SCALA

**“Quality means doing it Right,
When No one is looking”**

Launching an Application with an Object

- There are two ways to create a launching point for any application

1. Define an object that extends the App trait. In this approach command line arguments can be available through args object which is of Array[String] type from App trait.

2. Define an object with a properly defined main method

```
object Hello extends App {
if (args.length == 1)
println(s"Hello, ${args(0)}")
else
println("I didn't get your name.")
}
```

```
$ scalac
Hello.scala$ scala
Hello bhaskar Hello,
bhaskar
```

Launching an Application with an Object

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

Note that in both cases, Scala applications are launched from an object, not a class.

Creating Singletons with object

We can create Singleton objects in Scala with the object keyword. It is like defining static methods in a Java class and calling them without object creation. Since Scala doesn't support static keyword, we use singleton objects.

```
import java.util.Calendar
import java.text.SimpleDateFormat
object DateUtils {
// as "Thursday, November 29"
def getCurrentDate: String = getCurrentDateTime("EEEE, MMMM d")
def getCurrentTime: String = getCurrentDateTime("K:m aa")
// a common function used by other date/time functions
private def getCurrentDateTime(dateTimeFormat: String): String = {
val dateFormat = new SimpleDateFormat(dateTimeFormat)
val cal = Calendar.getInstance()
dateFormat.format(cal.getTime())
}}

Scala> DateUtils.getCurrentTime
Scala> DateUtils.getCurrentDate
```

Methods defined in an object instead of a class, can be called in the same way as a static method in Java

Creating Singletons with object

- A singleton can inherit methods from other classes/traits, which can't be done with statics.
- A singleton can exist within the scope of a surrounding class or method, just as Java can have inner classes
- It's also worth noting that a singleton doesn't have to be a companion, it's perfectly valid to define a singleton without also defining a companion class.

Companion object

Static methods of a class can be defined with Singleton object but instance methods (non static members) need to be defined in a class. If we define non static members in the class and static member in an object with same name as class then that object is known as companion object for that class.

```
// Pizza class
class Pizza (var crustType: String) {
override def toString = "Crust type is " + crustType
}
// companion object
object Pizza {
val CRUST_TYPE_THIN = "thin"
val CRUST_TYPE_THICK = "thick"
def getFoo = "Foo"
}
```

With the Pizza class and Pizza object defined in the same file assuming Pizza.scala, members of the Pizza object can be accessed just as static members of a Java class

Example

```
class Foo {
private val secret = 2
}
object Foo {
// access the private class field
'secret'
def double(foo: Foo) =
foo.secret * 2
}
object Driver extends App {
val f = new Foo
println(f.secret)    // not
allowed
println(Foo.double(f)) // prints
4
}
```

```
class Foo {
// access the private object field
'obj'
def printObj { println(s"I can see
${Foo.obj}") }
}
object Foo {
private val obj = "Foo's object"
}
object Driver extends App {
val f = new Foo
f.printObj
}
```

Similarly, instance member printObj can access the private field obj of the object Foo:

Creating Object Instances Without new Keyword

Scala code looks cleaner when we don't always have to use the new keyword to create a new instance of a class, like this

```
val a = Array(Person("Ram"), Person("Krishna"))
```

Two Ways for achieving

1. **Create a companion object for your class, and define an apply method in the companion object with the desired constructor signature.**
2. **Define our class as a case class.**

```
class Person {
var name: String = _   // _ sets null value to name
}
object Person {
def apply(name: String): Person = {
        var p = new Person
        p.name = name
        p
        }
}
(Or) case class Person (var name: String)
val bhaskar = Person("bhaskar")
val a = Array(Person("Ram"), Person("Krishna"))
```

✓ **Case class generates an apply method in a companion object**

Multiple constructors with additional apply methods

```
class Person {
var name = ""
var age = 0
}
object Person {
// a one-arg constructor
def apply(name: String): Person = {
        var p = new Person
        p.name = name
        p}
// a two-arg constructor
def apply(name: String, age: Int): Person = {
        var p = new Person
        p.name = name
        p.age = age
        p
}}
object main extends App{ val
bhaskar = Person("bhaskar")
val john = Person("John", 42)}
```

Case classes in Scala

- Case classes are nothing but similar to java beans or pojo or dto.
- Basic syntax of case class

```
case class Person(lastname: String, firstname: String, birthYear: Int)
```

- That’s it… this saves lot of code and time for developers.
- By default Case class parameters are val.
- Using of case class

```
val p = Person(“Ram", “Krishna", 1976)
// instead if the slightly more verbose:
val p = new Person(“Ram", “Krishna", 1976)
```

Important points about case class

- **Case classes can be pattern matched**

```
case class Person2(var name:String)
var a = Person2("bhaskar")
var b = Person2("Ram")
def fun(p:Person2) = p match {
case Person2("bhaskar") =>
println("bhaskar") case Person2("Ram")
=> println("Ram") case _ =>
println("default")
}
val f = fun(a)
```

- **Case classes automatically define hashcode and equals and**
- **Adds apply method in companion object automatically**
- **Case classes automatically define getter methods for the constructor arguments.**
- **Setters are not generated for case classes unless "var" is specified in the constructor argument, in which case you get the same getter/setter generation as regular classes**

Traits

A trait encapsulates method and field definitions, which can then be reused by mixing them into classes. Unlike class inheritance, in which each class must inherit from just one superclass, a class can mix in any number of trait

- **Traits are similar to interfaces in java, but in traits we can have default implementation**
- **When a class extends multiple traits, use extends for the first trait, and with for subsequent traits**

```
class Foo extends Trait1 with Trait2 with Trait3 with Trait4 { ...}
```

```
trait Vehicle {
def drive = { println("Driving") }
def race
}

class Car extends Vehicle {
def race ={ println("Racing the car") }
}
```

```
class Boat extends Vehicle {
override def drive= { println("float") }
def race= { println("Racing boat.") }
}
```

If we do not have default implementation in Trait then override keyword is not needed but if we have default implementation for a method in trait, we need verride keyword in extending class method definition

Traits

- **Limiting Which Classes Can Use a Trait by Inheritance**

```
class A
trait T extends A
class C extends A with T  //allows
class D extends T  // allows
class E
class F extends E with T // won't compile
```

- **To make sure a trait named MyTrait can only be mixed into a class that is a subclass of a type named BaseType, begin your trait with a this: BaseType => declaration**

```
trait MyTrait {
this: BaseType =>
def fun1
def fun2
}

class BaseType{... }
class SubType extends BaseType with MyTrait      { ... //allowed}//use abstract before
class
class Other extends MyTrait  {... } //not allowed
```

Abstract Class

- An abstract class can't be instantiated. It is made up of both abstract and non-abstract methods and fields
- Though Scala has abstract classes, it’s recommended to use traits instead of abstract classes to implement base behaviour because a class can extend only one abstract class, but it can implement multiple traits.
- Override Keyword Not Necessary for **var** Field. Override Keyword Necessary for **val** Field
- Abstract class will have constructor but not traits

```
abstract class Pet (var name: String) {
def speak // abstract
def ownerIsHome { println("excited") }
}
class Dog (name: String) extends Pet (name) {
def speak { println(“Boww..") }
override def ownerIsHome {
speak
}
}
```

```
trait CarTrait {
val door: Int    //abstract
var seat = 4    //concrete
}

class Car extends CarTrait {
override val door = 5  //override needed for val
seat = 5       //both var and override not needed
}

scala> val c = new Car()
scala> c.door
res0: Int = 5
scala> c.seat
res1: Int = 5
```

nterface VS Traits VS Abstract Classes

- Interfaces will not have constructors Traits will also not have constructors and thus trait declaration doesn’t accept constructor parameters. So below declaration is not valid

```
trait TestAbstract(val year:Int, var model:String){def move();
 def race()={println("race")};
}
```

- Where as abstract classes in Java can accept constructor Abstract classes in Scala also accepts the constructor parameters

```
abstract class TestAbstract(val year:Int, var model:String){def move();
 def race()={println("race")};
}
```

- In scala No need of abstract keyword prefixed for abstract methods in abstract class
- You can not instantiate either interface/trait or abstract class both in Java and Scala
- You can extend only one abstract class both in Java and Scala where as we can implement/extend multiple interfaces/traits in a single class in both Java and Scala

You can have doubt that if default implementation is provided in traits then there is a chance for “diamond problem”.

Scala handles this scenario very cleverly and it refers last value.

```
trait Base {
    def op: String
    }
trait Foo extends Base { override def op = "foo" }
trait Bar extends Base { override def op = "bar" }
trait Beer extends Base { override def op = "beer" }

class A extends Foo with Bar  //Here we can observe class A  extending
multiple traits.
class B extends Bar with Foo
(new A).op // res0: String = bar
(new B).op // res1: String = foo

class C extends Bar with Foo  with Beer
(new C).op // res2: String = beer
```

Exceptions

- Exceptions in scala are similar to exceptions in many other languages like Java. Instead of returning a value in the normal way, a method can terminate by throwing an exception.

- Throwing an exception looks the same as in Java. With help of throw key word.

- throw new IllegalArgumentException

- Handling exceptions using try catch blocks. The syntax of catch is quite different in Scala; you use pattern matching.

Example of try, catch and finally & Example for File Reading Line by Line

```
import java.io.FileReader
import java.io.FileNotFoundException
import java.io.IOException
import scala.io.Source


 object Test { def main(args: Array[String]) {
    try {
        val f = new FileReader("input.txt")    //args(0)
                BufferedReader br = new BufferedReader(fr);
                String s;
                while((s = br.readLine()) != null) {
                System.out.println(s);
}
                for(line <- Source.fromPath("myfile.txt").getLines())
                  println(line)

                 val writer = new PrintWriter(new File(args(1) ))

                      writer.write("Hello Scala")
                      writer.close()
    }
    catch {
        case ex: FileNotFoundException =>{ println("Missing file
exception")    }
        case ex: IOException => { println("IO Exception") }
    }
    finally {
         println("Exiting finally...") }
    } }
```

Mind the Catch Order

```
try {
        val str = “hello“
        println(str(31))
catch {
        case ex: Exception =>{ println(“Exception caught") }
        case ex: StringIndexOutOfBoundsException  => { println(“Invalid index") }
    }
```

The output from the previous code is shown here:

Exception caught

The first case matches Exception and all of its subclasses. When using multiple catch blocks, you must ensure that exceptions are being handled by the catch blocks you intend.

If in case of java, above example will raise compile time exception as
“exception java.lang.StringIndexOutOfBoundsException has already been caught.”

Methods with Exceptions

- Use the **@throws** annotation to declare the exception(s) that can be thrown.

- To declare that one exception can be thrown, place the annotation just before the method signature

```
@throws(classOf[IOException])
@throws(classOf[LineUnavailableException])
@throws(classOf[UnsupportedAudioFileException])
def playSoundFileWithJavaAudio {
// exception throwing code here ...
}
```

Custom Exceptions

Defining a custom exception in scala is very simple, as shown below

```
case class customException(msg:String) extends Exception(msg)
```

Using custom exception in code.

```
try{
        val stateCapitals = Map( "Alabama" -> "Montgomery",
        "Alaska" -> "Juneau",
        "Wyoming" -> "Cheyenne")
println("Alabama: " + stateCapitals.get ("Alabama").get)
}
catch{
case ex:Exception=>throw new CustomException("whatever")
}
```

Collections in Scala

- Scala has a rich set of collection library. Main categories of collection classes
  - ✓ Sequence
  - ✓ Map
  - ✓ Set
- A **sequence** is a linear collection of elements and may be indexed(Arraylist) or linear (a linked list).
- A **map** contains a collection of key/value pairs, like a Java Map.
- A **set** is a collection that contains no duplicate elements
- Other collection types are **Stack, Queue,** and Range. Few other classes that act like collections **tuples, enumerations**, and the **Option/Some/None**
- **Option** acts as a collection that contains zero or one elements.
- The **Some** class and **None** object extend Option.
- Some is a container for one element, and None holds zero elements.
- Tuple supports a heterogeneous collection of elements.
- There is no one “Tuple” class; tuples are implemented as case classes

ranging from Tuple1 to Tuple22, which support 1 to 22

Collections in Scala

- **Scala favours immutable collections and they are thread safe, even though mutable versions are also provided. If you want to modify a collection and your operations on the collection are all within a single thread, you can choose a mutable collection.**

- **You can choose between these versions by selecting a class in one of these two packages: scala.collection.mutable or scala.collection.immutable.**

List

Scala Lists are quite similar to arrays which means, all the elements of a list have the same type. Scala List class is immutable, so its size as well as the elements it refers to can't change. Example for defining a list. It's implemented as a linked list

```
// List of Strings
val fruit: List[String] = List("apples", "oranges", "pears")
// List of Integers
val nums: List[Int] = List(1, 2, 3, 4)
// Empty List
val empty: List[Nothing] = List()
// Two dimensional
val dim: List[List[Int]] = List( List(1, 0, 0), List(0, 1, 0), List(0, 0, 1) )
```

All the above lists can be defined as below way also:

```
val fruit = "apples" :: ("oranges" :: ("pears" :: Nil))       // List of Strings
val nums = 1 :: (2 :: (3 :: (4 :: Nil)))              // List of Integers
val empty = Nil                          // Empty List.
val dim = (1 :: (0 :: (0 :: Nil))) :: (0 :: (1 :: (0 :: Nil))) :: (0 :: (0 :: (1 :: Nil))) :: Nil // Two dimensional list
```

- **Basic operations in list**

```
val fruit = "apples" :: ("oranges" :: ("pears" :: Nil))
val nums = Nil
    println( "Head of fruit : " + fruit.head )
    println( "Tail of fruit : " + fruit.tail )
    println( "Check if nums is empty : " + nums.isEmpty )
```

Output:

```
Head of fruit : apples
Tail of fruit : List(oranges, pears)
Check if nums is empty : true
```

List is immutable, so we can't actually add elements to it. To work with a List, the general approach is to prepend items to the list with the :: method.

```
var x = List(2)
    scala> x = 1 :: x
    x: List[Int] = List(1, 2)
    scala> x = 0 :: x
    x: List[Int] = List(0, 1, 2)
```

```
scala> val x = List(1)
x: List[Int] = List(1)
scala> val y = 0 +: x
y: List[Int] = List(0, 1)
scala> val y = x :+ 2
y: List[Int] = List(1, 2)
```

Though using :: is very common, there are additional methods that let you prepend or append single elements to a List

- If we need mutable lists, use a ListBuffer, and convert the ListBuffer to a List when needed

```
import scala.collection.mutable.ListBuffer
var fruits = new ListBuffer[String]()
// add one element at a time to the ListBuffer
fruits += "Apple"
fruits += "Banana"
fruits += "Orange“
// add multiple elements
fruits += ("Strawberry", "Kiwi", "Pineapple")
// remove one element
fruits -= "Apple"
// remove multiple elements
fruits -= ("Banana", "Orange")
// remove multiple elements specified by another sequence
fruits --= Seq("Kiwi", "Pineapple")
// convert the ListBuffer to a List when you need to
val fruitsList = fruits.toList
```

- There are many important methods in scala, few of them are
- Concatenating list(You can use either ::: operator or ++ method or List.concat() method)
- List.reverse, List.addString, List.contains, List.copyToArray, List.distinct, List.drop etc…

Set

- Set is a collection that contains no duplicate elements. Example for defining a Set.
- By default we will get the immutable set created

```
// Empty set of integer type
var s : Set[Int] = Set()
// Set of integer type
var s : Set[Int] = Set(1,3,5,7)
Or
 var s = Set(1,3,5,7)
```

- Basic operations in Set

```
val fruit = Set("apples", "oranges", "pears")
val nums: Set[Int] = Set()
println( "Head of fruit : " + fruit.head )      // Head of fruit : apples
println( "Tail of fruit : " + fruit.tail )      // Tail of fruit : Set(oranges,
pears)
println( "Check if fruit is empty : " + fruit.isEmpty )     // Check if fruit is
empty : false
println( "Check if nums is empty : " + nums.isEmpty ) // Check if nums is
empty : true
```

- **Mutable set can be created with** scala.collection.mutable.Set and **use var.**

var s2 = scala.collection.mutable.Set(1,2,3,4)

- **Important methods in Set**

- **Concatenating Sets(You can use either ++ operator or Set.++() method to concatenate)**

```
val fruit1 = Set("apples", "oranges", "pears")
val fruit2 = Set("mangoes", "banana")
 // use two or more sets with ++ as operator
var fruit = fruit1 ++ fruit2 println( "fruit1 ++ fruit2 : " + fruit )
// use two sets with ++ as method

fruit = fruit1.++(fruit2) println( "fruit1.++(fruit2) : " + fruit )
```

- **Other methods in set.**

- **Set.min, Set.mkString, Set.remove, Set.size, Set.take etc…**

Map

- Scala map is a collection of key/value pairs. Any value can be retrieved based on its key.
- Keys are unique in the Map, but values need not be unique.
- By default Immutable maps will be created, for mutable we need to import it and use **var**.

```
scala> val states = Map("AL" -> "Alabama", "AK" -> "Alaska")
states: scala.collection.immutable.Map[String,String] =
Map(AL -> Alabama, AK -> Alaska)
scala> var states = collection.mutable.Map[String, String]()
states: scala.collection.mutable.Map[String,String] = Map()
scala> states += ("AL" -> "Alabama")
res0: scala.collection.mutable.Map[String,String] = Map(AL -> Alabama)
```

Printing keys and values in a map.

```
val colors = Map("red" -> "#FF0000",
                 "azure" -> "#F0FFFF",
                 "peru" -> "#CD853F")

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

Result:

```
Key = red Value = #FF0000
Key = azure Value = #F0FFFF
Key = peru Value = #CD853F
```

Tuples

Tuple is nothing but fixed number of items together so that they can be passed around as a single object/value.

Example for tuple

```
val t = (1, "hello")
```

or

```
val t1 = new Tuple3(1, "hello", ‘a’)
```

The datatype of tuple for t is Tuple2[Int, String] and for t1 is Tuple3[Int, String,Char]

Accessing elements in tuple

```
val t = (4,3,2,1)
val sum = t._1 + t._2 + t._3 + t._4
```

Iterator

- Iterator is a way to access the elements of a collection one by one. We can iterate using next and hasNext methods.

Example

```
scala> val list = List("one","two","three")
list: List[String] = List(one, two, three)
scala> val it = list.toIterator
it: Iterator[String] = non-empty iterator
scala> while(it.hasNext) {
println(it.next)
}
one
two
Three
```

- But using Iterator for collections is not good idea, as we have **map** and **foreach** methods in collections to iterate them.
- Best use case of using iterator is when reading file bcoz putting all file data into in-memory is not a good idea.

Scala Collections Examples

(electric boogaloo)
Processing collections with functional programming

```
val lst = List(1, 2, 3)
list.foreach(x => println(x)) // prints 1, 2, 3
list.foreach(println) // same
list.map(x => x + 2) // returns a new List(3, 4, 5)
list.map(_ + 2) // same
list.filter(x => x % 2 == 1)// returns a new List(1, 3)
list.filter(_ % 2 == 1) // same
list.reduce((x, y) => x + y) // => 6
list.reduce(_ + _) // same
```

All of these leave the list unchanged as it is immutable.

Using Java code in scala

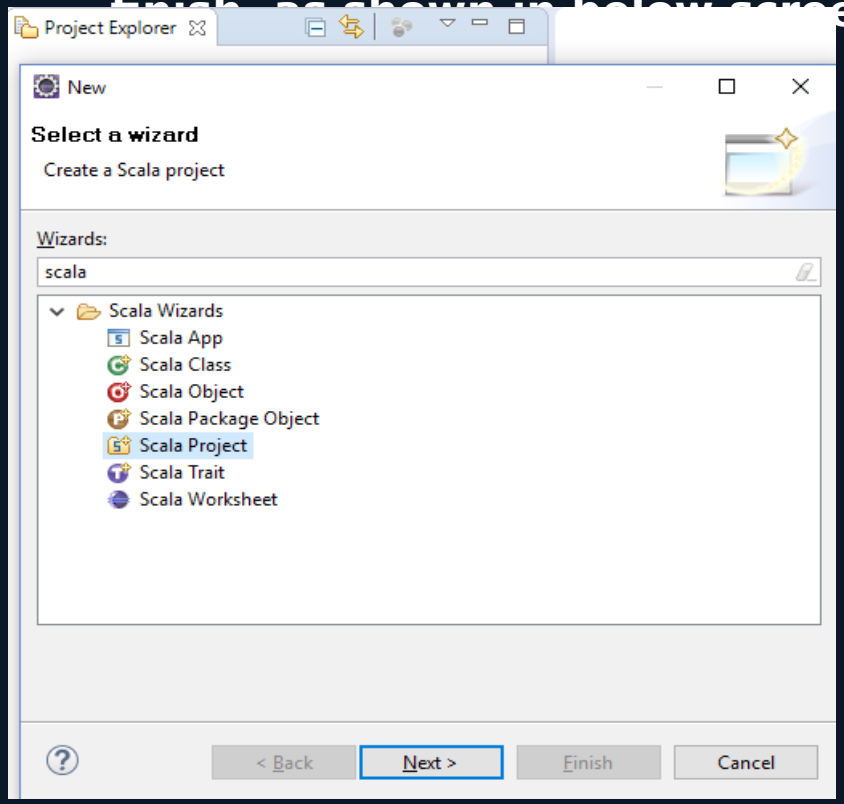
- One of Scala's strengths is that it makes it very easy to interact with Java code. All classes from the java.lang package are imported by default, while others need to be imported explicitly
- We can use Java classes in a Scala program directly, but you would of course have to use Scala syntax.

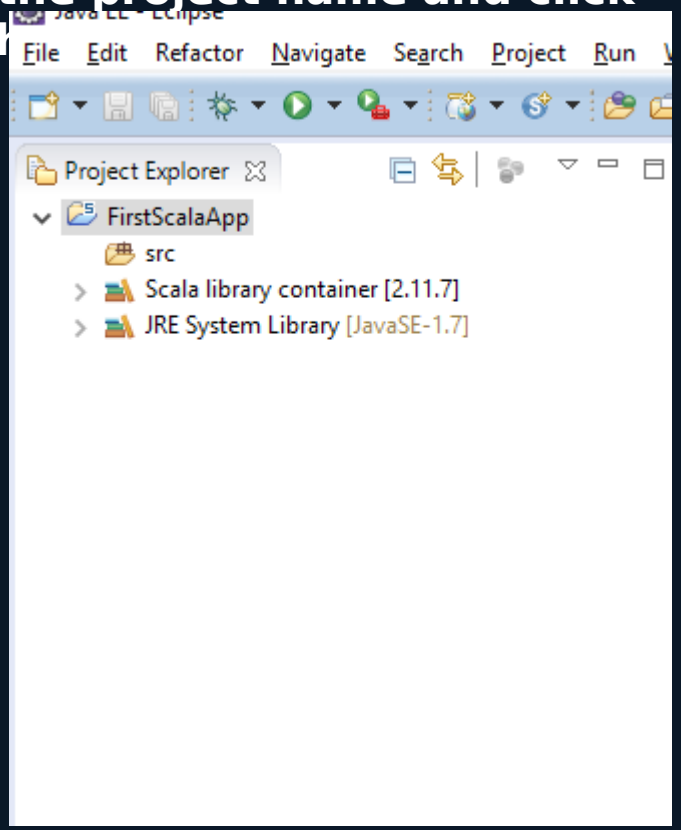
```
import java.util.{Date, Locale}
import java.text.DateFormat
import java.text.DateFormat._

      object FrenchDate {
            def main(args: Array[String]) {
            val now = new Date
            val df = getDateInstance(LONG, Locale.FRANCE)
            println(df format now)
}
}
```

Building scala applications using maven build

- Create a new Scala project in Eclipse for Scala.
- Right click on project explorer -> new -> type scala in search box
- Now select scala project and give the project name and click finish, as shown in below screen sh





Building scala applications using maven build

- Now right click on the project, and go to Configure.
- Select Convert to Maven Project.
- The project will convert to a Maven for Scala.
- Open the pom.xml (double click).
- Select the Dependencies tab.
- Choose Add…
- Insert the dependence as per your project requirement.
- Here we are inserting spark dependency.

```
Group ID: org.apache.spark
Artifact ID: spark-core_2.11
Version: 1.3.0
```

- Save the changes, wait for a download to complete.
- Under a src (default package), create a New Scala File, name it "SparkPi"
- To test the spark, cut-and-paste the following code:

Building scala applications using maven build

```
import scala.math.random
import org.apache.spark._
/** Computes an approximation to pi */
object SparkPi {
def main(args: Array[String]) {
val conf = new SparkConf().setAppName("Spark Pi")
.setMaster("local")

val spark = new SparkContext(conf)
val slices = if (args.length > 0) args(0).toInt else 2
val n = math.min(100000L * slices, Int.MaxValue).toInt // avoid
overflow
val count = spark.parallelize(1 until n, slices).map { i =>
val x = random * 2 - 1
val y = random * 2 - 1
if (x*x + y*y < 1) 1 else 0
}.reduce(_ + _)
println("Pi is roughly " + 4.0 * count / n)
spark.stop()
}
}
```

Run it as “Scala Application”

- The program should display a PI value among the other lines.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>SimpleSpark</groupId>
  <artifactId>SimpleSpark</artifactId>
  <version>0.0.1-SNAPSHOT</version>
<build>
<sourceDirectory>src</sourceDirectory>
<plugins>

     <plugin>
     <groupId>org.scala-tools</groupId>
     <artifactId>maven-scala-plugin</artifactId>
     <version>2.15.2</version>
     <executions>
          <execution>
               <goals>
               <goal>compile</goal>
               <goal>testCompile</goal>
               </goals>
          </execution>
     </executions>
     </plugin>

     <plugin>
          <artifactId>maven-compiler-plugin</artifactId>
          <configuration>
               <source>1.5</source>
               <target>1.5</target>
          </configuration>
     </plugin>
```

```
     <plugin>
          <artifactId>maven-assembly-plugin</artifactId>
          <configuration>
               <descriptorRefs>
               <descriptorRef>jar-with-dependencies</descriptorRef>
               </descriptorRefs>
          </configuration>

          <executions>
               <execution>
                    <id>make-assembly</id>
                    <phase>package</phase>
                    <goals>
                         <goal>single</goal>
                    </goals>
               </execution>
          </executions>
     </plugin>
     </plugins>
     </build>
     <dependencies>
          <dependency>
                <groupId>org.apache.spark</groupId>
               <artifactId>spark-core_2.11</artifactId>
               <version>1.3.0</version>
          </dependency>
      </dependencies>
</project>

Run as => Maven Build => clean install

$ spark-submit --class SparkPi SimpleSpark-0.0.1-SNAPSHOT-jar-with-dependencies.jar
</input_dir /output_dir>
```

Building scala applications using SBT build

- Install sbt from http://www.scala-sbt.org/release/docs/Setup.html and download .msi file
- Update environment variables to include sbt\bin into path
- Run sbt command from command prompt and update .sbt\0.13 in user home directory to add plugins\build.sbt file and add below entries to it. Space is mandatory between these lines.

```
addSbtPlugin("com.typesafe.sbteclipse" % "sbteclipse-plugin" % "2.2.0")

addSbtPlugin("com.github.mpeltonen" % "sbt-idea" % "1.6.0")

addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.14.1")
```

- Create a new folder for your project, e.g.: mkdir myProject and cd myProject
- Create the project folder structure:
  - mkdir -p src/main/scala (Mac/Linux)
  - mkdir src\main\scala (Windows)

Building scala applications using SBT build (Cont..)

- Create a file in src/main/scala e.g.
  - echo 'object Hi { def main(args: Array[String]) = println("Hi!") }' > hw.scala (Mac/Linux)
  - echo object Hi { def main(args: Array[String]) = println("Hi!") } > hw.scala (Windows)

- To allow managing dependencies, project name, Scala version etc, create a file named build.sbt in your project root (e.g. in myProject/build.sbt) for example:

  - name := "hello"
  - version := "1.0"
  - scalaVersion := "2.11.7«

- Now Run **sbt** command from myProject in command prompt. And **run** command from sbt shell. Run **eclipse** command from sbt shell to eclipse version of project, which can be imported into eclipse now.

- Now, we can add new scala classes, objects, traits to project from eclipse and building we need to again use **sbt run**

Any Question?

Thank

You

Visit

Once

Again