

SCALA PROGRAMMING





TABLE OF CONTENTS



1. Getting Started with Scala Programming: 3
2. Building Blocks of Scala:29
3. Shaping our Scala Program:118
4. Giving Meaning to Programs with Functions: 142



1: GETTING STARTED WITH SCALA PROGRAMMING



GETTING STARTED WITH SCALA PROGRAMMING

In this lesson, we will cover the following topics:

Introduction to Scala

Scala advantages

Working with Scala

Running our first program

INTRODUCTION TO SCALA



- Consider a scenario where you get a paragraph and a word and you are asked to get the number of occurrences for that word.
- Your solution might look like this:

```
String str = "Scala is a multi-paradigm language. Scala is scalable too."  
int count = 0;  
for (stringy: str.split (" ")) {  
    if (word.equals (stringy))  
        count++;  
}  
System.out.println ("Word" + word + " occurred " + count + " times.")
```

INTRODUCTION TO SCALA



- Now our Scalable language has a simple way of accomplishing this.
- Let's take a look at that:

```
val str = "Scala is a multi-paradigm language. Scala is scalable too."  
println ("Word" + word + " occurred " + str.split(" ").filter(_ == word).size + "  
times.")
```

A PROGRAMMING PARADIGM

A paradigm is simply a way of doing something. So a programming paradigm means a way of programming or a certain pattern of writing programs.

- **Imperative Paradigm:** First do this and then do that
- **Functional Paradigm:** Evaluate and use
- **Logical Paradigm:** Answer through solution
- **Object-Oriented Paradigm:** Send messages between objects to simulate temporal evolution of a set of real-world phenomena

OBJECT-ORIENTED VERSUS FUNCTIONAL PARADIGMS

- With its roots in the mathematics discipline, the functional programming paradigm is simple.
- It works on the theory of functions which produce values that are immutable.
- Immutable values mean they can't be modified later on directly.
- In the functional paradigm, all computations are performed by calling self/other functions.

SCALA IS MULTI-PARADIGM

- Scala, being a multi-paradigm language, supports both paradigms, As we're learning Scala, we have the power of both of these paradigms.
- We can create functions as we need them, and also have objects talking to other objects.
- We can have class hierarchies and abstractions, With this, dominance over a particular paradigm will not affect another.

SCALA IS MULTI-PARADIGM

- The following is an example of a trait defined in Scala, called Function1:

```
package scala
trait Function1[A, B] {
    def apply(x: A) : B
}
```



SCALA IS MULTI-PARADIGM

- We also refer to these as $A \Rightarrow B$ (we call it, A to B).
- It means this function takes a parameter of type A, does some operation as defined, and returns a value of type B:

```
val answer = new Functional[Int, Int] {  
    def apply(x: Int): Int = x * 2  
}
```

SCALA IS MULTI-PARADIGM



- One example could be a function which takes your date of birth and returns your age in terms of the number of years and months:

```
class YearsAndMonths(years: Int, months: Int)  
def age(birthdate: Date): YearsAndMonths = //Some  
Logic
```

SCALA ADVANTAGES

Runs on JVM

- Consider efficiency and optimization as factors for a language to be well performant. Scala utilizes JVM for this.
- JVM uses Just in Time (JIT) compilation, adaptive optimization techniques for improved performance.
- Running on JVM makes Scala interoperable with Java. You've multitudinous libraries available as tools for reuse.

SUPER SMART SYNTAX

- You are going to write succinct code with Scala.
- There are a lot of examples we can look at to see Scala's syntax conciseness.
- Let's take an example from Scala's rich collections and create a Map:

```
val words = Map ("Wisdom" -> "state of being  
wise")  
println(words("Wisdom"))
```

```
> state of being wise
```

TYPE IS THE CORE

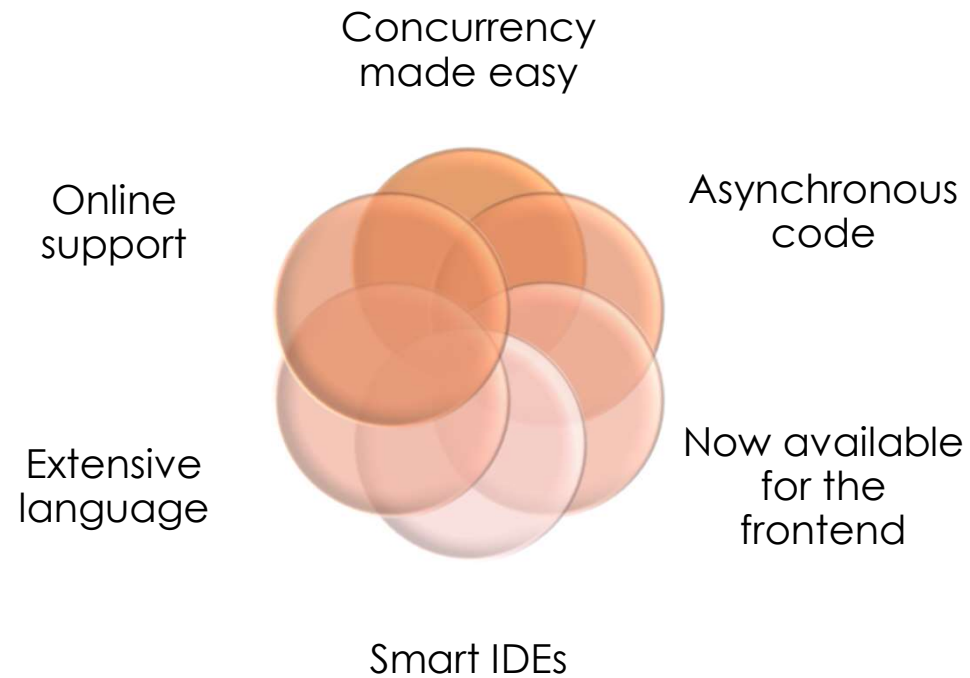


- In the early days (great, if even in the present) you may have come across this:

$f : \mathbb{R} \rightarrow \mathbb{N}$

This is the mathematical representation of a function.

SCALA ADVANTAGES



WORKING WITH SCALA

- Scala 2.12 was a step up from previous versions, mainly for support of Java and Scala lambda interoperability.
- Traits and functions are compiled directly to their Java 8 equivalents.



Java installation

SBT INSTALLATION

To install SBT on your machine, perform the following:

- Go to <http://www.scala-sbt.org/download.html>.
- You may choose from the available options suitable for your operating system.
- After installation, you may check the version, so open a command prompt/terminal and type this:

```
sbt sbt-version  
[info] 0.13.11
```

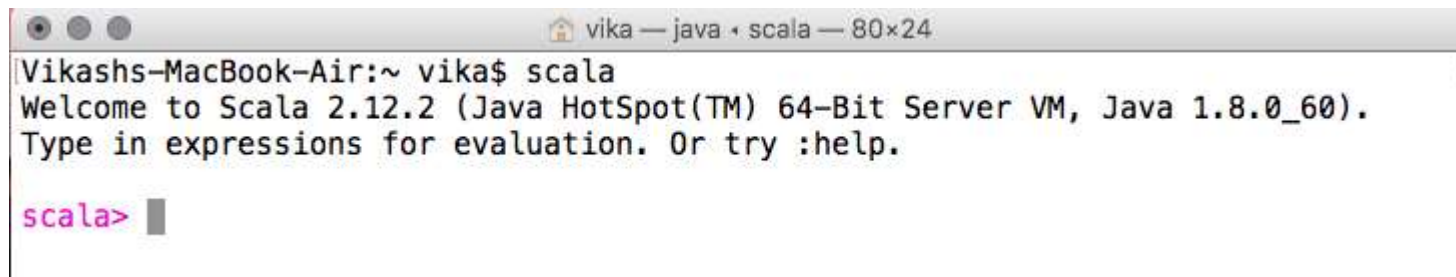
SCALA REPL

- There is more than one way of interacting with Scala.
- One of them is using Scala Interpreter (REPL).
- To run Scala REPL using SBT, just give the following command in the command prompt/terminal:

`sbt console`

SCALA REPL

- Try running the scala command, it should look something like this:



```
vika — java • scala — 80x24
Vikashs-MacBook-Air:~ vika$ scala
Welcome to Scala 2.12.2 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_60).
Type in expressions for evaluation. Or try :help.

scala> █
```

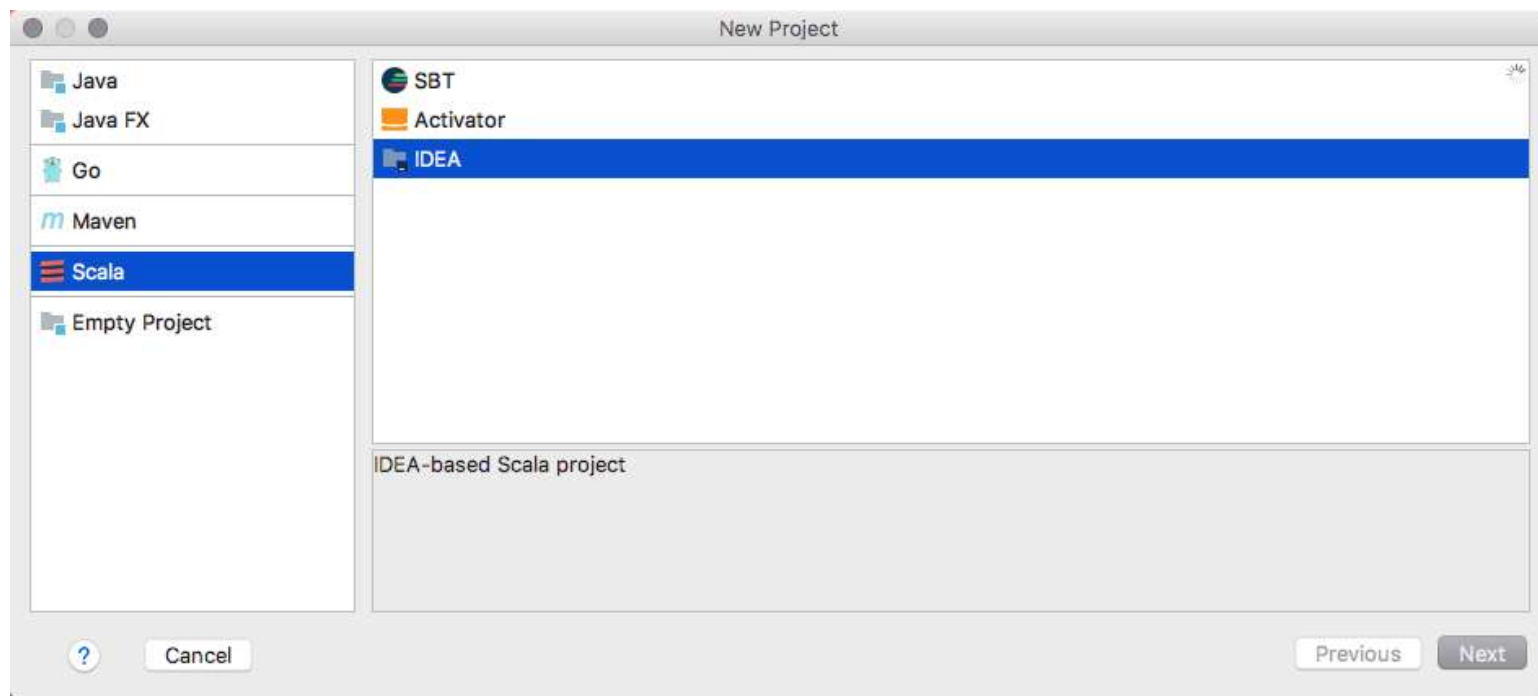
RUNNING OUR FIRST PROGRAM

- Click on the Create New Project function:

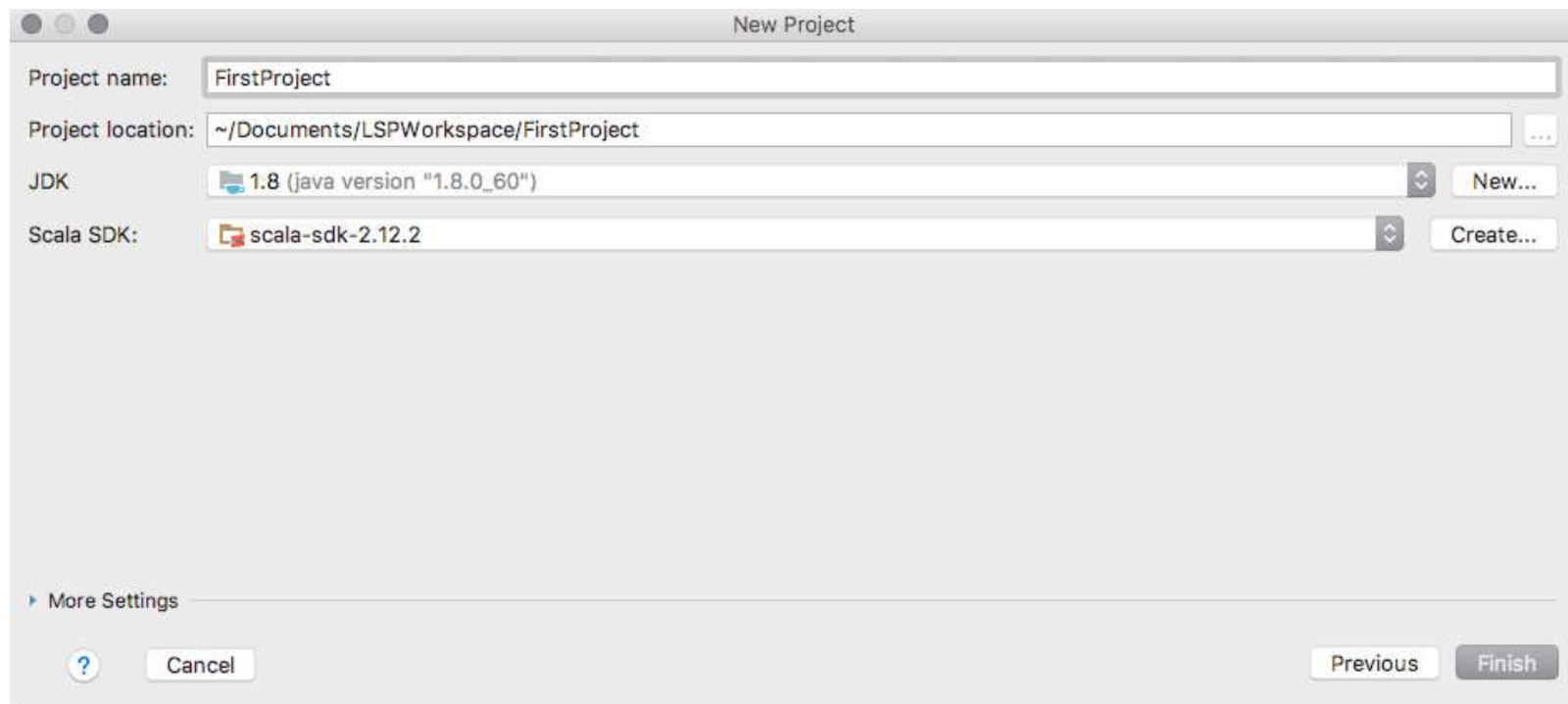


RUNNING OUR FIRST PROGRAM

- Select the Scala | IDEA option and click Next:



- Give Project name, Project location, select/locate Scala SDK, and Finish:



RUNNING OUR FIRST PROGRAM

- Let's write some code:

```
package lsp
```

```
object First {  
  def main(args: Array[String]): Unit = {  
    val double: (Int => Int) = _ * 2  
    (1 to 10) foreach double .andThen(println)  
  }  
}
```



RUNNING OUR FIRST PROGRAM

- In function literal applied on a range of integer values 1 to 10, represented by (1 to 10), gives back doubled values for each integer:

`(1 to 10) foreach double .andThen(println)`

RUNNING OUR FIRST PROGRAM

- With this example, you successfully wrote and understood your first Scala program.
- Even though the code was concise, there's a bit of overhead that can be avoided.
- For example, the main method declaration, The code can be written as follows:

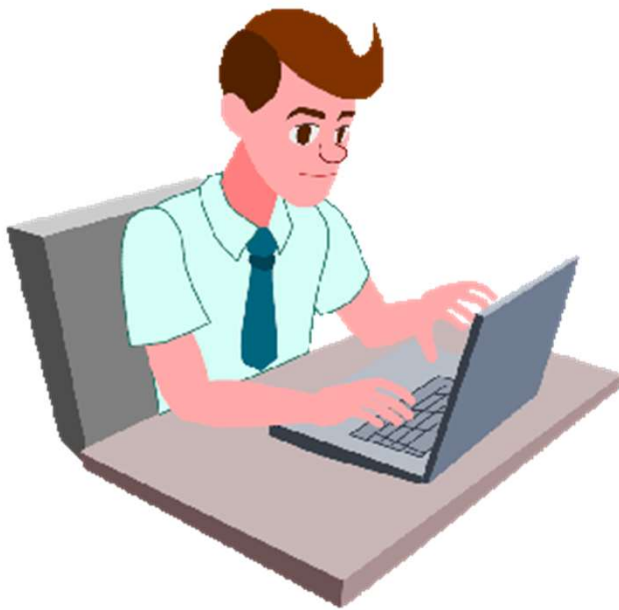
```
package lsp
```

```
object FirstApp extends App {  
  val double: (Int => Int) = _ * 2  
  (1 to 10) foreach double .andThen(print)  
}
```

SUMMARY



- This lesson was an introduction to Scala for us. We started learning about programming paradigms.
- After that, we discussed Scala's advantages over other available languages.
- Then we got our development environment ready, Finally, we wrote our first Scala program.



"COMPLETE LAB"

2: BUILDING BLOCKS OF SCALA



[illegible]

BUILDING BLOCKS OF SCALA

We'll go through:

- The `val` and `var` keywords
- Literals
- Data types
- Type Inference
- Operators
- Wrapper classes
- String interpolation

WHAT IS UNDERNEATH A SCALA PROGRAM?

- A Scala program is a tree of nested definitions.
- So the syntax is regular, just like any other programming language has keyword/name/classifier/bound-entity.
- Let's take an example, We'll use Scala REPL to see how a simple Scala program is built.
- For that, let's import a Scala package named universe:

```
scala> import scala.reflect.runtime.universe._  
import scala.reflect.runtime.universe._
```


WHAT IS UNDERNEATH A SCALA PROGRAM?

- let's execute this line and see what we get in response:

```
scala> val expr = reify {class Car {val segment="SUV"; def name="Q7"}}
expr: reflect.runtime.universe.Expr[Unit] =
Expr[Unit]({
  class Car extends AnyRef {
    def <init>() = {
      super.<init>();
      ()
    };
    val segment = "SUV";
    def name = "Q7"
  };
  ()
})
```

WHAT IS UNDERNEATH A SCALA PROGRAM?

- We'll use the `showRaw(expr.tree)` method to print tree:

```
scala> showRaw(expr.tree)
res0: String = Block(List(ClassDef(Modifiers(), TypeName("Car"), List(),
Template(List(Ident(TypeName("AnyRef"))), noSelfType, List(DefDef(Modifiers(),
termNames.CONSTRUCTOR, List(), List(List()), TypeTree(),
Block(List(Apply(Select(Super(This(typeNames.EMPTY), typeNames.EMPTY),
termNames.CONSTRUCTOR), List()), Literal(Constant(())))), ValDef(Modifiers(),
TermName("segment"), TypeTree(), Literal(Constant("SUV"))), DefDef(Modifiers(),
TermName("name"), List(), List(), TypeTree(), Literal(Constant("Q7"))))),
Literal(Constant(())))
```

VALS AND VARS

- When we use a `val` keyword to assign a value to any attribute, it becomes a value, We're not allowed to change that value in the course of our program.
- So a `val` declaration is used to allow only immutable data binding to an attribute, Let's take an example:

```
scala> val a = 10  
a: Int = 10
```

VALS AND VARS

- we have used a `val` keyword with an attribute named `a`, and assigned it a value `10`.
- Furthermore, if we try to change that value, the Scala compiler will give an error saying: `reassignment to val`:

```
scala> a = 12  
<console>:12: error: reassignment to val  
    a = 12
```

VALS AND VARS

- Scala recommends use of `val` as much as possible to support immutability.
- But if an attribute's value is going to change in the course of our program, we can use the `var` declaration:

```
scala> var b = 10  
b: Int = 10
```

VALS AND VARS

- When we define an attribute using a var keyword, we're allowed to change its value.
- The var keyword here stands for variable, which may vary over time:

```
scala> b = 12  
b: Int = 12
```



VALS AND VARS

- In Scala, we can explicitly give types after the attribute's name:

```
scala> val a: String = "I can be inferred."  
a: String = I can be inferred.
```

VALS AND VARS

- When we explicitly define type information for an attribute, then the value we give to it should justify to the type specified:

```
scala> val a: Int = "12"  
<console>:11: error: type mismatch;  
found   : String("12")  
required: Int  
    val a: Int = "12"
```


LITERALS

- If you're coming from a Java background, then quite a few will be the same for you: Integer, Floating point, Boolean, Character, and String are similar.
- Along with those, the Tuple and Function literals can be treated as something new to learn

Integer literals

Floating point literals

Boolean literals

Character literals

String literals

Symbol literals

Tuple literals

Function literals

INTEGER LITERALS

- Numeric literals can be expressed in the form of decimal, octal, or hexadecimal forms.
- Octal values are deprecated since version 2.10, so if you try out a numeric with a leading 0, it'll give you a compile time error:

```
scala> val num = 002
<console>:1: error: Decimal integer literals may
not have a leading zero. (Octal syntax is
obsolete.)
val num = 002
      ^
```

INTEGER LITERALS

- If you define a literal with prefix 0x or 0X, it's going to be a hexadecimal literal.
- Also, the Scala interpreter prints these values as a decimal value. For example:

```
scala> 0xFF  
res0: Int = 255
```

INTEGER LITERALS

Type	Minimum value	Maximum value
Int	-2^{31}	$2^{31} - 1$
Long	-2^{63}	$2^{63} - 1$
Short	-2^{15}	$2^{15} - 1$
Byte	-2^7	$2^7 - 1$

INTEGER LITERALS

- If we try to define any literal outside of these ranges for specified types, the compiler is going to give some error stating type mismatch:

```
scala> val aByte: Byte = 12  
aByte: Byte = 12
```

INTEGER LITERALS

- The compiler will try to convert that value to an integer, and then try to assign it to the attribute, but will fail to do so:

```
scala> val aByte: Byte = 123456  
<console>:20: error: type mismatch;  
found   : Int(123456)  
required: Byte  
      val aByte: Byte = 123456
```



INTEGER LITERALS

- What if we try to assign an attribute, an integer value that does not come under any of the mentioned ranges?
- Let's try:

```
scala> val outOfRange = 123456789101112131415  
<console>:1: error: integer number too large  
val outOfRange = 123456789101112131415
```

INTEGER LITERALS

- To define long literals, we put the character L or l at the end of our literal.
- Otherwise, we can also give type information for our attribute:

```
scala> val aLong = 909L
```

```
aLong: Long = 909
```

```
scala> val aLong = 909l
```

```
aLong: Long = 909
```

```
scala> val anotherLong: Long = 1
```

```
anotherLong: Long = 1
```


INTEGER LITERALS

- The Byte and Short values can be defined by explicitly telling the interpreter about the type:

```
scala> val aByte : Byte = 1  
aByte: Byte = 1
```

```
scala> val aShort : Short = 1  
aShort: Short = 1
```

FLOATING POINT LITERALS

- Floating point literals include a decimal point that can be at the beginning or in between decimal digits, but not at the end.
- What we mean by this is that if you write the following statement, it won't work:

```
scala> val a = 1. //Not possible!
```

FLOATING POINT LITERALS



- By default, Scala treats decimal point values as Double, if we don't specify it to be a Float:

```
scala> val aDoubleByDefault = 1.0  
aDoubleByDefault: Double = 1.0
```

FLOATING POINT LITERALS

- We can specify our values to be of Float type the same way we did for Long literals but with an invisible asterisk.
- Let's check that condition:

```
scala> val aFloat: Float = 1.0 //Compile Error!  
scala> val aFloat: Float = 1.0F //Works  
scala> val aFloat: Float = 1.0f //Works
```

FLOATING POINT LITERALS

- In Scala, to specify a literal to be of Float value, we'll have to give suffix f or F:

```
scala> val aFloat: Float = 1.0  
<console>:11: error: type mismatch;  
found   : Double(1.0)  
required: Float  
    val aFloat: Float = 1.0
```



BOOLEAN LITERALS

- The basic use of Boolean literals is for operating on comparisons, or conditions. These two are called Boolean literal, which can't be replaced by 0 or 1:

```
scala> val aBool: Boolean = 1
<console>:11: error: type mismatch;
found   : Int(1)
required: Boolean
      val aBool: Boolean = 1
```

BOOLEAN LITERALS

- To define a Boolean value, we simply give true or false:

```
scala> val aBool = true
aBool: Boolean = true
scala> val aBool = false
aBool: Boolean = false
```



CHARACTER LITERALS

- We represent Character literals in single quotes.
- Any Unicode character or escape sequence can be represented as a Character literal.
- What's an escape sequence, by the way? Let's take this backslash for example. If we try this:

```
scala> val aChar = '\\'  
<console>:1: error: unclosed character literal  
val aChar = '\\'
```


CHARACTER LITERALS

- To define these characters, we use this sequence:

```
scala> val doublequotes = "\""  
doublequotes: String = "  
scala> val aString = doublequotes + "treatme a  
string" + doublequotes  
aString: String = "treatme a string"
```

CHARACTER LITERALS

- We've a list of escape sequence characters shown in the following table:

Sequence	Value	Unicode
<code>\b</code>	Backspace	<code>\u0008</code>
<code>\t</code>	Horizontal Tab	<code>\u0009</code>
<code>\r</code>	Carriage Return	<code>\u000D</code>
<code>\n</code>	Line Feed	<code>\u000A</code>
<code>\f</code>	Form Feed	<code>\u000C</code>
<code>\"</code>	Double Quote	<code>\u0022</code>
<code>\\</code>	Backslash	<code>\u005C</code>
<code>\'</code>	Single Quote	<code>\u0027</code>

CHARACTER LITERALS

- You can also use the hex code to represent a Character literal, but we need to put a `\u` preceding it:

```
scala> val c = '\u0101'  
c: Char = ā
```



STRING LITERALS

- We'll take a look at how String literals in Scala are different, since there's more than one way to write String literals.
- Up till now we've declared String literals within double quotes:

```
scala> val boringString = "I am a String  
Literal."  
boringString: String = I am a String Literal.
```

STRING LITERALS

```
scala> val interestingString = """I am an Interesting String
    | Also span in multiple Lines!
    | Ok, That's it about me"""
interestingString: String =
"I am an Interesting String
```

```
Also span in multiple Lines!
Ok, That's it about me"
```

STRING LITERALS

- These multi-line string literals treat them as normal characters:

```
scala> val aString = """ / " ' """  
aString: String = " / " ' "  
scala> println(aString)
```

```
/ " '
```



SYMBOL LITERALS

- A symbol has a name, and it can be defined as a single quote (') followed by alphanumeric identifier:

```
scala> val aSymbol = 'givenName  
aSymbol: Symbol = 'givenName
```

```
scala> aSymbol.name  
res10: String = givenName
```



SYMBOL LITERALS

- We can check the absolute type for a symbol:

```
scala> import scala.reflect.runtime.universe._  
import scala.reflect.runtime.universe._
```

```
scala> typeOf[Symbol]  
res12:reflect.runtime.universe.Type=  
scala.reflect.runtime.universe.Symbol
```


TUPLE LITERALS

- First, let's take a look at how we can define a literal of the same type:

```
scala> val aTuple = ("Val1", "Val2", "Val3")
aTuple: (String, String, String) =
(Val1,Val2,Val3)
scala> println("Value1 is: " + aTuple._1)
Value1 is: Val1
```

TUPLE LITERALS

- Take a closer look at the REPL response for the first declaration:

```
aTuple: (String, String, String) =  
(Val1,Val2,Val3)
```



TUPLE LITERALS

- A tuple with two elements is also called a Pair, it can be defined using the arrow assoc (->) operator:

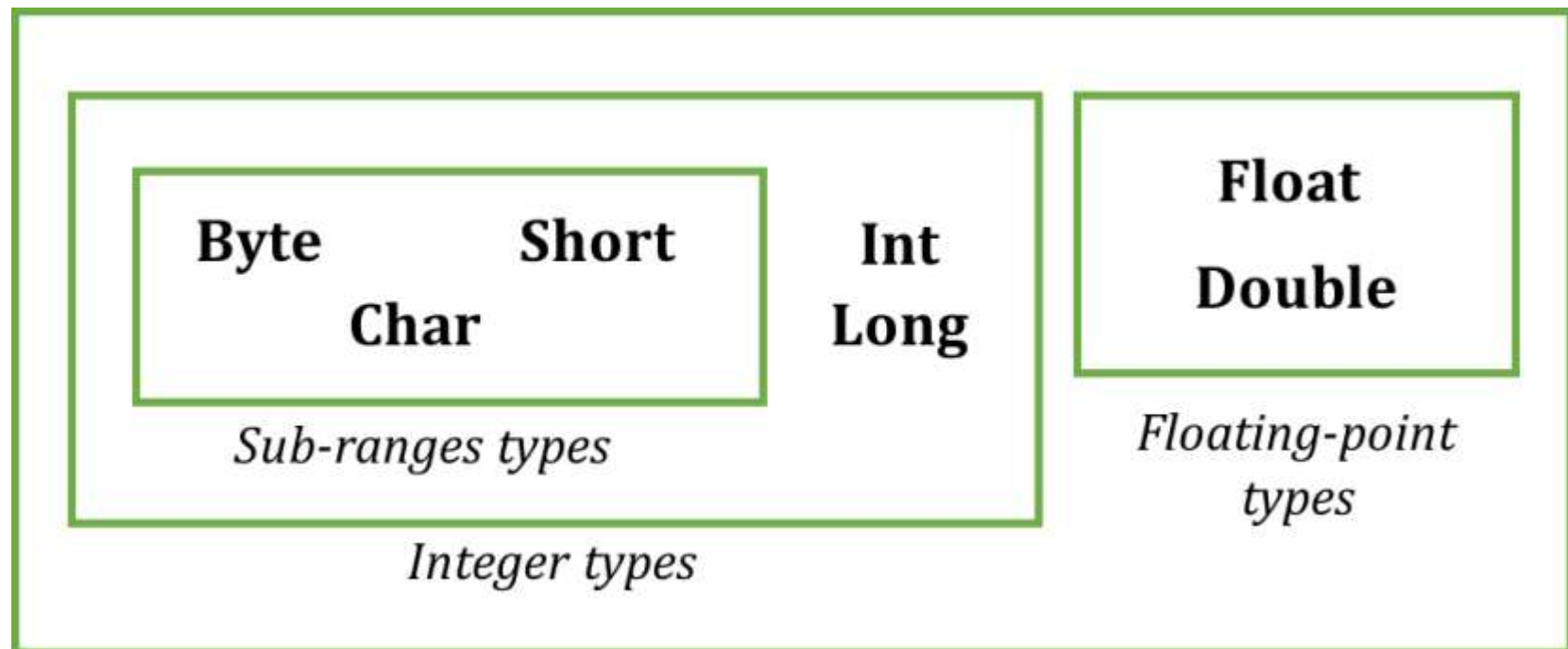
```
scala> val smartPair = 1 -> "One"  
smartPair: (Int, String) = (1,One)
```

FUNCTION LITERALS

- Function literals are a syntactical way of representing a function.
- The basic structure of a function is something that can take some parameters and return a response.
- If we've to represent a function that takes an Int value and respond in String, it will be like this:

`Int => String`

DATA TYPES



DATA TYPES

- let's take an example:

```
scala> val x = 10 //x is an object of Type Int  
x: Int = 10 //x is assigned value 10
```

```
scala> val y = 16 //y is an object of Type Int  
y: Int = 16 //y is assigned value 16
```

```
scala> val z = x + y //z is addition of x and y's  
value  
z: Int = 26
```

DATA TYPES

- The sign + here is a method on the Int object, which means more than just an operator, it's a method that is defined for Int types and expects a parameter of Int type.
- This is going to have a definition similar :

```
scala> def +(x: Int): Int = ??? //Some definition  
$plus: (x: Int)Int
```

DATA TYPES



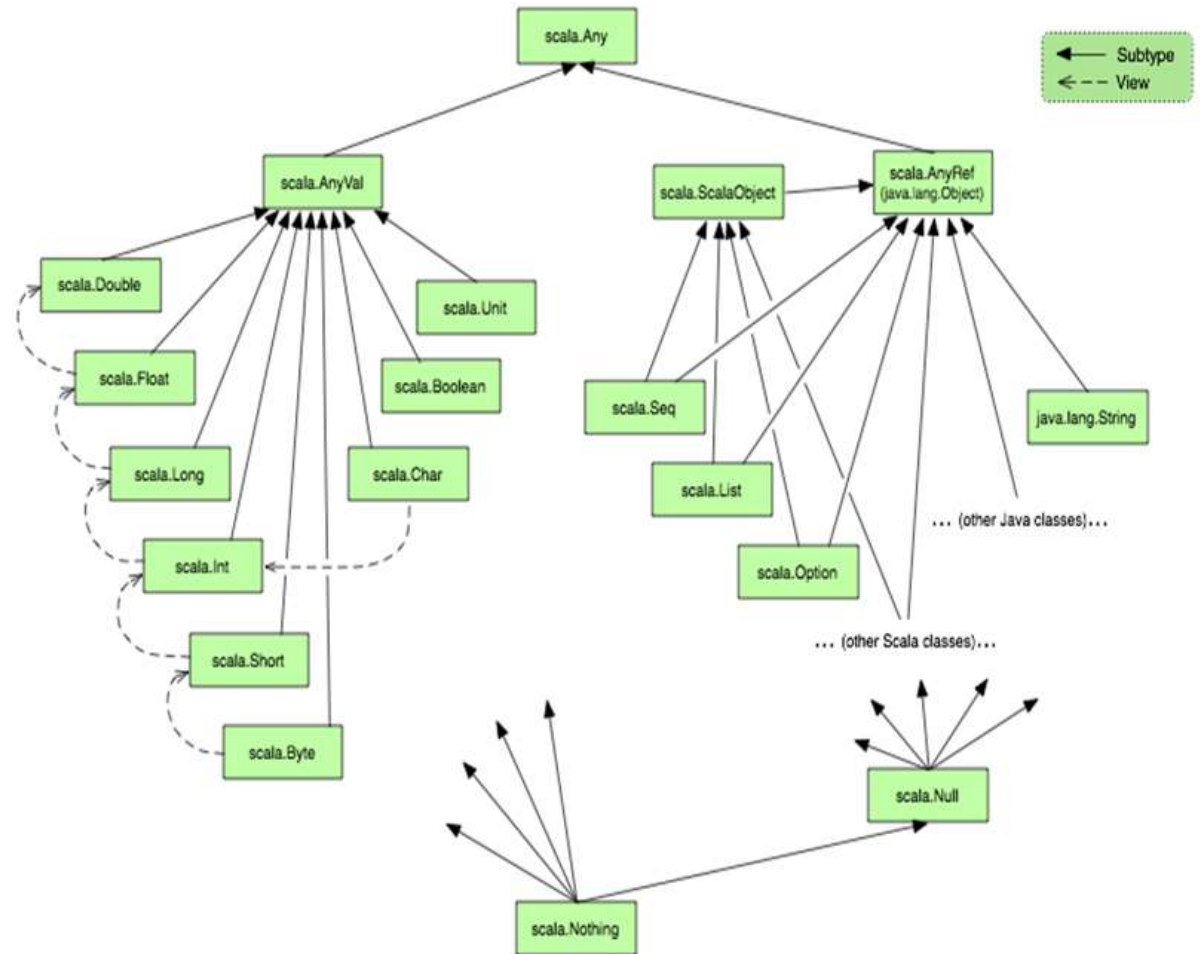
- Let's try this:

```
scala> val aCharAndAnInt = 12 + 'a'  
aCharAndAnInt: Int = 109
```

- This happened because there's a method + overloaded for the type character. Something like this:

```
scala> def +(x: Char): Int = ???  
$plus: (x: Char)Int
```


SCALA'S CLASS HIERARCHY



ANY

```
import java.util.UUID

class Item {
    val id: UUID = UUID.randomUUID()
}

class ElectronicItem(val name: String, val subCategory: String) extends Item {
    val uuid: String = "Elec_" + id
}

object CartApp extends App {

    def showItem(item: ElectronicItem) = println(s"Item id: ${item.id} uuid: ${item.uuid}
name: ${item.name}")

    showItem(new ElectronicItem("Xperia", "Mobiles"))
    showItem(new ElectronicItem("IPhone", "Mobiles"))
}
```

ANY

- The following is the result:

Item id: 16227ef3-2569-42b3-8c5e-b850474da9c4
uuid: Elec_16227ef3-2569-42b3-8c5e-b850474da9c4
name: Xperia

Item id: 1ea8b6af-9cf0-4f38-aefb-cd312619a9d3
uuid: Elec_1ea8b6af-9cf0-4f38-aefb-cd312619a9d3
name: iPhone

ANY

- Methods like `!=` , `==`, `asInstanceOf`, `equals`, `asInstanceOf`, `toString`, and `hashCode` are defined in `Any` class.
- These are in the form of:

```
final def != (that: Any): Boolean
final def == (that: Any): Boolean
def isInstanceOf[a]: Boolean
def equals(that: Any): Boolean
def ##: Int
def hashCode: Int
def toString: String
```

NULL AND NOTHING

- Null and Nothing are called Bottom types in Scala.
- Why do we need these Bottom types? Take a look at the code snippet:

```
def checkIF10AndReturn20(x: Int): Int = {  
    if(x == 10)  
        x * 2  
    else  
        throw new Exception("Sorry, Value wasn't 10")  
}
```

NULL AND NOTHING

Visualize Types:

Branch One

[Int] -> -> AnyVal -> Any

Branch Two

Nothing -> ... -> [Int] -> -> AnyVal -> Any



TYPE INFERENCE

- We can call type inference a built-in Scala feature that permits us to omit type information while writing code.
- This means we don't have to specify the type while declaring any variables; Scala compiler can do it for us:

```
scala> val treatMeAString = "Invisible"  
treatMeAString: String = Invisible
```

TYPE INFERENCE

- We did not specify our `val`, to be of `String` type, but seeing the value of `Invisible`, Scala compiler was able to infer its type.
- Also with some constraints, we can also omit the method's return types:

```
def checkMeImaString(x: Boolean) = if(x)  
  "True" else "False"
```


TYPE INFERENCE

- For recursive methods, this doesn't work.
- The famous factorial method expects you to specify the return type if implementation is recursive:

```
def recursiveFactorial(n: Int) = if(n == 0) 1  
else recursiveFactorial(n-1)  
//Recursive method recursiveFactorial needs  
result type
```

TYPE INFERENCE

- Scala compiler uses this approach to deduce constraints and then apply unification (explanation is beyond the scope of this course) to infer the type.
- In instances where we can't take out any statements about an expression, inferring type is impossible:

```
scala> val x = x => x  
<console>:11: error: missing parameter type  
      val x = x => x
```

TYPE INFERENCE

- Because of type inference only, we're able to use syntactic sugar for cases where we're not expected to specify types:

```
scala> List(1,4,6,7,9).filter(_+1 > 5)  
res0: List[Int] = List(6, 7, 9)
```

OPERATORS IN SCALA

- We use operators to perform some operation on operands, which is obvious, and the way we implement makes them infix, prefix, or postfix.
- A basic example of an infix operator is addition +:

```
scala> val x = 1 + 10
x: Int = 11
```



OPERATORS IN SCALA

- In our case, the addition (+) method is defined for Int.
- Along with this, there are several versions of overloaded methods that support other numeric value types as well.
- It means that we can pass in any other type and it'll be a normal addition operation performed, given that the overloaded version of that method is present:

```
scala> val y = 1 + 'a'  
y: Int = 98
```

OPERATORS IN SCALA

```
class Amount(val amt: Double) {  
    def taxApplied(tax: Double) = this.amt * tax/100 + this.amt  
}  
  
object Order extends App {  
    val tax = 10  
    val firstOrderAmount = 130  
  
    def amountAfterTax(amount: Amount) = amount taxApplied tax  
  
    println(s"Total Amount for order:: ${amountAfterTax(new Amount(firstOrderAmount))}")  
}
```

OPERATORS IN SCALA

- Operators are a way we can use methods, which is why we have this operator.
- We've used it in `object,Order` while defining a function, `amountAfterTax`:

> `amount taxApplied tax`

OPERATORS IN SCALA

- It can also be written as `amount.taxApplied(tax)`.
- There are also a few examples in Scala; for example, the `indexOf` operator that works on `String`:

```
scala> val firstString = "I am a String"  
firstString: String = I am a String
```

```
scala> firstString indexOf 'a'  
res1: Int = 2
```


OPERATORS IN SCALA

- The first one, prefix operators, sits before an operand.
- Examples of these are -, !, and so on:

```
scala> def truthTeller(lie: Boolean) = !lie  
truthTeller: (lie: Boolean)Boolean
```

```
scala> truthTeller(false)  
res2: Boolean = true
```

OPERATORS IN SCALA

- What happens in the background is that Scala uses `unary_!` to call these operators, and that's obvious because these operators use only one operand.
- So our implementation looks something like the following:

```
scala> def truthTeller(lie: Boolean) =  
lie.unary_  
truthTeller: (lie: Boolean)Boolean
```

OPERATORS IN SCALA

```
scala> 1.toString  
res4: String = 1
```

```
scala> "1".toInt  
res5: Int = 1
```

```
scala> "ABC".toLowerCase  
res7: String = abc
```



ARITHMETIC OPERATORS

- Let's take examples of others:

```
scala> val x = 10 - 1  
x: Int = 9  
scala> val y = 10 * 1  
y: Int = 10  
scala> val z = 10 / 1  
z: Int = 10  
scala> val yx = 10 % 9  
yx: Int = 1
```



ARITHMETIC OPERATORS

- These operators have their overloaded versions also defined, to see that we may try with different types as operands.
- Let's take an Int and Double:

```
scala> val yx = 10 % 9.0  
yx: Double = 1.0
```

RELATIONAL OPERATORS

```
scala> val equal_op = 10 == 10  
equal_op: Boolean = true
```

```
scala> val not_eq_op = 10 != 10  
not_eq_op: Boolean = false
```

```
scala> val gt_than_op = 10 > 10  
gt_than_op: Boolean = false
```

```
scala> val gt_than_op = 11 > 10  
gt_than_op: Boolean = true
```

```
scala> val lt_than_op = 11 < 10  
lt_than_op: Boolean = false
```

```
scala> val gt_eq_op = 11 >= 11  
gt_eq_op: Boolean = true
```

```
scala> val lt_eq_op = 11 <= 11  
lt_eq_op: Boolean = true
```

LOGICAL OPERATORS

- Logical operators include ! (NOT), && (AND), and || (OR), and obviously we use these to perform logical operations on operands.
- These methods are written for Boolean, so they expect Boolean operands:

```
scala> val log_not = !true
```

```
log_not: Boolean = false
```

```
scala> val log_or = true || false
```

```
log_or: Boolean = true
```

```
scala> val log_and = true && true
```

```
log_and: Boolean = true
```

BITWISE OPERATORS

- We can perform operations on individual bits of Integer types using Bitwise operators.
- These includes Bitwise AND (&), OR (|), and XOR (^):

```
scala> 1 & 2  
res2: Int = 0  
scala> 1 | 2  
res3: Int = 3  
scala> 1 ^ 2  
res5: Int = 3
```


BITWISE OPERATORS

- We can perform a logical not-operating using ~ operator:

```
scala> ~2  
res8: Int = -3
```



OPERATOR PRECEDENCE

- Operations such as $2 + 3 * 4 / 2 - 1$ can give different results if there's no rule for evaluation of these.
- Hence we have some precedence-based rules for these.
- We're going to talk about it in this part:

```
scala> 2 + 3 * 4 / 2 - 1  
res15: Int = 7
```

OPERATOR PRECEDENCE

Operator Precedence:

! ~
* / %
+ -
>> >>> <<
:
= !
> >= < <=
&
^
|
&&
||

OPERATOR PRECEDENCE

- If operators of the same precedence level appear together, the operands are evaluated from left to right.
- It means that the expression $1 + 2 + 3 * 3 * 4 - 1$ will result in 38:

```
scala> 1 + 2 + 3 * 3 * 4 - 1  
res16: Int = 38
```

WRAPPER CLASSES

<i>Rich Wrapper Classes:</i>	
<i>Base Type</i>	<i>Wrapper</i>
Byte	scala.runtime.RichByte
Short	scala.runtime.RichShort
Char	scala.runtime.RichChar
Int	scala.runtime.RichInt
Boolean	scala.runtime.RichBoolean
Float	scala.runtime.RichFloat
Double	scala.runtime.RichDouble
String	scala.collection.immutable.StringOps

WRAPPER CLASSES

- To see how it happens, let's see an example:

```
scala> val x = 10  
x: Int = 10
```

```
scala> x.isValidByte  
res1: Boolean = true
```

WRAPPER CLASSES

- The preceding expression tries to check if the value of x can be converted into a Byte, and suffices range of a Byte, and finds it to be true:

```
scala> val x = 260
x: Int = 260
scala> x.isValidByte
res2: Boolean = false
scala> val x = 127
x: Int = 127
scala> x.isValidByte
res3: Boolean = true
```



WRAPPER CLASSES

- For example, the `charAt` method does pretty good here:

```
scala> val x = "I am a String"
x: String = I am a String
scala> x.charAt(5)
res13: Char = a
```


WRAPPER CLASSES

- Now let's try some methods from StringOps:

```
scala> x.capitalize  
res14: String = I am a String
```

```
scala> x.toUpperCase  
res15: String = I AM A STRING
```

```
scala> x.toLowerCase  
res16: String = i am a string
```

```
scala> val rangeOfNumbers = 1 to 199
rangeOfNumbers: scala.collection.immutable.Range.Inclusive = Range 1 to 199

scala> val rangeOfNumbersUntil = 1 until 199
rangeOfNumbersUntil: scala.collection.immutable.Range = Range 1 until 199

scala> rangeOfNumbers contains 1
res17: Boolean = true

scala> rangeOfNumbersUntil contains 1
res18: Boolean = true

scala> rangeOfNumbersUntil contains 199
res19: Boolean = false

scala> rangeOfNumbers contains 199
res20: Boolean = true
```

WRAPPER CLASSES

- The first includes both values we use to build a Range; the latter includes only the beginning value.
- We've tried all these, As you can see, `rangeOfNumbersUntil` does not contain 199.
- We can also create a Range with some step difference:

```
scala> 1 to 10 by 2 foreach println
```

STRING INTERPOLATORS

- We've already used String Interpolators, it's hard to avoid using them when they are available to you.
- Remember when we used them? Yes! When we were learning to create operators on our own:

```
println(s"Total Amount for order::  
${amountAfterTax(new Amount(firstOrderAmount))}")
```

STRING INTERPOLATORS

- We can use any variable with a \$ and it'll be replaced by its value:

```
scala> val myAge = s"I completed my $age."  
myAge: String = I completed my 25.
```



THE S INTERPOLATOR

- Now, let's take an example that takes on expressions:

```
scala> val nextYearAge = s"Next Year, I'll  
complete ${age + 1}."  
nextYearAge: String = Next Year, I'll complete  
26.
```

THE S INTERPOLATOR

- It can be any expression. An arithmetic operation like we just did, or a method call:

```
scala> def incrementBy1(x: Int) = x + 1  
incrementBy1: (x: Int)Int
```

```
scala> val nextYearAge = s"Next Year, I'll  
complete ${incrementBy1(age)}."  
nextYearAge: String = Next Year, I'll complete 26.
```

F INTERPOLATOR

- To have something like printf styled formatting in Scala, we can use the f interpolator.
- We do this by using a f preceding the double quotes of our string, and then within the String we can use one of the format specifiers:

```
scala> val amount = 100
```

```
amount: Int = 100
```

```
scala> val firstOrderAmount = f"Your total amount  
is: $amount%.2f"
```

```
firstOrderAmount: String = Your total amount is:  
100.00
```


<i>Format Specifiers</i>	
<i>Specifiers</i>	<i>Description</i>
%c	Characters
%d	Decimal Numbers
%e	Exponentials
%f	Floating Point Numbers
%i	Integers
%o	Octal Numbers
%u	Unsigned decimal number
%x	Hexadecimal Number

THE RAW INTERPOLATOR

- The final one pre-existing interpolator in Scala is the raw interpolator.
- The way we write raw interpolator is almost similar to the other two interpolators. We precede our String with a raw keyword and it works for us:

```
scala> val rawString = raw"I have no escape \n  
character in the String \n "  
rawString: String = "I have no escape \n  
character in the String \n "
```

THE RAW INTERPOLATOR

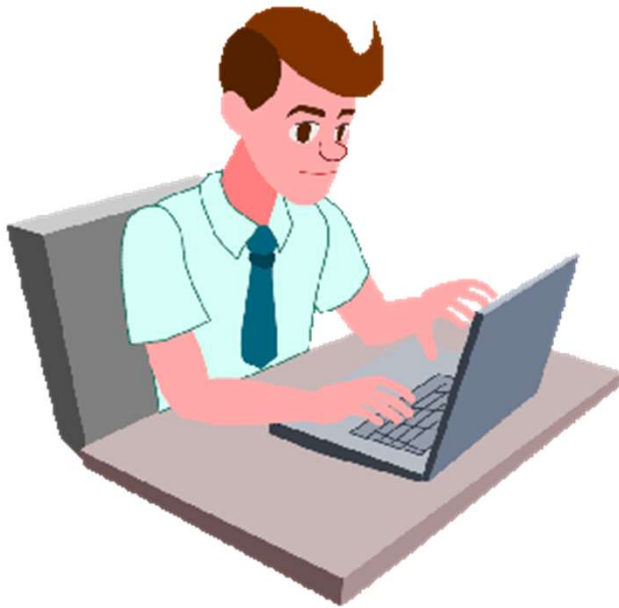
- In a normal string, `\n` would have converted into a newline character.

```
scala> val rawString = "I have no escape \n  
character in the String \n "  
rawString: String =  
"I have no escape  
character in the String  
"
```

SUMMARY



- We started with the most basic `val` and `var` variable constructs.
- Then, we learned how we can write literals, and what data types we have in Scala.
- We then studied the interesting class hierarchy in Scala, in which we talked about unified class hierarchy and value and reference type classes.
- Later, we learned one of the most important concepts of Type Inference in Scala.



"COMPLETE LAB"

3: SHAPING OUR SCALA PROGRAM



SHAPING OUR SCALA PROGRAM

A quick card for what's in there for us in this lesson:

- Looping(for, while, and do while loops)
- The for expressions: a quick go-through
- Recursion
- Conditional statements(if, and if else)
- Pattern matching

LOOPING

Take a look at the following program:

```
object PagePrinter extends App {  
  
  /*  
   * Prints pages page 1 to lastIndex for doc  
   */  
  def printPages(doc: Document, lastIndex: Int) = ??? //Yet to be defined  
  
  /*  
   * Prints pages page startIndex to lastIndex for doc  
   */  
  def printPages(doc: Document, startIndex: Int, lastIndex: Int) = ???  
  
  /*  
   * Prints pages with given Indexes for doc  
   */  
  def printPages(doc: Document, indexes: Int*) = ???  
  
  /*  
   * Prints pages  
   */  
  private def print(index: Int) = println(s"Printing Page $index.")  
}  
  
/*  
 * Declares a Document type with two arguments numOfPages, typeOfDoc  
 */  
case class Document(numOfPages: Int, typeOfDoc: String)
```


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:

```
scala> val stocks = List("APL", "GOOG", "JLR", "TESLA")  
stocks: List[String] = List(APL, GOOG, JLR, TESLA)
```

```
scala> stocks.foreach(x => println(x))  
APL  
GOOG  
JLR  
TESLA
```

```

object PagePrinter extends App{

  /*
   * Prints pages page 1 to lastIndex for doc
   */
  def printPages(doc: Document, lastIndex: Int) = if(lastIndex <= doc.numOfPages) for(i
<- 1 to lastIndex) print(i)

  /*
   * Prints pages page startIndex to lastIndex for doc
   */
  def printPages(doc: Document, startIndex: Int, lastIndex: Int) = if(lastIndex <=
doc.numOfPages && startIndex > 0 && startIndex < lastIndex) for(i <- startIndex to
lastIndex) print(i)

  /*
   * Prints pages with given Indexes for doc
   */
  def printPages(doc: Document, indexes: Int*) = for(index <- indexes if index <=
doc.numOfPages && index > -1) print(index)

  /*
   * Prints pages
   */

```

```

private def print(index: Int) = println(s"Printing Page $index.")

    println("-----Method V1-----")
    printPages(Document(15, "DOCX"), 5)

    println("-----Method V2-----")
    printPages(Document(15, "DOCX"), 2, 5)

    println("-----Method V3-----")
    printPages(Document(15, "DOCX"), 2, 5, 7, 15)

}

/*
 * Declares a Document type with two arguments numOfPages, typeOfDoc
 */
case class Document(numOfPages: Int, typeOfDoc: String)

```

THE FOR LOOP

The following is
the output:

```
-----Method V1-----  
Printing Page 1.  
Printing Page 2.  
Printing Page 3.  
Printing Page 4.  
Printing Page 5.  
-----Method V2-----  
Printing Page 2.  
Printing Page 3.  
Printing Page 4.  
Printing Page 5.  
-----Method V3-----  
Printing Page 2.  
Printing Page 5.  
Printing Page 7.  
Printing Page 15.
```

THE WHILE LOOP

- Like in most other languages, the while loop is another looping construct used.
- The while loop can do any repetitive task until a condition is satisfied.
- It means that the condition provided has to be true for the code execution to stop, Generic syntax for the while loop is:

```
while (condition check (if it's true))  
    ... // Block of Code to be executed
```

THE WHILE LOOP

- The condition to be checked is going to be a Boolean expression.
 - It gets terminated when the condition is false.
- One of the ways we can use them is:

```
scala> val stocks = List("APL", "GOOG", "JLR", "TESLA")
stocks: List[String] = List(APL, GOOG, JLR, TESLA)
```

```
scala> val iteraatorForStocks = stocks.iterator
iteraatorForStocks: Iterator[String] = non-empty iterator
```

```
scala> while(iteraatorForStocks.hasNext) println(iteraatorForStocks.next())
APL
GOOG
JLR
TESLA
```

THE DO WHILE LOOP

- The do while loop does not differ a lot from the while loop. Generic syntax for do while loop is:

do

```
... // Block of Code to be executed  
while(condition check (if it's true))
```

- The do while loop ensures that the code in block gets executed at least once and then checks for the condition defined in a while expression:

```
scala> do println("I'll stop by myself after 1 time!")  
while(false)
```

THE FOR EXPRESSIONS

```
object ForExpressions extends App {  
  
    val person1 = Person("Albert", 21, 'm')  
    val person2 = Person("Bob", 25, 'm')  
    val person3 = Person("Cyril", 19, 'f')  
  
    val persons = List(person1, person2, person3)  
  
    for {  
        person <- persons  
        age = person.age  
        name = person.name  
        if age > 20 && name.startsWith("A")  
    } {  
        println(s"Hey ${name} You've won a free Gift Hamper.")  
    }  
  
    case class Person(name: String, age: Int, gender: Char)  
}
```

The following is the result:

Hey Albert You've won a free Gift Hamper.

THE FOR EXPRESSIONS

<i>For Expressions</i>		
<i>Term</i>	<i>Ex.</i>	<i>Description</i>
Generator	<code>person <- <i>persons</i></code>	Generates a new element from sequence
Definition	<code>age = <i>person.age</i></code>	Defines a value in scope
Filter	<code><i>if</i> age > 20</code>	Filters out a value from scope

THE FOR YIELD EXPRESSIONS

```
object ForYieldExpressions extends App {  
  
    val person1 = Person("Albert", 21, 'm')  
    val person2 = Person("Bob", 25, 'm')  
    val person3 = Person("Cyril", 19, 'f')  
  
    val persons = List(person1, person2, person3)  
  
    val winners = for {  
        person <- persons  
        age = person.age  
        name = person.name  
        if age > 20  
    } yield name  
  
    winners.foreach(println)  
  
    case class Person(name: String, age: Int, gender: Char)  
}
```

The following is the result:

Albert
Bob

```
object RecursionEx extends App {  
  /*  
   * 2 to the power n  
   * only works for positive integers!  
   */  
  def power2toN(n: Int): Int = if(n == 0) 1 else 2 * power2toN(n - 1)  
  
  println(power2toN(2))  
  println(power2toN(4))  
  println(power2toN(6))  
}
```

The following is the result:

```
4  
16  
64
```

RECURSION

RECURSION

- Consider the following:

```
def power2toN(n: Int) = if(n == 0) 1 else (2 *  
power2toN(n - 1))
```

- The Scala compiler gives an error stating Recursive method power2N needs result type.
- This is a required condition by the Scala compiler.

RECURSION

```
import scala.annotation.tailrec

object TailRecursionEx extends App {

  /*
   * 2 to the power n
   * @tailrec optimization
   */
  def power2toNTail(n: Int): Int = {
    @tailrec
    def helper(n: Int, currentVal: Int): Int = {
      if(n == 0) currentVal else helper(n - 1, currentVal * 2)
    }
    helper(n, 1)
  }

  println(power2toNTail(2))
  println(power2toNTail(4))
  println(power2toNTail(6))
}
```

The following is the result:

```
4
16
64
```

CONDITIONAL STATEMENTS

The if else conditional expression

- In Scala, you can use if else to control program flow.
- The generic syntax for an if else statement goes as follows:

```
if (condition (is true))  
    ... //Block of code to be executed  
else  
    ... //Block of code to be executed
```

```
scala> val age = 17  
age: Int = 17  
scala> if(age > 18) println("You're now responsible adult.")  
else println("You should grow up.")  
You should grow up.
```

CONDITIONAL STATEMENTS

- Instead of just printing out strings, we can perform any operation as part of the control flow.
- In Scala, we can also declare and assign a value to our variables using if else expressions:

```
scala> val marks = 89  
marks: Int = 89
```

```
scala> val performance = if(marks >= 90) "Excellent"  
else if(marks > 60 && marks < 90) "Average" else "Poor"  
performance: String = Average
```

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.

PATTERN MATCHING

<i>Java's switch vs Scala's Pattern Matching</i>	
<i>Java's switch statements</i>	<i>Scala's Pattern Matching</i>
<pre>switch (expression){ case value1: <i>//code to be executed;</i> break; <i>//optional</i> case value2: <i>//code to be executed;</i> break; <i>//optional</i> case value3: <i>//code to be executed;</i> break; <i>//optional</i> default: <i>//code to be executed if all cases</i> <i>above are not matched;</i> }</pre>	<pre>value match { case value1 => <i>//Code Block to be</i> <i>executed</i> case value2 => <i>//Code Block to be</i> <i>executed</i> case value3 => <i>//Code Block to be</i> <i>executed</i> case _ => <i>//Code Block to be executed</i> }</pre>

PATTERN MATCHING

```
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")  
  }  
  
  matchAgainst(5)  
}
```

The following is the result:

```
Exception in thread "main" scala.MatchError: 5 (of class java.lang.Integer)  
    at PatternMatching$.matchAgainst(PatternMatching.scala:6)  
    at PatternMatching$.delayedEndpoint$PatternMatching$1(PatternMatching.scala:13)  
    at PatternMatching$delayedInit$body.apply(PatternMatching.scala:4)  
    at scala.Function0.apply$mcV$sp(Function0.scala:34)  
    at scala.Function0.apply$mcV$sp$(Function0.scala:34)  
    at scala.runtime.AbstractFunction0.apply$mcV$sp(AbstractFunction0.scala:12)
```

```
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)  
}
```

The following is the result:

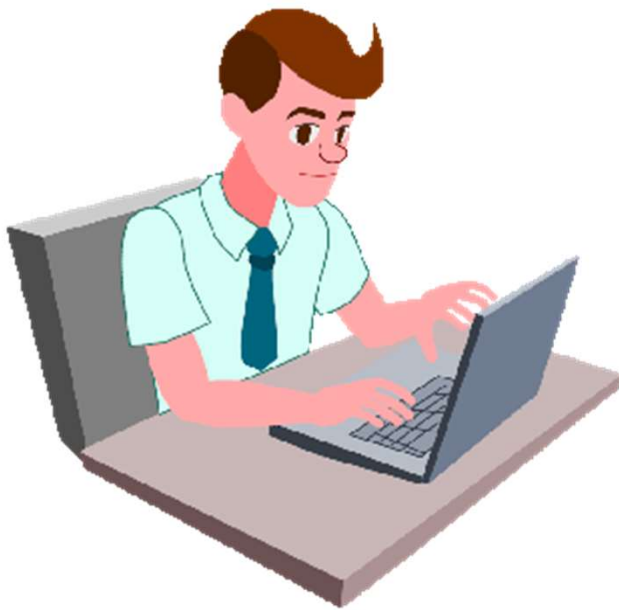
```
One  
Not in Range 1 to 4
```

PATTERN MATCHING

SUMMARY



- We discussed native looping constructs such as for, while, and do while loops. After that, we saw for expressions, along with for yield expressions.
- Then we understood alternatives to iteration, that is, recursion.
- We wrote a few recursive functions as well, Finally, we looked at if else conditional statements and pattern matching.

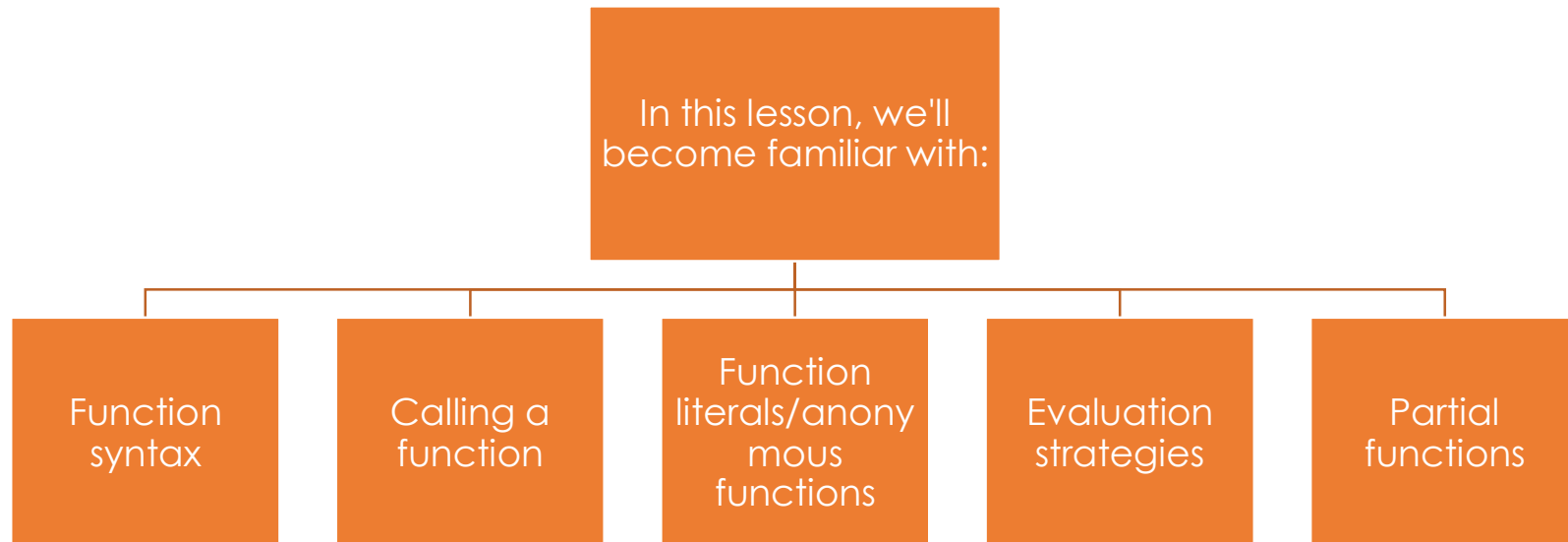


"COMPLETE LAB"

4: GIVING MEANING TO PROGRAMS WITH FUNCTIONS



GIVING MEANING TO PROGRAMS WITH FUNCTIONS



FUNCTION SYNTAX

- Functions in Scala can be written using the `def` keyword, followed by the name of the function, with some arguments supplied as inputs to the function.
- Let's take a look at the generic syntax for a function:

`modifiers...`

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


FUNCTION SYNTAX

- After declaring the function, we can give the definition body.
- Let's take a look at some concrete examples:

```
def compareIntegers(value1: Int, value2: Int):  
Int = if (value1 == value2) 0 else if (value1 >  
value2) 1 else -1
```

FUNCTION SYNTAX

- A recommended practice to define your function definition inline is when your function signature, along with the definition, is 30 characters or so.
- If it's more but still concise, we can start the definition on the next line, as follows:

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

FUNCTION SYNTAX

- The choice is yours; to make your code more readable, you may choose to define functions inline.
- If there are multiple lines in a function body, you may choose to encapsulate them within a pair of curly braces:

```
def compareIntegersV2(value1: Int, value2: Int): Int =  
{  
    println(s" Executing V2")  
    if (value1 == value2) 0 else if (value1 > value2) 1  
    else -1  
}
```

```

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
  }

  def compareIntegersV2(value1: Int, value2: Int): Int =
    if (value1 == value2) 0 else if (value1 > value2) 1 else -1

  println(compareIntegers(1, 2))
  println(compareIntegersV1(2, 1))
  println(compareIntegersV2(2, 2))
}

```

NESTING OF FUNCTIONS

```
object FunctionSyntaxOne extends App {  
  
  def compareIntegersV4(value1: Int, value2: Int): String = {  
    println("Executing V4")  
    val result = if (value1 == value2) 0 else if (value1 > value2) 1 else -1  
    giveAMeaningFullResult(result, value1, value2)  
  }  
  
  private def giveAMeaningFullResult(result: Int, value1: Int, value2: Int) = result  
  match {  
    case 0 => "Values are equal"  
    case -1 => s"$value1 is smaller than $value2"  
    case 1 => s"$value1 is greater than $value2"  
    case _ => "Could not perform the operation"  
  }  
  
  println(compareIntegersV4(2,1))  
}
```

NESTING OF FUNCTIONS

```
object FunctionSyntaxTwo extends App {  
  
  def compareIntegersV5(value1: Int, value2: Int): String = {  
    println("Executing V5")  
  
    def giveAMeaningFullResult(result: Int) = result match {  
      case 0 => "Values are equal"  
      case -1 => s"$value1 is smaller than $value2"  
      case 1 => s"$value1 is greater than $value2"  
      case _ => "Could not perform the operation"  
    }  
  
    val result = if (value1 == value2) 0 else if (value1 > value2) 1 else -1  
    giveAMeaningFullResult(result)  
  }  
  
  println(compareIntegersV5(2,1))  
}
```

CALLING A FUNCTION

Passing a variable number of arguments

- If you remember, we've already seen an example for functions that take a variable number of arguments and perform operations on them in the previous lesson:

```
/*  
 * Prints pages with given Indexes for doc  
 */  
def printPages(doc: Document, indexes: Int*) =  
  for(index <- indexes if index <= doc.numOfPages)  
    print(index)
```

It may be a signature with the `def` keyword, with a name and parameters, or just one *vararg*:

```
def average(numbers: Int*): Double = ???
```

The preceding code is the signature for our `average` function. The body for the function is yet to be defined:

```
object FunctionCalls extends App {  
  
  def average(numbers: Int*) : Double = numbers.foldLeft(0)((a, c) => a + c) /  
  numbers.length  
  
  def averageV1(numbers: Int*) : Double = numbers.sum / numbers.length  
  
  println(average(2,2))  
  println(average(1,2,3))  
  println(averageV1(1,2,3))  
  
}
```


CALLING A FUNCTION

- In the same way, we can define our function to support a variable number of arguments of any type.
- We can call the function accordingly.
- The only requirement is that the vararg parameter should come last in the function signature's parameters list:

```
def averageV1(numbers: Int*, wrongArgument: Int):  
Double = numbers.sum / numbers.length
```

CALLING A FUNCTION WITH A DEFAULT PARAMETER VALUE

```
def compareIntegersV6(value1: Int, value2: Int = 10): String = {  
  println("Executing V6")  
  
  def giveAMeaningFullResult(result: Int) = result match {  
    case 0 => "Values are equal"  
    case -1 => s"$value1 is smaller than $value2"  
    case 1 => s"$value1 is greater than $value2"  
    case _ => "Could not perform the operation"  
  }  
  
  val result = if (value1 == value2) 0 else if (value1 > value2) 1 else -1  
  giveAMeaningFullResult(result)  
}  
  
println(compareIntegersV6(12))
```

CALLING A FUNCTION WITH A DEFAULT PARAMETER VALUE

- The following is the result of previous code:

Executing V6

12 is greater than 10

- Here, while declaring the `compareIntegersV6` function, we gave a default value of 10 to parameter `value2`.
- At the end while calling the function, we passed only one argument:

`compareIntegersV6(12)`

CALLING A FUNCTION WITH A DEFAULT PARAMETER VALUE

- In cases of ambiguity, it does not allow you to call a function.
- Let's take an example:

```
def compareIntegersV6(value1: Int = 10, value2:  
Int) = ???
```

- For this function, let's try to call using the following function call:

```
println(compareIntegersV6(12)) // Compiler won't  
allow
```

CALLING A FUNCTION WHILE PASSING NAMED ARGUMENTS

```
def compareIntegersV6(value1: Int = 10, value2: Int): String = {  
  println("Executing V6")  
  
  def giveAMeaningFullResult(result: Int) = result match {  
    case 0 => "Values are equal"  
    case -1 => s"$value1 is smaller than $value2"  
    case 1 => s"$value1 is greater than $value2"  
    case _ => "Could not perform the operation"  
  }  
  
  val result = if (value1 == value2) 0 else if (value1 > value2) 1 else -1  
  giveAMeaningFullResult(result)  
}  
  
println(compareIntegersV6(value2 = 12))
```

CALLING A FUNCTION WHILE PASSING NAMED ARGUMENTS

- It's possible to call our function like this:

```
println(compareIntegersV6(value2 = 12, value1 =  
10))
```

- The following is the result:

```
Executing V6  
10 is smaller than 12
```

FUNCTION LITERALS

- We can pass a function in the form of a literal to another function, to work for us.
- Let's take the same `compareIntegers` function example:

```
def compareIntegersV6(value1: Int = 10, value2:  
Int): Int = ???
```

- If we take a look at the abstract form of our function, it will look like this:

```
(value1: Int, value2: Int) => Int
```

FUNCTION LITERALS

- We can say that this is in its literal form, also called function literals. Hence, it's also possible to assign this literal to any variable:

```
val compareFuncLiteral = (value1: Int, value2: Int) =>  
if (value1 == value2) 0 else if (value1 > value2) 1  
else -1
```

- Remember in PagePrinter from the last lesson, we had a print function that took an index and printed that page:

```
private def print(index: Int) = println(s"Printing Page  
$index.")
```


FUNCTION LITERALS

- If we look at the form our function takes, it takes an integer and prints pages.
- So the form will look as follows:

```
(index: Int) => Unit
```

FUNCTION LITERALS

```
object ColorPrinter extends App {  
  
  def printPages(doc: Document, lastIndex: Int, print: (Int) => Unit) = if(lastIndex <=  
doc.numOfPages) for(i <- 1 to lastIndex) print(i)  
  
  val colorPrint = (index: Int) => println(s"Printing Color Page $index.")  
  
  val simplePrint = (index: Int) => println(s"Printing Simple Page $index.")  
  
  println("-----Method V1-----")  
  printPages(Document(15, "DOCX"), 5, colorPrint)  
  
  println("-----Method V2-----")  
  printPages(Document(15, "DOCX"), 2, simplePrint)  
}  
  
case class Document(numOfPages: Int, typeOfDoc: String)
```

FUNCTION LITERALS

- The filter function expects a predicate that checks for a condition and responds with a Boolean response, based on which we filter out elements in a list or collection:

```
scala> val names =  
List("Alice","Allen","Bob","Catherine","Alex")  
names: List[String] = List(Alice, Allen, Bob, Catherine,  
Alex)
```

```
scala> val nameStartsWithA = names.filter((name) =>  
name.startsWith("A"))  
nameStartsWithA: List[String] = List(Alice, Allen, Alex)
```

FUNCTION LITERALS

- The part where we checked if the name starts with A is an example of a function literal:

```
(name) => name.startsWith("A")
```

- The Scala compiler only requires extra information where it is needed to infer type information; with this, it allows us to omit the parts that are just extra syntax, hence it's possible to write the proceeding syntax as:

```
scala> val nameStartsWithA =  
names.filter(_.startsWith("A"))  
nameStartsWithA: List[String] = List(Alice, Allen,  
Alex)
```

CALL BY NAME

- For that we can refactor our function:

```
def printPages(doc: Document, lastIndex: Int, print:  
(Int) => Unit, isPrinterOn: () => Boolean) = {  
  
    if(lastIndex <= doc.numOfPages && isPrinterOn()) for(i  
    <- 1 to lastIndex) print(i)  
  
}
```

- To call this function, we can use:

```
printPages(Document(15, "DOCX"), 16, colorPrint, () =>  
!printerSwitch)
```

```

object ColorPrinter extends App {

    val printerSwitch = false

    def printPages(doc: Document, lastIndex: Int, print: (Int) => Unit, isPrinterOn: => Boolean) = {

        if(lastIndex <= doc.numOfPages && isPrinterOn) for(i <- 1 to lastIndex) print(i)

    }

    val colorPrint = (index: Int) => {
        println(s"Printing Color Page $index.")
    }

    println("-----Method V1-----")
    printPages(Document(15, "DOCX"), 2, colorPrint, !printerSwitch)

}

case class Document(numOfPages: Int, typeOfDoc: String)

```

EVALUATION STRATEGIES

- Take a closer look and you'll be able to see that we removed the () parenthesis and added => in our function signature.
- This makes our code understand that this is a by name parameter, and to evaluate it only when it's called.
- This is the reason we are allowed to make this call:

```
printPages(Document(15, "DOCX"), 2, colorPrint,  
!printerSwitch)
```

CALL BY VALUE

- Call by value is a simple and common evaluation strategy, where an expression is evaluated and the result is bound to the parameter. We've already seen many examples for this strategy:

```
def compareIntegers(value1: Int, value2: Int):  
  Int =  
    if (value1 == value2) 0 else if (value1 >  
value2) 1 else -1
```

```
compareIntegers(10, 8)
```


PARTIAL FUNCTIONS

- To understand more, let's first define a partial function:

```
scala> val oneToFirst: PartialFunction[Int, String] = {  
    | case 1 => "First"  
    | }  
oneToFirst: PartialFunction[Int, String] =  
<function1>  
scala> println(oneToFirst(1))  
First
```

PARTIAL FUNCTIONS

- In the preceding code, we defined a partial function named `oneToFirst`.
- We also specified type parameters for our partial function; in our case we passed `Int`, `String`.
- The `PartialFunction` function is a trait in Scala, defined as:

```
trait PartialFunction[-A, +B] extends (A) => B
```

PARTIAL FUNCTIONS

- let's say 2, it'll throw a MatchError:

```
scala> println(oneToFirst(2))
scala.MatchError: 2 (of class java.lang.Integer)
  at scala.PartialFunction$$anon$1.apply(PartialFunction.scala:254)
  at scala.PartialFunction$$anon$1.apply(PartialFunction.scala:252)
  at $anonfun$1.applyOrElse(<console>:12)
  at $anonfun$1.applyOrElse(<console>:11)
  at scala.runtime.AbstractPartialFunction.apply(AbstractPartialFunction.scala:34)
```

PARTIAL FUNCTIONS

- We can check whether the partial function is applicable for a value or not using the `isDefinedAt` method:

```
scala> oneToFirst.isDefinedAt(1)  
res3: Boolean = true
```

```
scala> oneToFirst.isDefinedAt(2)  
res4: Boolean = false
```

```

object PartialFunctions extends App {

  val isPrimeEligible: PartialFunction[Item, Boolean] = {
    case item => item.isPrimeEligible
  }

  val amountMoreThan500: PartialFunction[Item, Boolean] = {
    case item => item.price > 500.0
  }

  val freeDeliverable = isPrimeEligible orElse amountMoreThan500

  def deliveryCharge(item: Item): Double = if(freeDeliverable(item)) 0 else 50

  println(deliveryCharge(Item("1", "ABC Keyboard", 490.0, false)))

}

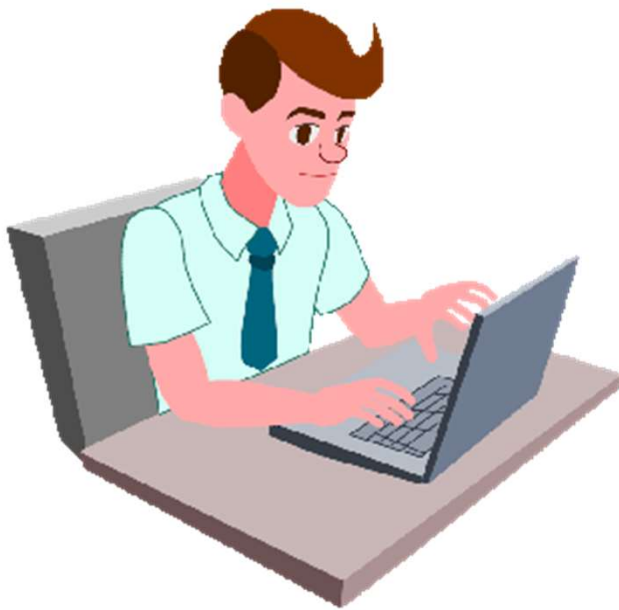
case class Item(id: String, name: String, price: Double, isPrimeEligible: Boolean)

```

SUMMARY



- It is important to know that we're allowed to nest functions and make our code look cleaner.
- We learned about how we can make function calls in a variety of ways, for example with a variable number of arguments, with a default parameter value, and with a named argument.
- Then we learned how to write function literals in Scala.



"COMPLETE LAB"