> for (i <- 1 to 5) yield i.toDouble
```

res1: scala.collection.immutable.IndexedSeq[Double] =
Vector(1.0, 2.0, 3.0, 4.0, 5.0)

- **List**

We can create a List like this:

```
val x = List(1, 2, 3)
```

or like this, using cons cells and a Nil element:

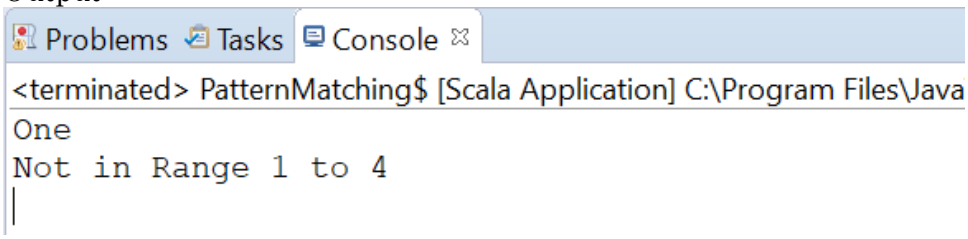
```
val y = 1 :: 2 :: 3 :: Nil
```

Exercise 6: Pattern Matching

Pattern matching is more like Java's switch statements with a few differences. With one expression/value to match against several case statements, whenever a match happens, the corresponding block of code is executed. This gives more than one option for our program flow to follow. Java's switch is a fall-through statement, which means it executes all the statements after the very first match until it confronts a break statement.

```
object PatternMatching extends App {

  def matchAgainst(i: Int) = i match {
    case 1 => println("One")
    case 2 => println("Two")
    case 3 => println("Three")
    case 4 => println("Four")
    case _ => println("Not in Range 1 to 4")
  }

  matchAgainst(1)
  matchAgainst(5)
}
```

Output



Note: In Scala, there's no break statement. Also, there's no default case in Scala's pattern matching. Instead, a wildcard "-_" is used that matches against any other case that has not been covered in previous case statements.

Pattern matching, “And” and “Or” expressions

Scala REPL(Matching Multiple Conditions with One Case Statement)

```
scala>val i = 5
scala>i match {
  case 1 | 3 | 5 | 7 | 9 => println("odd")
  case 2 | 4 | 6 | 8 | 10 => println("even")
}
```

We can also assign the results into another variable, as below

```
scala>val evenOrOdd = someNumber match {
  case 1 | 3 | 5 | 7 | 9 => println("odd")
  case 2 | 4 | 6 | 8 | 10 => println("even")
}
```

- Pattern Matching in List

  When doing a matching in List, always ensure to handle NIL cases as in the below example.

  Lets create a list, and will define two methods as sum() and multiple() which will match the cases and will do the aggregation.

```
scala>def sum(list: List[Int]): Int = list match {
  case Nil => 1
  case n :: rest => n + sum(rest)
}

scala>def multiply(list: List[Int]): Int = list match {
  case Nil => 1
  case n :: rest => n * multiply(rest)
}

scala> val nums = List(1,2,3,4,5)
nums: List[Int] = List(1, 2, 3, 4, 5)

scala> sum(nums)
res0: Int = 16

scala> multiply(nums)
res1: Int = 120
```

Note: When using this recipe, be sure to handle the Nil case, or you'll get the following error in the REPL.

Eclipse

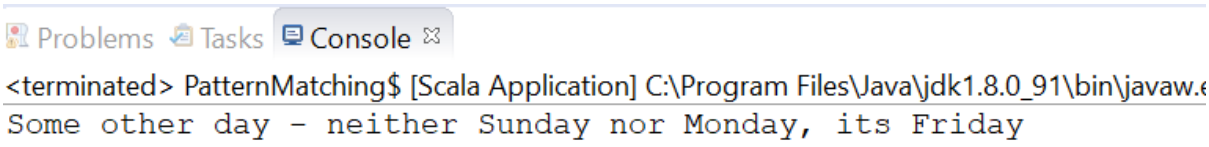
```
object PatternMatching extends App {

  val dayOfWeek = "Friday"

  val typeOfDay = dayOfWeek match {
    case "Monday" => "Manic Monday"
    case "Tuesday" | "Wednesday" | "Thursday" | "Friday" => "Other working day"
    case someOtherDay if someOtherDay == "Sunday" => "Sleepy Sunday"
    case someOtherDay if someOtherDay == "Saturday" => "Sizzing Saturday"
  }

  print(typeOfDay)
}
```

Check the output for the value for variable typeOfDay and someOtherDay.

Pattern matching: catch all to match all in a variable

```
object PatternMatching extends App {

 val dayOfWeek = "Friday"
 val typeOfDay = dayOfWeek match{
 case "Monday" => "Manic Monday"
 case "Sunday" => "Sleepy Sunday"
 case someOtherDay => {
   println(s"Some other day - neither Sunday nor Monday, its $someOtherDay")
   someOtherDay
 }
 }
}
```

Output



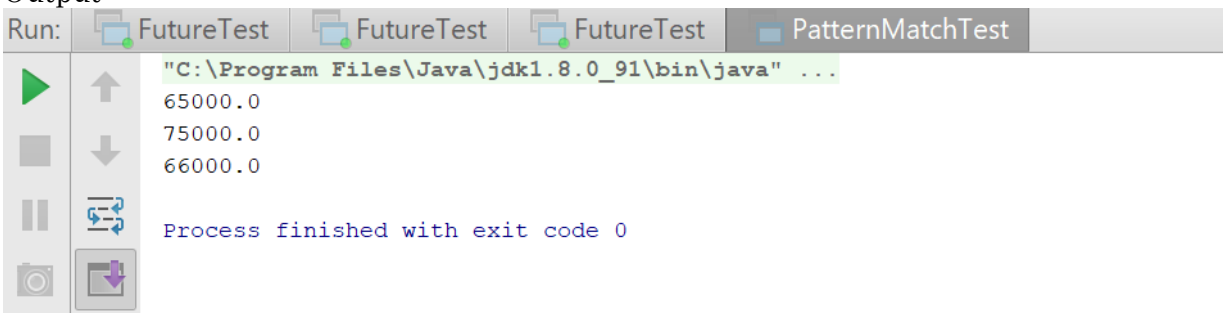
Pattern Matching using Constructors

We have defined three categories of employees: Developer, Consultant, and ContractEmp. We've a problem to solve: we'll have to process the joining bonus amount for specific employees in a specific category with some conditions.

```
trait Employee
case class ContractEmp(id: String, name: String) extends Employee
case class Developer(id: String, name: String) extends Employee
case class Consultant(id: String, name: String) extends Employee

object PatternMatchTest extends App{
 /*
  * Process joining bonus if
  *    :> Developer has ID Starting from "DL"  JB: 1L
  *    :> Consultant has  ID Starting from "CNL":  1L
  */
 def processJoiningBonus(employee: Employee, amountCTC: Double) = employee match {
  case ContractEmp(id, _) => amountCTC
  case Developer(id, _) => if(id.startsWith("DL")) amountCTC + 10000.0 else amountCTC
  case Consultant(id, _) => if(id.startsWith("CNL")) amountCTC + 10000.0 else amountCTC

  /* case Developer(id, _) if id.startsWith("DL") => amountCTC + 10000.0
  case Consultant(id, _) if id.startsWith("CNL") =>  amountCTC + 10000.0*/
 }

 val developerEmplEligibleForJB = Developer("DL0001", "Alex")
 val consultantEmpEligibleForJB = Consultant("CNL0001","Henry")
 val developer = Developer("DI0002", "Heith")

 println(processJoiningBonus(developerEmplEligibleForJB, 55000))
 println(processJoiningBonus(consultantEmpEligibleForJB, 65000))
```

```
  println(processJoiningBonus(developer, 66000))
}
```

Output:



Assignment: Create a Scala object, define a fibbonaci function for three cases:
Case 1. For n ==0 or n==1
Case 2: For n<=0
Case 3. Any other value of n.

After the method is defined, invoke the method and observe the output.

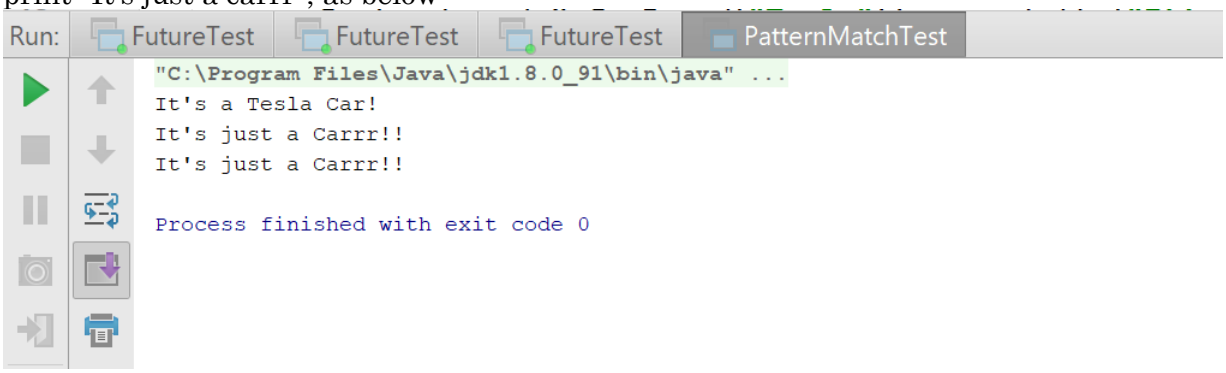
Assignment: match a case class that has another case class inside
We have a nested case class inside our Car, named CarBrand, and we performed a pattern match against that

```
case class Car(name: String, brand: CarBrand)
case class CarBrand(name: String)
```

Create the instance of car with different brands , as below:

```
val car = Car("Model X", CarBrand("Tesla"))
val anyCar = Car("Model XYZ", CarBrand("XYZ"))
val myCar = Car("Model XYZ", CarBrand("EON"))
```

Make a match over Tesla and print "It's a Tesla Car" and for i=other instances, print "It's just a carrr", as below:

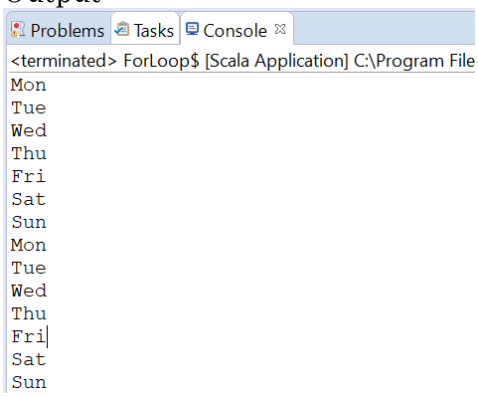


Exercise 7: The for loop

In Scala, a for loop, also called for comprehension takes a sequence of elements, and performs an operation on each of them. One of the ways we can use them is:

```
object ForLoop extends App {

 val daysOfWeekList = List("Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun")

 for (day <- daysOfWeekList) {
  println(day)
 }

 for (i <- 0 to daysOfWeekList.size - 1) {
  println(daysOfWeekList(i))
 }

}
```

Output:



We can also use conditional loop within for loop.

```
object ForLoopield extends App {
 val daysOfWeekList = List("Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun")

 for (day <- daysOfWeekList if day == "Mon") {
  println(day)
 }
}
```

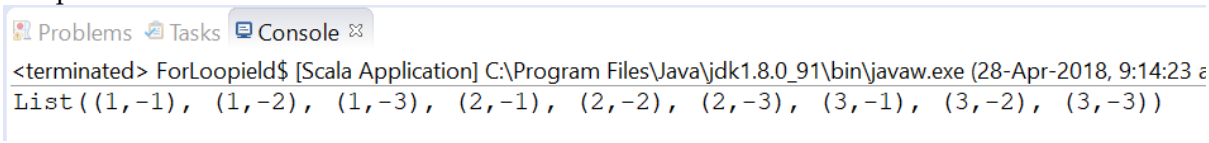
Output

Mon

For loop with yield : You can ofcourse use a for loop to iterate over a collection. If what you want to do is do something with each element of a collection and return a new collection with the transformed elements, you can use yield.

```
object ForLoopYield extends App {

  val l = List(1, 2, 3)
  val t = List(-1, -2, -3)
  val crossProduct = for (x <- l; y <- t) yield (x, y)
  println(crossProduct)
}
```

Output:



Scala REPL

Given a simple array:

```
scala>val a = Array("apple", "banana", "orange")
```

I prefer to iterate over the array with the following for loop syntax, because it's clean and easy to remember:

```
scala> for (e <- a) println(e)
```

apple
banana
orange

RETURNING VALUES FROM A FOR LOOP

```
scala> val newArray = for (e <- a) yield e.toUpperCase
```

newArray: Array[java.lang.String] = Array(APPLE, BANANA, ORANGE)

```
scala> for {
  i <- 1 to 2
  j <- 1 to 2
} println(s"i = $i, j = $j")
```

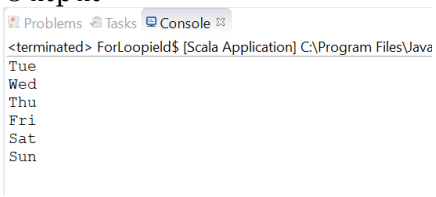
i = 1, j = 1
i = 1, j = 2
i = 2, j = 1
i = 2, j = 2

Exercise 8: Do while loops

The Scala while loop executes a certain block of code, as long as a certain condition is true.

```
object WhileLoop extends App {
 val daysOfWeekList = List("Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun")

 var x = 0;
 while (x < daysOfWeekList.size-1) {
  x += 1
  val day = daysOfWeekList(x)
  println(day)
 }

}
```

Output:



Exercise 9: Scala REPL

// NUMBERS and ARITHMETIC
// Code that goes along with tutorial
// Note: This is meant show what is to be run in REPL
// This won't work as a script!

// Two main types of Numbers:

// Integers (Whole)

```
scala> 100
```

res0: Int = 100

// Double (Floating Point)

```
scala> 2.5
```

res1: Double = 2.5

// Addition

```
scala> 1 + 1
```

res2: Int = 2

// Subtraction

```
scala> 2 - 1
```

res3: Int = 1

// Multiplication

```
scala> 2 * 5
```

res4: Int = 10

// Division with Integers (Classic)

```
scala> 1 / 2
```

res5: Int = 0

// Division with Doubles (True)

```
scala> 1.0/2
```

res6: Double = 0.5

```
scala> 1/2.0
```
res7: Double = 0.5

// Exponents
```
scala> math.pow(4,2)
```
res36: Double = 16.0

// Modulo (Remainder)
```
scala> 10 % 4
```
res8: Int = 2

```
scala> 9 % 4
```
res9: Int = 1

// You can call back results:
```
scala> res0
```
res10: Int = 100

// Order of Operations with Parenthesis
```
scala> (1 + 2) * (3 + 4)
```
res11: Int = 21

```
scala> 1 + 2 * 3 + 4
```
res12: Int = 11

// Quick understanding check
// Convert 3 feet to meters using scala
```
scala> 3 * 0.3048
```
res13: Double = 0.9144000000000001

Assignment:

1. What is 2 to the power of 5 ?

2. What is the remainder of 180 divided by 7?

3. Given variable pet_name = "Sammy", use string interpolation to print out "My dog's name is Sammy."

4. Use scala to find out if the letter sequence "xyz" is contained in: "sadfjshfjyuyxyzjkfuidkjklhasyysdfk"

5. What is the difference between a value and a variable?

Exercise 10: Methods

Methods in Scala can be written using the def keyword, followed by the name of the method, with some arguments supplied as inputs to the function. The generic syntax for a function.

```
def function_name(arg1: arg1_type, arg2: arg2_type,...): return_type = ???
```

Here, we have two statements in the method definition. The first is printing and the latter is evaluating the comparison. Hence, we encapsulated them using a pair of curly braces. The first method will explain we can avoid curly braces in Scala for writing Inline functions and the code becomes clean and concise.

```
object FunctionSyntax extends App{
 /*
  * Function compare two Integer numbers
  * @param value1 Int
  * @param value2 Int
  * return Int
  * 1  if value1 > value2
  * 0  if value1 = value2
  * -1 if value1 < value2
  */
 def compareIntegers(value1: Int, value2: Int): Int = if (value1 == value2) 0 else if (value1 > value2) 1 else -1

 def compareIntegersV1(value1: Int, value2: Int): Int = {
   if (value1 == value2) 0 else if (value1 > value2) 1 else -1
  }

 println(compareIntegers(1, 2))
 println(compareIntegersV1(2, 1))


}
```

Defining a Method That Returns Multiple Items (Tuples)

Let us consider you want to return multiple values from a method, but don't want to wrap those values in a makeshift class.

Scala also lets you return multiple values from a method using tuples. First, define a method that returns a tuple:

```
object FunctionSyntax extends App {
 // a method that returns a tuple
 def getStockInfo = {
  // other code here ...
  ("NFLX", 100.00, 101.00) // return a tuple
 }

 // call the method like this
 val (symbol, currentPrice, bidPrice) = getStockInfo

 // or call the method like this
 val x = getStockInfo
 x._1 // String = NFLX
 x._2 // Double = 100.0
 x._3 // Double = 101.0

 println(x._1 + "--" +x._2 + "--"+x._3)
}
```

Note: the getStockInfo method returned a tuple with three elements, so it is a Tuple3. Tuples can contain up to 22 variables and are implemented as Tuple1 through Tuple22 classes. As a practical matter, you don't have to think about those specific classes; just create a new tuple by enclosing elements inside parentheses and Scala will take care.

Let us understand what is Tuple?
Tuple s1 can be declared as;

val s1 = new Tuple2(12,"Harry")
The type of the tuple depends upon the number of elements contained and the data types of those elements. The type (12,"Harry") is of type Tuple2[Int,String]

Exercise on Tuples

Scala REPL

```
// TUPLES

// An ordered sequence of values
// Can be of multiple data types
scala> val my_tup = (1,2,"hello",23.2,true)
my_tup: (Int, Int, String, Double, Boolean) = (1,2,hello,23.2,true)

// Can also be nested:
scala> (3,1,(2,3))
res46: (Int, Int, (Int, Int)) = (3,1,(2,3))

// Accessing elements with ._n notation

// Indexing starts at 1
```

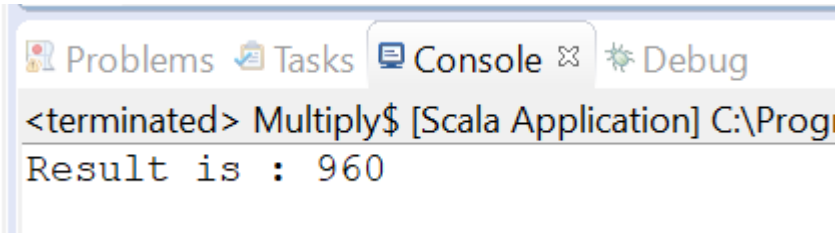
```
scala> val greeting = my_tup._3
greeting: String = hello

scala> my_tup._1
res37: Int = 1
```

Assignment
Given the tuple (1,2,3,(4,5,6)) retrieve the number 6.

Example 1. Accessing Objects in Tuple

```
object Multiply {
 def main(args: Array[String]) {
  val m1 = (20, 12, 16, 4)

  val mul = m1._1 * m1._2 * m1._4

  println("Result is : " + mul)
 }
}
```

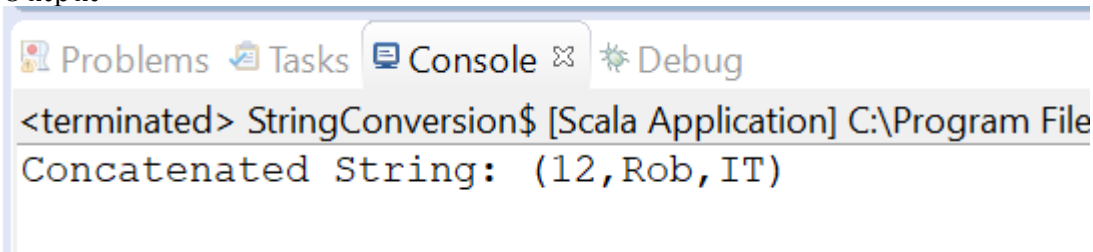
Output:



Example 2: Conversion to String
The Tuple.toString() method concatenates all the elements of the tuple into a string.

```
object StringConversion {
 def main(args: Array[String]) {
  val student = new Tuple3(12, "Rob", "IT")

  println("Concatenated String: " + student.toString())
 }
}
```

Output:



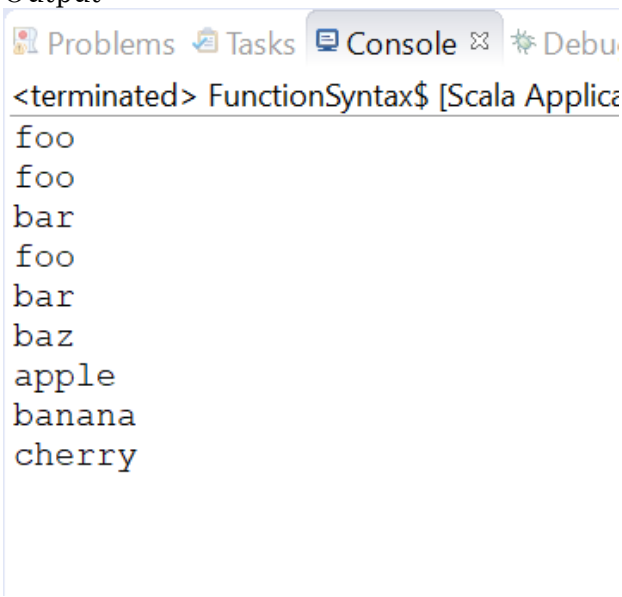
Creating Methods That Take Variable-Argument Fields

Problem
To make a method more flexible, you want to define a method parameter that can take a variable number of arguments, i.e., a varargs field.

Solution
Define a varargs field in your method declaration by adding a * character after the field type:

```
object FunctionSyntax extends App {
 def printAll(strings: String*) {
  strings.foreach(println)
 }

 // these all work
 printAll()
 printAll("foo")
 printAll("foo", "bar")
 printAll("foo", "bar", "baz")

 // a sequence of strings
 val fruits = List("apple", "banana", "cherry")

 // pass the sequence to the varargs field
 printAll(fruits: _*)
}
```

Output:



Note: If you come from a Unix background, it may be helpful to think of _* as a "splat" operator. This operator tells the compiler to pass each element of the sequence to printAll as a separate argument, instead of passing fruits as a single argument.

Exercise 11: Functional Programming

In scala, functions are first class values. You can store function value, pass function as an argument and return function as a value from other function. You can create function by using def keyword. You must mention return type of parameters while defining function and return type of a function is optional. If you don't specify return type of a function, Scala compiler will type infer it.

Scala REPL

- Anonymous Functions

You can create anonymous functions.

```
scala> (x: Int) => x + 1
```

res2: (Int) => Int = <function1>
This function adds 1 to an Int named x.

```
scala> res2(1)
```

res3: Int = 2
You can pass anonymous functions around or save them into vals.

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

addOne: (Int) => Int = <function1>

```
scala> addOne(1)
```

res4: Int = 2

- Creating a method and converting into a function.

```
scala> def test1 = (str: String) => str + str
```

test1: (String) => java.lang.String

```
scala> val test2 = test1
```

test2: (String) => java.lang.String = <function1>

- Converting methods into functions using wild card expressions.

```
scala> def add(i: Int, j: Int) = i + j
```

add: (i: Int,j: Int)Int

```
scala> val addF = add(_, _)
```
addF: (Int, Int) => Int = <function2>

```
scala> val addF2 = add _
```
addF2: (Int, Int) => Int = <function2>

- Invoking functions with tuples as parameters

```
scala> def getRectangleArea (length:Double, breadth:Double):Double = {length * breadth}
```
getRectangleArea: (length: Double, breadth: Double)Double

```
scala> val area = getRectangleArea(5,8)
```
area: Double = 40.0

```
scala> val perimeterOfSquare = 20.0
```
perimeterOfSquare: Double = 20.0

```
scala> (getRectangleArea _).tupled({val sideOfSquare = perimeterOfSquare/4; (sideOfSquare,sideOfSquare)})
```
res0: Double = 25.0

- **Create Function With Option Return Type**

Let's define a method named dailyCouponCode() which will assume a database lookup to provide our customers with a daily coupon code.
Since there may or may not be a daily coupon code available, it would be a good idea for the users of our dailyCouponCode() function to be aware explicitly of the possibility that the daily coupon code may be empty.

As such, you can define the dailyCouponCode() function's return type to be an Option of the type String.

```
scala> println(s"Step 1: Define a function which returns an Option of type String")
```
Step 1: Define a function which returns an Option of type String

```
scala> def dailyCouponCode(): Option[String] = {
     |   // look up in database if we will provide our customers with a coupon today
     |   val couponFromDb = "COUPON_1234"
     |   Option(couponFromDb).filter(_.nonEmpty)
     | }
```
dailyCouponCode: ()Option[String]

NOTE:

We are also lifting the couponFromDb value into an Option which will perform a null check.
We then remove any empty strings using the filter() function.

call function with Option return type using getOrElse

- Since the dailyCouponCode() function from Step 1 returns an Option of type String, you should use getOrElse() function when retrieving its return value.

```
scala> println(s"\nStep 2: Call function with Option return type using getOrElse")
```

Step 2: Call function with Option return type using getOrElse

```
scala> val todayCoupon: Option[String] = dailyCouponCode()
```

todayCoupon: Option[String] = Some(COUPON_1234)

```
scala> println(s"Today's coupon code = ${todayCoupon.getOrElse("No Coupon's Today")}")
```

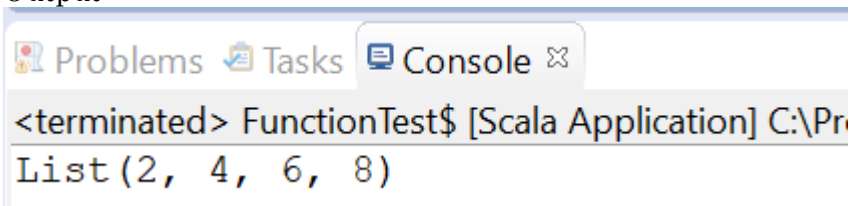
Today's coupon code = COUPON_1234

Eclipse:

Let us create an Object where we can pass an anonymous function to the List's filter method to create a new List that contains only even numbers:

```
object FunctionTest extends App {
  val x = List.range(1, 10)
  val evens = x.filter((i: Int) => i % 2 == 0)

  //val evens = x.filter(_ % 2 == 0) //Syntactic sugar way of writing
  println(evens)
}
```

Output



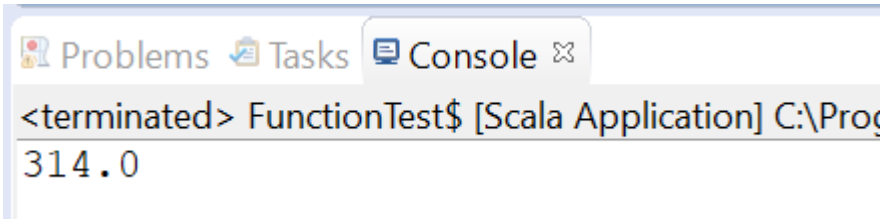
Note: Scala lets you use the _ wildcard instead of a variable name when the parameter appears only once in your function.

Using Methods converting into Functions as Variables

Scala is very flexible, and just like you can define an anonymous function and assign it to a variable, you can also define a method and then pass it around like an instance variable.

```
object FunctionTest extends App {
  val PI = 3.14

  val r = 10

  def getcircleArea(r: Double): Double = PI * r * r

  val calcCircleArea = getcircleArea(r)
  println(calcCircleArea)
}
```

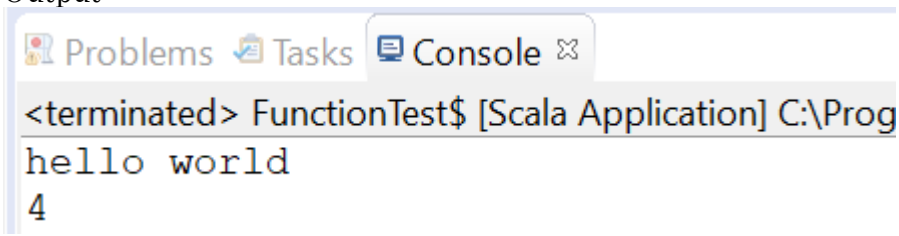
Output:



If your method is made up of many expressions, you can use {} to give yourself some breathing room.

```
object FunTest1 extends App {
 def timesTwo(i: Int): Int = {
  println("hello world")
  i * 2
 }

 val a = timesTwo(2)
 println(a)
}
```

Output:



Passing function inside function

Create two functions : a sum function and a multiply function:
val sum = (x: Int, y: Int) => x + y
val multiply = (x: Int, y: Int) => x * y

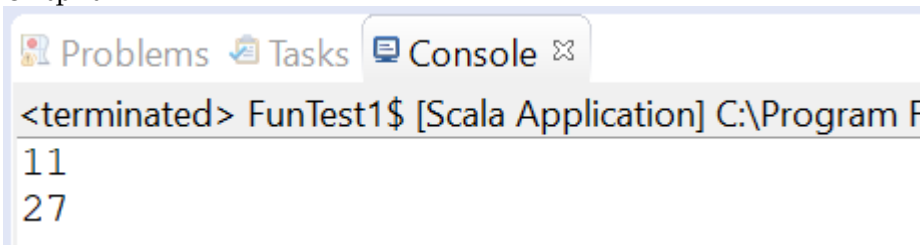
Now let us define another method which will take function as one of the parameter.
def executeAndPrint(f:(Int, Int) => Int, x: Int, y: Int) {
  val result = f(x, y)
  println(result)
}

Let us wrap them inside a object, to show its behaviour in Eclipse

```
object FunTest1 extends App {
 val sum = (x: Int, y: Int) => x + y
 val multiply = (x: Int, y: Int) => x * y

 def executeAndPrint(f: (Int, Int) => Int, x: Int, y: Int) {
  val result = f(x, y)
  println(result)
 }
 executeAndPrint(sum, 2, 9) // prints 11
 executeAndPrint(multiply, 3, 9) // prints 27
```

```
}
```

Output:



Note: executeAndPrint method doesn't know what algorithm is actually run. All it knows is that it passes the parameters x and y to the function it is given and then prints the result from that function. This is similar to defining an interface in Java and then providing concrete implementations of the interface in multiple classes.

Scala REPL

```
//Returning Unit
def simple(): Unit = {
  println("Simple Print")
}
simple()

// Adding two inputs
def adder(num1:Int,num2:Int): Int = {
  return num1 + num2
}

val x = adder(2,3)
println(x)

// Greeting Someone
def greetName(name:String): String = {
  return s"Hello $name"
}
val fullgreet = greetName("Jose")
println(fullgreet)

// Check if a number is prime
def isPrime(numcheck:Int): Boolean = {
  for(n <- Range(2,numcheck)){
    if(numcheck%n == 0){
      return false
    }
    }
    return true
```

```
}


println(isPrime(10))
println(isPrime(23))

// Using Collections

val numbers = List(1,2,3,7)
def check(nums:List[Int]): List[Int]={
  return nums
}

println(check(numbers))

// "One Line Functions"
def quickgreet(name:String) = s"Hello $name"
println(quickgreet("Sammy"))
```

Assignment:

1.) Check for Single Even:
Write a function that takes in an integer and returns a Boolean indicating whether
or not it is even. See if you can write this in one line!

2.) Check for Evens in a List:
Write a function that returns True if there is an even number inside of a List, otherwise, return False.

3.) Lucky Number Seven:
Take in a list of integers and calculate their sum. However, sevens are lucky and they should be counted twice, meaning their value is 14 for the sum. Assume the list isn't empty.

4.) Can you Balance?
Given a non-empty list of integers, return true if there is a place to
split the list so that the sum of the numbers on one side
is equal to the sum of the numbers on the other side. For example, given the list (1,5,3,3) would return true, you can split it in the middle. Another example (7,3,4) would return true 3+4=7. Remember you just need to return the boolean, not the split index point.

5.) Palindrome Check
Given a String, return a boolean indicating whether or not it is a palindrome. (Spelled the same forwards and backwards). Try exploring methods to help you.

Exercise 12: Higher Order Functions

A function which accepts another function as arguments or returns a function is a higher-order function.

Scala REPL
Let's start by writing a simple Function which can be applied to a list of integer values, resulting into another filtered list, containing only all even values of the incoming list:

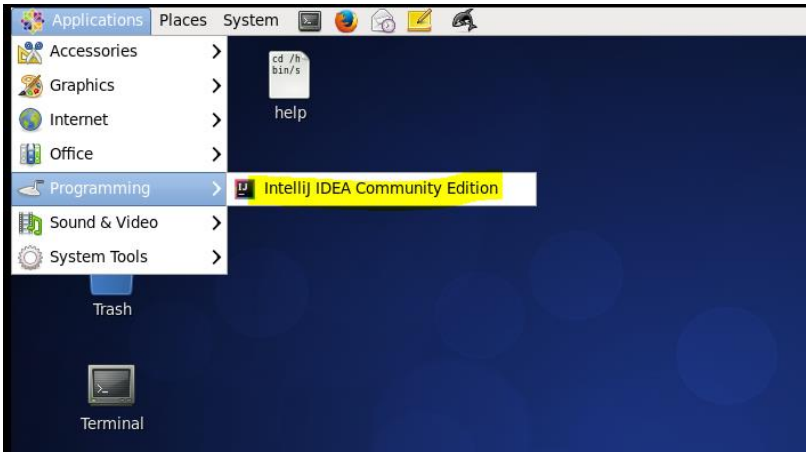
```
scala> val even =  ( x :Int ) => x % 2 == 0
even: Int => Boolean = <function1>

scala> val odd =  ( x :Int ) => x % 2 == 1
odd: Int => Boolean = <function1>

scala> val candidates = List( 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 )
candidates: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

Now let us define that function (with that needed type) as another incoming value to the argument list of our filter function and be done. Here filter is the higher order function which is accepting a function as an input arguments.

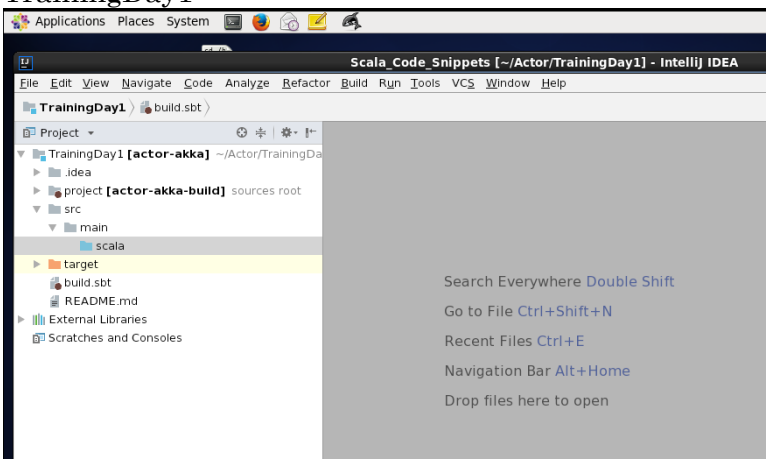
```
scala> val  filter = ( predicate :Int => Boolean, xs :List[Int] )  =>  {
     |
     |     for(  x <- xs;  if predicate( x )  )  yield x
     | }
filter: (Int => Boolean, List[Int]) => List[Int] = <function2>

scala> val evenValues = filter( even, candidates )
evenValues: List[Int] = List(2, 4, 6, 8, 10)

scala> val oddValues = filter( odd, candidates )
oddValues: List[Int] = List(1, 3, 5, 7, 9)
```

Creating a worksheet in IDE- IntelliJ

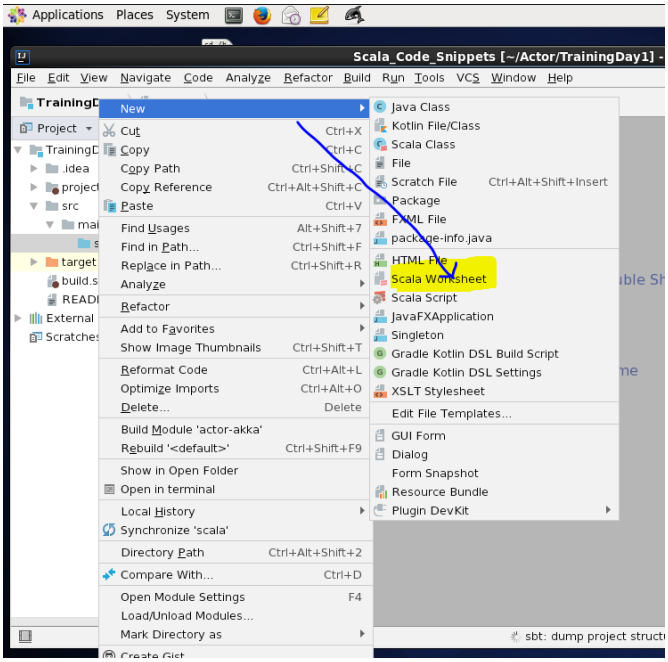
To create a worksheet in IntelliJ, Navigate to /home/acadgild/eclipse and click on the eclipse Icon.



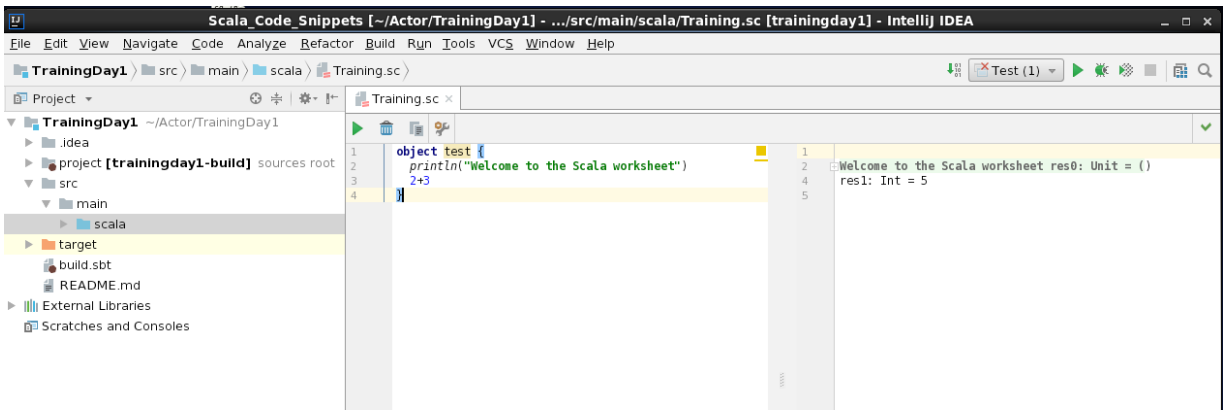
After IntelliJ opens, You will see a project already created for you as TrainingDay1



Create new Scala worksheet as below with Name as Training.



You will observe Training.sc gets created. Go ahead and start writing your code and in the right side you will see the execution results.



The higher order example in the Scala Worksheet, Paste the content in the worksheet and observe the output.

```
object higherOrderFns {
 println("Welcome to the Scala worksheet")      //> Welcome to the Scala worksheet

 val double = (i:Int) => i * 2                 //> double  : Int => Int = <function1>
 def higherOrder(x:Int, y:Int=>Int) = y(x)      //> higherOrder: (x: Int, y: Int => Int)Int

 higherOrder(6, double)                        //> res0: Int = 12

 val triple = (i:Int) => i * 3                 //> triple  : Int => Int = <function1>
 higherOrder(6, triple)                        //> res1: Int = 18

 def sayHello = (name:String) => {"Hello" + " " + name}
                                               //> sayHello: => String => String
 var message = sayHello("Peggy")               //> message  : String = Hello Peggy

 var y = 5                                     //> y  : Int = 5
 val multiplier = (x:Int)=> x * y              //> multiplier  : Int => Int = <function1>
 multiplier(10)                                //> res2: Int = 50
}
```

Placeholder Syntax for Invocation of Methods/Functions in Higher Order Functions

We will define first the IsSpecialName function,

```
def IsSpecialName(firstName:String, lastName:String):Boolean = {
 firstName == "Donald" && lastName == "Trump"
}
```

Let us define our Higher Order function IsVIP, which has input as IsSpecialName

```
def IsVIP(firstName:String, lastName:String, IsSpecialName:(String,String) => Boolean):Boolean = {
 IsSpecialName (firstName,lastName)
}
```

Invocation of Higher Order function using full name of inner function.

```
IsVIP("Donald","Trump", IsSpecialName)
```

Invocation of Higher Order function using wild card syntax.

```
IsVIP("Donald","Truman",_ == "Donald" && _ == "Trump")
```

Assignment 1

In this challenge your objectives is to create a new project in the Scala IDE, to create a worksheet in this project, write code for a payroll function. The payroll function will calculate the weekly paycheck for an hourly employee. In our example hourly employees receive overtime, which means the first 40 hours are calculated at the regular rate. Anything about 40 hours should be calculated at 1.5 times their hourly rate.
Use an hourly rate of $10.50. Check to see if the employee is hourly or salaried before calculating their paycheck. If they're salaried, just print a message. Make sure you test your program with several different types of employees. Maybe an hourly employee who worked 40 hours, another hourly employee who worked 45 hours, an hourly employee who only worked 25 hours, and finally make sure you test what happens if you have a salaried employee.

Assignment 2: Implement the different sorting logic

```
object HofSort {

 def sort(cmp: (String, String)=>Boolean, list: List[String]):List[String] ={
  list.sortWith(cmp)
 }

 def comparator(a: String, b: String): Boolean ={
  if(a < b) true
  else false
 }
 def sortAlphabetically(list: List[String]):List[String]= {
  ??? //TODO use the sort method with a function to comparate elements
 }

 def sortReverseAlphabetically(list: List[String]):List[String]={
  ??? //TODO use the sort method with a function to comparate elements
 }

 def sortIncreasingSize(list: List[String]):List[String]={
  ??? //TODO use the sort method with a function to comparate elements
 }

}
```

Exercise 12A: Using Partially Applied Functions

When applying a function, when we do not to pass in all the arguments defined by the function. But provides only for some of them, leaving remaining parameters.

Consider the example below, where we define a method to calculate product price after discount. The method takes in two parameters — the first is the discount to be applied, and the second is product price.

def calculateProductPrice(discount: Double, productPrice: Double): Double =
(1 - discount/100) * productPrice

We cannot ask shopkeepers to provide discounts every time. We will set the discount once for all the products.

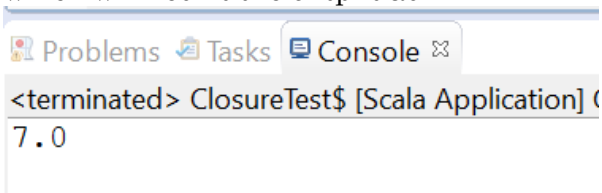
```
object ClosureTest extends App {

 def calculateProductPrice(discount: Double, productPrice: Double): Double =
  (1 - discount / 100) * productPrice

  val dicountApplied = calculateProductPrice(30, _: Double)
  println(dicountApplied.toString())
}
```

The function calculateProductPrice did not return any value but a function. calculateProductPrice partially applied function since we did not provide all the values required.

Later, you can call the method and pass the discount whenever needed, as below

```
dicountApplied(10.0)
```

which will result the output as

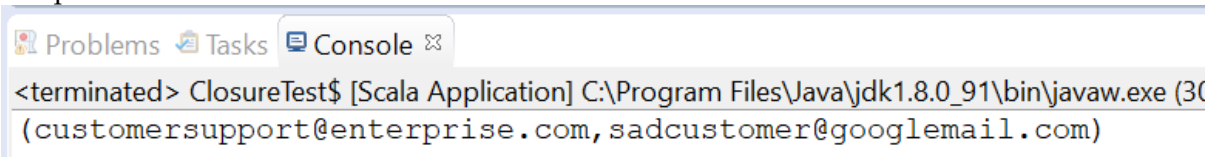


Example 2: Consider an application that send email to customer support customersupport@enterprise.com. We may have a function that looks like

```
object PartialMethodTest extends App {

def email(to:String, from:String) = (println(to,from))

 def emailCustomerSupport = email("customersupport@enterprise.com", _:String)

 emailCustomerSupport("sadcustomer@googlemail.com")
}
```

So instead of customer knowing the email for customer support or entering it (assuming the email for support would not change), we can have a partially applied function that looks like above.

Output:



Assignment:
Create a greeting method that returns an appropriate greeting based on the language specified: English and Spanish along with the message. Check the user if it has name starts with English send him greeting using English language and else to remaining send using Spanish greeting. Use Partially applied function to achieve it.

Exercise 13: Currying

Currying – transformation of a function with multiple arguments into a chain of single-argument functions, partially applied function – function which was called with less number of arguments.

Let's start with a partially applied function and then curry it for demonstration purposes in Scala REPL

Let's demonstrate this with illustrating different uses of a "factor function":

```
def factorOperation(a: Int, b: Int) = b%a == 0
```

To shortcut the use of the function without re-using any parameters, you can just use the wildcard:

```
val someFactor = factorOperation_
val a = someFactor(8,110)
```

But what if we wanted to retain the value of the first parameter? let's say I want to retain the number 5 and find if "factorOperations" second parameter is a multiple of the number 5. Here multipleOfA is a **partially applied function** because it has some, though not all, the parameters for factorOperation..

```
val multipleOfA = factorOperation(5, _: Int)
val b = multipleOfA(80)
```

Let us curry the function to make it cleaner.

```
def factorOperation (a: Int)(b: Int) = b % a == 0
```

... and the impact downstream would be:

```
val multipleOfA = factorOperation(5)_
val c = multipleOfA(80)
```

So when curried, the function type takes that of a chained function.

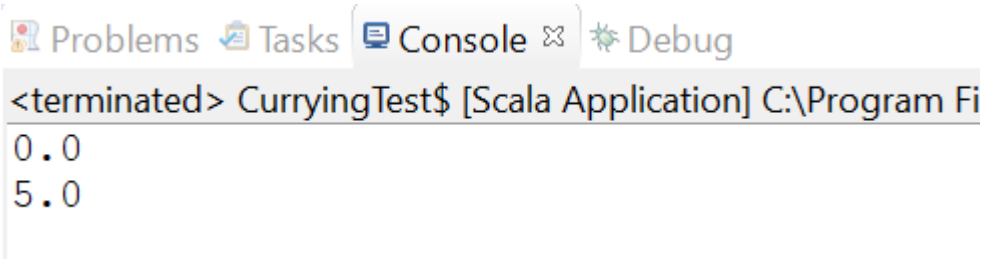
```
scala> def factorOperation (a: Int)(b: Int) = b % a == 0
factorOperation: (a: Int)(b: Int)Boolean

scala> val multipleOfA = factorOperation(5)_
multipleOfA: Int => Boolean = <function1>

scala> val c = multipleOfA(80)
c: Boolean = true

scala>
```

Note: Remember that the output will not change.

Example 2: Look at the following function definition that takes two parameters. Try this in Eclipse creating a new Object:

```
trait TaxStrategy {
  def taxIt(product: String): Double
}

object CurryingTest extends App {
  val taxIt: (TaxStrategy, String) => Double = (s, p) => s.taxIt(p)

  class TaxFree extends TaxStrategy {
    override def taxIt(product: String) = 0.0
  }

  val taxFree = taxIt.curried(new TaxFree)
  taxFree("someProduct")
  println(taxFree("someProduct"))

  class TaxCharged extends TaxStrategy {
    override def taxIt(product: String) = 5.0
  }

  val taxCharged = taxIt.curried(new TaxCharged)
  taxCharged("someProductTaxed")
  println(taxCharged("someProductTaxed"))

}
```

Output:



Explanation:

```
trait TaxStrategy
{
def taxIt(product: String): Double
}
val taxIt: (TaxStrategy, String) => Double = (s, p) => s.taxIt(p)
```

The taxIt function takes TaxStrategy and String parameters and returns Double value. To turn this function into a curried function, you can invoke the built-in curried method defined for function types:

```
taxIt.curried
```

It turned the taxIt function into a function that takes one parameter and returns another function that takes the second parameter

```
class TaxFree extends TaxStrategy { override def taxIt(product:
    String) = 0.0 }

val taxFree = taxIt.curried(new TaxFree)
taxFree("someProduct")

scala> taxIt.curried
res2: TaxStrategy => String => Double = <function1>
```

It allows you to turn generalized functions into specialized ones. For example, turning the taxIt function to function currying allows you to create a more specialized function like taxFree.

You can also turn existing methods to curried functions using an underscore (_). The following code example redefines the taxIt function as a method and then converts it to a curried function.

Assignment 1:

Implement the multiply and multiplyByTwo.

```
object CurryingObject {

 def multiply(x : Int, y: Int):Int= ??? //TODO define a function who multiply x by y

 def multiplyByTwo(x:Int): Int = ??? //TODO define a function who multiply x by y

}
```

Assignment 2: Implement the area computation

```
object Area {


 def area(x : Double, y:Double):Double =(x, y) match { // it's pattern matching
    // you can use it to define some differents implementations
    // as a function of the parameters
  case (_, math.Pi) => x*x*math.Pi // for example here you can define a specific implementation for circles
  case (_, _) if x>0 && y>0 => y*x //and another one for rectangle
 }


 def circleArea(radius: Double): Double= ??? //TODO Compute the circle area using the existing method

 def squareArea(side: Double): Double= ??? //TODO Compute the square area using the existing method


 //let's create circleArea and squareArea from the area function
}
```

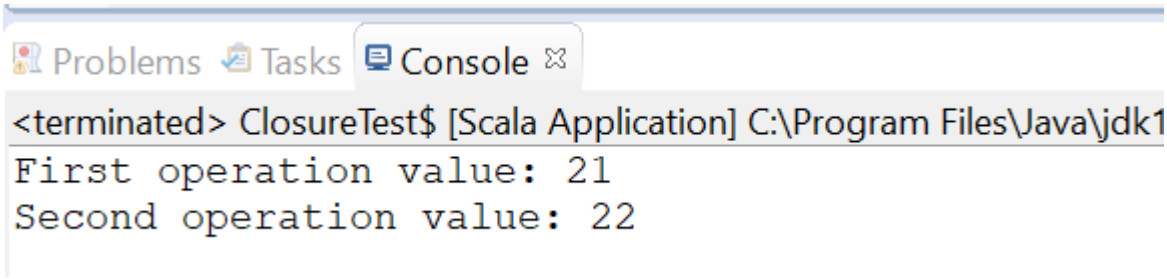
Exercise 14: Closures

It is a function which return value is depends on the one or more variables values which are declared outside this function.

In this Object ClosureTest, i is a formal parameter whereas value is not a formal parameter. So sum value is depends on the outside variable value.

```
object ClosureTest {

 def main(args: Array[String]) {

   println("First operation value: " + sum(1))

   println("Second operation value: " + sum(2))

 }

 var value = 20

 val sum = (i: Int) => i + value

}
```

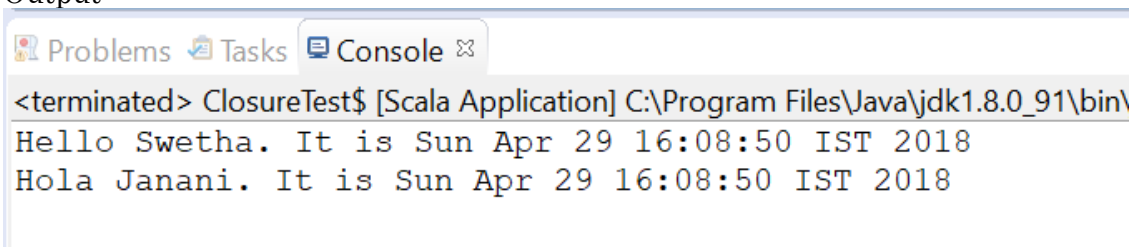
Output:



Let us define another Object, which will define a greeting method, and it will send the greeting based on the user name passed as the argument, and the time which is declared outside the function.

```
import java.util.Calendar
object ClosureTest {

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

Output:



Assignment:

1. Create a simple function named isOfVotingAge that validates age has to be greater than 18, instead of hardcoding the value 18 into the function, let the function refer to the variable votingAge that's in scope when you define the function. Now call the function and pass different values, and should return as below

```
isOfVotingAge(16)  // false
isOfVotingAge(20)  // true
```

2. define a method “buyStuff” that accepts a function with addToBasket’s signature and a String as the fruit name:

   Hint: def buyStuff(f: String => Unit, s: String) {
     f(s)
   }

Now define addToBasket function which will accept input as String which will be the fruit to be added in the collection of Fruits defined outside the function but in the scope, as below. The string passed from buyStuff will be passed as the input argument of the addToBasket function.

Hint: import scala.collection.mutable.ArrayBuffer
val fruits = ArrayBuffer("apple")

Exercise 14A: Collections

There are three main categories of collection classes to choose from:

List

Sequence

Map

Set

A sequence is a linear collection of elements and may be indexed or linear (a linked list). A map contains a collection of key/value pairs, like a Java Map, Ruby Hash, or Python dictionary. A set is a collection that contains no duplicate elements.

1. List

The elements of the list have same data type. Lists are similar to arrays with two differences that is Lists are immutable and list represents a linked list whereas arrays are flat.

A list can be created as

val StudentNames[String] = List("Rohan", "Andreas", "Rob", "John")

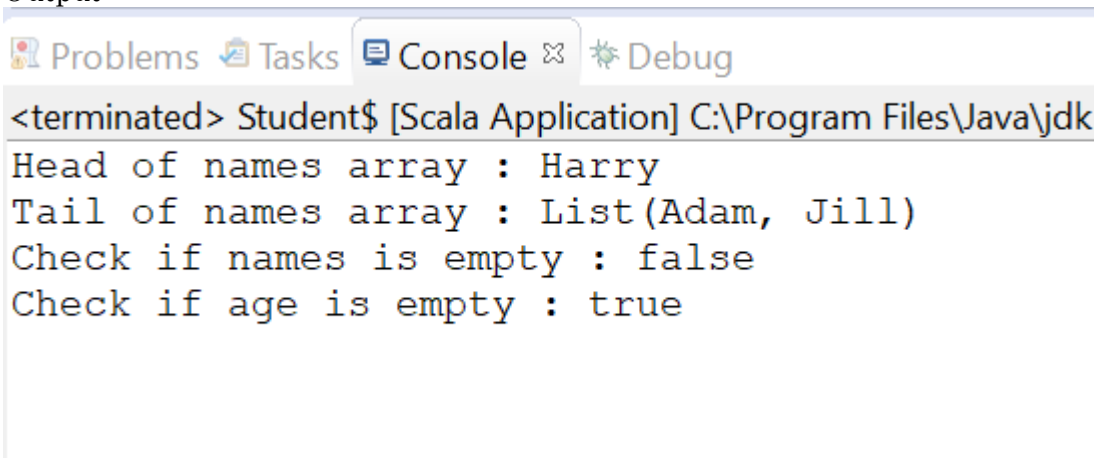
Empty list can be created as

val em: List[Nothing] = List()

Example1: Let us create an Object Student with names and Age List, and we will do some operations on the list.

```
object Student {
def main(args:Array[String]) {
   val names= "Harry" :: ("Adam" :: ("Jill" :: Nil))
   val age = Nil
     println( "Head of names array : " + names.head )
     println( "Tail of names array : " + names.tail )
     println( "Check if names is empty : " + names.isEmpty )
     println( "Check if age is empty : " + age.isEmpty )
  }
}
```

Output:

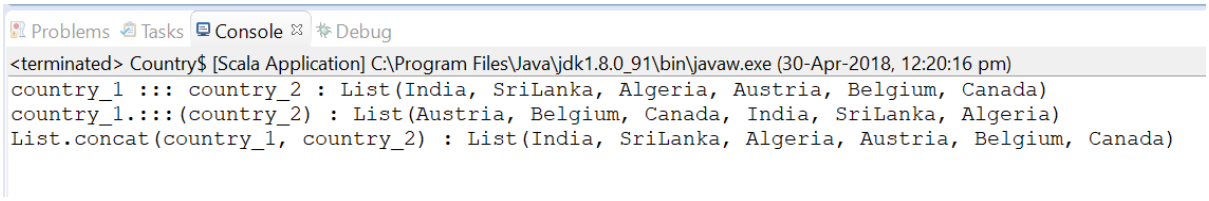


Example 2: Concatenating lists

```
object Country {
            def main(args:Array[String]) {
            val country_1 =  List("India","SriLanka","Algeria")
            val country_2 = List("Austria","Belgium","Canada")

            val country = country_1 ::: country_2
            println( "country_1 ::: country_2 : " + country )

            val cont = country_1.:::(country_2)
        println( "country_1.:::(country_2) : " + cont )
            val con = List.concat(country_1, country_2)
        println( "List.concat(country_1, country_2) : " + con  )

  }
```

```
}
```

Output:



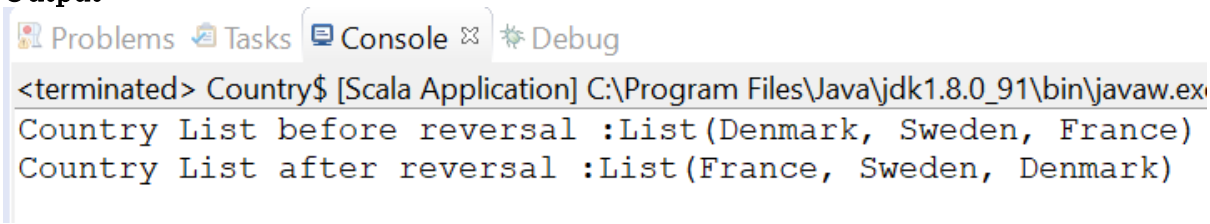
We have created two lists country_1 and country_2. We are concatenating two lists country_1 and country_2 into a single country list using ::: operator. In the second case we are using .::: operator to concat country_1 and country_2 into "cont" list and the second list country_2 data will be first inserted and then country_1 list data is displayed.

Finally we use concat method to concatenate country_1 and country_2 into the list "con"

Example 3: Reverse List Order

```
object Country {
        def main(args:Array[String]) {
        val country = List("Denmark","Sweden","France")
        println("Country List before reversal :" + country)
        println("Country List after reversal :" + country.reverse)
}
}
```

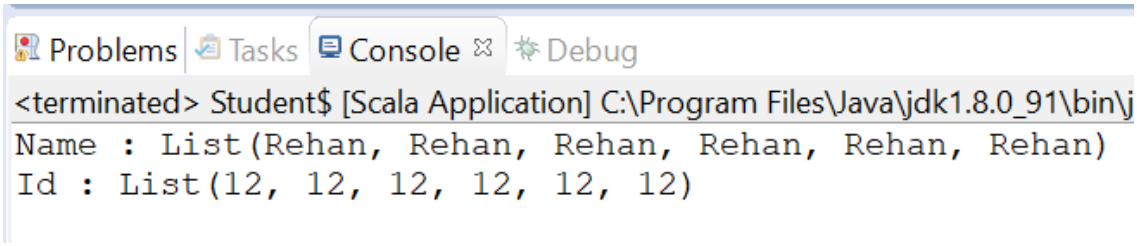
Output:



Example 4: Creating Uniform Lists

```
object Student {
  def main(args: Array[String]) {
    val name = List.fill(6)("Rehan")
    println( "Name : " + name  )

    val id = List.fill(6)(12)
    println( "Id : " + id  )
  }
}
```

Output:



Example 5: Scala REPL with List

Creating a List

```
scala> val weekDays = List("Mon","Tue","Wed","Thu","Fri")
```

weekDays: List[String] = List(Mon, Tue, Wed, Thu, Fri)

```
scala> weekDays.head
```

res6: String = Mon

```
scala> weekDays.tail
```

res7: List[String] = List(Tue, Wed, Thu, Fri)

```
scala> weekDays.size
```

res8: Int = 5

```
scala> weekDays.reverse
```

res9: List[String] = List(Fri, Thu, Wed, Tue, Mon)

```
scala> weekDays.reverse
```

res10: List[String] = List(Fri, Thu, Wed, Tue, Mon)

```
scala> weekDays(1)
```

res11: String = Tue

```
scala> weekDays.contains("Mon")
```

res12: Boolean = true

```
scala> weekDays contains "Mon"
```

res13: Boolean = true

```
scala> for (c <- weekDays) println(c)
```

Mon
Tue
Wed
Thu
Fri

scala>

```
scala> weekDays.size
```

res15: Int = 5

```
scala> var restOfWeek = weekDays
```
restOfWeek: List[String] = List(Mon, Tue, Wed, Thu, Fri)

```
scala> while (! restOfWeek.isEmpty)
     | {
     |   println(s"Grr..today is ${restOfWeek.head}, ${restOfWeek.size} days left for the weekend")
     |   restOfWeek = restOfWeek.tail
     | }
```
Grr..today is Mon, 5 days left for the weekend
Grr..today is Tue, 4 days left for the weekend
Grr..today is Wed, 3 days left for the weekend
Grr..today is Thu, 2 days left for the weekend
Grr..today is Fri, 1 days left for the weekend

```
scala> var restOfWeek = weekDays
```
restOfWeek: List[String] = List(Mon, Tue, Wed, Thu, Fri)

```
scala> while (restOfWeek != Nil)
     | {
     |   println(s"Grr..today is ${restOfWeek.head}, ${restOfWeek.size} days left for the weekend")
     |   restOfWeek = restOfWeek.tail
     | }
```
Grr..today is Mon, 5 days left for the weekend
Grr..today is Tue, 4 days left for the weekend
Grr..today is Wed, 3 days left for the weekend
Grr..today is Thu, 2 days left for the weekend
Grr..today is Fri, 1 days left for the weekend

```
scala> val weekEnds = List("Sat","Sun")
```
weekEnds: List[String] = List(Sat, Sun)

```
scala> val weeklyHolidays = List("Sat","Sun")
```
weeklyHolidays: List[String] = List(Sat, Sun)

```
scala> weekEnds == weeklyHolidays
```
res18: Boolean = true

```
scala> weekDays.distinct
```
res19: List[String] = List(Mon, Tue, Wed, Thu, Fri)

Example 6: Scala REPL with List – Higher Order Functions

Creating a List in Scala REPL

```
scala> val weekDays = List("Mon","Tue","Wed","Thu","Fri")
```
weekDays: List[String] = List(Mon, Tue, Wed, Thu, Fri)

Prints the content of the list

```
scala> val printValue = (x:Any) => {println(x)}:Unit
printValue: Any => Unit = <function1>
```

Map function:-

```
scala> weekDays.map(_ == "Mon")
res22: List[Boolean] = List(true, false, false, false, false)
```

Declaring a function

```
scala> val IsManicMonday = (x:String) => {x == "Mon"}:Boolean
IsManicMonday: String => Boolean = <function1>
```

Mapping the function as defined with every element of the list

```
scala> weekDays.map(IsManicMonday)
res23: List[Boolean] = List(true, false, false, false, false)
```

Filter the contents of the List.

```
scala> weekDays.filter(IsManicMonday)
res24: List[String] = List(Mon)
```

Map with Anonymous function.

```
scala> weekDays.map(_ != "Mon")
res25: List[Boolean] = List(false, true, true, true, true)
```

Sort by function with the first element of the list element.

```
scala> weekDays.sortBy(_(0))
res26: List[String] = List(Fri, Mon, Tue, Thu, Wed)
```

Creating a List.

```
scala> val someNumbers = List(10,20,30,40,50,60)
someNumbers: List[Int] = List(10, 20, 30, 40, 50, 60)
```

```
scala> someNumbers.scanRight(0)(_ - _)
res27: List[Int] = List(-30, 40, -20, 50, -10, 60, 0)
```

```
scala> someNumbers.scanLeft(0)(_ - _)
res28: List[Int] = List(0, -10, -30, -60, -100, -150, -210)
```

```
scala> someNumbers.scan(0)(_ - _)
res29: List[Int] = List(0, -10, -30, -60, -100, -150, -210)
```

```
scala> someNumbers.foldRight(0)(_ - _)
res30: Int = -30
```

```
scala> someNumbers.foldLeft(0)(_ - _)
res31: Int = -210
```

```
scala> someNumbers.fold(0)(_ - _)
res32: Int = -210
```

```
scala> someNumbers.reduceRight(_ - _)
```
res33: Int = -30

```
scala> someNumbers.reduceLeft(_ - _)
```
res34: Int = -190

```
scala> someNumbers.reduce(_ - _)
```
res35: Int = -190

2. Set

Set is a collection of unique elements of the same type. Unlike list Set does not contain duplicate elements. Sets can be mutable or immutable. When the object is immutable the object itself cannot be changed.

val country = Set("Russia", "Denmark", "Sweden")

Or, Set of Integer data type can be created as

var id : Set[Int] = Set(4,5,6,7,8,9)

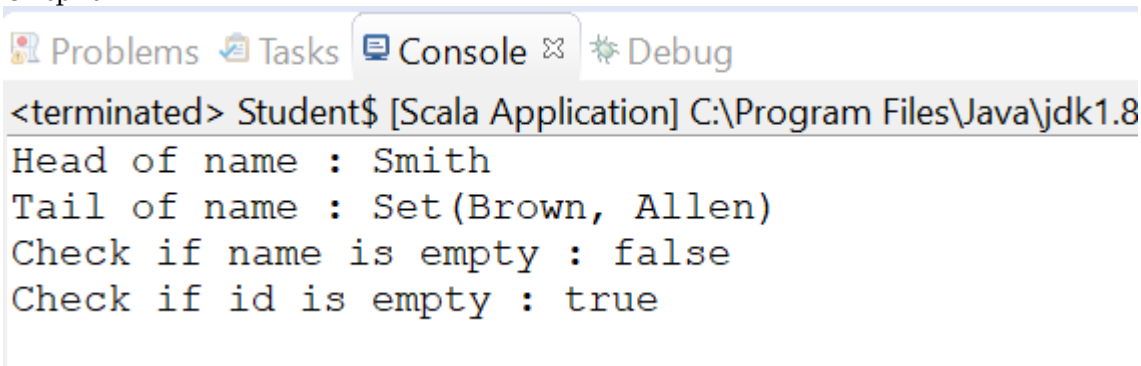
Empty Set can be created as;

var age = Set()

Example 1. Let us create student object with two sets "names" and "id". We are doing head, tail and empty operations on these arrays and printing the result.

```
object Student {
  def main(args: Array[String]) {
    val name = Set("Smith", "Brown", "Allen")
    val id: Set[Int] = Set()

    println( "Head of name : " + name.head )
    println( "Tail of name : " + name.tail )
    println( "Check if name is empty : " + name.isEmpty )
    println( "Check if id is empty : " + id.isEmpty )
  }
}
```

Output:



Example 2: Concatenating Sets- We are creating two sets furniture_1 and furniture_2 and concatenating these two sets into furniture and furn using ++ and Set.++ operators.

```
object Furniture {
  def main(args: Array[String]) {
    val furniture_1 = Set("Sofa", "Table", "chair")
    val furniture_2 = Set("Bed", "Door")

     var furniture = furniture_1 ++ furniture_2
    println( "furniture_1 ++ furniture_2 : " + furniture )

    var furn = furniture_1.++(furniture_2)
    println( "furniture_1.++(furniture_2) : " + furn )
  }
}
```

Output

```
Problems  Tasks  Console  Debug
<terminated> Furniture$ [Scala Application] C:\Program Files\Java\jdk1.8.0_91\bin\javaw.exe (30-Apr-
furniture_1 ++ furniture_2 : Set(Door, Sofa, Bed, Table, chair)
furniture_1.++(furniture_2) : Set(Door, Sofa, Bed, Table, chair)
```

Example 3: Common Values in a Set- We are creating two sets of numbers n1 and n2 and finding common elements present in both the sets.

```
object Numbers {
  def main(args: Array[String]) {
    val n1 = Set(11,45,67,78,89,86,90)
    val n2 = Set(10,20,45,67,34,78,98,89)

    println( "n1.&(n2) : " + n1.&(n2) )
    println( "n1.intersect(n2) : " + n1.intersect(n2) )
  }
}
```

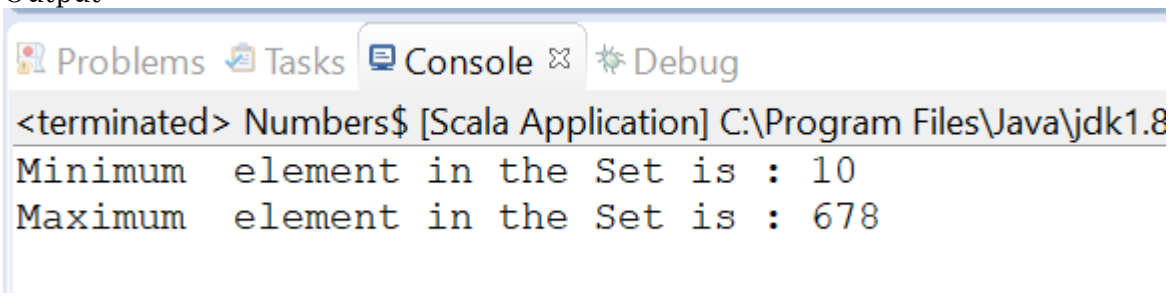
Output:

```
Problems  Tasks  Console  Debug
<terminated> Numbers$ [Scala Application] C:\Program Files\
n1.&(n2) : Set(78, 89, 45, 67)
n1.intersect(n2) : Set(78, 89, 45, 67)
```

Example 4: Maximum and Minimum elements in a Set- We are finding the minimum and maximum numbers in set “num1” using the min and max methods.

```
object Numbers {
  def main(args: Array[String]) {
    val num1 = Set(125,45,678,34,20,322,10)

    println( "Minimum  element in the Set is : " + num1.min )
    println( "Maximum  element in the Set is : " + num1.max )
  }
}
```

Output



3. Map

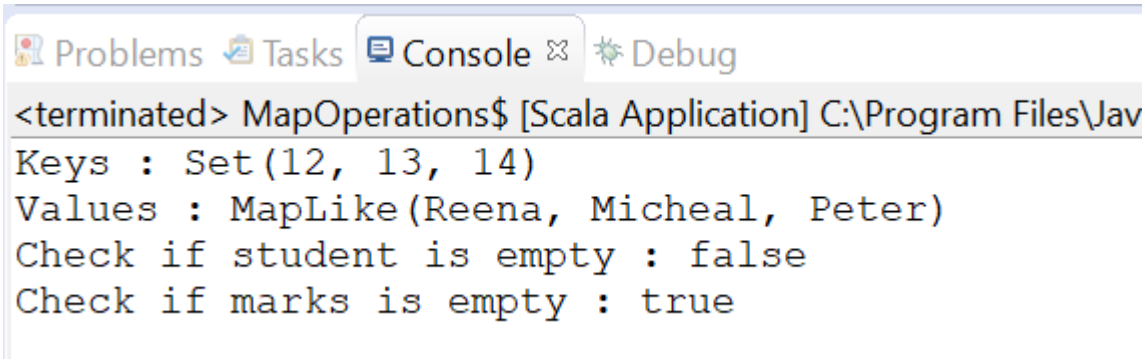
The immutable Map class is in scope by default, so you can create an immutable map without an import.

Consider an example of map with key as student ids and student names as the value.
val student = Map(12 -> "Reena", 13 -> "Micheal" , 14 -> "Peter")
The map created above is immutable, to create mutable Maps, we have to import the package
var states = scala.collection.mutable.Map("AL" -> "Alabama")

Example 1: Basic Operations on Map

```
object MapOperations {
 def main(args: Array[String]) {
  val student = Map(12 -> "Reena", 13 -> "Micheal", 14 -> "Peter")

  val marks: Map[String, Int] = Map()

  println("Keys : " + student.keys)
  println("Values : " + student.values)
  println("Check if student is empty : " + student.isEmpty)
  println("Check if marks is empty : " + marks.isEmpty)
 }
}
```

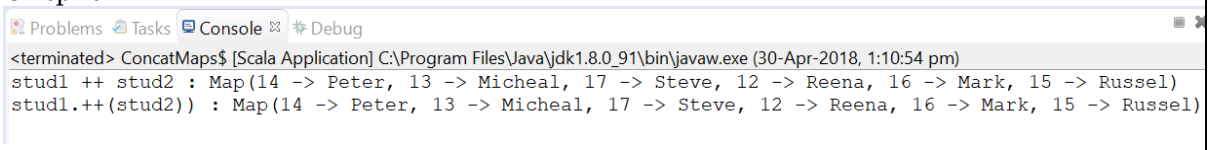
Output:



Example 2: Concatenating Maps

```
object ConcatMaps {
  def main(args: Array[String]) {
            val stud1 = Map(12 -> "Reena", 13 -> "Micheal" , 14 -> "Peter")
            val stud2 = Map(15 -> "Russel", 16 -> "Mark" , 17 -> "Steve")

            var student = stud1 ++ stud2
            println( "stud1 ++ stud2 : " + student )

            val stu = stud1.++(stud2)
            println( "stud1.++(stud2)) : " + stu )

  }
}
```

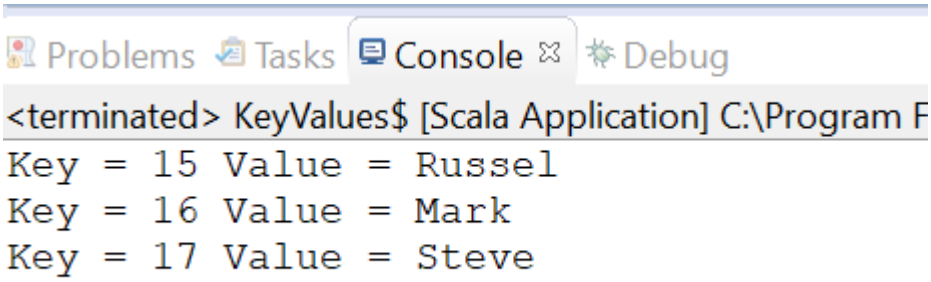
Output:



Example 3: Print keys and values from a Map
Foreach loop is used to iterate through all the key and value pairs in a map.

```
object KeyValues {
  def main(args: Array[String]) {
            val stud2 = Map(15 -> "Russel", 16 -> "Mark" , 17 -> "Steve")


            stud2.keys.foreach{ i =>
                        print( "Key = " + i )
                        println(" Value = " + stud2(i) )}
  }
}
```

Output:

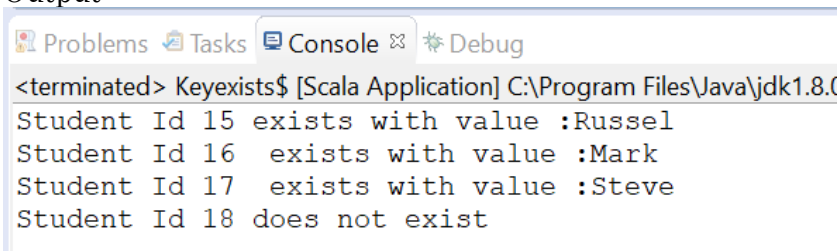


Example 4: Check for a key in a Map

The Map.contains method to test if a given key exists in the map or not.

```
object Keyexists {
  def main(args: Array[String]) {
            val stud2 = Map(15 -> "Russel", 16 -> "Mark" , 17 -> "Steve")

            if( stud2.contains( 15 )){
            println("Student Id 15 exists with value :"  + stud2(15))
            }else{
            println("Student Id with 15 does not exist")
            }
            if( stud2.contains( 16 )){
            println("Student Id 16  exists with value :"  + stud2(16))
            }else{
            println("Student Id 16 does not exist")
            }
            if( stud2.contains( 17 )){
            println("Student Id 17  exists with value :"  + stud2(17))
            }else{
            println("Student Id 17 does not exist")
            }
            if( stud2.contains( 18 )){
            println("Student Id 18  exists with value :"  + stud2(18))
            }else{
            println("Student Id 18 does not exist")
            }

  }
}
```

Output:



4. Immutable to Mutable Collection

Let us create a List

```
val someNumbers = collection.immutable.List(10,20,30,40,50,60)
```

Let us create a Map

```
val stateCodes = collection.immutable.Map("California" -> "CA",("Vermont","VT"))
```

Let us create Set

```
val stateSet = collection.immutable.Set("California","Vermont")
```

Let us create a Mutable List

```
val someNumbers = collection.mutable.Buffer(10,20,30,40,50)
```

Let us create a Mutable Map

```
val stateCodes = collection.mutable.Map("California" -> "CA",("Vermont","VT"))
```

Let us create a Mutable Set.

```
val stateSet = collection.mutable.Set("California","Vermont")
```

Conversion of a Immutable List to Mutable Buffer i.e. List.

```
val listBuilder = List.newBuilder[Int]
someNumbers.foreach(listBuilder+=_)
val reconstructedList = listBuilder.result
```

Output:

```
scala> val someNumbers = collection.immutable.List(10,20,30,40,50,60)
someNumbers: List[Int] = List(10, 20, 30, 40, 50, 60)

scala> val stateCodes = collection.immutable.Map("California" -> "CA",("Vermont","VT"))
stateCodes: scala.collection.immutable.Map[String,String] = Map(California -> CA, Vermont -> VT)

scala> val stateSet = collection.immutable.Set("California","Vermont")
stateSet: scala.collection.immutable.Set[String] = Set(California, Vermont)

scala>

scala> val someNumbers = collection.mutable.Buffer(10,20,30,40,50)
someNumbers: scala.collection.mutable.Buffer[Int] = ArrayBuffer(10, 20, 30, 40, 50)

scala> val stateCodes = collection.mutable.Map("California" -> "CA",("Vermont","VT"))
stateCodes: scala.collection.mutable.Map[String,String] = Map(California -> CA, Vermont -> VT)

scala> val stateSet = collection.mutable.Set("California","Vermont")
stateSet: scala.collection.mutable.Set[String] = Set(Vermont, California)

scala>

scala> val listBuilder = List.newBuilder[Int]
listBuilder: scala.collection.mutable.Builder[Int,List[Int]] = ListBuffer()

scala> someNumbers.foreach(listBuilder+=_)

scala> val reconstructedList = listBuilder.result
reconstructedList: List[Int] = List(10, 20, 30, 40, 50)
```

5. Option Collection

```
scala> def fraction(numer:Double,denom:Double): Option[Double] = {
    | if (denom == 0)
    |   None
    | else
    |   Option(numer/denom)
```

```
  | }
```

fraction: (numer: Double, denom: Double)Option[Double]

```
scala> val piOption = fraction(22,7)
```

piOption: Option[Double] = Some(3.142857142857143)

6. Higher Order Functions on Collections

To understand strict and lazy collections, it helps to first understand the concept of a transformer method. A transformer method is a method that constructs a new collection from an existing collection. This includes methods like map, filter, reverse, etc.—any method that transforms the input collection to a new output collection. They are also called as Functional Combinators.

Try the below scripts in the **Scala REPL**.

1. Map
   Evaluates a function over each element in the list, returning a list with the same number of elements.

```
scala> val numbers = List(1, 2, 3, 4)
```

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

```
scala> numbers.map((i: Int) => i * 2)
```

res0: List[Int] = List(2, 4, 6, 8)
or pass in a function (the Scala compiler automatically converts our method to a function)

```
scala> def timesTwo(i: Int): Int = i * 2
```

timesTwo: (i: Int)Int

```
scala> numbers.map(timesTwo)
```

res0: List[Int] = List(2, 4, 6, 8)

2. Foreach
   foreach is like map but returns nothing. foreach is intended for side-effects only.

```
scala> numbers.foreach((i: Int) => i * 2)
```

returns nothing.

You can try to store the return in a value but it'll be of type Unit (i.e. void)

```
scala> val doubled = numbers.foreach((i: Int) => i * 2)
```

doubled: Unit = ()

3. Filter
   Removes any elements where the function you pass in evaluates to false. Functions that return a Boolean are often called predicate functions.

```
scala> numbers.filter((i: Int) => i % 2 == 0)
```

res0: List[Int] = List(2, 4)

```
scala> def isEven(i: Int): Boolean = i % 2 == 0
```

isEven: (i: Int)Boolean

```
scala> numbers.filter(isEven)
```

res2: List[Int] = List(2, 4)

4. Zip
   zip aggregates the contents of two lists into a single list of pairs.

```
scala> List(1, 2, 3).zip(List("a", "b", "c"))
```

   res0: List[(Int, String)] = List((1,a), (2,b), (3,c))

5. Partition
   partition splits a list based on where it falls with respect to a predicate function.

```
scala> val numbers = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
scala> numbers.partition(_ % 2 == 0)
```

   res0: (List[Int], List[Int]) = (List(2, 4, 6, 8, 10),List(1, 3, 5, 7, 9))

6. Find
   find returns the first element of a collection that matches a predicate function.

```
scala> numbers.find((i: Int) => i > 5)
```

   res0: Option[Int] = Some(6)

7. Drop and dropwhile

   drop drops the first i elements

```
scala> numbers.drop(5)
```

   res0: List[Int] = List(6, 7, 8, 9, 10)
   dropWhile removes the first element that match a predicate function. For example, if we dropWhile odd numbers from our list of numbers, 1 gets dropped (but not 3 which is “shielded” by 2).

```
scala> numbers.dropWhile(_ % 2 != 0)
```

   res0: List[Int] = List(2, 3, 4, 5, 6, 7, 8, 9, 10)

8. foldLeft

```
scala> numbers.foldLeft(0)((m: Int, n: Int) => m + n)
```

   res0: Int = 55
   0 is the starting value (Remember that numbers is a List[Int]), and m acts as an accumulator.

   Seen visually:

```
scala> numbers.foldLeft(0) { (m: Int, n: Int) => println("m: " + m + " n: " + n); m + n }
```

m: 0 n: 1
m: 1 n: 2
m: 3 n: 3
m: 6 n: 4
m: 10 n: 5
m: 15 n: 6
m: 21 n: 7
m: 28 n: 8
m: 36 n: 9
m: 45 n: 10
res0: Int = 55

9. foldRight
   Is the same as foldLeft except it runs in the opposite direction.

```
scala> numbers.foldRight(0) { (m: Int, n: Int) => println("m: " + m + " n: " + n); m + n }
```

m: 10 n: 0
m: 9 n: 10
m: 8 n: 19
m: 7 n: 27
m: 6 n: 34
m: 5 n: 40
m: 4 n: 45
m: 3 n: 49
m: 2 n: 52
m: 1 n: 54
res0: Int = 55
flatten
flatten collapses one level of nested structure.

```
scala> List(List(1, 2), List(3, 4)).flatten
```

res0: List[Int] = List(1, 2, 3, 4)

10. flatMap
    flatMap is a frequently used combinator that combines mapping and flattening. flatMap takes a function that works on the nested lists and then concatenates the results back together.

```
scala> val nestedNumbers = List(List(1, 2), List(3, 4))
```

nestedNumbers: List[List[Int]] = List(List(1, 2), List(3, 4))

```
scala> nestedNumbers.flatMap(x => x.map(_ * 2))
```

res0: List[Int] = List(2, 4, 6, 8)
Think of it as short-hand for mapping and then flattening:

```
scala> nestedNumbers.map((x: List[Int]) => x.map(_ * 2)).flatten
```

res1: List[Int] = List(2, 4, 6, 8)

The example calling map and then flatten is an example of the “combinator”-like nature of these functions.

11. All of the functional combinators shown work on Maps, too. Maps can be thought of as a list of pairs so the functions you write work on a pair of the keys and values in the Map.

```
scala> val extensions = Map("steve" -> 100, "bob" -> 101, "joe" -> 201)
```

extensions: scala.collection.immutable.Map[String,Int] = Map((steve,100), (bob,101), (joe,201))
Now filter out every entry whose phone extension is lower than 200.

```
scala> extensions.filter((namePhone: (String, Int)) => namePhone._2 < 200)
```

res0: scala.collection.immutable.Map[String,Int] = Map((steve,100), (bob,101))
Because it gives you a tuple, you have to pull out the keys and values with their positional accessors. Yuck!

Lucky us, we can actually use a pattern match to extract the key and value nicely.

```
scala> extensions.filter({case (name, extension) => extension < 200})
```

res0: scala.collection.immutable.Map[String,Int] = Map((steve,100), (bob,101))

12. // filter the items, then double them

```
scala> val nums = List(1,2,3,4,5).filter(_ < 4).map(_ * 2)
```

nums: List[Int] = List(2, 4, 6)

Assignment:

1.) Can you figure out what method you can use to find out if the list: List(1,2,3,4,5) contains the number 3?

2.) How can you add all the elements of the previous list?

3.) Create an Array of all the odd numbers from 0 to 15

4.) What are the unique elements in the list: List(2,3,1,4,5,6,6,1,2)?

5.) Create a mutable map that maps together Names to Ages.
It should have the following key value pairs:
Sammy, 3
Frankie, 7
John, 45

6.) Write a Scala program to get the largest element of an array using foldLeft.
Implement the factorial function using to and foldLeft, without a loop or recursion.
7.) Write a Scala code which reverses the lines of a file (makes the first line as the last one, and so on).
8.) Write a Scala code which reads a file and prints all words with more than 10 characters. Can you write all of it in a single line?

Exercise 15: Classes and Objects

Fields declared in the body of a Scala class are handled in a manner similar to Java; they are assigned when the class is first instantiated.

- **Creating a Primary Constructor**
  You want to create a primary constructor for a class, and you quickly find that the approach is different than Java.

The following class demonstrates constructor parameters, class fields, and statements in the body of a class.

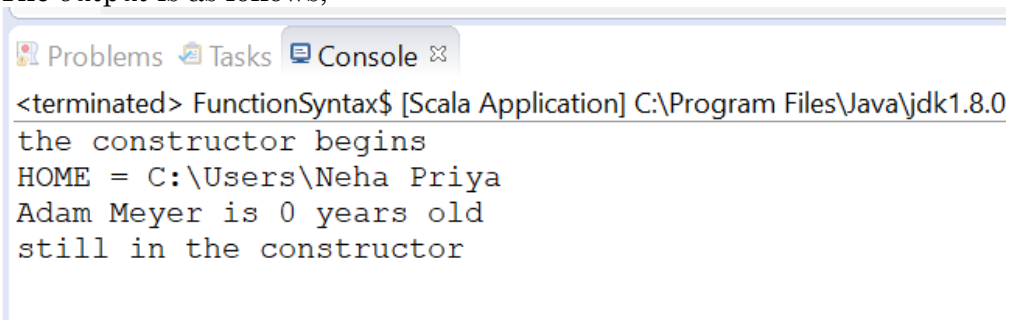
```
object ClassDemo extends App {
 class Person(var firstName: String, var lastName: String) {

  println("the constructor begins")

  // some class fields
  private val HOME = System.getProperty("user.home")
  var age = 0

  // some methods
  override def toString = s"$firstName $lastName is $age years old"
  def printHome { println(s"HOME = $HOME") }
  def printFullName { println(this) } // uses toString

  printHome
  printFullName
  println("still in the constructor")
```

```
 }
 val p = new Person("Adam", "Meyer")
p.firstName = "Scott"
p.lastName = "Jones"

println(p.firstName)
println(p.lastName)
}
```

Because the methods in the body of the class are part of the constructor, when an instance of a Person class is created, you'll see the output from the println statements at the beginning and end of the class declaration, along with the call to the printHome and printFullName methods near the bottom of the class

val p = new Person("Adam", "Meyer")

The output is as follows,



Note: the two constructor arguments firstName and lastName are defined as var fields, which means that they're variable, or mutable; they can be changed after they're initially set. Because the fields are mutable, Scala generates both accessor and mutator methods for them.
You want to control the visibility of fields that are used as constructor parameters in a Scala class.
The visibility of constructor fields in a Scala class is controlled by whether the fields are declared as val, var, without either val or var, and whether private is also added to the fields.

If a field is declared as a var, Scala generates both getter and setter methods for that field.

If the field is a val, Scala generates only a getter method for it.

If a field doesn't have a var or val modifier, Scala gets conservative, and doesn't generate a getter or setter method for the field.

Additionally, var and val fields can be modified with the private keyword, which prevents getters and setters from being generated.

Scala REPL

```
scala> class Person(private var name: String) { def getName {println(name)} }
```

defined class Person

```
scala> val p = new Person("Alvin Alexander")
```

p: Person = Person@3cb7cee4

```
scala> p.name
```

```
<console>:10: error: variable name in class Person cannot be accessed in Person
        p.name
          ^
```

```
scala> p.getName
```

```
Alvin Alexander
```

Assignment:
Create a class Employee(val name: String). Observe what happens when you create an Object of the Person class and try to assign some value to the name field.

Exercise 16: Defining Auxiliary Constructors

Problem
You want to define one or more auxiliary constructors for a class to give consumers of the class different ways to create object instances.

Solution
We can define multiple auxiliary constructors, but they must have different signatures (parameter lists). Also, each constructor must call one of the previously defined constructors.

```
object test extends App{
class Pizza (var crustSize: Int, var crustType: String) {

 // one-arg auxiliary constructor
 def this(crustSize: Int) {
  this(crustSize, Pizza.DEFAULT_CRUST_TYPE)
  println(this)
 }

 // one-arg auxiliary constructor
 def this(crustType: String) {
  this(Pizza.DEFAULT_CRUST_SIZE, crustType)
  println(this)
 }

 // zero-arg auxiliary constructor
 def this() {
  this(14, Pizza.DEFAULT_CRUST_TYPE)
  println(this)
 }

 override def toString = s"A $crustSize inch pizza with a $crustType crust"

}

object Pizza {
 val DEFAULT_CRUST_SIZE = 10
 val DEFAULT_CRUST_TYPE = "DOUBLE THIN"
}
```

```
val p1 = new Pizza(12)
val p2 = new Pizza("THIN")
val p3 = new Pizza
}
```

Given these constructors, the same pizza can be created in the above following ways.
We have created the class as Pizza and the companion object as Pizza. A companion object is simply an object that's defined in the same file as a class, where the object and class have the same name. any method declared in a companion object will appear to be a static method on the object

Note:
Auxiliary constructors are defined by creating methods named "this".
Each auxiliary constructor must begin with a call to a previously defined constructor.
Each constructor must have a different signature.
One constructor calls another constructor with the name this.

Exercise 17: Case Classes

A case class is a special type of class that generates a lot of boilerplate code for you.

Assume that you start with this case class in a file named Person.scala:

```
// initial case class
case class Person (var name: String, var lastName: String)
```

This lets you create a new Person instance without using the new keyword, like this:

```
val p = Person("John", "Smith")
```

When you write this line of code:
val p = Person("John", "Smith")

behind the scenes, the Scala compiler converts it into this:
val p = Person.apply("John", "Smith")

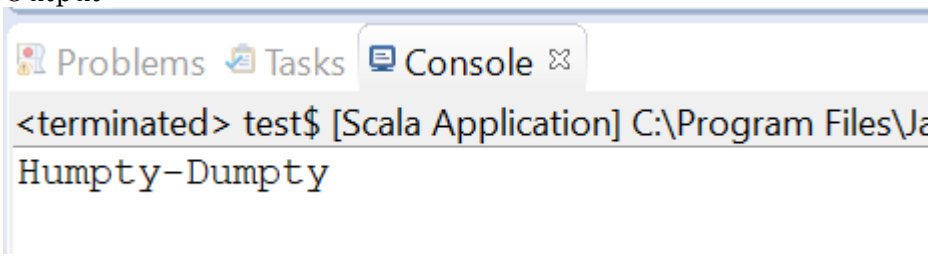
This is a call to an apply method in the companion object of the Person class. You don't see this, you just see the line that you wrote, but this is how the compiler translates your code. As a result, if you want to add new "constructors" to your case class, you write new apply methods. (To be clear, the word "constructor" is used loosely here.)

Eclipse:-
Case classes are mainly used in pattern matching.

```
case class Person(firstName:String, lastName:String)


object test extends App{

val person1 = Person("Vitthal","Srinivasan")
```

```
val person2 = Person("Janani","Ravi")


person1 match {
 case Person("Vitthal","Srinivasan") => "Humpty-Dumpty";println("Humpty-Dumpty")
 case x => x.firstName;println(x)
}

}
```

Output:



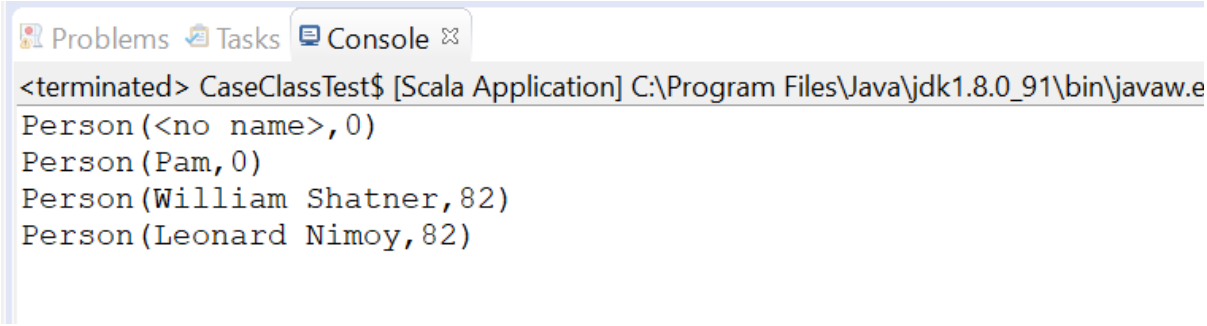
If you decide that you want to add auxiliary constructors to let you create new Person instances (a) without specifying any parameters, and (b) by only specifying their name, the solution is to add apply methods to the companion object of the Person case class in the Person.scala file:

```
// the case class
case class Person (var name: String, var age: Int)

// the companion object
object Person {

 def apply() = new Person("<no name>", 0)
 def apply(name: String) = new Person(name, 0)

}
```

The following test code demonstrates that this works as desired:

```
object CaseClassTest extends App {

 val a = Person()       // corresponds to apply()
 val b = Person("Pam")   // corresponds to apply(name: String)
 val c = Person("William Shatner", 82) //default apply

 println(a)
 println(b)
 println(c)

 // verify the setter methods work
 a.name = "Leonard Nimoy"
 a.age = 82
 println(a)
}
```

Output



Note: Do not use Case Classes if:

Your class carries mutable state.
Your class includes some logic.
Your class is not a data representation and you do not require structural equality.

Generating Boilerplate Code with Case Classes

Defining a class as a case class results in a lot of boilerplate code being generated, with the following benefits:

- An apply method is generated, so you don't need to use the new keyword to create a new instance of the class.
- Accessor methods are generated for the constructor parameters because case class constructor parameters are val by default. Mutator methods are also generated for parameters declared as var.
- A good, default toString method is generated.
- An unapply method is generated, making it easy to use case classes in match expressions.
- equals and hashCode methods are generated.
- A copy method is generated.
- When you define a class as a case class, you don't have to use the new keyword to create a new instance

```
case class Person(name: String, relation: String)

val emily = Person("Emily", "niece")
```

Case class constructor parameters are val by default, so accessor methods are generated for the parameters, but mutator methods are not generated:

`emily.name` will result the output whereas,

```
emily.name = "Fred"
```

<console>:10: error: reassignment to val

emily.name = "Fred"

Let us observe what code gets generated for a case class.

Create a file and paste the below code in the Person.scala.

```
case class Person(var name: String, var age: Int)
```

Open terminal and type scalac, then compile the file:

```
$ scalac Person.scala
```

This creates two class files, Person.class and Person$.class. Disassemble Person.class with this command:

```
$ javap Person
```

```
[acadgild@localhost scala]$ pwd
/home/acadgild/scala
[acadgild@localhost scala]$ source ~/.bash_profile
[acadgild@localhost scala]$ ls
Person~  Person.scala  Person.scala~
[acadgild@localhost scala]$ scalac Person.scala
[acadgild@localhost scala]$ ls
Person~  Person.class  Person$.class  Person.scala  Person.scala~
[acadgild@localhost scala]$ javap Person
Compiled from "Person.scala"
public class Person implements scala.Product,scala.Serializable {
  public static scala.Option<scala.Tuple2<java.lang.String, java.lang.String>> u
napply(Person);
  public static Person apply(java.lang.String, java.lang.String);
  public static scala.Function1<scala.Tuple2<java.lang.String, java.lang.String>
, Person> tupled();
  public static scala.Function1<java.lang.String, scala.Function1<java.lang.Stri
ng, Person>> curried();
  public java.lang.String name();
  public java.lang.String relation();
  public Person copy(java.lang.String, java.lang.String);
  public java.lang.String copy$default$1();
  public java.lang.String copy$default$2();
  public java.lang.String productPrefix();
  public int productArity();
  public java.lang.Object productElement(int);
  public scala.collection.Iterator<java.lang.Object> productIterator();
  public boolean canEqual(java.lang.Object);
  public int hashCode();
  public java.lang.String toString();
  public boolean equals(java.lang.Object);
  public Person(java.lang.String, java.lang.String);
}
[acadgild@localhost scala]$
```

Compiled from "Person.scala"

```
public class Person extends java.lang.Object implements scala.ScalaObject,scala.Product,scala.Serializable{
  public static final scala.Function1 tupled();
  public static final scala.Function1 curry();
  public static final scala.Function1 curried();
```

```
  public scala.collection.Iterator productIterator();
  public scala.collection.Iterator productElements();
  public java.lang.String name();
  public void name_$eq(java.lang.String);
  public int age();
  public void age_$eq(int);
  public Person copy(java.lang.String, int);
  public int copy$default$2();
  public java.lang.String copy$default$1();
  public int hashCode();
  public java.lang.String toString();
  public boolean equals(java.lang.Object);
  public java.lang.String productPrefix();
  public int productArity();
  public java.lang.Object productElement(int);
  public boolean canEqual(java.lang.Object);
  public Person(java.lang.String, int);
}
```

Then disassemble Person$.class:

```
$ javap Person$
```

Compiled from "Person.scala"

```
public final class Person$ extends scala.runtime.AbstractFunction2 implements
scala.ScalaObject,scala.Serializable{
  public static final Person$ MODULE$;
  public static {};
  public final java.lang.String toString();
  public scala.Option unapply(Person);
  public Person apply(java.lang.String, int);
  public java.lang.Object readResolve();
  public java.lang.Object apply(java.lang.Object, java.lang.Object);
}
```

Scala generates a lot of source code when you declare a class as a case class.

Overriding Default Accessors and Mutators for Class

When you define a constructor parameter to be a var field, Scala makes the field private to the class and automatically generates getter and setter methods that other classes can use to access the field. For instance, given a simple class like this:

```
// intentionally left the 'private' modifier off _symbol
class Stock (var _symbol: String) {

  // getter
  def symbol = _symbol

  // setter
  def symbol_= (s: String) {
    this.symbol = s
    println(s"symbol was updated, new value is $symbol")
  }
```

```
}
```

Assignment:
We are going to start by creating a new project in your Scala IDE then we're going to add a Scala app. What I'd like you to do is inside that app, I want you to add a class to create a bank account. The bank account will include the client name, the starting balance and the account type, checking or savings. Then, use a singleton for the account number, start it at 5000. In the main program, create several instances of bank accounts and print out your results.

Exercise 18: Creating Inner Classes

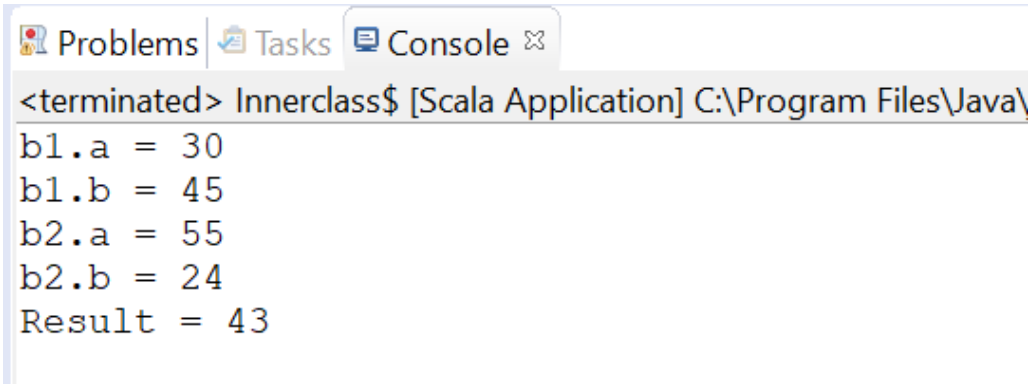
Problem
You want to create a class as an inner class to help keep the class out of your public API, or to otherwise encapsulate your code.

Solution
Declare one class inside another class. we define an Outer class Add and inside this class we define one more class Addtwonumbers which is the inner class. In the inner class Addtwonumbers, we are defining variables a, b, c and storing result of a+b in c. We are assigning values for variables a and b and accessing it by creating instances of the outer class and then instance of inner class through this outer class instance.

```
class Add {
  class Addtwonumbers {
 var a = 12;
 var b = 31;
 var c = a + b;

  }
}

object Innerclass {

def main(args:Array[String]) {

val a1 = new Add
val a2 = new Add
val b1 = new a1.Addtwonumbers
val b2 = new a2.Addtwonumbers

b1.a = 30;
b1.b = 45;
b2.a = 55;
b2.b = 24;
println(s"b1.a = ${b1.a}")
println(s"b1.b = ${b1.b}")
println(s"b2.a = ${b2.a}")
```

```
println(s"b2.b = ${b2.b}")
println(s"Result = ${b2.c}")

}
}
```

Output:



- **Problem**

You want to override the getter or setter methods that Scala generates for you in Classes.

Solution

This is a bit of a trick problem, because you can't override the getter and setter methods Scala generates for you, at least not if you want to stick with the Scala naming conventions.

If you have a class named Person with a constructor parameter named name, and attempt to create getter and setter methods according to the Scala conventions, your code won't compile:

```
// error: this won't work
class Person(private var name: String) {
 // this line essentially creates a circular reference
 def name = name
 def name_=(aName: String) { name = aName }
}
```

Attempting to compile this code generates three errors:

```
Person.scala:3: error: overloaded method name needs result type
 def name = name
         ^
Person.scala:4: error: ambiguous reference to overloaded definition,
both method name_= in class Person of type (aName: String)Unit
and  method name_= in class Person of type (x$1: String)Unit
match argument types (String)
 def name_=(aName: String) { name = aName }
```

```
                    ^
Person.scala:4: error: method name_= is defined twice
  def name_=(aName: String) { name = aName }
      ^
three errors found
```

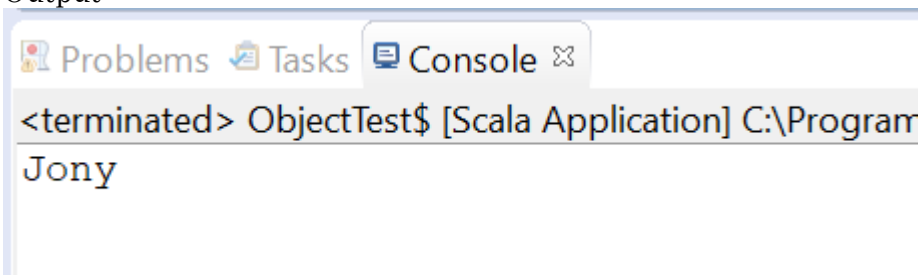
To solve the errors, Change the name of the field you use in the class constructor so it won't collide with the name of the getter method you want to use. A common approach is to add a leading underscore to the parameter name

```
class Person(private var _name: String) {
  def name = _name // accessor
  def name_=(aName: String) { _name = aName } // mutator
}

object ObjectTest extends App {
  val p = new Person("Jonathan")
  p.name = "Jony" // setter
  println(p.name) // getter
}
```

Creating a getter method named name and a setter method named name_= conforms to the Scala convention and lets a consumer of your class write code like this:

```
val p = new Person("Jonathan")
p.name = "Jony"    // setter
println(p.name)    // getter
```

Output:



Setting Uninitialized var Field Types

Imagine that you're starting a social network, and to encourage people to sign up, you only ask for a username and password during the registration process. Therefore, you define username and password as fields in your class constructor. However, later on, you'll also want to get other information from users, including their age, first name, last name, and address. But what do you do when you get to

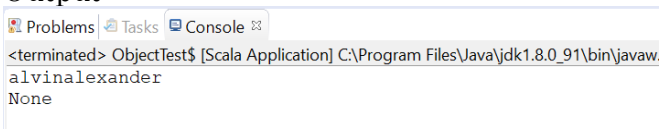
the address as it will be an Object and we cannot assign a default type unlike to Strings and Integers?

The solution is to define the address field as an Option, as shown here:

```
case class Person(var username: String, var password: String) {

  var age = 0
  var firstName = ""
  var lastName = ""
  var address = None: Option[Address]

}

case class Address(city: String, state: String, zip: String)
```

```
object ObjectTest extends App {
  val p = Person("alvinalexander", "secret")
  println(p.username)
  println(p.address)
}
```

Output:

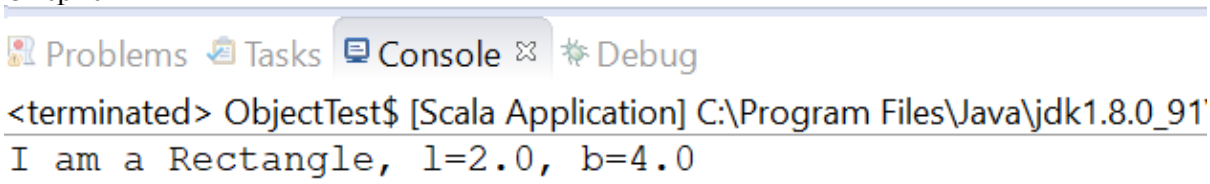


Exercise 19: Extending a Class

Let us consider a base class called Shape with name as properties and there are two sub classes of Shape: Rectangle and Square

```
class Shape(name:String) {
  val shapeName = name
  override def toString = s"I am a $shapeName"
}


class Rectangle(l:Double, b:Double, shapeName:String="Rectangle") extends Shape(shapeName) {
  val length = l
  val breadth = b
  override def toString = s"I am a $shapeName, l=$length, b=$breadth"
}


class Square(s:Double) extends Rectangle(s,s,"Square") {
}


object ObjectTest extends App {
  val p = new Rectangle(2,4,"Rectangle")
  println(p)
}
```

Output:



Assignment:

1. First define a Person base class, Next define Employee as a subclass of Person, so that it takes the constructor parameters name, address, and age. The name and address parameters are common to the parent Person class, so leave the var declaration off those fields, but age is new, so declare it as a var.
2. With this Employee class and an Address case class(case class Address (city: String, state: String)) and you can create a new Employee.
3. Test with the employee object and print name, address and age.

Exercise 20: Abstract class

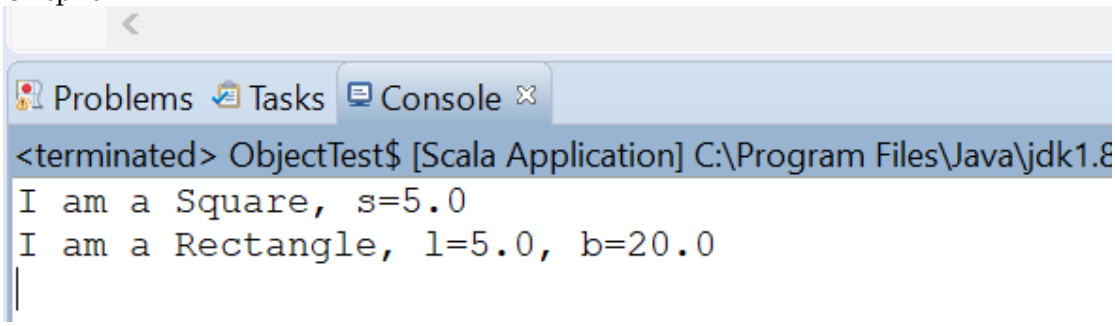
There are two main reasons to use an abstract class in Scala:

You want to create a base class that requires constructor arguments.
The code will be called from Java code.

Let us create an abstract class as Shape, and two concrete classes as Rectangle and Square implementing getArea() for both of these.

```
abstract class Shape(name: String) {
  val shapeName = name
  override def toString = s"I am a $shapeName"

  def getArea: Double //getArea is the abstract method which has Double as return type
}

class Rectangle(l: Double, b: Double, shapeName: String = "Rectangle") extends Shape(shapeName) {
  val length = l
  val breadth = b
  override def toString = s"I am a $shapeName, l=$length, b=$breadth"
  def getArea = l * b
}

class Square(s: Double, shapeName: String = "Square") extends Shape(shapeName) {
  override def toString = s"I am a $shapeName, s=$s"
  override def getArea = s * s
}

class AreaCalculator[T](s: T) {
  val shape: T = s
  override def toString = shape.toString
}
object ObjectTest extends App {
  val s = new Square(5)
  val r = new Rectangle(5, 20)
  println(s)
  println(r)
}
```

Output:



Note: A class can only extend one Abstract class.

Verify the output by creating a scala class as Animal in Eclipse:

```
abstract class Animal {
  val greeting = { println("Animal"); "Hello" }
}

class Dog extends Animal {
  override val greeting = { println("Dog"); "Woof" }
}

object Test extends App {
  new Dog
}
```

Exercise 21: Traits

Scala Traits consists of method and field definitions that can be reused by mixing classes. The class can mix any number of traits.

Traits define the objects by specifying the signature of the supported methods. Traits are declared in the same way as class except that the keyword trait is used.

Let us create a Trait with one concrete implementation for getTaxOnPrice().

```
trait Book {

   val id : Int
   val name : String
   val isbn : Long
   val price : Double
   //Concrete variable
   val category = "Uncategorized"

  //Concrete implementation
  def getTaxOnPrice : Double = {
    (price * 14)/100
  }

}
```

Extending traits

Similar to java, scala has the extends keyword which can be used for extending classes and traits.

Traits can be extended by other traits,abstract classes,concrete classes and case classes as well.

Classes extending traits

```
class ScienceBook extends Book{

  override val id: Int = 1000
  override val name: String = "A Brief History of Time"
  override val isbn: Long = 9783499605550l
  override val price: Double = 7.43

  override val category: String = "Science book"

  override def getTaxOnPrice : Double = {
    (price * 10)/100
  }

}
```

Create the object of ScienceBook and call getTaxOnPrice() for it and observe the output.

The with keyword

There is no concept of interfaces and implements keyword in scala, how would you go about extending a trait and then a class at the same time?

Let's consider another abstract class called Product.

```
abstract class Product {

 val prodID : Int
 val skuID : Int


}
```

Now since a book is a product we can combine the logic.

```
class ScienceBook extends Product with Book{

 override val id: Int = 1000
 override val name: String = "A Brief History of Time"
 override val isbn: Long = 9783499605550l
 override val price: Double = 7.43

 override val category: String = "Science book"

 override def getTaxOnPrice : Double = {
  (price * 10)/100
 }

 //Members of product abstract class
 override val prodID: Int = 20001504
 override val skuID: Int = 4574555
}
```

Note: Traits are more related to abstract classes than to interfaces. Main difference being traits do not have a constructor. Whenever you need to have a constructor for your OOP logic, then an abstract class will suit better, for all else traits are much better.

Assignment:

1. Write a class AccountInfo with methods deposit and withdraw, and a read-only property balance.

2. Write an object Conversions with methods inchestoFeet, milestoKms and poundsToKilos and invoke its methods from a class of your choice.

4. Extend the following BankAccount class to a CheckingAccount class that charges $1 for every deposit and withdrawal

Class BankAccount (initBal: Double) {
a. private var balance = initBal
b. def deposit(amount: Double) = { balance += amount; balance }
c. def withdraw(amount: Double) = { balance -= amount; balance }
}

Exercise 22: Objects

Scala is more object-oriented than Java because in Scala, we cannot have static members. Instead, Scala has singleton objects. A singleton is a class that can have only one instance, i.e., Object. You create singleton using the keyword object instead of class keyword. Since you can't instantiate a singleton object, you can't pass parameters to the primary constructor.

A singleton object is declared using the object keyword. Here is an example:

```
object Main {
   def sayHi() {
      println("Hi!");
   }
}
```

This example defines a singleton object called Main. You can call the method sayHi() like this:

```
Main.sayHi();
```

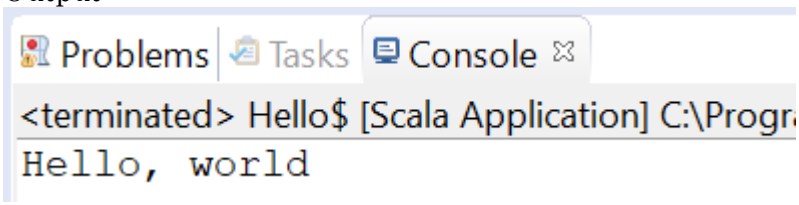
Note: It is like calling a static method in Java, except you are calling the method on a singleton object instead.

Launching an Application with an Object

There are two ways to create a launching point for your application: define an object that extends the App trait, or define an object with a properly defined main method.

```
Method 1:
object Hello extends App {
  println("Hello, world")
}
```

Output:

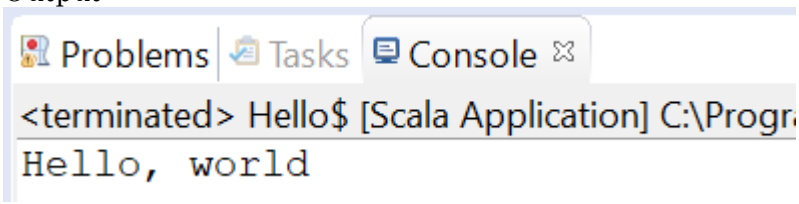


The code in the body of the object is automatically run, just as if it were inside a main method.

Method 2: The second approach to launching an application is to manually implement a main method with the correct signature in an object, in a manner similar to Java:

```
object Hello2 {
  def main(args: Array[String]) {
    println("Hello, world")
  }
}
```

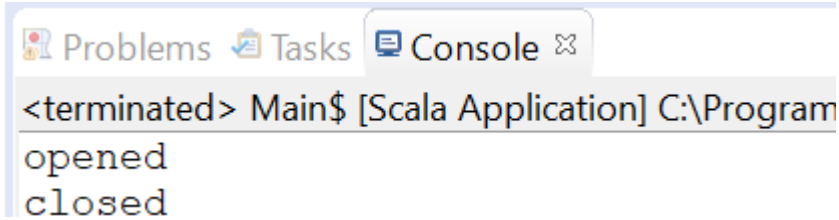
Output:



We might create a Singleton object to represent something like a keyboard, mouse, or perhaps a cash register in a pizza restaurant.

```
object CashRegister {
  def open { println("opened") }
  def close { println("closed") }
}

object Main extends App {
  CashRegister.open
  CashRegister.close
}
```

Output



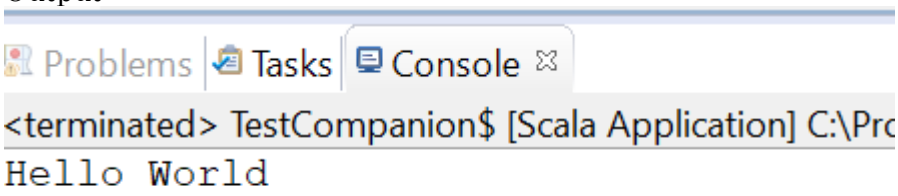
With CashRegister defined as an object, there can be only one instance of it, and its methods are called just like static methods on a Java class

Companion Objects

When a singleton object is named the same as a class, it is called a companion object. A companion object must be defined inside the same source file as the class.

```
class Main {
  def sayHelloWorld() {
    println("Hello World");
  }
}

object Main {
  def sayHi() {
    println("Hi!");
  }
}
object TestCompanion extends App {
  var aMain: Main = new Main();
  aMain.sayHelloWorld();
}
```

Output:



Creating Object Instances Without Using the new Keyword

There are two ways to do this:

1. Create a companion object for your class, and define an apply method in the companion object with the desired constructor signature.

2. Define your class as a case class.

CREATING A COMPANION OBJECT WITH AN APPLY METHOD

Let us define a Person class and Person object in the same file. Define an apply method in the object that takes the desired parameters. This method is essentially the constructor of your class

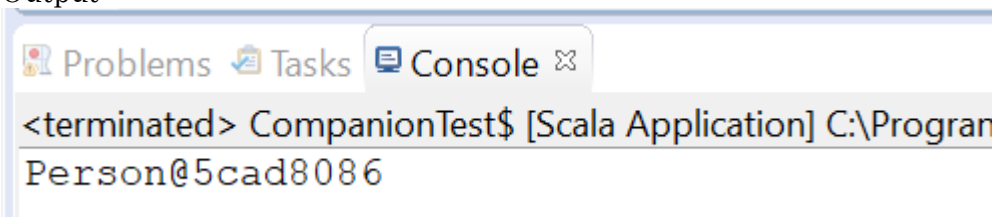
```
class Person {
  var name: String = _
}

object Person {
  def apply(name: String): Person = {
```

```
    var p = new Person
    p.name = name
    println(p)
    p
  }
}
```

Now we can create new Person instances without using the new keyword, as shown:

```
object CompanionTest extends App{
val dawn = Person("Dawn")
}
```

Output:

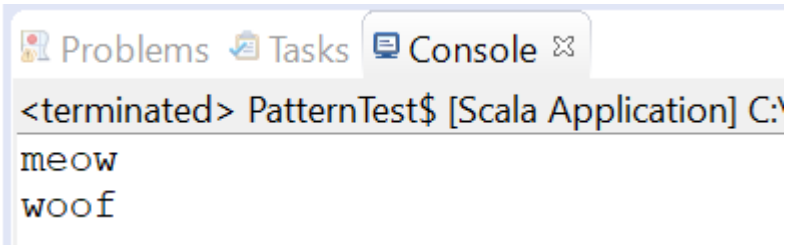


Implement the Factory Method in Scala with apply

You want to create an Animal factory that returns instances of Cat and Dog classes, based on what you ask for. By writing an apply method in the companion object of an Animal class, users of your factory can create new Cat and Dog instances.

```
trait Animal {
  def speak
}
object Animal {

  private class Dog extends Animal {
    override def speak { println("woof") }
  }

  private class Cat extends Animal {
    override def speak { println("meow") }
  }

  // the factory method
  def apply(s: String): Animal = {
    if (s == "dog") new Dog
    else new Cat
  }
}
object PatternTest extends App {
  val cat = Animal("cat") // returns a Cat
  cat.speak
  val dog = Animal("dog") // returns a Dog
  dog.speak
}
```

Output:



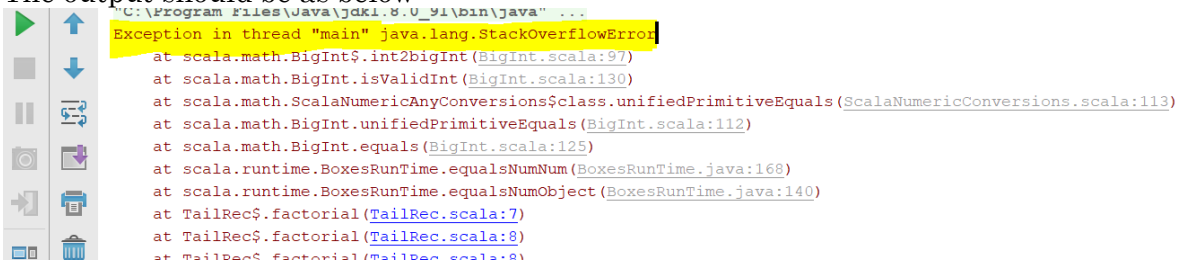
Assignment: Create a case class as Person with name as var field. Create an object of it and set the different values for it.

Exercise 23: Recursion

Assignment: Define a Factorial function that calculates the factorial of number as 100.

Once you get the output for the factorial 100, call the function to calculate the factorial of 10000.

The output should be as below:



Note: The Factorial method almost worked well until we call on the number fairly as large as 10000.

Here, you ended up writing a recursive method which may run out of stack memory. Scala offers to convert recursive method into a tail recursive method. Scala has an annotation as @scala.annotation.tailrec in the package scala.annotation.tailrec. It provides compiler assistance.

Tip: Tail recursive methods after evaluation must be returned.

You can make simple changes to factorial to eliminate both of these problems.

Step 1. First, you could move the recursive code into a local function within the method, so that it cannot be overridden.
Step 2. Second, you could introduce an accumulator so that multiplication happens before the recursive call.
Step 3. Finally, you could add a @tailrec annotation so that you can be sure that your changes have worked.

Create a new Scala object as TailRecursion.scala in IntelliJ and paste the below code:

```
import scala.annotation.tailrec
/**
  * Created by Neha Priya on 05-06-2018.
  */
object TailRecursion extends App{
  @tailrec
  def factorial(n: BigInt,acc:BigInt): BigInt = {
     if (n <= 1) acc
     else factorial(n - 1,n * acc)

  }

  def fact(n:Int)=factorial(n,1)

  fact(10000)
}
```

Try to execute the fact() method defined for the number 10000 and observe the output.

Assignment 1: How to calculate sum of numbers between two numbers?
For example from 0 to 1000, calculate the sum of all the numbers between using Tail Recursion.

Assignment 2: Implement the Fibonacci sequence

```
import scala.annotation.tailrec

object Recursion {

  /**
    * Implement an algorithm which returns a list with the X first elements of the Fibonacci sequence ,
    * from bigger to smaller
    * @param list the previous elements (empty list at first call)
    * @param size the number of elements you want
    */
  def fibonacci(list : List[Int], size : Int): List[Int]= ???


  /***
    * implement an algorithm which returns a list with the X first elements of the Fibonacci sequence ,
    * from bigger to smaller with tail recusion
    * @param size the number of elements you want
    *
    */
  def tailFibonacci(size: Int): List[Int]= ???
}
```

Note: Do not create an array of all elements and then iterate through it, adding all elements

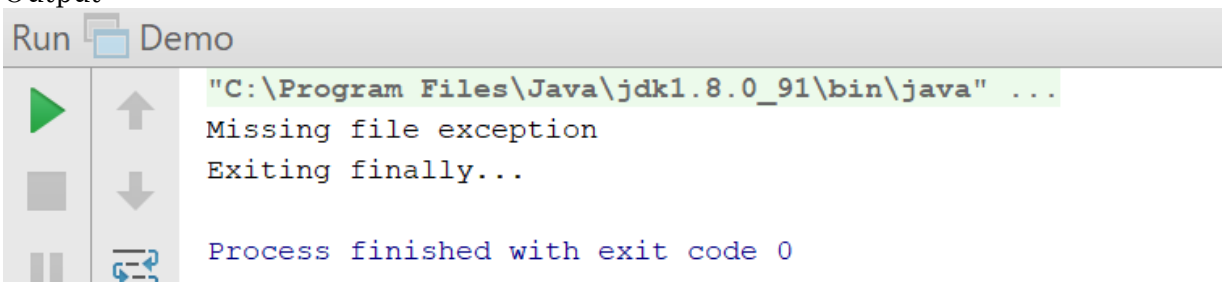
Exercise 24: Exceptions

Method 1. Using Try/catch:-

Let us create a Scala Object Demo.scala which will try to access the file "input.txt".

```
import java.io.{FileNotFoundException, FileReader, IOException}

/**
 * Created by Neha Priya on 09-04-2018.
 */
object Demo {
 def main(args: Array[String]) {
  try {
   val f = new FileReader("input.txt")
  } catch {
   case ex: FileNotFoundException => {
    println("Missing file exception")
   }

   case ex: IOException => {
    println("IO Exception")
   }
  }
   finally {
    println("Exiting finally...")
   }
  }

}
```

Output:



Note: There is (_) wildcard operator to catch any other exceptions that might occur.

Method 2. Try has 2 cases, Success[T] for the successful case and Failure[T] for the failure case. The main difference is that the failure can only be of type Throwable. You can use it instead of a try/catch block to postpone exception handling.

```
object Demo {
 def main(args: Array[String]) {
   // Throws a NumberFormatException when the integer cannot be parsed
   def parseIntException(value: String): Int = value.toInt

   // Catches the NumberFormatException and returns a Failure containing that exception
   // OR returns a Success with the parsed integer value
   def parseInt(value: String): Try[Int] = Try(value.toInt)
   parseIntException("Test")
   parseInt("Test")
   }

}
```

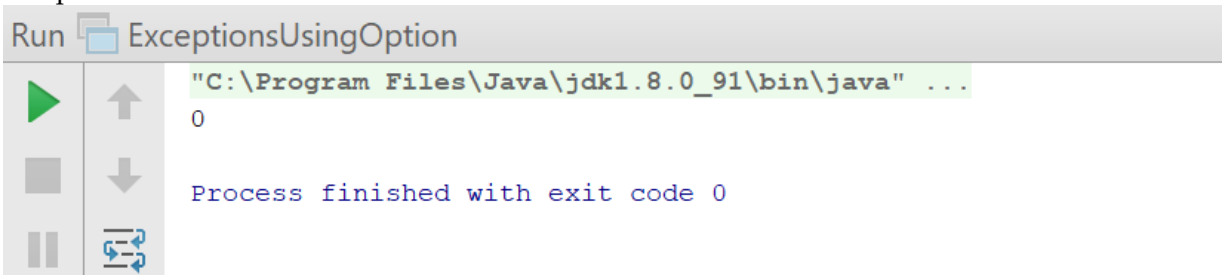
Try the output for both of the function calls.
The first function needs describing that an exception can be thrown. The second function describes in its signature what can be expected and requires the user of the function to take the failure case into account. Try is typically used when exceptions need to be propagated.

Method 3: Using Option to wrap exception
Options are super useful as a wrapper around null/None or any other default value. However, they can also be used to handle exceptions. The resulted value from an operation is provided if it ran successfully otherwise a None is returned.

Let us define a Scala Object, with method as divide which will divide two integers, wrapping it in Option. The call to the method returns the option's value if the option is nonempty, otherwise return the result of evaluating `default`.

```
object ExceptionsUsingOption extends App {

 def divide(x: Int, y: Int): Option[Int] =
   try {
     Option[Int](x / y)
   } catch {
     case _ => None
   }

 override def main(args: Array[String]): Unit = {
   val result: Option[Int] = divide(42, 0)
   println(result.getOrElse(0))
 }
}
```

Output:



Method 4: Custom Exceptions

Let us define a user defined exception as “invalidageexcepton”, when age is less than 18 throw the user defined exception with the custom messages.

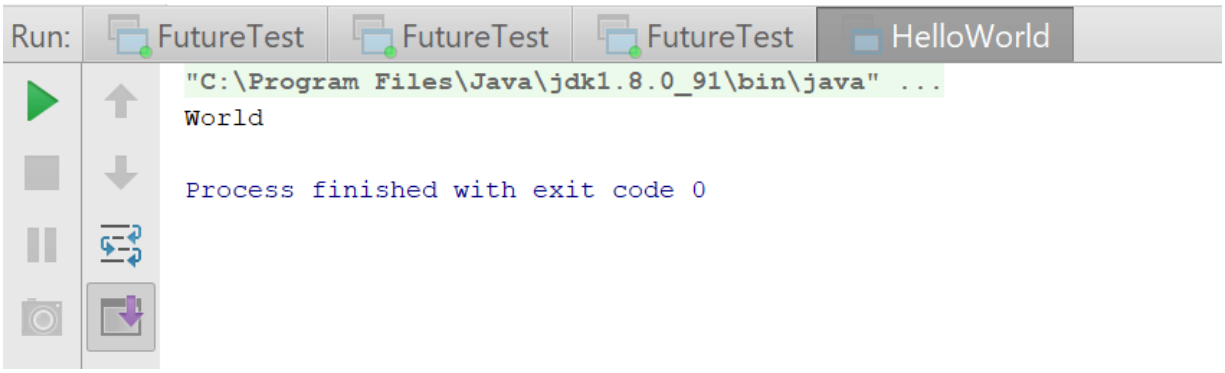
```
import java.io.IOException

class invalidageexcepton(smth: String) extends Exception(smth) {
}

object testexe {
 def validate(age: Int) {
  if (age < 18) {
   throw new invalidageexcepton("notvalid")
  } else {
   println("welcome to vote")

  }
 }
 def main(args: Array[String]) {
  try {
   validate(13)
  } catch {
   case ex: Exception => {
    println("output:" + ex)
   }
  }
 }
}
```

Exercise 25: Implicits, Futures and Promises

How do implicits work in Scala?

```
object HelloWorld {
 case class Text(content: String)
 case class Prefix(text: String)

 implicit def String2Text(content: String)(implicit prefix: Prefix) = {
  Text(prefix.text + " " + content)
 }

 def printText(implicit text: Text): Unit = {
  println(text.content)
 }

 implicit val test = Text("World")

 // Best to hide this line somewhere below a pile of completely unrelated code.
 // Better yet, import its package from another distant place.
 implicit val prefixLOL = Prefix("Hello")

 def main(args: Array[String]): Unit = {
  printText
  String2Text("Hello")
 }
}
```

Output:



Assignment:

Let us say that we have created operator everoading for adding two Complex numbers, as below:

```
class Complex(val real : Double, val imag : Double) {

 def +(that: Complex) =
        new Complex(this.real + that.real, this.imag + that.imag)

 def -(that: Complex) =
        new Complex(this.real - that.real, this.imag - that.imag)

 override def toString = real + " + " + imag + "i"

}

object Complex {
 def main(args : Array[String]) : Unit = {
     var a = new Complex(4.0,5.0)
     var b = new Complex(2.0,3.0)
     println(a)  // 4.0 + 5.0i
     println(a + b)  // 6.0 + 8.0i
     println(a - b)  // 2.0 + 2.0i
 }
}
```

Now suppose that we need to add a normal number to a complex number, how would we do that?
For Example: How to fix the highlighted error.

```
object Complex extends App{
     var a = new Complex(4.0,5.0)
     var b = new Complex(2.0,3.0)
     println(a)  // 4.0 + 5.0i
     println(a + b)  // 6.0 + 8.0i
     println(a - b)  // 2.0 + 2.0i

    val sum = a + 8.5
    }
```

Hint:Try to use Implicits conversions

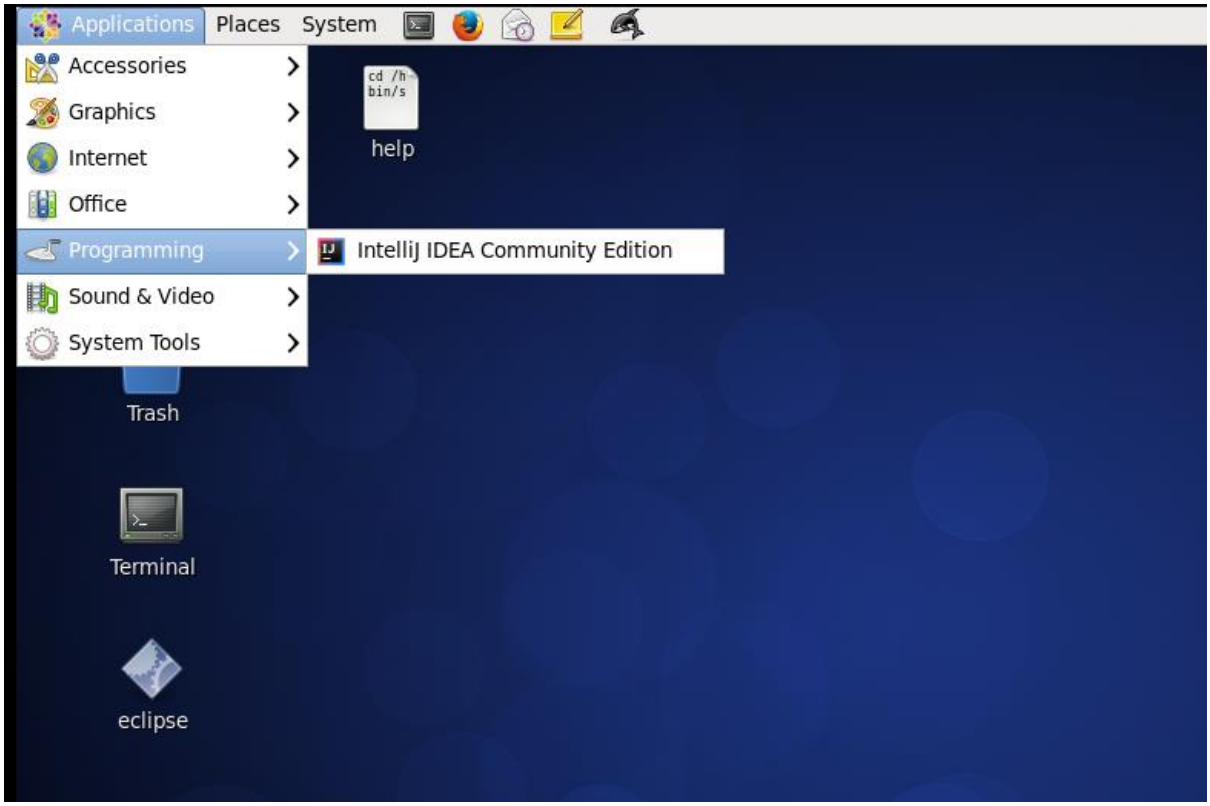
Futures

Future is an object that represents a computation unit. It has some value at some point in the future. In other words, it is a programming construct in Scala used to write concurrency programs very easily. When a Future finishes its computation, we say that the Future is completed.

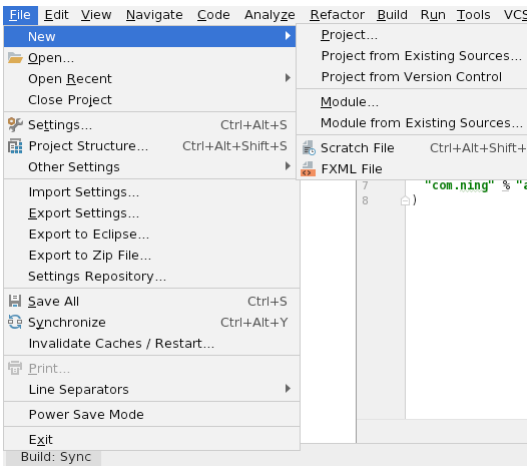
However, it may be completed either successfully, or unsuccessfully.
If a Future is completed successfully, it has some computed value. If it has failed or completed unsuccessfully, it has some error.

Let's start now with the Scala Hello World Future App

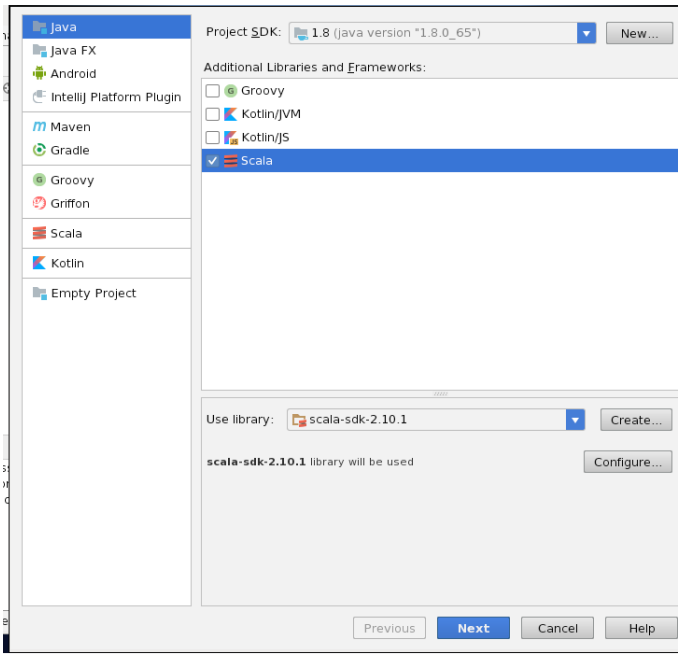
- Open IntelliJ from Applications > Programming > IntelliJ Idea Community Edition.

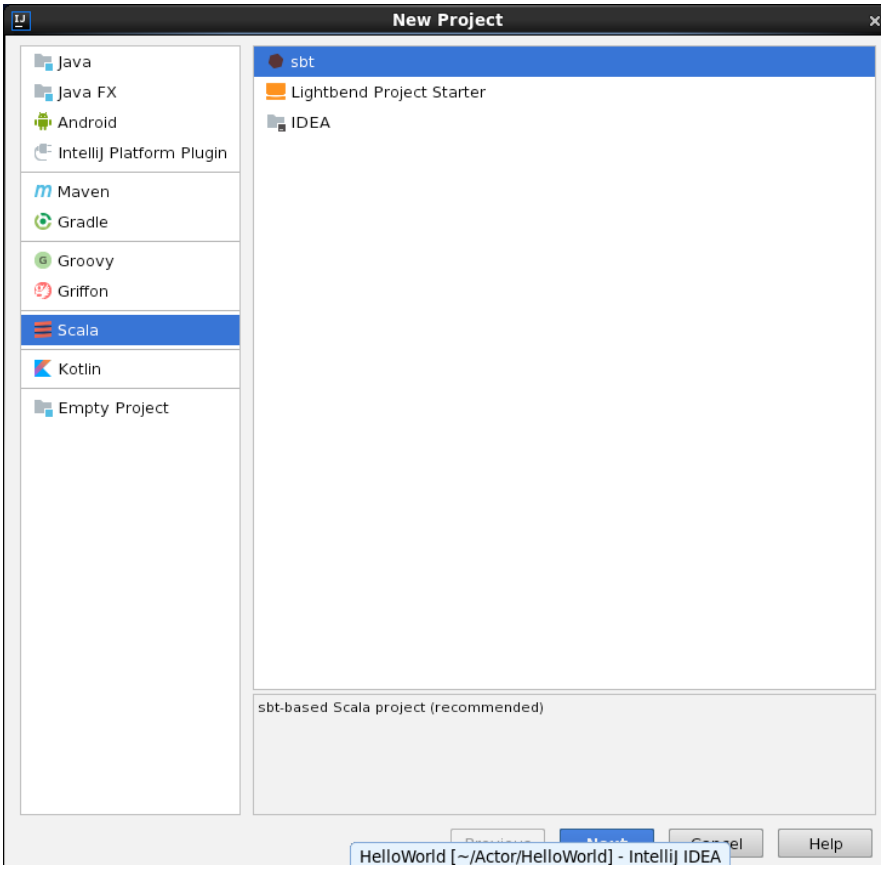


- Let us create an SBT project, as follows



Click on Next,





Click on Next,

Project Name: scala-future-app

- build.sbt:

```
name := "scala-future-app"
version := "1.0"
scalaVersion := "2.12.3"
```

- Create a Scala App:

```
object ScalaFutureHelloWorldApp extends App{
}
```

- Create a HelloWorld Future Object:

ScalaFutureHelloWorldApp.scala:

```
import scala.concurrent.Future
object ScalaFutureHelloWorldApp extends App{
  val helloWordlFuture = Future("Hello World")
}
```

- When we run this program, we can observe the following error:

```
[error] .../scala-future-
app/src/main/scala/com/packt/publishing/concurrent/future/ScalaFutureHelloWolrdAp
p.scala:8: Cannot find an implicit ExecutionContext. You might pass
[error] an (implicit ec: ExecutionContext) parameter to your method
[error] or import scala.concurrent.ExecutionContext.Implicits.global.
[error] val helloWordlFuture = Future("Hello World")
```

When we make a call to Future(someObject), internally it makes a call to the Future.apply() method and it has the following signature:

```
def apply[T](b: =>T)(implicit e: ExecutionContext): Future[T]
```

As Future objects run in ExecutionContext, they need that object, which is why the Future.apply() function has an implicit object of type ExecutionContext.Scala that provides a default implicit ExecutionContext object so we can use it by using the following import:

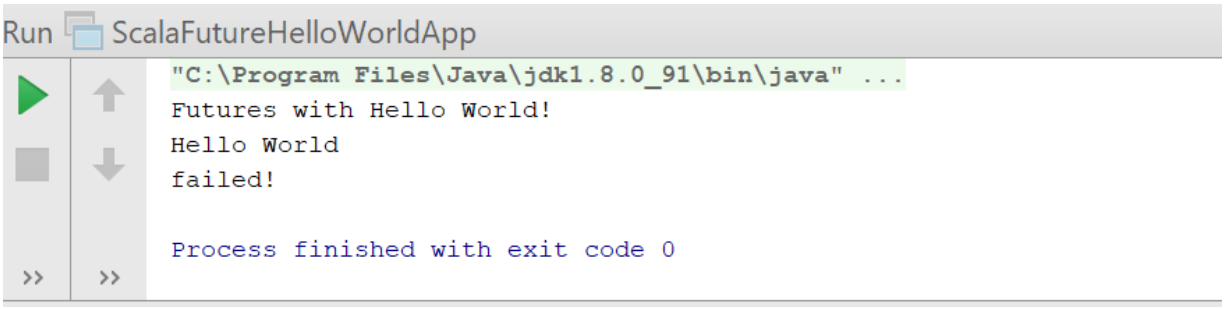
```
import scala.concurrent.ExecutionContext.Implicits.global
```

We then import the global execution context from the Implicits object. This makes sure that Future computations execute on global, the default ExecutionContext.

Now, the scala class looks like:-

```
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.Future

object ScalaFutureHelloWorldApp extends App{
     println("Futures with Hello World!")

     // Create one Future
   val f1 = Future {
    "Hello World"
   }

  f1.foreach(n => println(n))
  f1.failed.foreach(ex => println("failed!"))

    Thread.sleep(2000)


  val helloWordlFuture2 = Future {
    throw new Throwable("Hello World")
  }
  helloWordlFuture2.failed.foreach(ex => println("failed!"))
}
```

Output:



Let us try below code in the Scala REPL

```
scala> import scala.concurrent.Future

scala> import scala.concurrent.ExecutionContext.Implicits.global

scala val nameFuture = Future("Rams")

scala> nameFuture.value

scala> nameFuture.value.get

scala> nameFuture.value.get.get
```

Scala Future's onComplete() callback function

Scala Future's onComplete callback function. It refers to the scala.util.Try construct to check the Future value or exception.

```
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.Future
import scala.util.{Success,Failure}
object ScalaFutureWithOnCompleteCallbackApp extends App{

  val arithemeticFuture = Future {
    100/0
  }
  arithemeticFuture onComplete {
    case Success(_) => println("Future completed successfully.")
    case Failure(error) => println(s"Future completed with error: ${error}.")
  }
  Thread.sleep(1000)
}
```

Output:

Assignment:

Create a future that waits
1. for a random period of upto 1 second
2. gets a random Int between 0 and 100
3. prints it and returns it.

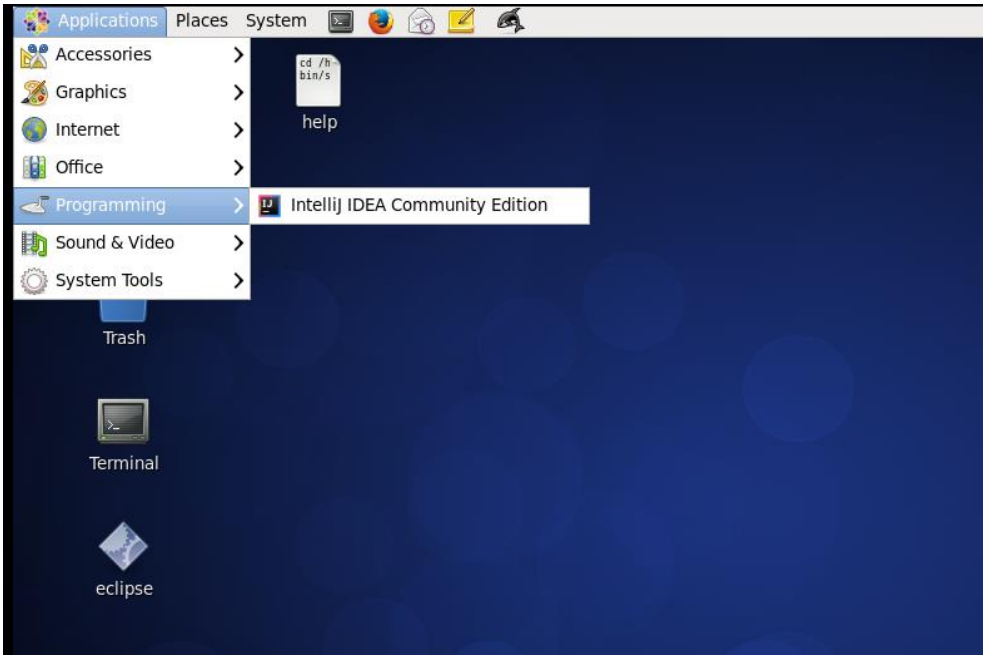
Create a second future that does the same thing
1. Add a for comprehension that waits for two Futures to complete
2. It should return boolean , representing v1>v2
3. Add a handler for the Success case that prints out the value
4. You may need to add a thread.sleep(2000) to your main so that main will wait for the code execution
5. Observe the output

Scala Promise

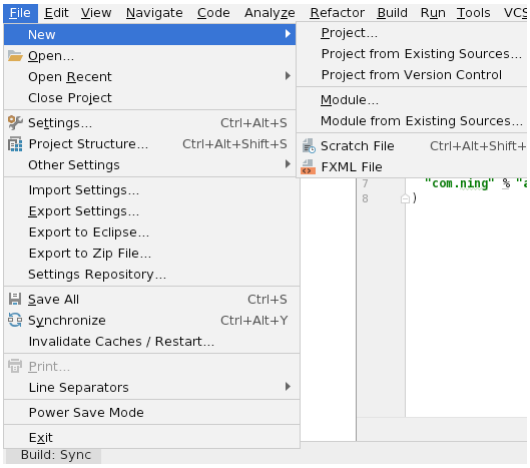
Future is a placeholder to hold a computation unit (which returns a value or exception and does not yet exist). It is used to read that result, whereas a Promise is used to finish that computation and write a value into the Future.

Exercise 26: Akka Actor - Concurrency

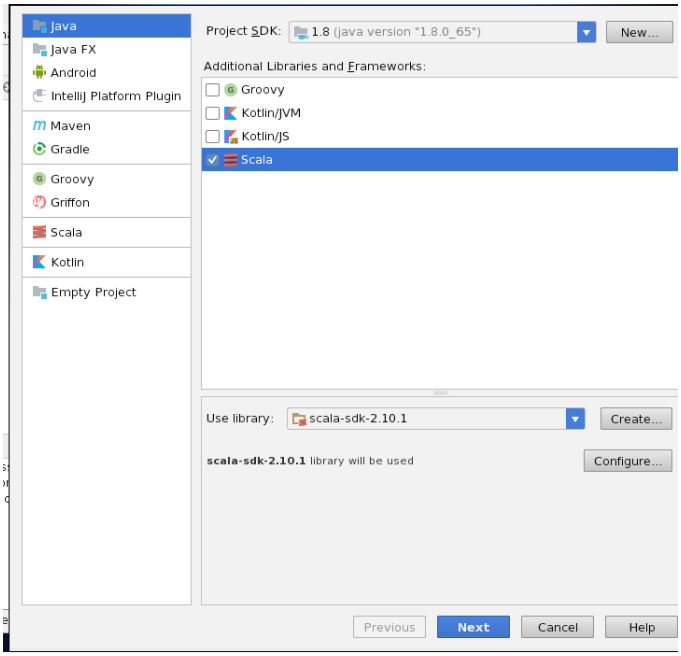
- Open IntelliJ from Applications > Programming > IntelliJ Idea Community Edition.

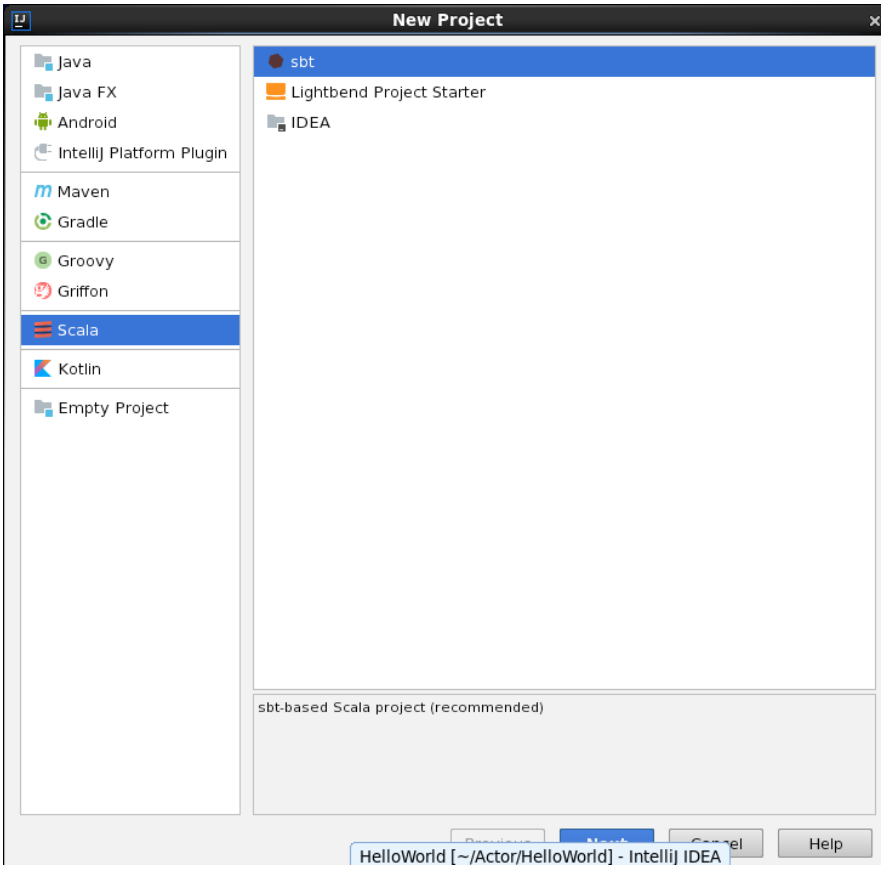


- Let us create an SBT project, as follows

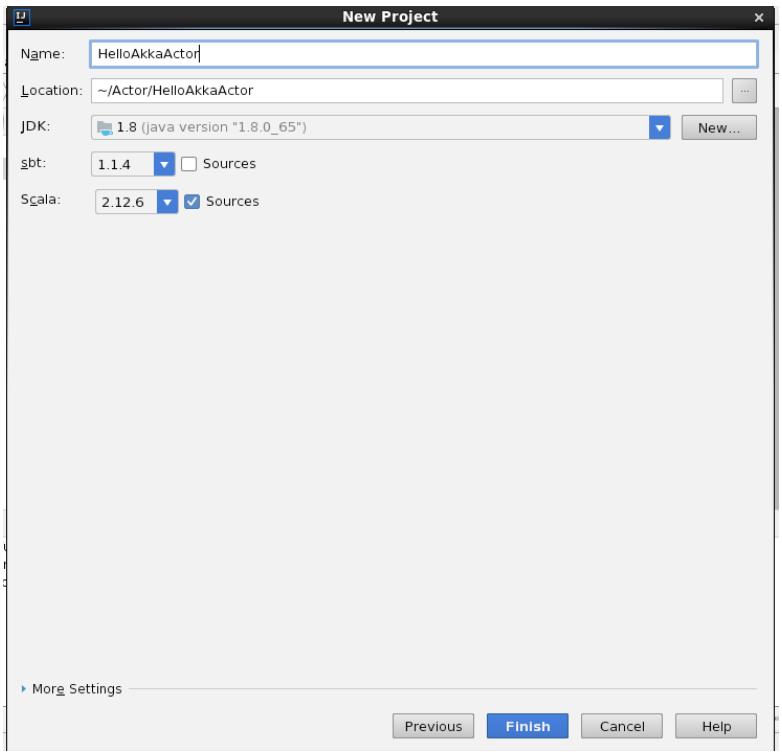


Click on Next,



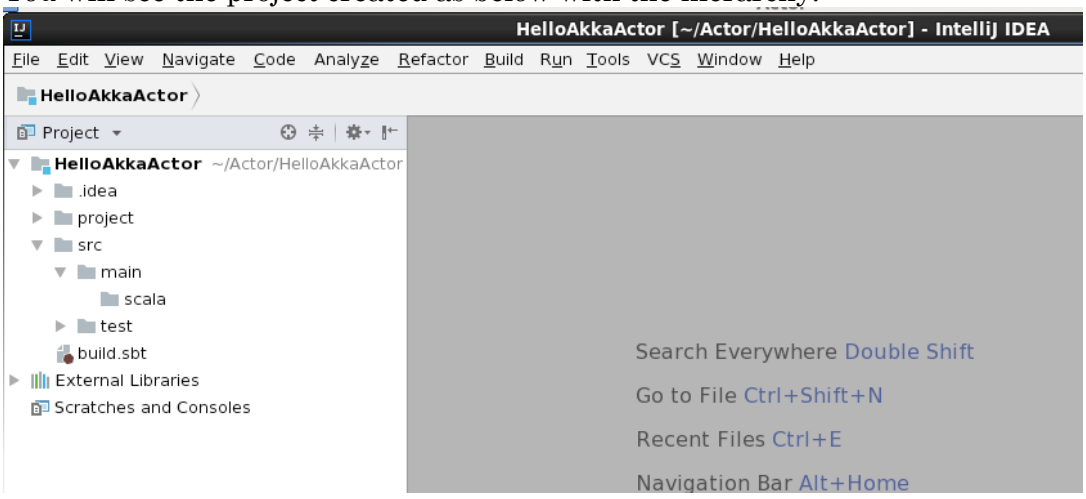


Click on Next,



Click on Finish.

You will see the project created as below with the hierarchy.



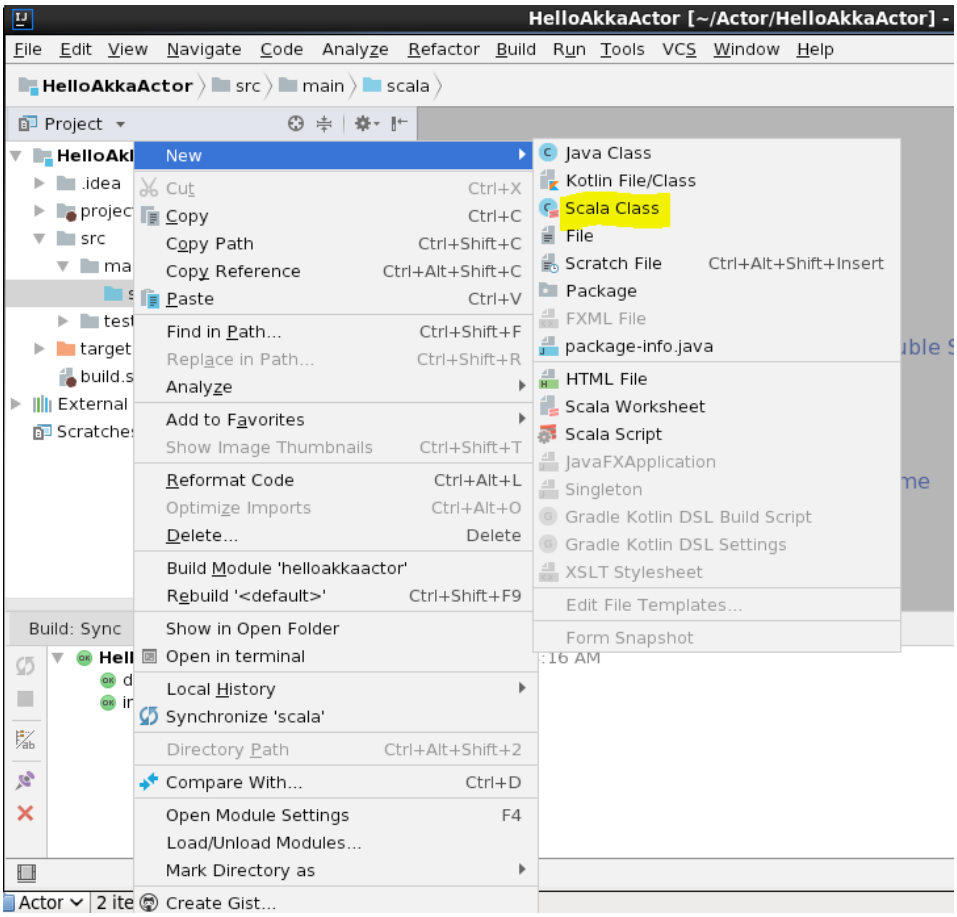
- Create an SBT project directory named HelloAkka, move into that directory, and then add the necessary Akka resolver and dependency information to your build.sbt file.

```
name := "Hello Test #1"

version := "1.0"

scalaVersion := "2.10.0"
```

```
resolvers += "Typesafe Repository" at
"http://repo.typesafe.com/typesafe/releases/"

libraryDependencies += "com.typesafe.akka" %% "akka-actor" % "2.1.2"
```

- Let us define an actor that responds when it receives the String literal hello as a message.

Let us create Hello.scala in the root directory of your SBT project. Notice how the literal hello is used in the first case statement in the receive method of the HelloActor class.



Paste the below code in the HelloActor class just created.

```
import akka.actor.Actor
import akka.actor.ActorSystem
import akka.actor.Props
class HelloActor extends Actor {
  def receive = {
    case "hello" => println("hello back at you")
    case _       => println("huh?")
  }
}

object Main extends App {

  // an actor needs an ActorSystem
```

```
  val system = ActorSystem("HelloSystem")

  // create and start the actor
  val helloActor = system.actorOf(Props[HelloActor], name = "helloactor")

  // send the actor two messages
  helloActor ! "hello"
  helloActor ! "buenos dias"

  // shut down the system
  system.shutdown

}
```

Description of the code:

1. The import statements import the members that are needed.
2. An Actor named HelloActor is defined.
3. HelloActor's behavior is implemented by defining a receive method, which is implemented using a match expression.
4. When HelloActor receives the String literal hello as a message, it prints the first reply, and when it receives any other type of message, it prints the second reply.
5. The Main object is created to test the actor.
6. In Main, an ActorSystem is needed to get things started, so one is created. The ActorSystem takes a name as an argument, so give the system a meaningful name. The name must consist of only the [a-zA-Z0-9] characters, and zero or more hyphens, and a hyphen can't be used in the leading space.
7. Actors can be created at the ActorSystem level, or inside other actors. At the ActorSystem level, actor instances are created with the system.actorOf method. The helloActor line shows the syntax to create an Actor with a constructor that takes no arguments.
8. Actors are automatically started (asynchronously) when they are created, so there's no need to call any sort of "start" or "run" method.
9. Messages are sent to actors with the ! method, and Main sends two messages to the actor with the ! method: hello and buenos dias.
10. helloActor responds to the messages by executing its println statements.
11. The ActorSystem is shut down.

Then run the application like this:

```
$ sbt run
```

Note: The project is already created in IntelliJ in the location /home/acadgild/Actor/Actor-Akka.
Browse to the location and type sbt run in the command terminal.

```
[acadgild@localhost Actor-Akka]$ sbt run
[info] Loading project definition from /home/acadgild/Actor/Actor-Akka/project
```

After SBT downloads the Akka JAR files and their dependencies, you should see the following output from the println statements in the HelloActor class.

```
[info] Packaging /home/acadgild/Actor/Actor-Akka/target/scala-2.11/scala_code_snippets_2.11-0.1.0-SNAPSHOT.jar ...
[info] Done packaging.
[info] Running BankAccount.Test
hello back at you
huh?
```

Use Case:

You can try to withdraw money from ATM terminal, and at the same time you can access your account via mobile and try to transfer the money from account to your friends account.
The challenge here is what happens when both the transection hits your account to debit the money on the same time. lets say you have 10000 rupees and you want to take the 10000rs from ATM and also want to send the 10000 rupees to your friend. if you will not use multithreading with the methods synchronized then, Your bank account will go in negative.

Assignment:

1. Create an ActorAccount class with companion object.
2. The initial amount in the account Rs 500. Create a case class Withdrawal() which will be always confirming amount transferred is greater than zero.
3. Write the definition for Withdrwal() in the ActorAccount class for handling withdrawal of money from the account.
4. The conditions to check are

```
if (ActorAccount.accountbalance >=amt) {
ActorAccount.accountbalance -=amt
println(s"the balance is deducted by $amt and the remaining balance is" +
ActorAccount.acountbalance)
}
else
{
  println(s"the balance in your account is "+ActorAccount.accountbalance+ s"
and you are trying to take $amt,the balance is not sufficient for transaction")
    }
```

5. Create the TestObject and send messages to the ActorAccount to withdraw() money in three times.

```
  actor1.!(actoraccount.withdrawal(100))
  actor1.!(actoraccount.withdrawal(200))
  actor1.!(actoraccount.withdrawal(300))
```

   Now verify how Actor is managing three requests without putting any lock in your resources.

Exercise 27: Scala Futures

Create a Gradle project in IntelliJ and include the gradle dependency, as below:-

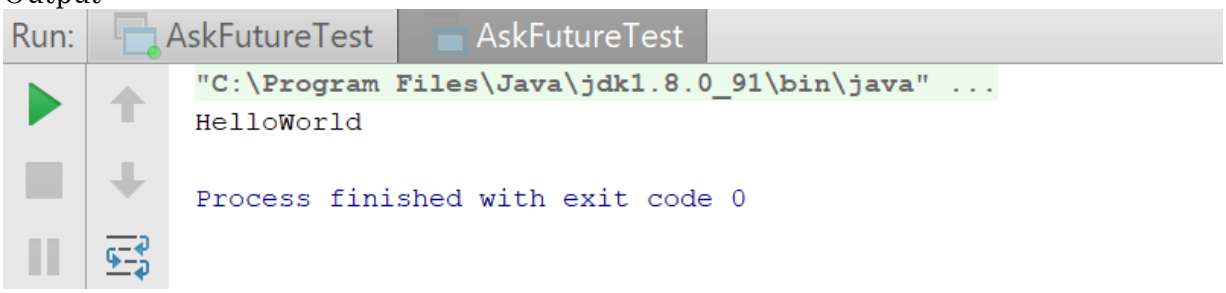
```
dependencies {
 compile group: 'com.typesafe.akka', name: 'akka-actor_2.12', version: '2.5.13'
}
```

Example 1: Hello World Example

Let us create a simple example of Future

```
import scala.concurrent.{Await, ExecutionContext, Future}
import scala.concurrent.duration._
import scala.language.postfixOps
import scala.concurrent.ExecutionContext.Implicits.global


object AskFutureTest extends App {

 val future = Future {
  "Hello" + "World"
 }

 val result = Await.result(future, 1 second)
 println(result)

}
```

Output:

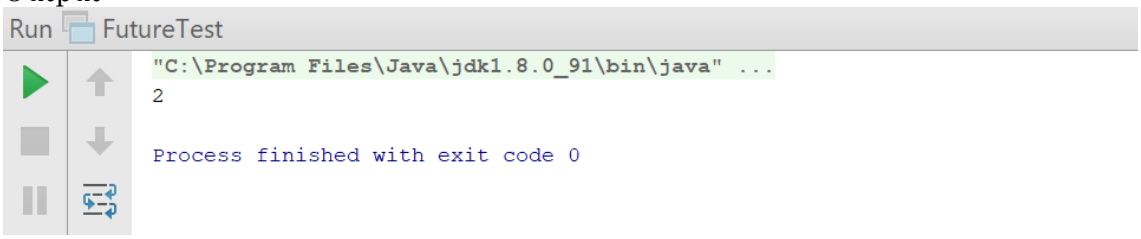


- **Assignment**: Let us create a scala object which will perform the calculation 1 + 1 at some time in the future. When it's finished with the calculation, it returns its result.

```
import scala.concurrent.{Await, Future}
import scala.concurrent.duration._
import scala.concurrent.ExecutionContext.Implicits.global


object FutureTest extends App {
 def sleep(time: Long) { Thread.sleep(time) }

 // used by 'time' method
 implicit val baseTime = System.currentTimeMillis

 // 2 - create a Future
```

```
 val f = Future {
  sleep(500)
  1 + 1
 }

 // 3 - this is blocking (blocking is bad)
 val result = Await.result(f, 1 second)
 println(result)
 sleep(1000)

}
```

Output:



Explaination of the code.

- The import statements bring the code into scope that's needed.
- The ExecutionContext.Implicits.global import statement imports the "default global execution context." Execution context is very similar as being a thread pool, and this is a simple way to get access to a thread pool.
- A Future is created after the second comment. Creating a Future is simple; you just pass it a block of code you want to run. This is the code that will be executed at some point in the future.
- The Await.result method call declares that it will wait for up to one second for the Future to return. If the Future doesn't return within that time, it throws a java.util.concurrent.TimeoutException.
- The sleep statement at the end of the code is used so the program will keep running while the Future is off being calculated. You won't need this in real-world programs, but in small example programs like this, you have to keep the JVM running.

Example 2: Sending a message to an Actor

Using an Actor's ? method to send a message will return a Future. To wait for and retrieve the actual result the simplest method is.

Let us create a Scala class as below:

```
import akka.actor._
import akka.pattern.ask
import akka.util.Timeout
import scala.concurrent.{Await, ExecutionContext, Future}
import scala.concurrent.duration._
import scala.language.postfixOps


case object AskNameMessage
```

```
class TestActor extends Actor {
 def receive = {
  case AskNameMessage => // respond to the 'ask' request
    sender ! "Test message"
  case _ => println("that was unexpected")
 }
}

object AskFutureTest extends App {

 // create the system and actor
 val system = ActorSystem("AskTestSystem")
 val myActor = system.actorOf(Props[TestActor], name = "myActor")

 implicit val timeout = Timeout(5 seconds)
 val future = myActor ? AskNameMessage // enabled by the “ask” import
 val result = Await.result(future, timeout.duration).asInstanceOf[String]
 println(result)
 system.terminate()

}
```

This will cause the current thread to block and wait for the Actor to ‘complete’ the Future with its reply. Blocking is discouraged though as it will cause performance problems. The blocking operations are located in Await.result and Await.ready to make it easy to spot where blocking occurs.

Note:

Await.result and Await.ready are provided for exceptional situations where you must block, a good rule of thumb is to only use them if you know why you must block.

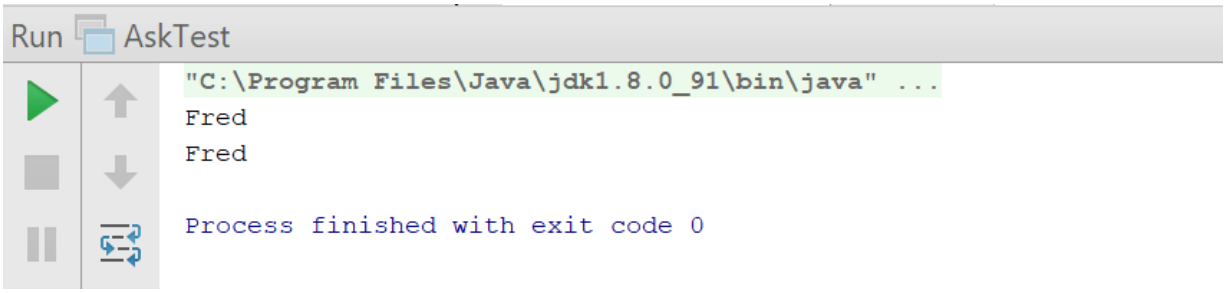
Example 3: Sending a Message to an Actor and Waiting for a Reply

Let us consider we have one actor that needs to ask another actor for some information, and needs an immediate reply. (The first actor can’t continue without the information from the second actor.)

```
import akka.actor._
import akka.pattern.ask
import akka.util.Timeout
import scala.concurrent.{Await, ExecutionContext, Future}
import scala.concurrent.duration._
import scala.language.postfixOps

case object AskNameMessage

class TestActor extends Actor {
 def receive = {
  case AskNameMessage => // respond to the 'ask' request
                sender ! "Fred"
  case _ => println("that was unexpected")
 }
}

object AskTest extends App {

 // create the system and actor
 val system = ActorSystem("AskTestSystem")
 val myActor = system.actorOf(Props[TestActor], name = "myActor")
```

```
// (1) this is one way to "ask" another actor for information
implicit val timeout = Timeout(5 seconds)
val future = myActor ? AskNameMessage
val result = Await.result(future, timeout.duration).asInstanceOf[String]
println(result)
// (2) a slightly different way to ask another actor for information
val future2: Future[String] = ask(myActor, AskNameMessage).mapTo[String]
val result2 = Await.result(future2, 1 second)
println(result2)
system.shutdown
}
```

Output:



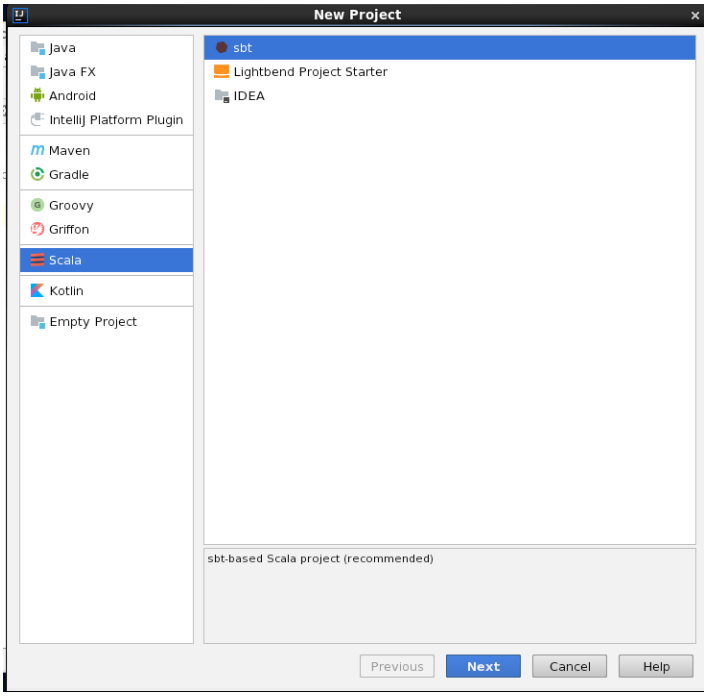
Explanation of the code:

- Send a message to an actor using either ? or ask instead of the usual ! method.
- The ? and ask methods create a Future, so you use Await.result to wait for the response from the other actor.
- The actor that's called should send a reply back using the ! method, as shown in the example, where the TestActor receives the AskNameMessage and returns an answer using sender ! "Fred".

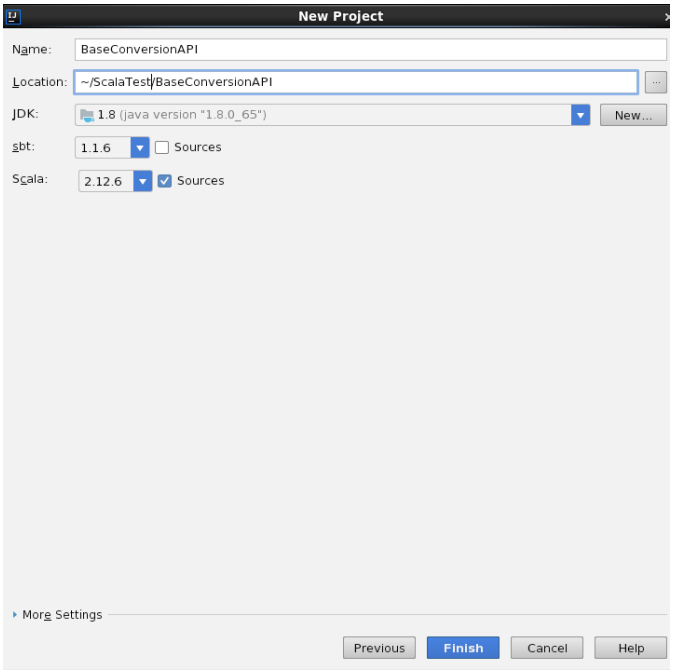
Exercise 27: Scala Test

We will need to define a project structure similar to the one we did in in Exercise 23. This time though, we have IntelliJ at our side, so we can use IntelliJ to create the structure for us. Follow these steps:

1. From the welcome window for IntelliJ, select Create New Project.
2. From the next dialog, select Scala from the left-hand margin and SBT from the right hand. Click on Next:



3. In the next dialog box, enter the project and version. This includes the name of project and its location. Name the project **BaseConversionAPI**. It also asks for version information about Scala and SBT you are using. Leave other options as default for now. Click on Finish.



4. Now add ScalaTest as a dependency in build.sbt:

```
libraryDependencies += "org.scalatest" %% "scalatest" % "3.0.5"
```

5. Writing tests with ScalaTest

Method 1. TDD style unit to write test classes extend the ScalaTest FunSuite, and then write your tests.

- Let us define Pizza.scala inside src→main→scala, with functions which will add Toppings, remove toppings and display the list of toppings in the Pizza.

```
package com.training.pizza

import scala.collection.mutable.ArrayBuffer

class Pizza {

 private val toppings = new ArrayBuffer[Topping]

 def addTopping (t: Topping) { toppings += t}
 def removeTopping (t: Topping) { toppings -= t}
 def getToppings = toppings.toList

 def boom { throw new Exception("Boom!") }
}
```

- Here's the Topping class inside src→main→scala:

```
package com.training.pizza

case class Topping(name: String)
```

- Let us start writing Test driven development test cases for each methods as defined.

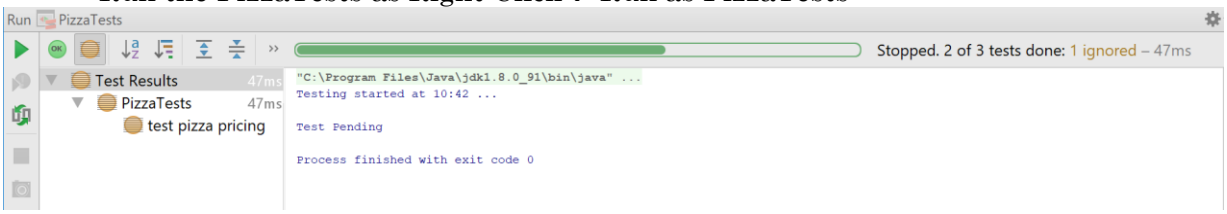
The following demonstrates a simple set of TDD tests for a Pizza class:
Create a Scala class inside src→test→scala as PizzaTests.

```
package com.training.pizza
import org.scalatest.FunSuite
import org.scalatest.BeforeAndAfter

class PizzaTests extends FunSuite with BeforeAndAfter {
 var pizza: Pizza = _

 before {
  pizza = new Pizza
 }

 test("new pizza has zero toppings") {
  assert(pizza.getToppings.size == 0)
 }

 test("adding one topping") {
  pizza.addTopping(Topping("green olives"))
  assert(pizza.getToppings.size === 1)
 }

 // mark that you want a test here in the future
 test ("test pizza pricing") (pending)
}
```

- The before method lets you do any setup work that needs to be performed before each test is run. Similarly, the after method lets you perform any teardown work that should be performed after each test.
- The pending feature shown in the last test is very helpful. This is a nice way of noting that you need to add a test in the future, but for one reason

or another you're not adding that test right now. As you saw, this ends up in the printed output like this.

Run the PizzaTests as Right Click→ Run as PizzaTests



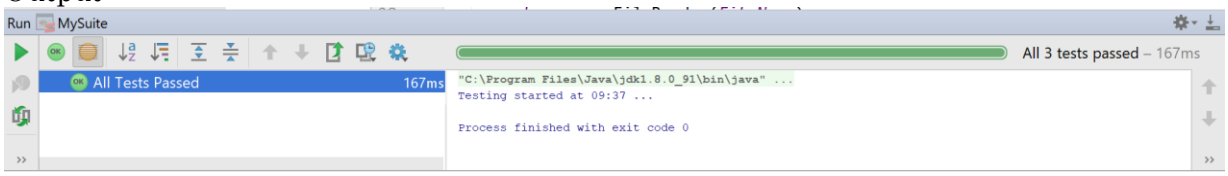
Setup and Tear Down

Create a Scala class as MySuite, and create a temporary file before each test, and delete it afterwords. We can use BeforeAndAfterEach trait for implementing set up and tear down methods

```
import org.scalatest.FunSuite
class MySuite extends FunSuite with BeforeAndAfterEach {

  private val FileName = "TempFile.txt"
  private var reader: FileReader = _

  // Set up the temp file needed by the test
  override def beforeEach() {
    val writer = new FileWriter(FileName)
    try {
      writer.write("Hello, test!")
    }
    finally {
      writer.close()
    }

    // Create the reader needed by the test
    reader = new FileReader(FileName)
  }

  // Close and delete the temp file
  override def afterEach() {
    reader.close()
    val file = new File(FileName)
    file.delete()
  }

  test("reading from the temp file") {
    var builder = new StringBuilder
    var c = reader.read()
    while (c != -1) {
      builder.append(c.toChar)
      c = reader.read()
    }
    assert(builder.toString === "Hello, test!")
  }

  test("first char of the temp file") {
    assert(reader.read() === 'H')
  }

  test("without a fixture") {
```

```
  assert(1 + 1 === 2)
 }
}
```

Output:

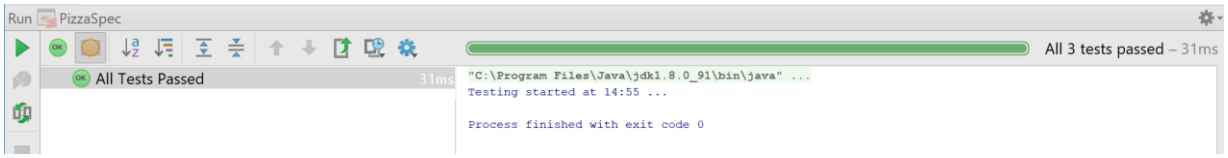


Method 2. BDD(behavior-driven development) style unit to write test classes extend the ScalaTest FunSpec trait, typically with the BeforeAndAfter trait.

Let us add test class for the Pizza class, begins with the describe method, with individual tests declared in it methods.

```
package com.training.pizza

import org.scalatest.FunSpec
import org.scalatest.BeforeAndAfter

class PizzaSpec extends FunSpec with BeforeAndAfter {

 var pizza: Pizza = _

 before {
  pizza = new Pizza
 }

 describe("A Pizza") {

  it("should start with no toppings") {
   assert(pizza.getToppings.size == 0)
  }

  it("should allow addition of toppings") {
   pizza.addTopping(Topping("green olives"))
   assert(pizza.getToppings.size === 1)
   println(pizza.getToppings)
  }

  it("should allow removal of toppings") {
   pizza.addTopping(Topping("green olives"))
   pizza.removeTopping(Topping("green olives"))
   assert(pizza.getToppings.size == 0)
  }
 }

}
```

Right Click →Run PizzaSpec.



Note: Use **it** in the singular context, and **they** when referring to a plural context.

Example 2: Scala test BDD-like by adding "Given/When/Then" behavior to them.

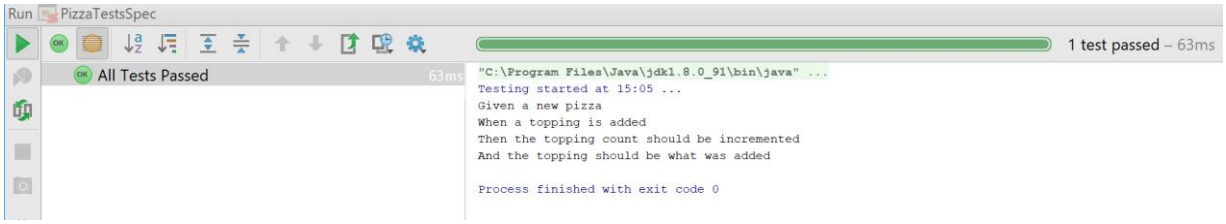
Let us mix the GivenWhenThen trait into your FunSpec BDD test, then add the Given/When/Then conditions, as shown in the following code.

```
package com.training.pizza

import org.scalatest.FunSpec
import org.scalatest.BeforeAndAfter
import org.scalatest.GivenWhenThen

class PizzaTestsSpec extends FunSpec with GivenWhenThen {

 var pizza: Pizza = _

 describe("A Pizza") {

  it ("should allow the addition of toppings") {
    Given("a new pizza")
    pizza = new Pizza

    When("a topping is added")
    pizza.addTopping(Topping("green olives"))

    Then("the topping count should be incremented")
    assertResult(1) {
      pizza.getToppings.size
    }

    And("the topping should be what was added")
    val t = pizza.getToppings(0)
    assert(t === new Topping("green olives"))
  }
 }

}
```

This is an example that might look like this:

1. Given a new pizza
2. When a topping is added
3. Then the pizza should have one topping

Let us run the test class and test the behaviour of the class, as expected or not.



Assignment:-
Develop a Behaviour driven development of test cases to ensure below behaviours for the Pizza class can be achieved or not.

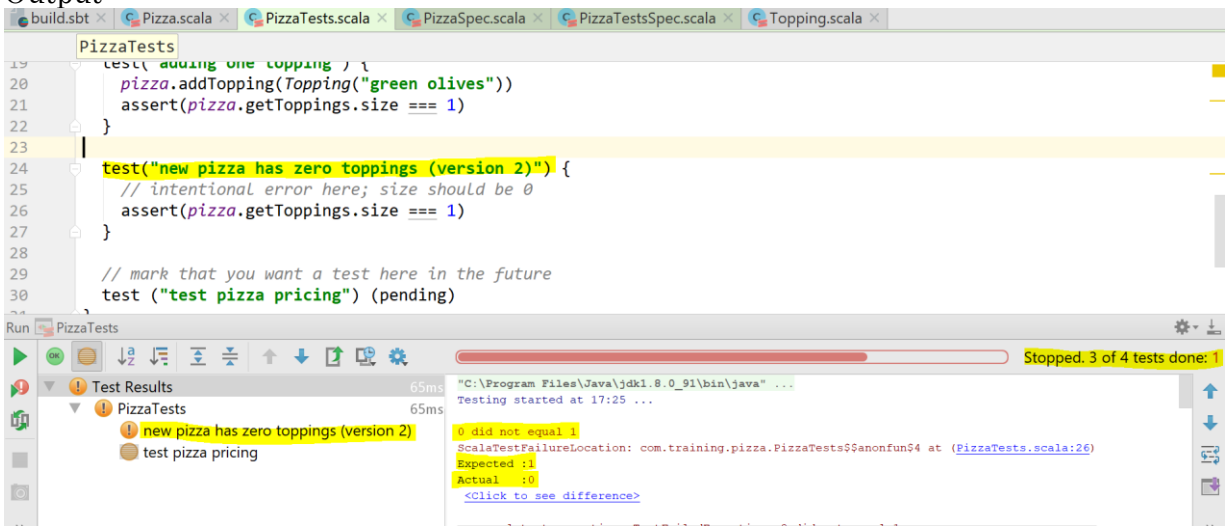
1. Given a new pizza
2. It Should start with no toppings
3. It Should allow removal of toppings
4. When the topping is removed then the topping count should be zero

- **Problem 1**. If we want to have better output from your ScalaTest assert tests, output that shows the expected and actual values.
- **Solution 1**. Use the === method instead of == for the test cases in case they are failing with the asserts.

Let us add another test case in PizzaTests.scala created earlier, which is likely to fail and we will observe the errors in the console wrt the expectations.

```
test("new pizza has zero toppings (version 2)") {
  // intentional error here; size should be 0
  assert(pizza.getToppings.size === 1)
}
```

Output:-

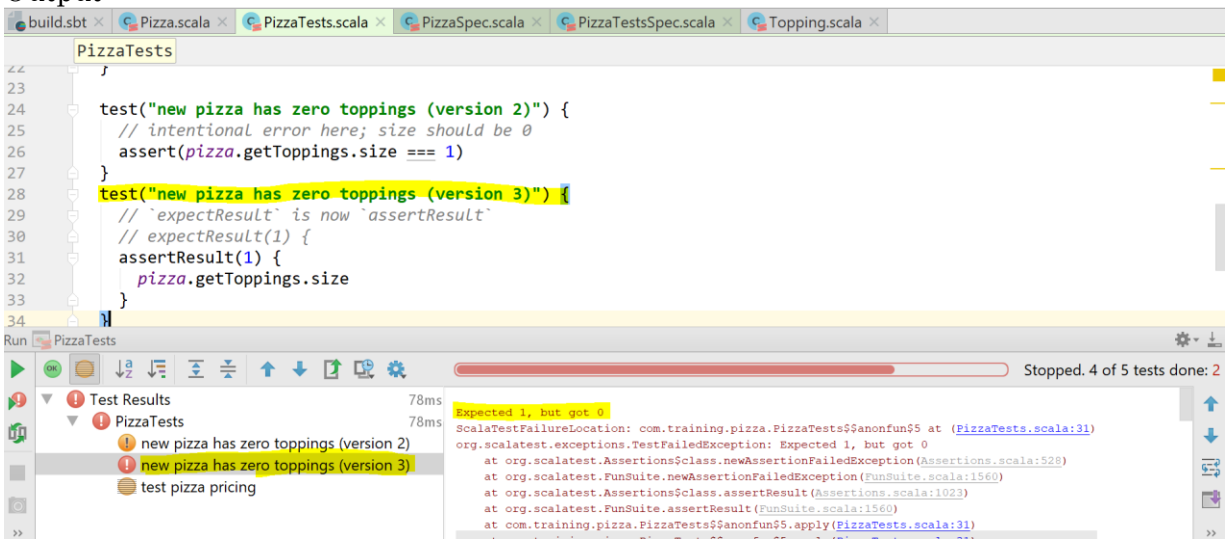


- **Solution 2**.

Another approach is to use assertResult instead of assert.
Let us add another test case in PizzaTests.scala as below and run the test class as ScalaTest.

```
test("new pizza has zero toppings (version 3)") {
 // `expectResult` is now `assertResult`
 // expectResult(1) {
 assertResult(1) {
  pizza.getToppings.size
 }
}
```

Output:



Intercepting Exceptions

When we need to test the code that is throwing an expected exception. In Scala, the intercept method lets you catch that exception, so you can verify that this portion of the method works as desired.

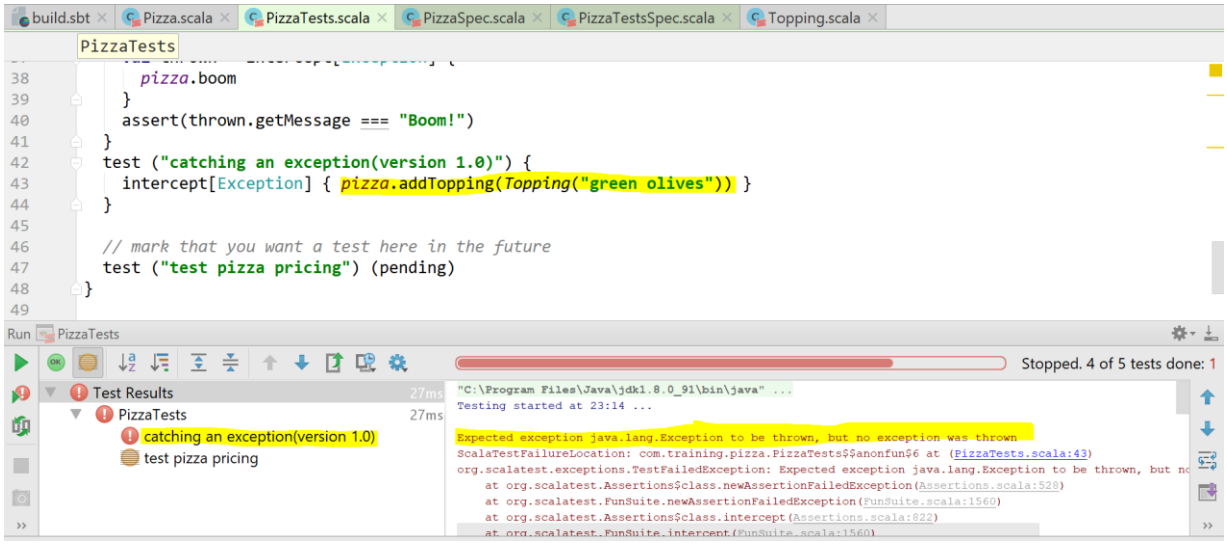
In Pizza.scala class we have a method as “boom” which is expected to throw Exeption. In order to TDD for the method boom. We add the below test case in PizzaTests.scala class

```
test ("catching an exception") {
 intercept[Exception] { pizza.boom }
}
```

If your code throws the exception, intercept catches it, and the test succeeds.

If the code which has scala test cases intercepting exceptions, and in case the method is not throwing exceptions and you have written intercept then it will be likely giving you the error as

Output:



Create fixtures

When a fixture is used by only one test method, then the definitions of the fixture objects can be local to the method.

Let us define a fixture in a class "ExampleSuite".

```
import org.scalatest.FunSuite
import collection.mutable.ListBuffer
class ExampleSuite extends FunSuite {
 def fixture =
  new {
   val builder = new StringBuilder("ScalaTest is ")
   val buffer = new ListBuffer[String]
  }

 test("easy") {
  val f = fixture
  f.builder.append("easy!")
  assert(f.builder.toString === "ScalaTest is easy!")
  assert(f.buffer.isEmpty)
  f.buffer += "sweet"
 }

 test("fun") {
  val f = fixture
  f.builder.append("fun!")
  assert(f.builder.toString === "ScalaTest is fun!")
  assert(f.buffer.isEmpty)
 }
}
```

Temporarily disabling unit tests

In Scalatests when we have to temporarily disable one or more tests.

Approach 1. In Test Driven Development

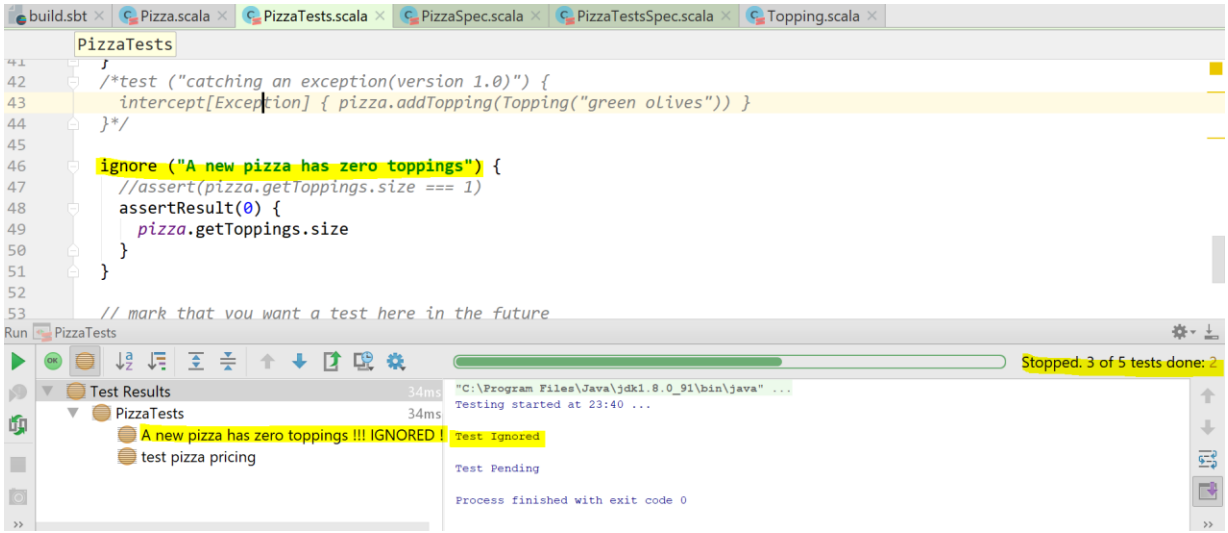
Change test method calls to ignore for all the tests which need not to be tested when running the entire test class.

Let us add a test case illustrating ignore so that the particular test block will not be tested.

Add the particular test case in PizzaTests.scala, as below:-

```
ignore ("A new pizza has zero toppings") {
  //assert(pizza.getToppings.size === 1)
  assertResult(0) {
   pizza.getToppings.size
  }
 }
```

Now, run the PizzaTests.scala test class, and observe the output

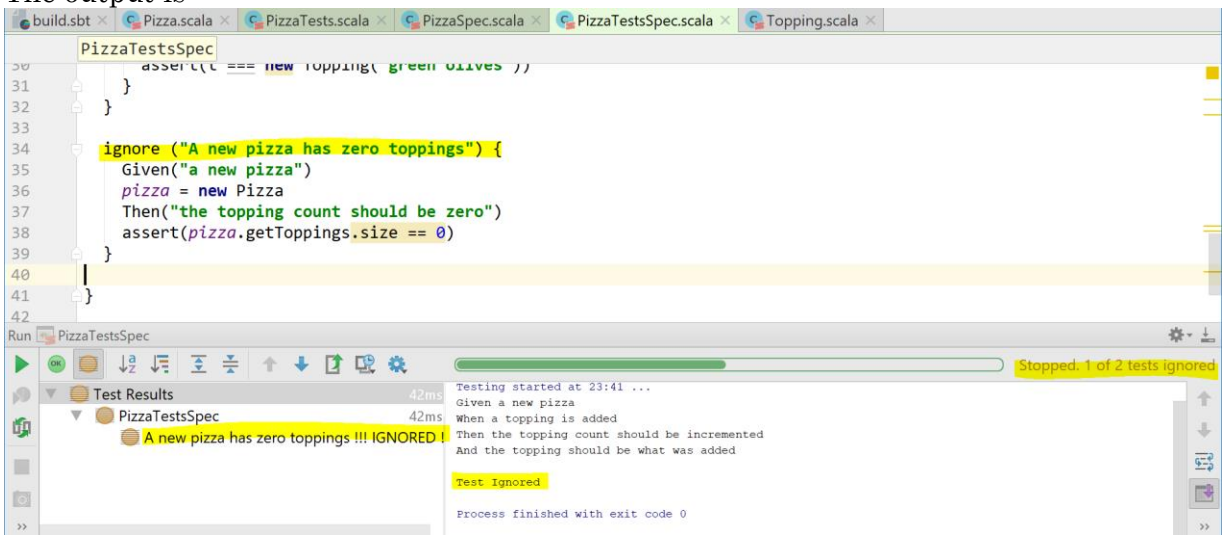


Approach 2: When using BDD-style tests, change it method calls to ignore:

Let us add the below piece of code in PizzaTestsSpec.scala

```
ignore ("A new pizza has zero toppings", DatabaseTest) {
 Given("a new pizza")
 pizza = new Pizza
 Then("the topping count should be zero")
 assert(pizza.getToppings.size == 0)
}
```

The output is



Exercise 28: Launching Spark

- Spark shell can be launched in two ways i.e. Scala and Python.
- To launch Scala spark shell, follow below steps

```
$ spark-shell
```

```
[cloudera@quickstart ~]$ spark-shell
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/lib/zookeeper/lib/slf4j-log4j12-1.7.5.jar!/org/
slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/jars/slf4j-log4j12-1.7.5.jar!/org/slf4j/impl/St
aticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel).
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.6.0
      /_/

Using Scala version 2.10.5 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_67)
```

You will see several INFO and WARNING message on the command prompt after launching spark-shell, which can be disregarded.

Logs will appear as below and finally you will get Scala Prompt

```
SQL context available as sqlContext.

scala>
```

- Spark creates a SparkContext object called sc, verify that the object exists

```
scala> sc
res0: org.apache.spark.SparkContext = org.apache.spark.SparkContext@a23c46d
```

- To know various SparkContext methods which are available, type sc. (sc followed by dot) and then TAB key

```
scala> sc.
accumulable                   accumulableCollection
accumulator                   addFile
addJar                        addSparkListener
appName                       applicationAttemptId
applicationId                 asInstanceOf
binaryFiles                   binaryRecords
broadcast                     cancelAllJobs
cancelJobGroup                clearCallSite
clearFiles                    clearJars
clearJobGroup                 defaultMinPartitions
defaultMinSplits              defaultParallelism
emptyRDD                      externalBlockStoreFolderName
files                         getAllPools
getCheckpointDir              getConf
getExecutorMemoryStatus       getExecutorStorageStatus
getLocalProperty              getPersistentRDDs
getPoolForName                getRDDStorageInfo
getSchedulingMode             hadoopConfiguration
hadoopFile                    hadoopRDD
initLocalProperties           isInstanceOf
isLocal                       isStopped
jars                          killExecutor
killExecutors                 makeRDD
```

Spark is based on the concept of Resilient Distributed Dataset (RDD), which is fault tolerant collection of elements that can be operated in parallel.

- Invoke python spark shell by using 'pyspark'

```
[cloudera@quickstart ~]$ pyspark
Python 2.6.6 (r266:84292, Jul 23 2015, 15:22:56)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-11)] on linux2
Type "help", "copyright", "credits" or "license" for more information.

Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 1.6.0
      /_/

Using Python version 2.6.6 (r266:84292, Jul 23 2015 15:22:56)
SparkContext available as sc, HiveContext available as sqlContext.
>>>
```

Two ways to create RDDs:

- Parallelizing an existing collection
- Referencing a dataset from an External Storage System

Parallelized collection:

To create a parallelized collection holding numbers 1 to 6, use **sc.parallelize** method

```
val data = Array (1,2,3,4,5,6)
val distData = sc.parallelize(data)
```

Once 'distData' RDD is created, we can perform **Action** such as:

- Finding the sum, mean & variance

```
distData.sum
distData.mean
distData.variance
```

External Storage System:

Spark can create distributed datasets from any storage system supported by Hadoop, Spark supports text files, Sequence Files and any other Hadoop Input Format.

Load a file from your Local Filesystem say batting.txt using **sc.textFile** method.

```
val textFile = sc.textFile("file:/home/cloudera/datasets/story.txt")
```

Operations on RDD will be discussed in next section.

Exploring Data using RDD operations:

Transformations

filter

Return a new dataset formed by selecting those elements of the source on which *func* returns true.

```
val a = sc.parallelize(1 to 10)
val b = a.filter(_% 2 == 0)
b.collect
```

```
scala> val a = sc.parallelize(1 to 10)
a: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:21

scala> val b = a.filter(_% 2 == 0)
b: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[1] at filter at <console>:23

scala> b.collect
16/08/30 06:59:25 INFO spark.SparkContext: Starting job: collect at <console>:26
```

```
16/08/30 06:59:28 INFO scheduler.DAGScheduler: ResultStage 0 (collect at <console>:26) finished in 0.316 s
16/08/30 06:59:28 INFO scheduler.TaskSetManager: Finished task 0.0 in stage 0.0 (TID 0) in 233 ms on localhost (1/1)
16/08/30 06:59:28 INFO scheduler.TaskSchedulerImpl: Removed TaskSet 0.0, whose tasks have all completed, from pool
16/08/30 06:59:28 INFO scheduler.DAGScheduler: Job 0 finished: collect at <console>:26, took 2.285340 s
res0: Array[Int] = Array(2, 4, 6, 8, 10)
```

map

Return a new distributed dataset formed by passing each element of the source through a function *func*.

```
val a = sc.parallelize(List("dog", "salmon", "salmon", "rat", "elephant"))

val b = a.map(_.length)

val c = a.zip(b)

c.collect
```

```
scala> val a = sc.parallelize(List("dog", "salmon", "salmon", "rat", "elephant"))
a: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[2] at parallelize at <console>:21

scala> val b = a.map(_.length)
b: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[3] at map at <console>:23

scala> val c = a.zip(b)
c: org.apache.spark.rdd.RDD[(String, Int)] = ZippedPartitionsRDD2[4] at zip at <console>:25

scala> c.collect
16/08/30 07:03:58 INFO spark.SparkContext: Starting job: collect at <console>:28
```

```
16/08/30 07:03:58 INFO executor.Executor: Finished task 0.0 in stage 1.0 (TID 1). 1147 bytes result sent to driver
16/08/30 07:03:58 INFO scheduler.DAGScheduler: ResultStage 1 (collect at <console>:28) finished in 0.062 s
16/08/30 07:03:58 INFO scheduler.DAGScheduler: Job 1 finished: collect at <console>:28, took 0.108580 s
16/08/30 07:03:58 INFO scheduler.TaskSetManager: Finished task 0.0 in stage 1.0 (TID 1) in 64 ms on localhost (1/1)
res1: Array[(String, Int)] = Array((dog,3), (salmon,6), (salmon,6), (rat,3), (elephant,8))
```

Using Python Shell:

```
 nums = sc.parallelize([1, 2, 3, 4])

squared = nums.map(lambda x: x * x).collect()

for num in squared: print "%i " % (num)
```

```
>>> for num in squared: print "%i " % (num)
...
1
4
9
16
>>>
```

distinct

Return a new dataset that contains the distinct elements of the source dataset.

```
val c = sc.parallelize(List("Gnu", "Cat", "Rat", "Dog", "Gnu", "Rat"),
2) . c.distinct.collect
```

```
scala> val c = sc.parallelize(List("Gnu", "Cat", "Rat", "Dog", "Gnu", "Rat"), 2)
c: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[5] at parallelize at <console>:21

scala> c.distinct.collect
16/08/30 07:07:16 INFO spark.SparkContext: Starting job: collect at <console>:24
```

```
16/08/30 07:07:19 INFO executor.Executor: Finished task 1.0 in stage 3.0 (TID 5). 1171 bytes result sent to driver
16/08/30 07:07:19 INFO scheduler.TaskSetManager: Finished task 1.0 in stage 3.0 (TID 5) in 76 ms on localhost (2/2)
16/08/30 07:07:19 INFO scheduler.TaskSchedulerImpl: Removed TaskSet 3.0, whose tasks have all completed, from pool
16/08/30 07:07:19 INFO scheduler.DAGScheduler: ResultStage 3 (collect at <console>:24) finished in 0.385 s
16/08/30 07:07:19 INFO scheduler.DAGScheduler: Job 2 finished: collect at <console>:24, took 2.185199 s
res2: Array[String] = Array(Dog, Cat, Gnu, Rat)
```

cartesian

When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).

```
val x = sc.parallelize(List(1,2,3,4,5)) .

val y = sc.parallelize(List(6,7,8,9,10)) .

x.cartesian(y).collect
```

```
scala> val x = sc.parallelize(List(1,2,3,4,5))
x: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[9] at parallelize at <console>:21

scala> val y = sc.parallelize(List(6,7,8,9,10))
y: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[10] at parallelize at <console>:21

scala> x.cartesian(y).collect
16/08/30 07:10:11 INFO spark.SparkContext: Starting job: collect at <console>:26
```

```
16/08/30 07:10:11 INFO scheduler.DAGScheduler: ResultStage 4 (collect at <console>:26) finished in 0.145 s
16/08/30 07:10:11 INFO scheduler.DAGScheduler: Job 3 finished: collect at <console>:26, took 0.264577 s
16/08/30 07:10:11 INFO scheduler.TaskSetManager: Finished task 0.0 in stage 4.0 (TID 6) in 153 ms on localhost (1/1)
16/08/30 07:10:11 INFO scheduler.TaskSchedulerImpl: Removed TaskSet 4.0, whose tasks have all completed, from pool
res3: Array[(Int, Int)] = Array((1,6), (1,7), (1,8), (1,9), (1,10), (2,6), (2,7), (2,8), (2,9), (2,10), (3,6), (3,7), (3,8), (3,
9), (3,10), (4,6), (4,7), (4,8), (4,9), (4,10), (5,6), (5,7), (5,8), (5,9), (5,10))
```

coalesce

Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset.

```
val y = sc.parallelize(1 to 10, 10)

val z = y.coalesce(2, false)

z.partitions.length
```

```
scala> val y = sc.parallelize(1 to 10, 10)
y: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[12] at parallelize at <console>:21

scala> val z = y.coalesce(2, false)
z: org.apache.spark.rdd.RDD[Int] = CoalescedRDD[13] at coalesce at <console>:23

scala> z.partitions.length
res4: Int = 2
```

filterByRange

Returns an RDD containing only the items in the key range specified.

```
val randRDD = sc.parallelize(List( (2,"cat"), (6, "mouse"),(7, "cup"),
(3, "book"), (4, "tv"), (1, "screen"), (5, "heater")), 3)

val sortedRDD = randRDD.sortByKey()

sortedRDD.filterByRange(1,3).collect
```

```
16/08/30 07:14:40 INFO executor.Executor: Finished task 0.0 in stage 7.0 (TID 13). 1357 bytes result sent to driver
16/08/30 07:14:40 INFO scheduler.DAGScheduler: ResultStage 7 (collect at <console>:26) finished in 0.080 s
16/08/30 07:14:40 INFO scheduler.DAGScheduler: Job 5 finished: collect at <console>:26, took 0.527731 s
res5: Array[(Int, String)] = Array((1,screen), (2,cat), (3,book))
```

flatMap

Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item).

```
val a = sc.parallelize(1 to 10, 5)

a.flatMap(1 to _).collect
```

```
16/08/30 07:18:44 INFO executor.Executor: Finished task 4.0 in stage 9.0 (TID 20). 974 bytes result sent to
16/08/30 07:18:44 INFO scheduler.TaskSetManager: Finished task 4.0 in stage 9.0 (TID 20) in 24 ms on localho
16/08/30 07:18:44 INFO scheduler.DAGScheduler: ResultStage 9 (collect at <console>:24) finished in 0.141 s
16/08/30 07:18:44 INFO scheduler.DAGScheduler: Job 7 finished: collect at <console>:24, took 0.193634 s
res6: Array[Int] = Array(1, 1, 2, 1, 2, 3, 1, 2, 3, 4, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6, 7,
, 8, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

```
val b = sc.parallelize(List(1, 2, 3), 2).flatMap(x => List(x,
x, . x)).collect
res85: Array[Int] = Array(1, 1, 1, 2, 2, 2, 3, 3, 3)
```

```
16/08/30 07:17:29 INFO executor.Executor: Running task 1.0 in stage 8.0 (TID 15)
16/08/30 07:17:29 INFO executor.Executor: Finished task 1.0 in stage 8.0 (TID 15). 922 bytes result sent to driver
16/08/30 07:17:29 INFO scheduler.DAGScheduler: ResultStage 8 (collect at <console>:23) finished in 0.104 s
16/08/30 07:17:29 INFO scheduler.DAGScheduler: Job 6 finished: collect at <console>:23, took 0.126929 s
b: Array[Int] = Array(1, 1, 1, 2, 2, 2, 3, 3, 3)
```

```
val lines = sc.parallelize(List("hello world",
"hi")) . val words = lines.flatMap(line =>
line.split(" ")) . words.first() // returns "hello"
```

```
16/08/30 07:21:01 INFO scheduler.DAGScheduler: ResultStage 10 (first at <console>:26) finished in 0.006 s
16/08/30 07:21:01 INFO scheduler.DAGScheduler: Job 8 finished: first at <console>:26, took 0.035576 s
res7: String = hello
```

Using python shell:

```
lines = sc.parallelize(["hello world", "hi"])
words = lines.flatMap(lambda line: line.split(" "))
words.first()  # returns "hello"
```

```
>>> lines = sc.parallelize(["hello world", "hi"])
>>> words = lines.flatMap(lambda line: line.split(" "))
>>> words.first()
16/09/17 09:38:59 INFO spark.SparkContext: Starting job: runJob at PythonRDD.sca
la:393

'hello'
>>>
```

groupBy

```
val a = sc.parallelize(1 to 9, 3)
a.groupBy(x => { if (x % 2 == 0) "even" else "odd" }).collect
```

```
16/08/30 07:23:14 INFO scheduler.DAGScheduler: ResultStage 12 (collect at <console>:26) finished in 0.182 s
16/08/30 07:23:14 INFO scheduler.DAGScheduler: Job 9 finished: collect at <console>:26, took 0.447740 s
res8: Array[(String, Iterable[Int])] = Array((even,CompactBuffer(2, 4, 6, 8)), (odd,CompactBuffer(1, 3, 5, 7
, 9)))
```

keys

Extracts the keys from all contained tuples and returns them in a new RDD.

```
val a = sc.parallelize(List("dog", "tiger", "lion", "cat",
"panther", "eagle"), 2)
val b = a.map(x => (x.length, x))
b.keys.collect
```

```
16/08/30 07:25:04 INFO scheduler.DAGScheduler: ResultStage 13 (collect at <console>:28) finished in 0.089 s
16/08/30 07:25:04 INFO scheduler.DAGScheduler: Job 10 finished: collect at <console>:28, took 0.131419 s
res9: Array[Int] = Array(3, 5, 4, 3, 7, 5)
```

union

```
val seta = sc.parallelize(1 to 10)
val setb = sc.parallelize(5 to 15)
(seta union setb).collect
```

```
16/08/30 07:26:33 INFO scheduler.DAGScheduler: ResultStage 14 (collect at <console>:26) finished in 0.143 s
16/08/30 07:26:33 INFO scheduler.DAGScheduler: Job 11 finished: collect at <console>:26, took 0.318252 s
res10: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)
```

zip

Joins two RDDs by combining the i-th of either partition with each other. The resulting RDD will consist of two-component tuples which are interpreted as key-value pairs by the methods provided by the PairRDDFunctions extension.

```
val a = sc.parallelize(1 to 100, 3)
val b = sc.parallelize(101 to 200, 3)
a.zip(b).collect
```

```
16/08/30 07:30:02 INFO scheduler.TaskSetManager: Finished task 2.0 in stage 15.0 (TID 34) in 35 ms on localh
ost (3/3)
16/08/30 07:30:02 INFO scheduler.TaskSchedulerImpl: Removed TaskSet 15.0, whose tasks have all completed, fr
om pool
res11: Array[(Int, Int)] = Array((1,101), (2,102), (3,103), (4,104), (5,105), (6,106), (7,107), (8,108), (9,
109), (10,110), (11,111), (12,112), (13,113), (14,114), (15,115), (16,116), (17,117), (18,118), (19,119), (2
0,120), (21,121), (22,122), (23,123), (24,124), (25,125), (26,126), (27,127), (28,128), (29,129), (30,130),
(31,131), (32,132), (33,133), (34,134), (35,135), (36,136), (37,137), (38,138), (39,139), (40,140), (41,141)
, (42,142), (43,143), (44,144), (45,145), (46,146), (47,147), (48,148), (49,149), (50,150), (51,151), (52,15
2), (53,153), (54,154), (55,155), (56,156), (57,157), (58,158), (59,159), (60,160), (61,161), (62,162), (63,
163), (64,164), (65,165), (66,166), (67,167), (68,168), (69,169), (70,170), (71,171), (72,172), (73,173), (7
4,174), (75,175), (76,176), (77,177), (78...
scala>
```

Action

collect

Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.

```
val c = sc.parallelize(List("Gnu", "Cat", "Rat", "Dog", "Gnu", "Rat"),
2) c.collect
```

```
16/08/30 07:31:52 INFO scheduler.TaskSetManager: Finished task 1.0 in stage 16.0 (TID 36) in 7 ms on localho
st (2/2)
16/08/30 07:31:52 INFO scheduler.TaskSchedulerImpl: Removed TaskSet 16.0, whose tasks have all completed, fr
om pool
16/08/30 07:31:52 INFO scheduler.DAGScheduler: ResultStage 16 (collect at <console>:24) finished in 0.010 s
16/08/30 07:31:52 INFO scheduler.DAGScheduler: Job 13 finished: collect at <console>:24, took 0.018261 s
res12: Array[String] = Array(Gnu, Cat, Rat, Dog, Gnu, Rat)
```

collectAsMap

Similar to collect, but works on key-value RDDs and converts them into Scala maps to preserve their key-value structure.

```
val a = sc.parallelize(List(1, 2, 1, 3), 1)

val b = a.zip(a)

b.collectAsMap
```

```
16/08/30 07:33:33 INFO scheduler.TaskSetManager: Finished task 0.0 in stage 17.0 (TID 37) in 145 ms on local
host (1/1)
16/08/30 07:33:33 INFO scheduler.TaskSchedulerImpl: Removed TaskSet 17.0, whose tasks have all completed, fr
om pool
res13: scala.collection.Map[Int,Int] = Map(2 -> 2, 1 -> 1, 3 -> 3)
```

count

Return the number of elements in the dataset.

```
val c = sc.parallelize(List("Gnu", "Cat", "Rat", "Dog"), 2)

c.count

res2: Long = 4
```

```
16/08/30 07:34:52 INFO scheduler.TaskSetManager: Finished task 1.0 in stage 18.0 (TID 39) in 11 ms on localh
ost (2/2)
16/08/30 07:34:52 INFO scheduler.TaskSchedulerImpl: Removed TaskSet 18.0, whose tasks have all completed, fr
om pool
16/08/30 07:34:52 INFO scheduler.DAGScheduler: ResultStage 18 (count at <console>:24) finished in 0.040 s
16/08/30 07:34:52 INFO scheduler.DAGScheduler: Job 15 finished: count at <console>:24, took 0.054256 s
res14: Long = 4
```

reduce

Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.

```
val a = sc.parallelize(1 to 100, 3)

a.reduce(_+_)
```

```
16/08/30 08:32:49 INFO scheduler.TaskSchedulerImpl: Removed TaskSet 19.0, whose tasks have all completed, fr
om pool
16/08/30 08:32:49 INFO scheduler.DAGScheduler: ResultStage 19 (reduce at <console>:24) finished in 0.099 s
16/08/30 08:32:49 INFO scheduler.DAGScheduler: Job 16 finished: reduce at <console>:24, took 0.315500 s
res15: Int = 5050
```

take

Return an array with the first *n* elements of the dataset.

```
val b = sc.parallelize(List("dog", "cat", "ape", "salmon", "gnu"), 2)

b.take(2)
```

```
16/08/30 08:34:15 INFO scheduler.TaskSchedulerImpl: Removed TaskSet 20.0, whose tasks have all completed, fr
om pool
16/08/30 08:34:15 INFO scheduler.DAGScheduler: ResultStage 20 (take at <console>:24) finished in 0.035 s
16/08/30 08:34:15 INFO scheduler.DAGScheduler: Job 17 finished: take at <console>:24, took 0.142047 s
res16: Array[String] = Array(dog, cat)
```

```
val b = sc.parallelize(1 to 100, 5)

b.take(30)
```

```
16/08/30 08:35:57 INFO scheduler.TaskSchedulerImpl: Removed TaskSet 22.0, whose tasks have all completed, fr
om pool
16/08/30 08:35:58 INFO scheduler.DAGScheduler: ResultStage 22 (take at <console>:24) finished in 0.004 s
16/08/30 08:35:58 INFO scheduler.DAGScheduler: Job 19 finished: take at <console>:24, took 0.064874 s
res17: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
 24, 25, 26, 27, 28, 29, 30)
```

first

Return the first element of the dataset (similar to take(1)).

```
val c = sc.parallelize(List("Gnu", "Cat", "Rat", "Dog"), 2)

c.first
```

```
16/08/30 08:37:15 INFO scheduler.TaskSchedulerImpl: Removed TaskSet 23.0, whose tasks have all completed, fr
om pool
16/08/30 08:37:15 INFO scheduler.DAGScheduler: ResultStage 23 (first at <console>:24) finished in 0.006 s
16/08/30 08:37:15 INFO scheduler.DAGScheduler: Job 20 finished: first at <console>:24, took 0.016040 s
res18: String = Gnu
```

countByValue

Returns a map that contains all unique values of the RDD and their respective occurrence counts.

```
val b = sc.parallelize(List(1,2,3,4,5,6,7,8,2,4,2,1,1,1,1,1))

b.countByValue
```

```
16/08/30 08:38:18 INFO scheduler.DAGScheduler: ResultStage 25 (countByValue at <console>:24) finished in 0.0
40 s
16/08/30 08:38:18 INFO scheduler.DAGScheduler: Job 21 finished: countByValue at <console>:24, took 0.495353
s
res19: scala.collection.Map[Int,Long] = Map(5 -> 1, 1 -> 6, 6 -> 1, 2 -> 3, 7 -> 1, 3 -> 1, 8 -> 1, 4 -> 2)
```

lookup

Scans the RDD for all keys that match the provided value and returns their values as a Scala sequence.

```
val a = sc.parallelize(List("dog", "tiger", "lion", "cat",
"panther", "eagle"), 2)
val b = a.map(x => (x.length, x))

b.lookup(5)
```

```
16/08/30 08:39:47 INFO scheduler.TaskSchedulerImpl: Removed TaskSet 26.0, whose tasks have all completed, fr
om pool
16/08/30 08:39:47 INFO scheduler.DAGScheduler: ResultStage 26 (lookup at <console>:28) finished in 0.087 s
16/08/30 08:39:47 INFO scheduler.DAGScheduler: Job 22 finished: lookup at <console>:28, took 0.188596 s
res20: Seq[String] = WrappedArray(tiger, eagle)
```

max

Returns the largest element in the RDD

```
val y = sc.parallelize(10 to 30)

y.max
```

```
16/08/30 08:41:08 INFO scheduler.TaskSchedulerImpl: Removed TaskSet 27.0, whose tasks have all completed, fr
om pool
16/08/30 08:41:08 INFO scheduler.DAGScheduler: ResultStage 27 (max at <console>:24) finished in 0.042 s
16/08/30 08:41:08 INFO scheduler.DAGScheduler: Job 23 finished: max at <console>:24, took 0.115601 s
res21: Int = 30
```

```
val a = sc.parallelize(List((10, "dog"), (3, "tiger"), (9, "lion"),
(18, "cat")))
a.max
```

```
16/08/30 08:42:29 INFO scheduler.TaskSchedulerImpl: Removed TaskSet 28.0, whose tasks have all completed, fr
om pool
16/08/30 08:42:29 INFO scheduler.DAGScheduler: ResultStage 28 (max at <console>:24) finished in 0.031 s
16/08/30 08:42:29 INFO scheduler.DAGScheduler: Job 24 finished: max at <console>:24, took 0.054414 s
res22: (Int, String) = (18,cat)
```

min

Returns the smallest element in the RDD

```
val y = sc.parallelize(10 to 30)
y.min
```

```
16/08/30 08:43:49 INFO executor.Executor: Finished task 0.0 in stage 29.0 (TID 53). 1031 bytes result sent t
o driver
16/08/30 08:43:49 INFO scheduler.DAGScheduler: ResultStage 29 (min at <console>:24) finished in 0.075 s
16/08/30 08:43:49 INFO scheduler.DAGScheduler: Job 25 finished: min at <console>:24, took 0.170125 s
res23: Int = 10
```

```
val a = sc.parallelize(List((10, "dog"), (3, "tiger"), (9, "lion"),
(8, "cat")))
a.min
```

```
16/08/30 08:45:21 INFO scheduler.TaskSchedulerImpl: Removed TaskSet 30.0, whose tasks have all completed, fr
om pool
16/08/30 08:45:21 INFO scheduler.DAGScheduler: ResultStage 30 (min at <console>:24) finished in 0.012 s
16/08/30 08:45:21 INFO scheduler.DAGScheduler: Job 26 finished: min at <console>:24, took 0.020695 s
res24: (Int, String) = (3,tiger)
```

mean

Calls stats and extracts the mean component.

```
val a = sc.parallelize(List(9.1, 1.0, 1.2, 2.1, 1.3, 5.0, 2.0, 2.1,
7.4, 7.5, 7.6, 8.8, 10.0, 8.9, 5.5), 3)
a.mean
```

```
16/08/30 08:46:49 INFO scheduler.TaskSchedulerImpl: Removed TaskSet 31.0, whose tasks have all completed, fr
om pool
16/08/30 08:46:49 INFO scheduler.DAGScheduler: ResultStage 31 (mean at <console>:24) finished in 0.425 s
16/08/30 08:46:49 INFO scheduler.DAGScheduler: Job 27 finished: mean at <console>:24, took 0.526866 s
res25: Double = 5.3
```

variance

Calls stats and extracts either variance-component or corrected sampleVariance-component.

```
val a = sc.parallelize(List(9.1, 1.0, 1.2, 2.1, 1.3, 5.0, 2.0, 2.1, 7.4,
7.5, 7.6, 8.8, 10.0, 8.9, 5.5), 3)
a.variance
```

```
16/08/30 08:48:05 INFO scheduler.TaskSchedulerImpl: Removed TaskSet 32.0, whose tasks have all completed, fr
om pool
16/08/30 08:48:05 INFO scheduler.DAGScheduler: ResultStage 32 (variance at <console>:24) finished in 0.018 s
16/08/30 08:48:05 INFO scheduler.DAGScheduler: Job 28 finished: variance at <console>:24, took 0.032808 s
res26: Double = 10.605333333333332
```

PairRDD

countByKey

Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.

```
val c = sc.parallelize(List((3, "Gnu"), (3, "Yak"), (5, "Mouse"),
(3, "Dog")), 2)
c.countByKey
```

```
16/08/30 08:49:39 INFO scheduler.TaskSchedulerImpl: Removed TaskSet 34.0, whose tasks have all completed, fr
om pool
16/08/30 08:49:39 INFO scheduler.DAGScheduler: ResultStage 34 (countByKey at <console>:24) finished in 0.073
 s
16/08/30 08:49:39 INFO scheduler.DAGScheduler: Job 29 finished: countByKey at <console>:24, took 0.384196 s
res27: scala.collection.Map[Int,Long] = Map(3 -> 3, 5 -> 1)
```

groupByKey

When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs.

```
val a = sc.parallelize(List("dog", "tiger", "lion", "cat",
"spider", "eagle"), 2)
val b = a.keyBy(_.length)
b.groupByKey.collect
```

```
16/08/30 08:51:02 INFO executor.Executor: Finished task 1.0 in stage 36.0 (TID 68). 1703 bytes result sent t
o driver
16/08/30 08:51:02 INFO scheduler.DAGScheduler: ResultStage 36 (collect at <console>:26) finished in 0.042 s
16/08/30 08:51:02 INFO scheduler.DAGScheduler: Job 30 finished: collect at <console>:26, took 0.415007 s
res28: Array[(Int, Iterable[String])] = Array((4,CompactBuffer(lion)), (6,CompactBuffer(spider)), (3,Compact
Buffer(dog, cat)), (5,CompactBuffer(tiger, eagle)))
```

reduceByKey

When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) => V

```
val a = sc.parallelize(List("dog", "cat", "owl", "gnu", "ant"),

2) , val b = a.map(x => (x.length, x)) , b.reduceByKey(_ +

).collect
```

```
16/08/30 08:52:58 INFO scheduler.TaskSetManager: Finished task 1.0 in stage 38.0 (TID 72) in 25 ms on localh
ost (2/2)
16/08/30 08:52:58 INFO scheduler.DAGScheduler: ResultStage 38 (collect at <console>:28) finished in 0.033 s
16/08/30 08:52:58 INFO scheduler.DAGScheduler: Job 31 finished: collect at <console>:28, took 0.413629 s
res29: Array[(Int, String)] = Array((3,dogcatowlgnuant))
```

foldByKey

Very similar to *fold*, but performs the folding separately for each key of the RDD. This function is only available if the RDD consists of two-component tuples.

```
val deptEmployees =
  List
  (
   ("dept1",("kumar1",1000.0)),
   ("dept1",("kumar2",1200.0)),
   ("dept2",("kumar3",2200.0)),
   ("dept2",("kumar4",1400.0)),
   ("dept2",("kumar5",1000.0)),
   ("dept2",("kumar6",800.0)),
   ("dept1",("kumar7",2000.0)),
   ("dept1",("kumar8",1000.0)),
   ("dept1",("kumar9",500.0))
   )


val employeeRDD = sc.makeRDD(deptEmployees)


val maxByDept =
employeeRDD.foldByKey(("dummy",Double.MinValue))((acc,element) => if(acc._2
> element._2)acc else element)

println("Maximum salaries in each dept" + maxByDept.collect().toList)
```

```
16/08/30 08:56:35 INFO executor.Executor: Finished task 0.0 in stage 40.0 (TID 74). 1375 bytes result sent t
o driver
16/08/30 08:56:35 INFO scheduler.DAGScheduler: ResultStage 40 (collect at <console>:28) finished in 0.087 s
16/08/30 08:56:35 INFO scheduler.DAGScheduler: Job 32 finished: collect at <console>:28, took 0.351474 s
Maximum salaries in each deptList((dept2,(kumar3,2200.0)), (dept1,(kumar7,2000.0)))
```

cogroup

When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable<V>, Iterable<W>)) tuples.

```
val a = sc.parallelize(List(1, 2, 1, 3),
1) . val b = a.map((_, "b")) . val c =
a.map((_, "c")) . b.cogroup(c).collect
```

```
16/08/30 08:58:50 INFO executor.Executor: Finished task 0.0 in stage 43.0 (TID 77). 2177 bytes result sent t
o driver
16/08/30 08:58:50 INFO scheduler.DAGScheduler: ResultStage 43 (collect at <console>:28) finished in 0.249 s
16/08/30 08:58:50 INFO scheduler.DAGScheduler: Job 33 finished: collect at <console>:28, took 0.668327 s
res31: Array[(Int, (Iterable[String], Iterable[String]))] = Array((1,(CompactBuffer(b, b),CompactBuffer(c, c
))), (3,(CompactBuffer(b),CompactBuffer(c))), (2,(CompactBuffer(b),CompactBuffer(c))))
```

```
val x = sc.parallelize(List((1, "apple"), (2, "banana"), (3, "orange"),
(4, . "kiwi")), 2)
val y = sc.parallelize(List((5, "computer"), (1, "laptop"), (1,
"desktop"), . (4, "iPad")), 2)
x.cogroup(y).collect
```

```
16/08/30 09:00:55 INFO executor.Executor: Finished task 1.0 in stage 46.0 (TID 83). 2189 bytes result sent t
o driver
16/08/30 09:00:55 INFO scheduler.DAGScheduler: ResultStage 46 (collect at <console>:26) finished in 0.123 s
16/08/30 09:00:55 INFO scheduler.DAGScheduler: Job 34 finished: collect at <console>:26, took 0.624333 s
res32: Array[(Int, (Iterable[String], Iterable[String]))] = Array((4,(CompactBuffer(kiwi),CompactBuffer(iPad
))), (2,(CompactBuffer(banana),CompactBuffer())), (1,(CompactBuffer(apple),CompactBuffer(laptop, desktop))),
 (3,(CompactBuffer(orange),CompactBuffer())), (5,(CompactBuffer(),CompactBuffer(computer))))
```

Join

Performs an inner join using two key-value RDDs.

```
val a = sc.parallelize(List("dog", "salmon", "salmon", "rat",
"elephant"), . 3)
val b = a.keyBy(_.length)
```

```
val c =
sc.parallelize(List("dog","cat","gnu","salmon","rabbit","turkey","wolf","be
ar","bee"), 3)
val d = c.keyBy(_.length)
b.join(d).collect
```

```
16/08/30 09:02:41 INFO scheduler.TaskSchedulerImpl: Removed TaskSet 49.0, whose tasks have all completed, fr
om pool
16/08/30 09:02:41 INFO scheduler.DAGScheduler: ResultStage 49 (collect at <console>:30) finished in 0.470 s
16/08/30 09:02:41 INFO scheduler.DAGScheduler: Job 35 finished: collect at <console>:30, took 1.019005 s
res33: Array[(Int, (String, String))] = Array((6,(salmon,salmon)), (6,(salmon,rabbit)), (6,(salmon,turkey)),
 (6,(salmon,salmon)), (6,(salmon,rabbit)), (6,(salmon,turkey)), (3,(dog,dog)), (3,(dog,cat)), (3,(dog,gnu)),
 (3,(dog,bee)), (3,(rat,dog)), (3,(rat,cat)), (3,(rat,gnu)), (3,(rat,bee)))
```

leftOuterJoin

Performs an left outer join using two key-value RDDs.

```
val a = sc.parallelize(List("dog", "salmon", "salmon", "rat",
"elephant"), 3)
val b = a.keyBy(_.length)
val c =
sc.parallelize(List("dog","cat","gnu","salmon","rabbit","turkey","wolf","be
ar","bee"), 3)
val d = c.keyBy(_.length)
b.leftOuterJoin(d).collect
```

```
16/08/30 09:03:59 INFO scheduler.DAGScheduler: Job 36 finished: collect at <console>:30, took 0.675634 s
16/08/30 09:03:59 INFO scheduler.TaskSchedulerImpl: Removed TaskSet 52.0, whose tasks have all completed, fr
om pool
res34: Array[(Int, (String, Option[String]))] = Array((6,(salmon,Some(salmon))), (6,(salmon,Some(rabbit))),
(6,(salmon,Some(turkey))), (6,(salmon,Some(salmon))), (6,(salmon,Some(rabbit))), (6,(salmon,Some(turkey))),
(3,(dog,Some(dog))), (3,(dog,Some(cat))), (3,(dog,Some(gnu))), (3,(dog,Some(bee))), (3,(rat,Some(dog))), (3,
(rat,Some(cat))), (3,(rat,Some(gnu))), (3,(rat,Some(bee))), (8,(elephant,None))
```

rightOuterJoin

Performs an right outer join using two key-value RDDs.

```
val a = sc.parallelize(List("dog", "salmon", "salmon", "rat",
"elephant"), 3)
val b = a.keyBy(_.length)
val c =
sc.parallelize(List("dog","cat","gnu","salmon","rabbit","turkey","wolf","be
ar","bee"), 3)
val d = c.keyBy(_.length)
```

```
b.rightOuterJoin(d).collect
```

```
16/08/30 09:05:30 INFO scheduler.TaskSchedulerImpl: Removed TaskSet 55.0, whose tasks have all completed, fr
om pool
16/08/30 09:05:30 INFO scheduler.DAGScheduler: ResultStage 55 (collect at <console>:30) finished in 0.181 s
16/08/30 09:05:30 INFO scheduler.DAGScheduler: Job 37 finished: collect at <console>:30, took 1.003812 s
res35: Array[(Int, (Option[String], String))] = Array((6,(Some(salmon),salmon)), (6,(Some(salmon),rabbit)),
(6,(Some(salmon),turkey)), (6,(Some(salmon),salmon)), (6,(Some(salmon),rabbit)), (6,(Some(salmon),turkey)),
(3,(Some(dog),dog)), (3,(Some(dog),cat)), (3,(Some(dog),gnu)), (3,(Some(dog),bee)), (3,(Some(rat),dog)), (3,
(Some(rat),cat)), (3,(Some(rat),gnu)), (3,(Some(rat),bee)), (4,(None,wolf)), (4,(None,bear)))
```

keyBy

Constructs two-component tuples (key-value pairs) by applying a function on each data item. The result of the function becomes the key and the original data item becomes the value of the newly created tuples.

```
val a = sc.parallelize(List("dog", "salmon", "salmon", "rat",
"elephant"), 3)
val b = a.keyBy(_.length)
b.collect
```

```
16/08/30 09:14:29 INFO scheduler.TaskSetManager: Finished task 2.0 in stage 56.0 (TID 113) in 7 ms on localh
ost (3/3)
16/08/30 09:14:29 INFO scheduler.TaskSchedulerImpl: Removed TaskSet 56.0, whose tasks have all completed, fr
om pool
16/08/30 09:14:29 INFO scheduler.DAGScheduler: ResultStage 56 (collect at <console>:26) finished in 0.053 s
16/08/30 09:14:29 INFO scheduler.DAGScheduler: Job 38 finished: collect at <console>:26, took 0.073531 s
res36: Array[(Int, String)] = Array((3,dog), (6,salmon), (6,salmon), (3,rat), (8,elephant))
```

mapValues

Takes the values of a RDD that consists of two-component tuples, and applies the provided function to transform each value. Then, it forms new two-component tuples using the key and the transformed value and stores them in a new RDD.

```
val a = sc.parallelize(List("dog", "tiger", "lion", "cat",
"panther", "eagle"), 2)
val b = a.map(x => (x.length, x))
b.mapValues("x" + _ + "x").collect
```

```
16/08/30 09:21:21 INFO executor.Executor: Finished task 1.0 in stage 57.0 (TID 115). 1118 bytes result sent
to driver
16/08/30 09:21:21 INFO scheduler.DAGScheduler: ResultStage 57 (collect at <console>:28) finished in 0.082 s
16/08/30 09:21:21 INFO scheduler.DAGScheduler: Job 39 finished: collect at <console>:28, took 0.182411 s
res37: Array[(Int, String)] = Array((3,xdogx), (5,xtigerx), (4,xlionx), (3,xcatx), (7,xpantherx), (5,xeaglex
))
```

cache

The cache( ) method is a shorthand for using the default storage level, which is StorageLevel.MEMORY_ONLY (store deserialized objects in memory)

```
val c = sc.parallelize(List("Gnu", "Cat", "Rat", "Dog", "Gnu", "Rat"),
2) c.getStorageLevel
```

```
scala> val c = sc.parallelize(List("Gnu", "Cat", "Rat", "Dog", "Gnu", "Rat"), 2)
c: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[99] at parallelize at <console>:21

scala> c.getStorageLevel
res38: org.apache.spark.storage.StorageLevel = StorageLevel(false, false, false, false, 1)
```

```
c.cache

c.getStorageLevel
```

```
scala> c.cache
res39: c.type = ParallelCollectionRDD[99] at parallelize at <console>:21

scala> c.getStorageLevel
res40: org.apache.spark.storage.StorageLevel = StorageLevel(false, true, false, true, 1)
```

repartition

Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.

```
val rdd = sc.parallelize(List(1, 2, 10, 4, 5, 2, 1, 1, 1), 3)

rdd.partitions.length

val rdd2 = rdd.repartition(5)

rdd2.partitions.length
```

```
scala> val rdd = sc.parallelize(List(1, 2, 10, 4, 5, 2, 1, 1, 1), 3)
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[102] at parallelize at <console>:21

scala> rdd.partitions.length
res43: Int = 3

scala> val rdd2 = rdd.repartition(5)
rdd2: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[106] at repartition at <console>:23

scala> rdd2.partitions.length
res44: Int = 5
```

Developing with Spark

REPL

Example: Counting the occurrence of lines having a particular word in it

```
val textFile = sc.textFile("file:/home/cloudera/datasets/Joyce.txt")
```

```
textFile.count() //Return the number of elements in the dataset
```

```
16/08/30 09:34:43 INFO executor.Executor: Finished task 0.0 in stage 59.0 (TID 136). 2082 bytes result sent
to driver
16/08/30 09:34:43 INFO scheduler.DAGScheduler: ResultStage 59 (count at <console>:24) finished in 2.238 s
16/08/30 09:34:43 INFO scheduler.DAGScheduler: Job 41 finished: count at <console>:24, took 2.310447 s
res45: Long = 33056
```

```
textFile.first() //Return the first element of the dataset
```

```
16/08/30 09:35:03 INFO executor.Executor: Finished task 0.0 in stage 60.0 (TID 137). 2101 bytes result sent
to driver
16/08/30 09:35:03 INFO scheduler.DAGScheduler: ResultStage 60 (first at <console>:24) finished in 0.035 s
16/08/30 09:35:03 INFO scheduler.DAGScheduler: Job 42 finished: first at <console>:24, took 0.091601 s
res46: String = The Project Gutenberg EBook of Ulysses, by James Joyce
```

```
val linesWithJoyce = textFile.filter(line => line.contains("Joyce"))

linesWithJoyce.count() //How many lines contains "Joyce"
```

```
16/08/30 09:35:26 INFO executor.Executor: Finished task 0.0 in stage 61.0 (TID 138). 2082 bytes result sent
to driver
16/08/30 09:35:26 INFO scheduler.DAGScheduler: ResultStage 61 (count at <console>:26) finished in 0.064 s
16/08/30 09:35:26 INFO scheduler.DAGScheduler: Job 43 finished: count at <console>:26, took 0.094628 s
res47: Long = 4
```

Example: Add up the sizes of all the lines

```
val lineLength = textFile.map(s => s.length)
val totalLength = lineLength.reduce((a, b) => a + b) //Aggregate the
elements of the dataset using a function
```

```
16/08/30 09:40:54 INFO executor.Executor: Finished task 0.0 in stage 62.0 (TID 139). 2160 bytes result sent
to driver
16/08/30 09:40:54 INFO scheduler.DAGScheduler: ResultStage 62 (reduce at <console>:31) finished in 0.262 s
16/08/30 09:40:54 INFO scheduler.DAGScheduler: Job 44 finished: reduce at <console>:31, took 0.337723 s
totalLength: Int = 1506967
```

Example: To find out the line with maximum words

```
textFile.map(line => line.split(" ").size).reduce((a,b) => if(a>b) a
else , b)
```

```
16/08/30 09:41:48 INFO executor.Executor: Finished task 0.0 in stage 63.0 (TID 140). 2160 bytes result sent
to driver
16/08/30 09:41:48 INFO scheduler.DAGScheduler: ResultStage 63 (reduce at <console>:30) finished in 1.048 s
16/08/30 09:41:48 INFO scheduler.DAGScheduler: Job 45 finished: reduce at <console>:30, took 1.070554 s
res48: Int = 22
```