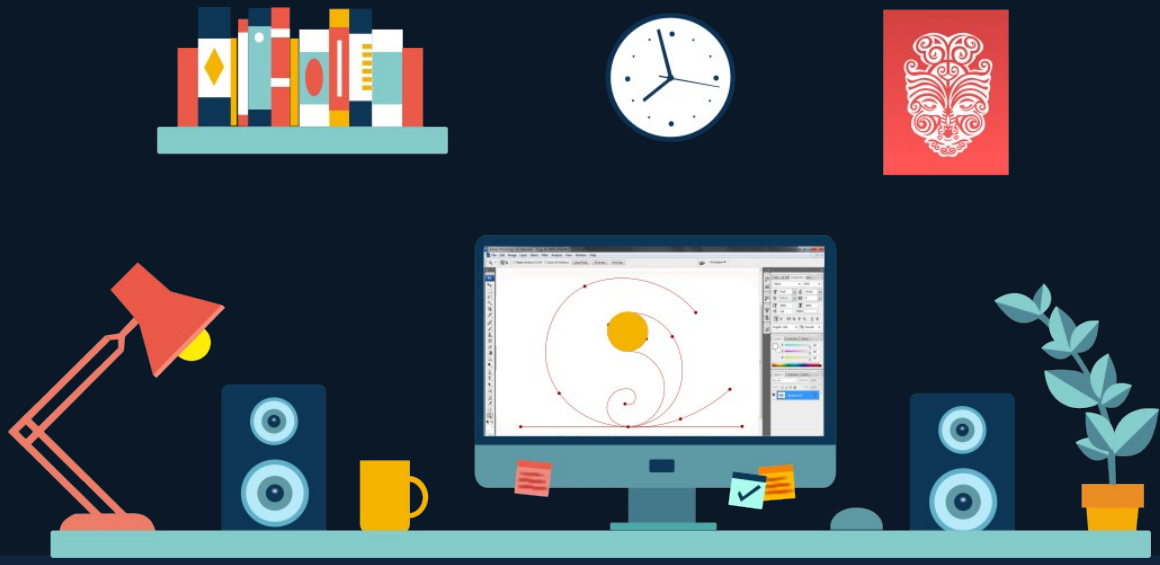
SCALA

"Quality means doing it Right,
When No one is looking"

- Presented By
--- Siva Kumar Bhuchipalli

Contents

- What is Scala
- Why Scala
- Scala Installation
- Eclipse IDE for Scala



Scala

High-level language for the JVM

- **Object oriented + functional programming**
- **Statically typed**

- **Comparable in speed to Java***

- **Interoperates with Java**
- **Can use any Java class (inherit from, etc**
- **Can be called from Java code**

Designed by Prof.Martin Odersky at EPFL

Why Scala?

- Scala designed for concurrency, expressiveness, and scalability
- Best fit for distributed applications/frameworks and concurrency programming
- Purely object-oriented and Interoperates with Java
- Clean & Concise - 100 lines of Java Code can be achieved within 10-20 lines of code in Scala some times
- Sensible static typing, closures, immutable collections, and elegant pattern matching

Key Features of Scala

- Scala source code compiles to Java bytecode, so that the resulting executable code runs on a Java virtual machine. we can use Java classes in Scala and Scala classes in Java, and can take advantage of Scala in Java applications where you need concurrency or conciseness.

- Unlike Java, Scala has many features of functional programming languages including type inference, immutability, lazy evaluation, currying and pattern matching.

- Scala had extensive support for functional programming from the beginning, Java remained a purely object oriented language until the introduction of lambda expressions with Java 8 in 2014.

- It is purely object-oriented.
  - For example, 2.toString() will generate a compilation error in Java. However, that is valid in Scala—we're calling the toString() method on an instance of Int.

- Semicolons are optional in Scala. In some cases the dot operator (.) is optional as well.
  - So, instead of writing s1.equals(s2);, we can write s1 equals s2

History

- Started in 2001 at EPFL by Martin Odersky.
- First released publicly in early 2004 on Java Platform & .NET platform
- The .NET support was officially dropped in 2012.
- In 2011, Odersky and collaborators launched Typesafe Inc., a company to provide commercial support, training, and services for Scala

Installing Scala in Linux

- First, download the most recent stable version of Scala—just visit
  http://www.scala-lang.org

  and click the “Download Scala” link

- For example, we have downloaded scala-2.7.4.final.tar.gz

- Installing Scala:
  - Untar the file using tar -xf scala-2.7.4.final.tar
  - Then copy scala-2.7.4.final to the /opt/scala directory.
  - Now add the path environment variable in ~/.bashrc file as shown below
  - /opt/scala/scala-2.7.4.final/bin

- In a new terminal window, type scala -version, and make sure it reports the right version of Scala you just installed. You’re all set to use Scala!

Hello World

```
object HelloWorld extends App {
  println("Hello, World!")
}
```

- **Unlike the stand-alone Hello World application for Java, there is no class declaration and nothing is declared to be static; a singleton object created with the object keyword is used instead.**

- **With the program saved in a file named HelloWorld.scala, it can be compiled from the command line:**

  $ scalac HelloWorld.scala

- **To run it:**

  $ scala HelloWorld

- **Scala includes scripting support, can be run as a script without prior compilation using:**

  $ scala HelloWorld2.scala

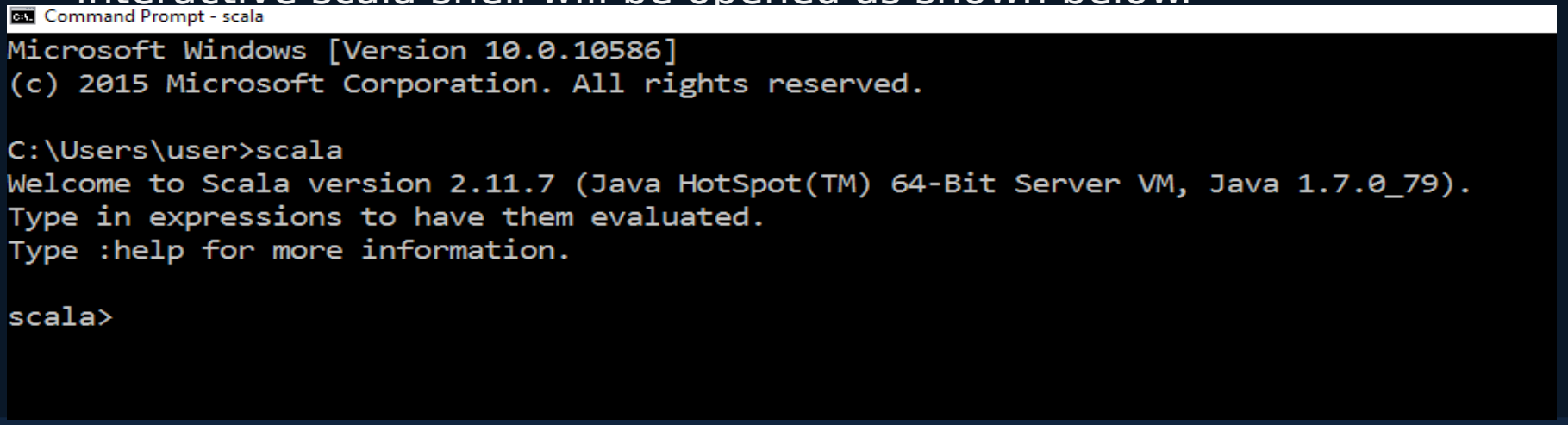
- **Commands can also be entered directly into the Scala interpreter, using the option -e:**

  $ scala -e 'println("Hello, World!")'

Scala Installation For Windows

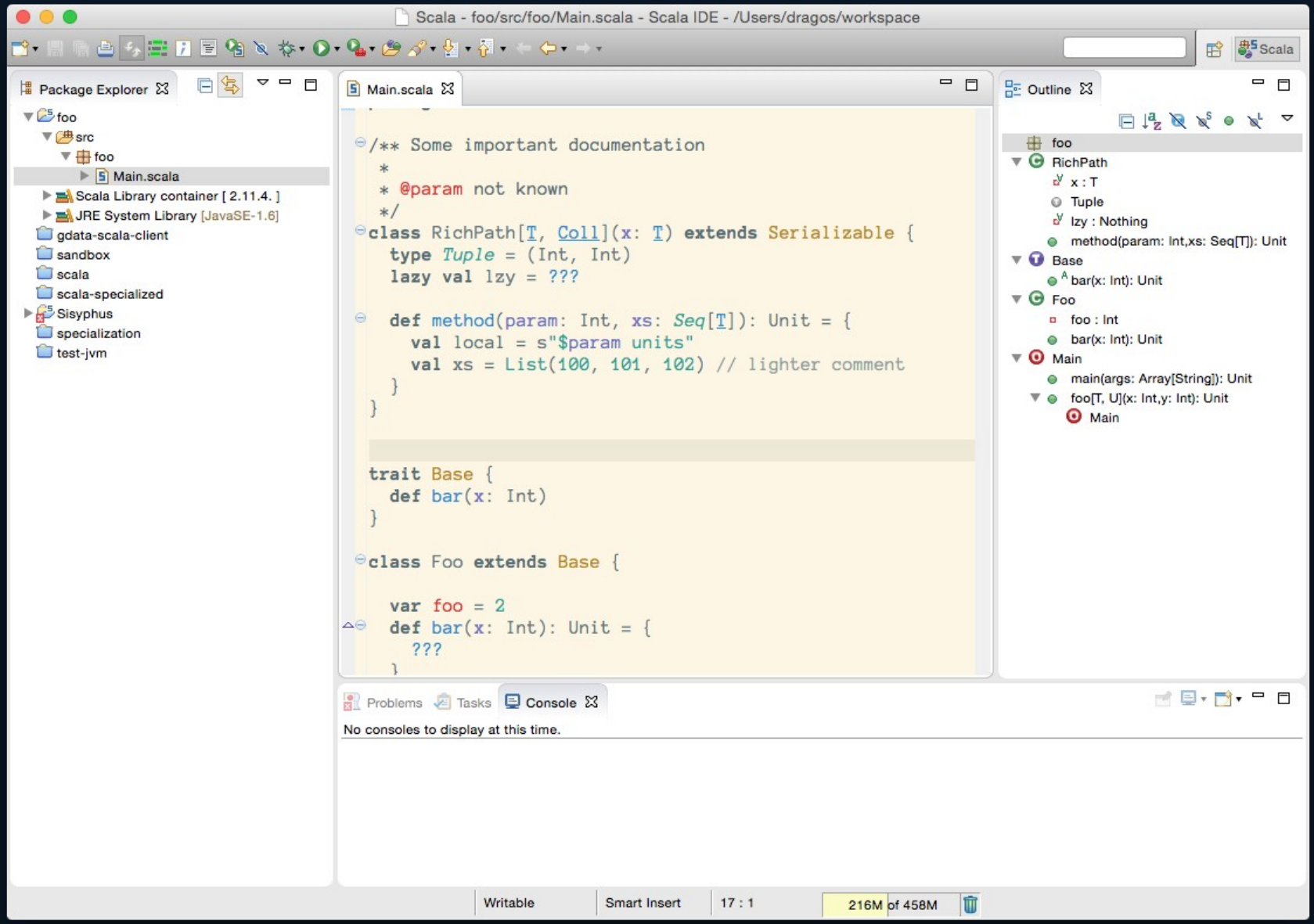
- Before installing scala make sure that you installed java in that system.
- Installing scala in windows is very easy. Just download Scala binaries from below URL
  http://www.scala-lang.org/download/

- Now Install it. That's it you can play with
- Open command prompt and type scala
- Interactive scala shell will be opened as shown below.



Eclipse IDE for scala :

- **First, download the most recent stable version of Scala—just visit http://www.scala-lang.org, and click the "Download Scala" link**
- **Download eclipse scala ide from below URL and just unzip/untar it.** http://scala-ide.org/download/sdk.html
- **If you already having java eclipse IDE, use below url plugin installation** http://scala-ide.org/download/current.html
- **Before using eclipse, you should have jdk installed in your system.**
- **Now you can use this IDE to build the scala applications as shown in the screen shot.**


Scala - foo/src/foo/Main.scala - Scala IDE - /Users/dragos/workspace
Package Explorer
foo
src
foo
Main.scala
Scala Library container [2.11.4.]
JRE System Library [JavaSE-1.6]
gdata-scala-client
sandbox
scala
scala-specialized
Sisyphus
specialization
test-jvm
Main.scala
/** Some important documentation
*
* @param not known
*/
class RichPath[T, Coll](x: T) extends Serializable {
type Tuple = (Int, Int)
lazy val lzy = ???
def method(param: Int, xs: Seq[T]): Unit = {
val local = s"$param units"
val xs = List(100, 101, 102) // lighter comment
}
}
trait Base {
def bar(x: Int)
}
class Foo extends Base {
var foo = 2
def bar(x: Int): Unit = {
???
}
Outline
foo
RichPath
x : T
Tuple
lzy : Nothing
method(param: Int,xs: Seq[T]): Unit
Base
bar(x: Int): Unit
Foo
foo : Int
bar(x: Int): Unit
Main
main(args: Array[String]): Unit
foo[T, U](x: Int,y: Int): Unit
Main
Problems
Tasks
Console
No consoles to display at this time.
Writable
Smart Insert
17 : 1
216M of 458M

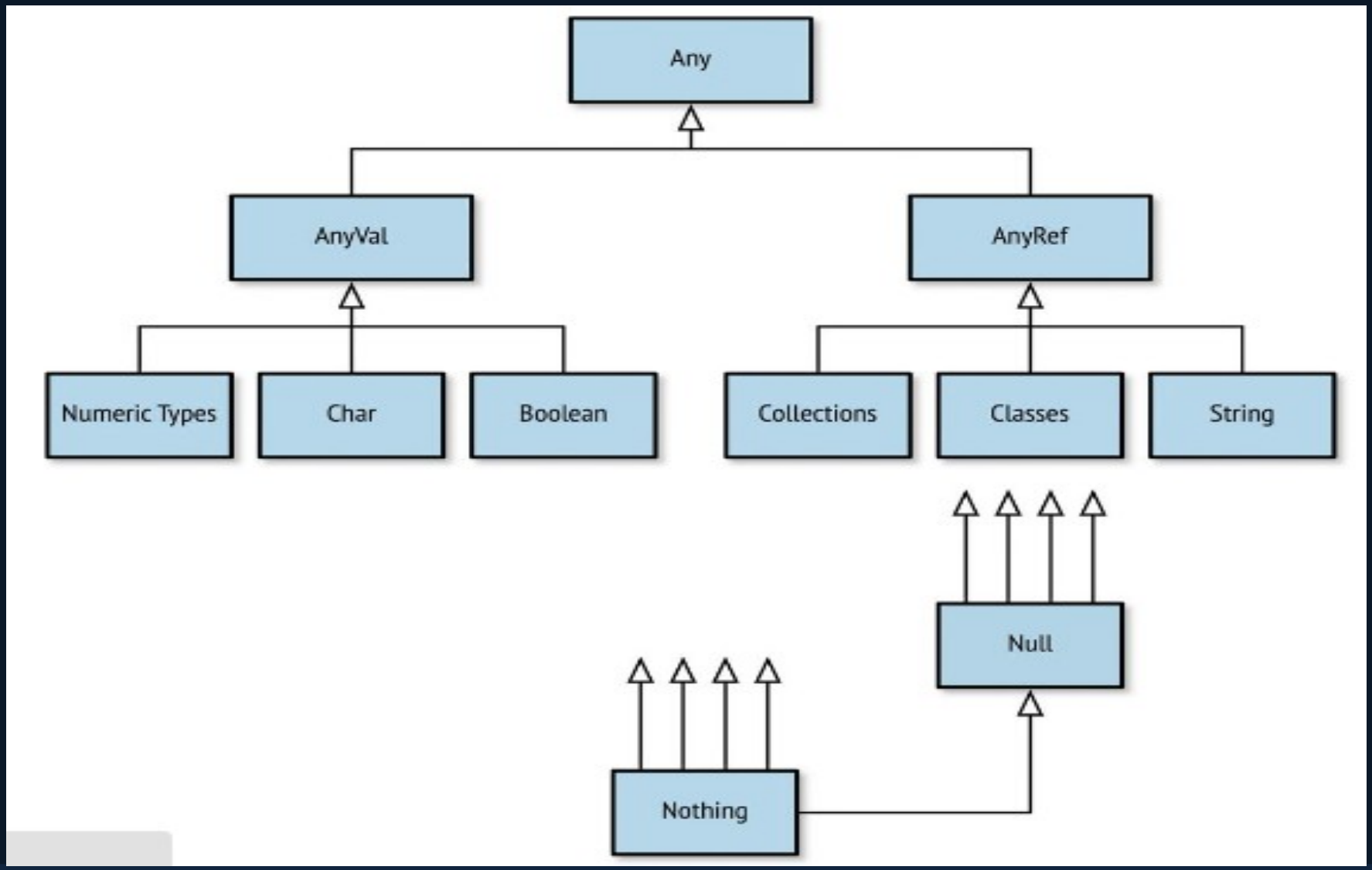
Data Types

- In Scala, all the data types are objects, including Byte, Char, Double, Float, Int, Long, and Short. These seven numeric types extend the AnyVal trait, as do the Unit and Boolean classes.
- All data in Scala corresponds to a specific type, and all Scala types are defined as classes with methods that operate on the data

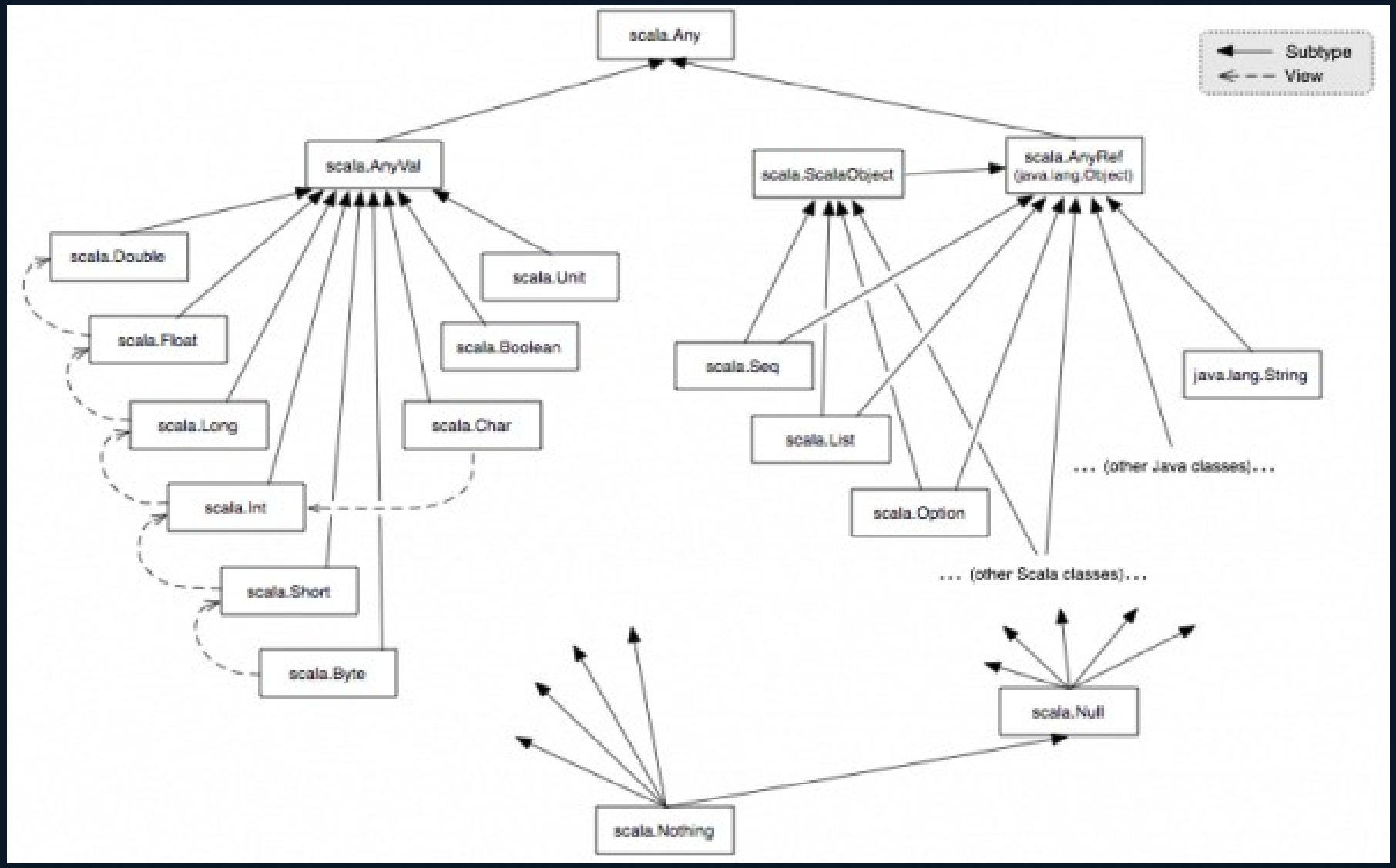
Data type	Range
Char	16-bit unsigned Unicode character
Byte	8-bit signed value
Short	16-bit signed value
Int	32-bit signed value
Long	64-bit signed value
Float	32-bit IEEE 754 single precision float
Double	64-bit IEEE 754 single precision float

- Scala doesn’t support operators like ++ and −− but += and −= methods can be used

Data Types Hierarchy



Data Types Hierarchy



Non Numeric Data Types

Name	Description	Instantiable
Any	The root of all types in Scala	No
AnyVal	The root of all value types	No
AnyRef	The root of all reference (nonvalue) types	No
Nothing	The subclass of all types	No
Null	The subclass of all AnyRef types signifying a null value	No
Char	Unicode character	Yes
Boolean	true or false	Yes
String	A string of characters (i.e., text)	Yes
Unit	Denotes the lack of a value	No

Variable Declaration

Scala has the different syntax for the declaration of variables and they can be defined as value, i.e., constant or a variable. Following is the syntax to define a variable using var keyword:

```
var myVar :String = "Foo"
```

Here, myVar is declared using the keyword var. Following is the syntax to define a variable using val keyword:

```
val myVal :String = "Foo"
```

Here, myVal can not be changed and this is called immutable variable.

Variable Data Types

The type of a variable is specified after the variable name, and before equals sign. You can define any type of Scala variable by mentioning its data type as follows:

```
val or var VariableName :DataType [=  Initial Value]
```

If you do not assign any initial value to a variable, then it is valid as follows:

```
var myVar :Int;
val myVal  :String;
```

String in Scala is nothing but java.lang.String. Scala can automatically convert a String to scala.runtime.RichString— which allows to apply some methods like capitalize(
), lines( ), and reverse.

- **Now we will declare a variable and initialize it**

```
scala> 2*3
res0: Int = 6
scala> val a = 10
a: Int = 10
scala> val b = a + 1
b: Int = 11
scala> a = a + 1
<console>:8: error: reassignment to val
a = a + 1
scala>
```

- **Val is Constant in Scala**
- **Var is variable**

Examples

```
def fun(a:Int) ={
      println("Hello")
      }
fun: (a: Int)Unit
```

Return type is Unit, A method with return type Unit is analogous to a Java method which is declared void.

```
scala> var a:Any=500
a: Any = 500

scala> var a:AnyRef=500
<console>:10: error: the result type of an implicit conversion must be more specific than AnyRef
       var a:AnyRef=500
                    ^

scala> var a:AnyVal=500
a: AnyVal = 500
```

We can only use Nothing if the method never returns (meaning it cannot complete normally by returning, it could throw an exception).

Strings in scala

- **Strings in scala are similar to strings in java.**

Creating Strings

```
var greeting = "Hello world!";
        or
var greeting:String = "Hello world!";
```

- **As we are using same java string object in scala. We will have same methods/features in scala string which are present in java.**
- **In addition to that we can use methods in** scala.collection.immutable.StringLike
- **Few methods in StringLike class are**

apply **- Retrieve the n-th character of the string**

capitalize **- Returns this string with first character converted to upper case**

toByte **– Converts to byte**

toDouble **- Converts to double**

toFloat **- Converts to float**

toInt **- Converts to int**

toLong **- Converts to long**

toShort **- Converts to short**

Strings Operations Examples

- Strings operations.

```
scala> "scala".drop(2).take(2).capitalize
res2: String = Al
scala> "scala".drop(2)
res0: String = ala
```

Next, the take(2) method retains the first two elements from the collection it's given,and discards the rest:

```
scala> "scala".drop(2).take(2)
res1: String = al
```

- Strings Comparison in scala

```
scala> val s1 = "Hello"
scala> val s2 = "Hello"
scala> val s3 = "H" + "ello"
scala> s1 == s2
res0: Boolean = true
scala> s1 == s3
res1: Boolean = true
```

In Java, == tests for reference equality (whether they are the same object)
.equals() tests for value equality (whether they are logically "equal")

```
// These two have the same value
new String("test").equals("test") // --> true
// ... but they are not the same object
new String("test") == "test" // --> false
“F” + “alse” not == “False”
```

- ✓ In Scala, we test object equality with the == method.
- ✓ This is different than Java, where we use the equals method to compare two objects.
- ✓ In Scala, the == method checks for null values, and then calls the equals method on the first object (i.e., this) to see if the two objects are equal.
- ✓ As a result, we don’t have to check for null values when comparing strings.
- ✓ Multi Line Strings

```
val foo = """This is
a multiline
String"""“
J=100;
println(s"Hello $j $foo")
```

Variable Type Inference:

When you assign an initial value to a variable, the Scala compiler can figure out the type of the variable based on the value assigned to it. This is called variable type inference. Therefore, you could write these variable declarations like this:

```
var myVar = 10;
val myVal = "Hello, Scala!";
```

Multiple Assignments:

Scala supports multiple assignments. If a code block or method returns a Tuple, the Tuple can be assigned to a val variable.
[Note: We will study Tuple in subsequent chapters.]

```
val (myVar1: Int, myVar2: String) = Pair(40, "Foo")
```

And the type inference gets it right:

```
val (myVar1, myVar2) = Pair(40, "Foo")
```

Variable Scopes

Fields nothing but instance variables. Fields can be either mutable or immutable types and can be defined using either var or val

Method Parameters are variables, which are used to pass the value inside a method when the method is called

- Method parameters are only accessible from inside the method but the objects passed in may be accessible from the outside, if you have a reference to the object from outside the method
- **Method parameters are always immutable and defined by val keyword**

Local Variables are variables declared inside a method

- Local variables are only accessible from inside the method, but the objects you create may escape the method if you return them from the method
- Local variables can be either mutable or immutable types and can be defined using either var or val

Expression Oriented Language

```
val i = 3
val p = if (i > 0) -1 else -2
val q = if (true) "hello" else "world"
println(p)
println(q)
```

Unlike languages like C/Java, almost everything in Scala is an "expression", ie, something which returns a value! Rather than programming with "statements", we program with "expressions"

- An expression is a single unit of code that returns a value

```
scala> "hel" + 'l' + "o“
scala> val x = 5 * 20; val amount = x + 10
```

- Expression blocks can span as many lines as you need

```
scala> val amount = {
         val x = 5 * 20
         x + 10
       }
```

Switch - Case Statement

```
def errorMsg(errorCode: Int) = errorCode match {
case 1 => "File not found"
case 2 => "Permission denied"
case 3 => "Invalid operation“
case _ => "Default Value"
}
println(errorMsg(2))
```

Case automatically "returns" the value of the expression corresponding to the matching pattern. Subsequent case statements will not get executed

```
scala> val status = 500
scala> val message = status match {
        case 200 =>   "ok"
        case 400 => {
           println("ERROR - we called the service incorrectly")
           "error"
        }
        case 500 => {
           println("ERROR - the service encountered an error")
           "error"
        }}
```

Arrays

Scala provides a data structure, the array, which stores a fixed-size sequential collection of elements of the same type

- ***Declaring***

  var z:Array[String] = new Array[String](3)
  or

  var z = new Array[String](3)

- ***Initializing***

  z(0) = “Siva"; z(1) = “Ram"; z(4/2) = “Kris
  Or z(5/2) = "Ayan“ It assigns to z(2)

  var z = Array(“Siva", “Ram", “Krish")

```
for ( x <- z ) { println( x ) }
```

Loop Control Statements

- There are three types of loops in scala
  - while
  - do… while
  - for loop

- Scala does not support break or continue statement like Java
- But there is a way to break the loops
  - break

While Loop

- **A while loop statement repeatedly executes a target statement as long as a given condition is true.**
- **Syntax:**

```
while(condition){
   statement(s);
}
```

- **Example**

```
object Test {
  def main(args: Array[String]) {
// Local variable declaration:
var a = 10
// while loop execution
while( a < 20 ){
println( "Value of a: " + a )
a = a + 1
} } }
```

Do...while Loop

- **A do...while loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.**
- **Syntax:**

```
do{
   statement(s);
} while(condition)
```

- **Example**

```
object Test {
  def main(args: Array[String]) {
// Local variable declaration:
var a = 10
// do while loop execution
do {
          println( "Value of a: " + a )
          a = a + 1
          } while( a < 20 )
} }
```

For Loop

- **A for loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times**
- **Syntax:**

```
for(var x <- Range ){
    statement(s);
}
```

- **Example**

```
object Test {
  def main(args: Array[String]) {
// Local variable declaration:
var a = 0
// while loop execution
for( a <- 1 to 10 ){
println( "Value of a: " + a )
a = a + 1
} } }
```

For Loop with collections

- **A for loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times**
- **Syntax:**

```
for(var x <- List ){
    statement(s);
}
```

- **Example**

```
object Test {
  def main(args: Array[String]) {
var a = 0;
val numList = List(1,2,3,4,5,6);
// for loop execution with a collection
for( a <- numList ){
println( "Value of a: " + a );
} } }
```

Example on arrays

```
object Test {
def main(args: Array[String]) {
var myList = Array(1.9, 2.9, 3.4, 3.5)
val a = Array("apple", "banana", "orange")

// Print all the array elements
for ( x <- myList ) { println( x ) }

for (e <- a) { val s = e.toUpperCase;  println(s) }

// Summing all elements
var total = 0.0;
for ( i <- 0 to (myList.length - 1)) { total += myList(i); }
println("Total is " + total);

// Finding the largest element
var max = myList(0);
for ( i <- 1 to (myList.length - 1) ) {
if (myList(i) > max) max = myList(i);
} println("Max is " + max); } }
```

For Loop With Yield

In cases where you want to build a new collection from the input collection, use the for/yield combination:

```
scala> val newArray = for (e <- a) yield e.toUpperCase
newArray: Array[java.lang.String] = Array(APPLE, BANANA, ORANGE)
```

OR

```
val newArray = for (e <- a) yield {
 // imagine this requires multiple lines
 val s = e.toUpperCase
 s
 }
```

For Loop With Multiple Counters

```
for (i <- 1 to 2; j <- 1 to 2) println(s"i = $i, j = $j")
```

Break Statement

- **Break statement in scala is different when compared to java break statement.**

- **Syntax:**

```
// import following package
import scala.util.control._

// create a Breaks object as follows
val loop = new Breaks;

// Keep the loop inside breakable as follows
loop.breakable{
    // Loop will go here
    for(...){
    ....
    // Break will go here
    loop.break;
} }
```

Break Statement example

```
   import scala.util.control._
object Test {
def main(args: Array[String]) {
var a = 0;
val numList = List(1,2,3,4,5,6,7,8,9,10);

val loop = new Breaks;
loop.breakable {
     for( a <- numList){
        println( "Value of a: " + a );
       if( a == 4 ){
         loop.break;
     } } }
println( "After the loop" );
} }
```

There is no Continue Statement as well in Scala

Function Definition

```
def fun(a: Int):Int = {
a + 1
a - 2
a * 3
}
val p:Int = fun(10)
println(p)

def isPalindrome(str: String) = str == str.reverse.toString()

isPalindrome("mom" )
```

There is no explicit "return" statement! The value of the last expression in the body is automatically returned.

Type Inference in Functions

```
def add(a:Int, b:Int) = a + b
val m = add(1, 2)
println(m)
```

We have NOT specified the return type of the function or the type of the variable "m". Scala "infers" that!

```
def add(a, b) = a + b
val m = add(1, 2)
println(m)
```

This does not work! Scala does NOT infer type of function parameters, unlike languages like Haskell/ML.

Function Parameters Aren't Mutable

```
scala> def m1(val i:Int) = i+2
<console>:1: error: identifier expected but 'val' found.
def m1(val i:Int) = i+2
       ^

scala> def m1(var i:Int) = i+2
<console>:1: error: identifier expected but 'var' found.
def m1(var i:Int) = i+2
       ^

scala> def m1(i:Int) = i+2
m1: (i: Int)Int

scala> def m1(i:Int) = {i = i+2; i}
<console>:10: error: reassignment to val
       def m1(i:Int) = {i = i+2; i}          // I.e Default is Val
for parameters
                       ^
```

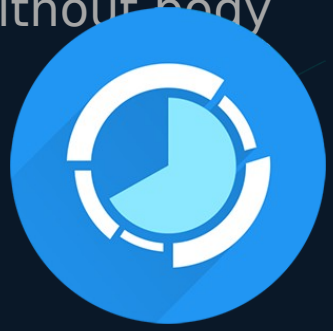
Function Definition (Achieving Mutability of Method Parameters)

```
class Field[T]( var value: T )   //Class declaration without body

class Foo {
  val x = new Field(0)
  val y = new Field(0)
}

class Bar {
  def changeFooField( field: Field[Int] ) {
    field.value = 1
  }
}

val f = new Foo
(new Bar).changeFooField( f.x )
println( f.x.value + " / " + f.y.value ) // prints "1 / 0"
```



Setting default values for method parameters

You can set default values for method parameters:

```
class Connection {

  // default values specified here
  def makeConnection(timeout:Int = 5000, protocol:String = "http") {
     println("timeout = %d, protocol = %s".format(timeout, protocol))
     // more code here ...
  }
}

val c = new Connection
c.makeConnection()                // timeout = 5000, protocol = http
c.makeConnection(2000)            // timeout = 2000, protocol = http
c.makeConnection(3000, "https")   // timeout = 3000, protocol =
https
```

Using parameter names when calling a method

We can use parameter names when calling a method:

```
c.makeConnection(timeout = 10000, protocol = "https")   // specify the parameter names
c.makeConnection(protocol = "https", timeout = 10000)   // names can be in any order
```

Returning multiple items from a method

A method can return multiple items (a Tuple):

```
// 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
```

Quick Tour

Quick Tour of Scala:

Declaring variables:

```
var x: Int = 7
var x = 7 // type inferred
val y = “hi” // read-only
```

Functions:

```
def square(x: Int): Int = x*x
def square(x: Int): Int = {
x*x
}
def announce(text: String) =
{
println(text)
}
```

Java equivalent:

```
int x = 7;
final String y = “hi”;
```

Java equivalent:

```
int square(int x) {
return x*x;
}
void announce(String
text) {
System.out.println(tex
t);
}
```

Recursion

```
// sum n + (n-1) + (n-2) + ... + 0

def sum(n: Int): Int =
if (n == 0) 0 else n + sum(n - 1)
val m = sum(10)
println(m)
```

Try calling the function "sum" with a large number (say 10000) as parameter! You get a stack overflow!

Tail Calls

```
def sum(n: Int, acc: Int):Int =
if(n == 0) acc else sum(n - 1, acc + n)
val r = sum(10000, 0)
println(r)
```

This is a "tail-recursive" version of the previous function – the Scala compiler converts the tail call to a loop, thereby avoiding stack overflow!

```
(sum 4)
(4 + sum 3)
(4 + (3 + sum 2))
(4 + (3 + (2 + sum 1)))
(4 + (3 + (2 + (1 + sum 0))))
(4 + (3 + (2 + (1 + 0))))
(4 + (3 + (2 + 1)))
(4 + (3 + 3))
(4 + 6)
(10)
------------------------------
(sum 4 0)
(sum 3 4)
(sum 2 7)
(sum 1 9)
(sum 0 10)
(10)
```

Summation Once Again

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

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

def sumSimple(a: Int, b: Int): Int =
if (a == b) a else a + sumSimple(a + 1, b)

def sumSquares(a: Int, b: Int): Int =
if (a == b) sqr(a) else sqr(a) + sumSquares(a +

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

Higher Order Function

```
def identity(x: Int) = x

def sqr(x: Int) = x * x

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

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

println(sum(identity, 1, 10))
println(sum(sqr, 1, 10))
println(sum(cube, 1, 10))
```

"sum" is now a "higher order" function! It's first parameter is a function which maps an Int to an Int. The type "Int" represents a simple Integer value. The type Int => Int represents a function which accepts an Int and returns an Int.

Anonymous Function

```
def sum(f: Int=>Int, a: Int, b: Int): Int = if (a == b) f(a) else
f(a) + sum(f, a + 1, b)
println(sum(x=>x, 1, 10))
println(sum(x=>x*x, 1, 10))
println(sum(x=>x*x*x, 1, 10))
```

We can create ”anonymous” functions on-the-fly! x => x*x is a function which takes an ”x” and returns x*x

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
```

If our function is made up of many expressions, we can use {}

```
def timesTwo(i: Int): Int = { println("hello world");   i * 2}
```

This is also true of an anonymous function.

```
scala> { i: Int =>
  println("hello world")
  i * 2
}
res0: (Int) => Int = <function1>
```

Functions with Place Holder Syntax

```
val someNumbers = List(1,2,3,4,5,6,7)
scala> someNumbers.filter(_ > 0)   //With Placeholder
  res9: List[Int] = List(5, 10)

  scala> someNumbers.filter(x => x > 0)
  res10: List[Int] = List(5, 10)

  scala> val f = _ + _
  <console>:4: error: missing parameter type for expanded
  function ((x$1, x$2) => x$1.$plus(x$2))
        val f = _ + _
                ^


  scala> val f = (_: Int) + (_: Int)
  f: (Int, Int) => Int = <function>

  scala> f(5, 10)
  res11: Int = 15
```

Nested /Local Function / Functions

Returning Functions

```
// fun returns a function of type Int => Int
def fun():Int => Int = {
def sqr(x: Int):Int = x * x
   sqr
}
val f = fun()
println(f(10))
```

def fun():Int=>Int says ”fun is a function which does not take any argument and returns a function which maps an Int to an Int. Note that it possible to have ”nested” function definitions.

Lexical Closure

- The function ”fun1” returns a ”closure”.
- A ”closure” is a function which carries with it references to the environment in which it was defined.
- When we call fun1(10), the ”add” function gets executed with the environment it had when it was defined - in this environment, the value of ”y” is 1

```
def fun1():Int => Int = {
      val y = 1
      def add(x: Int) = x + y  //Nested function that can access local
variables other than param
                  println(add(10))
      add
   }
def fun2() = {
            val y = 2
            val f  = fun1()
            println(f(10)) // what does it print? 11 or 12 □ 11
            }
fun2()
```

Simple Closure Examples

```
var votingAge = 18
val isOfVotingAge = (age: Int) => age >= votingAge

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

scala> var more = 1
scala> val addMore = (x: Int) => x + more //addMore: (Int) => Int = <function>
scala> addMore(10)   //11
scala> more = 9999
scala> addMore(10)   //10009

def removeLowScores(a: List[Int], threshold: Int): List[Int] =
a.filter(score => score >= threshold)

val a = List(95, 87, 20, 45, 35, 66, 10, 15)
println(removeLowScores(a, 30))
```

- The anonymous function "score => score >= threshold" is the closure here.
- How do you know that it is a closure? Its body uses a variable ”threshold” which is not in its local environment (the local environment, in this case, is the parameter list consisting of a single parameter ”score”)

Block Structure / Scope

The ”y” in the inner scope shadows the ”y” in the outer scope

```
def fun(x: Int) = {
val y = 1

val r = {
        val y = 2
        x + y
          }
println(r)
println(x + y)
}
fun(10)
```

Currying

In mathematics and computer science, currying is the technique of transforming a function that takes multiple arguments (or a tuple of arguments) in such a way that it can be called as a chain of functions, each with a single argument. It was originated by Moses Schonfinkel and later re-discovered by Haskell Curry.

Currying - two argument functions

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

val a = addA(10, 20)
val b = addB(10)(20)

println(a)
println(b)
```

```
scala> val f = addB(10)
f: Int => Int =
<function1>

scala> println(f(20))
30
```

Currying - three argument functions

```
def addA(x: Int, y: Int, z: Int) = x + y + z

def addB(x: Int): Int => (Int => Int) =
    y => (z => x + y + z)

val a = addA(1, 2, 3)

val b = addB(1)(2)(3)

println(a)
println(b)
```

It is now easy to see how the idea can be generalized to N argument functions!

Immutability

- ”val” is immutable in the sense once you bind a value to a symbol defined as ”val”, the binding can’t be changed

```
val a = 1
a = 2 // does not work
a = a + 1 // does not work
var b = 1
b = 2
b = b + 1
```

- Here is a similar program in Scala

```
val a = List(20, 30, 40, 50)
val b = 10 :: a
println(a)    // still List(20, 30, 40, 50)
println(b)    // List(10, 20, 30, 40, 50)

val c = a  ++  List(60)
println(c) // List(20, 30, 40, 50, 60)
println(a) // still List(20, 30, 40, 50)
```

Non – Strict Evaluation

```
def myIf(cond: Boolean, thenPart: Int, elsePart: Int) =
if (cond) thenPart else elsePart

println(myIf((1 < 2), 10, 20))
```

We are trying to write a function which behaves similar to the built-in ”if” control structure in Scala ... does it really work properly? Let’s try another example!

```
def fun1() = {
      println("fun1")
      10
}
def fun2() = {
      println("fun2")
      20
}
def myIf(cond: Boolean, thenPart: Int, elsePart: Int) =
if (cond) thenPart else elsePart
println(myIf((1 < 2), fun1(), fun2()))
```

- **The behaviour of ”if” is ”non-strict”: In the expression ”if (cond) e1 else e2”, if ”cond” is true e2 is NOT EVALUATED. Also, if ”cond” is false, e1 is NOT EVALUATED.**

- **By default, the behaviour of function calls in Scala is ”strict”: In the expression ”fun(e1, e2, ..., en), ALL the expressions e1, e2 ... en are evaluated before the function is called.**

- **There is a way by which we can make the evaluation of function parameters non-strict. If we define a functions as "def fun(e1: => Int)", the expression passed as a parameter to ”fun” is evaluated ONLY when its value is needed in the body of the function. This is the ”call-by-name” method of parameter passing, which is a ”non-strict” strategy.**

```
def fun1() = {
      println("fun1")
      10
}
def fun2() = {
      println("fun2")
      20
}
def myIf(cond: Boolean, thenPart: => Int,
      elsePart: => Int) =
if (cond) thenPart else elsePart
```

LAZY Val's

```
def hello() = {
      println("hello")
      10
}

val a = hello()
```

The program prints ”hello” once, as expected. The value of the val ”a” will be 10.

```
lazy val a = hello()
```

Strange, the program does NOT print ”hello”! Why? The expression which assigns a value to a ”lazy” val is executed only when that lazy val is used somewhere in the code!

```
println(a + a)
```

Unlike a ”call-by-name” parameter, a lazy val is evaluated only once and the value is stored! This is called ”lazy” or ”call by need” evaluation.

Pure Functions

```
var balance = 1000

def withdraw(amount: Int) = {
      balance = balance - amount
      balance }

println(withdraw(100))
println(withdraw(100))
```

- In what way is our ”withdraw” function different from a function like ”sin”?
- A pure function always computes the same value given the same parameters; for example, sin(0) is always 0. It is ”Referentially Transparent”.
- Evaluation of a pure function does not cause any observable ”side effects” or output - like mutation of global variables or output to I/O devices.

Methods On Collections: Map/Filter/Reduce

- **Map applies a function on all elements of a sequence.**
- **Filter selects a set of values from a sequence based on the Boolean value returned by a function passed as its parameter - both functions return a new sequence.**
- **Reduce combines the elements of a sequence into a single element.**

```
val a = List(1,2,3,4,5,6,7)

val b = a.map(x => x * x)      //returns another list with same element count
val c = a.filter(x => x < 5)     //returns another list but element count might vary
val d = a.reduce((x, y)=>x+y)  //returns an aggregated value but not the list

println(b)
println(c)
println(d)
```

More Methods On Collections

```
def even(x: Int) = (x % 2) == 0

val a = List(1,2,3,4,5,6,7)
val b = List(2, 4, 6, 5, 10, 11, 13, 12)

// are all members even?
println(a.forall(even))

// is there an even element in the sequence?
println(a.exists(even))

//take while the element is even
//stop at the first odd element
println(b.takeWhile(even))

//partition into two sublists: even and odd
println(a.partition(even))
```

Some List Operations

```
val a = List(1,2,3)
val b = Nil
val c = List()
val d = 0::a
val e = 0::b

println(b)
println(c)
println(d)  // List(0,1,2,3)
println(e)  // List(0)
```

- Nil and List() are both ”empty” lists
- a::b returns a new list with ”a” as the first item (the ”head”) and remaining part b (called the ”tail”)

Pattern Matching With Lists

```
def fun(a: List[Int]) = a match {
      case List(0, p, q) => p + q
      case _ => -1
}
println(fun(List(0, 10, 20)))   ===> 30
println(fun(List(0, 1, 2, 3)))   ===> -1
println(fun(List(1, 10, 20)))   ===> -1
```

Pattern matching helps you ”pull-apart” complex data structures and analyse their components in an elegant way. In the above case, the function will return the sum of the second and third elements for any three element List of integers starting with a 0 the value -1 is returned for any other list

```
def fun(a: List[Int]) = a match {
      case List(0, p, q) => p + q
      case List() => -1
}
```

```
def length(a: List[Int]):Int = a match {
      case Nil => 0
      case h::t => 1 + length(t)
}

println(length(List()))
println(length(List(10,20,30,40)))
```

A simple recursive routine for computing length of a list. If list is empty, length is 0, otherwise, it is 1 + length of remaining part of the list.

If we have a List(1,2,3) and if we match it with a ”case h::t”, ”h” will be bound to 1 and ”t” will be bound to the ”tail” of the list, ie, List(2,3)

Any Question?

Thank You

Visit

Once

Again