Scala

What is Scala

- Scalable Language
- Scala is a multi-paradigm programming language to express common programming patterns in a concise, elegant and type-safe way
- Written by Martin Odersky
- Statically types
- Runs on JVM,
- Fully Object Oriened
- Funcional
- No Primitives
- Support ststic class members through Singleton Object Concept
- Improve Support for OOP through Traits

Keywords

abstract	case	catch	class
def	do	else	extends
false	final	finally	for
forSome	if	implicit	import
lazy	match	new	Null
object	override	package	private
protected	return	sealed	super
this	throw	trait	Try
true	type	val	Var
while	with	yield	
-	:	=	=>
<-	<:	<%	>:
#	@		

Install Scala

1. Download java and set path
2. Verify Your Java Installation
   java –version
3. Set Your Java Environment

Set JAVA_HOME to C:\ProgramFiles\java\jdk1.7.0_60 or
Export JAVA_HOME=/usr/local/java-current

4. Export PATH=$PATH:$JAVA_HOME/bin/
5. $java –jar scala-2.9.0.1-installer.jar

Scala User

Scala - Operators

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators

Scala Framework

- Akka
- Spark
- Kafka
- Scalding
- Play

Scala Uses

- Scripting
- Web Application
- Messaging

- Mobile Android Apps
- GUI (Graphical User Interface)
- Digital Subscriber Line

REPL

- REPL: Read -Evaluate -Print -Loop

- Easiest way to get started with Scala, acts as an interactive shell interpreter

- Even though it appears as interpreter, all typed code is converted to Bytecode and executed

The line includes:

- An automatically generated or user-defined name to refer to the computed value (res0, which means result 0),

Check your Understanding

Scala REPL acts as scala Interpreter

a)True

b)False

Check your Understanding

Scala supports primitive and wrapper classes ?

a)True
b)False

Data Types in Scala

- A Data type tells the compiler about the type of the value to be stored in a location

- Scala comes with the following built-in data types which you can use for your Scala variables

Type	Description	Example
Byte	8-bit signed integer	3
Short	16-bit signed integer	32
Int	32-bit signed integer	327
Long	64-bit signed integer	32754L
Double	8-byte floating point	3.1415
Float	4-byte floating point	3.1415F
Char	Single character	'c' (single quotes)
String	Sequence of characters	"iFruit" (double quotes)
Boolean	Either true or false	true (case sensitive)

Special Unit **Type**

Unit

– When a function passes back “nothing” in Scala, it passes back Unit

– This is equivalent to the void return type in a Java method

– There is only one Unit in Scala; it is non-instantiable

E,g

```
val myreturn = println("Hello, world")
> myreturn: Unit = ()
```

Special Any **Type and Explicit Casting of Type**

- **Any**

– Used when Scala cannot determine which specific type to use

– Can be cast to a specific type using the method asInstanceOf[*type*]

e.g.

```
val myreturn = if (true) "hi"
>myreturn: Any = hi

val mystring = myreturn.asInstanceOf[String]
> mystring: String = hi
```

Using Booleans **to Control Program Flow**

Boolean variables are used to control program flow

– Such as branching, conditional execution, and looping

▪ **Boolean variables can be set to** true **or** false

– Lower case true and false only

✔ val gpsStatus: Boolean = false

val gpsStatus = true

✘ val gpsStatus = "true"

Properties of Scala Variables: Mutability

Variables must be initialized when declared

- **Variables are either *mutable* or *immutable***

– Mutable: can reassign a value of the same type
– Syntax: var *name*: *type* = *value*

– Immutable: value cannot be reassigned after initialization
– Syntax: val *name*: *type* = *value*

Properties of Scala Variables:

Types may either be explicitly declared or inferred

- Scala makes a best guess based on assignment
- You can also explicitly declare the type

Example: Type inference

```
var phoneModel = 3
phoneModel: Int = 3
```

Example: Explicit typing

```
var phoneModel: Short = 3
> phoneModel: Short = 3
```

Properties of Scala Variables:

Variables are statically typed

– Scala does not support dynamic typing

– The type is established on first use and never reassigned

– Using the same variable name with a different type will cause an error

```
var phoneModel = 3
> phoneModel: Int = 3
phoneModel = "iFruit 9000"
> error: type mismatch;
found : String("iFruit 9000")
required: Int
Reference:
```

Redefining Variables

Although you cannot reassign an immutable variable, or assign a new type to any defined variable, you can redefine a variable

```
var phoneModel = 3
phoneModel = "iFruit 9000"
> error: type mismatch;
found : String("iFruit 9000")
required: Int
var phoneModel = "iFruit 9000"
> phoneModel: String = iFruit 9000
```

Substituting Variables in Output

Let's define some variables to use:

```
val phoneName = "Titanic"
val phoneTemp = 35
```

- **Precede a string with *s* to substitute the value of the named variable**

```
println(s"Name: $phoneName")
> Name: Titanic
println(s"Name: $phoneName", s" Temp: $phoneTemp")
> (Name: Titanic, Temp: 35)
```

Formatting Output Using Format Strings

Given:

val phoneTemp = 46

- **Use** f **to format the string using C language-style format strings**

```
println(f"Temp: $phoneTemp%f")

println(f"Temp: $phoneTemp%.2f")

println(f"Temp: $phoneTemp%h as hex")
```

Format Strings

%c - character
%s - string
%d - decimal
%e - exponential
%f - floating point
%i - integer
%o - octal
%h - hexadecimal

Assigning Block Expression

- In Java or C++ a code block is a list of statements in curly braces { }

- In Scala, a { } block is a list of expressions, and result is also an expression

- The Value of a block is the value of the last expression of it

Note: You can assign an anonymous function result to a variable/value in Scala

Lazy Values

- One nice feature built into Scalaare "lazy val" values.
- Lazy value initialization is deferred till it’s accessed for first time
- Lazy values are very useful for delaying costly initialization instructions
- Lazy values don’t give error on initialization, whereas no lazy value do give error

Check your Understanding

If val a = (1, 2, 4,11, “Robert”,5,9,11,2.5) then a.-5?

a)No value , its wrong syntax
b)5
c)Nil
d)“Robert”

Control Structures in Scala

- Control Structures controls the flow of execution

- Scalaprovides various tools to control the flow of program's execution
- Some of them are:

- if..else
- while
- do-while
- foreach
- for

if Statement

```
if(Boolean_expression)
 { // Statements will execute if the Boolean
expression is true }
```

```
if(Boolean_expression)
{ //Executes when the Boolean expression is true }
 else
{ //Executes when the Boolean expression is false }
```

```
if(Boolean_expression 1)
{ //Executes when the Boolean expression 1 is true }
else if
(Boolean_expression 2)
{ //Executes when the Boolean expression 2 is true }
else if(Boolean_expression 3)
{ //Executes when the Boolean expression 3 is true }
else { //Executes when the none of the above condition is true. }
```

Loop Statements

- **As such Scala does not support break or continue statement**
- **The ++i, or i++ operators don't work in Scala, use i+=1 or i=i+1 expressions instead**

```
object Demo { def main(args:
Array[String]) { var a = 10; // An
infinite loop. while( true ){
println( "Value of a: " + a ); } } }
```

While loop

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

- In Scala while and do-while loops are same as Java

```
While(condition)
{
// Block of code ;
}
```

do-While Loop

```
do
{
//Block of code
} while(condition);
```

foreach Loop

```
var args  = “Welcome”

Args.foreach(println)
```

for loop

for Loop:

for loop can execute a block of code for specific number of times.

Scala doesn't have for (initialize; test; update) syntax

for(varx <-n) { here, n --> Range

//Block of statements; <-operator is called a generator

}

Scala: For Loop : to vs. until

You can use either the keyword to or until when creating a Range object. The difference is, that to includes the last value in the range, whereas until leaves it out. Here are two examples:

can have multiple generators in for loop

Check your Understanding

What is the output of the following program?

for (x <-'a' until 'f‘) print(x)

a)Error
b)abcde
c)abcdef
d)None of these

Collection

In Scala there are a large number of collection classes available

– Classes are optimized for use in particular circumstances

– They may be optimized for head/tail access or for fast update

- **Collection classes vary in the methods they support**

– Immutable Collection classes are defined in package scala.collection.immutable

– Mutable Collection classes are defined in package scala.collection.mutable

Traversable

Declare an object of type Traversable **to use the very important** foreach
method which facilitates parallel and distributed processing
– Performs a specified action on all members of the collection

```
val modelTrav = Traversable("MeToo", "Ronin", "iFruit")
modelTrav.foreach(println)
```

Set: **Storing Data with Automatic De-duplication**

Set **removes duplicates**

- **Does not change ordering**
- Set(*value*) **returns** true **or** false

```
val mySet = Set("MeToo", "Ronin", "iFruit")

mySet("Banana")
val myset2 = mySet.drop(1)
```

Map: **Storing Key-Value Pairs**

```
val wifiStatus = Map(
"disabled" -> "Wifi off",
"enabled" -> "Wifi on but disconnected",
"connected" -> "Wifi on and connected")

wifiStatus("enabled")
```

List

A List **is a finite immutable sequence**
– Very commonly used in Scala programming
– Accessing the first element and adding an element to the front of the list are
constant-time operations
▪ **A** List **literal can be constructed using** :: **(cons operator) and** Nil

```
val newList = "a" :: "b" :: "c" :: Nil

Newlist(1)

val randomlist = List("iFruit", 3, "Ronin", 5.2)
```

```
val devices = List(("Sorrento", 10), ("Sorrento", 20),
("iFruit", 30))

val myList: List[Int] = List(1, 5, 7, 1, 3, 2)

myList.sum
```

- **Use** :+ **to append to a list**

```
val myListE = myList :+ 10
```

Comparing Literals Using match ... case

```
val phoneWireless = "enabled"
var msg = "Radio state Unknown"
phoneWireless match {
case "enabled" => msg = "Radio is On"
case "disabled" => msg = "Radio is Off"
case "connected" => msg = "Radio On, Protocol Up“
case default => msg = "Radio state unknown"
}
println(msg)
```

Classes in Scala

▷Classes in Scala are static templates that can be instantiated into many objects at runtime

▷A Class can contain information about:

- Fields
- Constructors
- Methods
- Superclasses(inheritance)
- Interfaces implemented by the class, etc.

Simple class definition in Scala: class Myclass{}

Code is organized by *classes*

– Object-oriented code organization
– Contains data elements and methods
– Establishes local scope within the class
– Enables instantiation of objects

- **Classes are grouped together into *packages***

– Separate namespaces to avoid collision of classes

- **Packages are defined in Scala source files**

– Multiple packages can be contained in a single file

- **Use the** import **keyword to import libraries to use in your code**

Check your Understanding

What is the output of the following code ?

```
class add{
var x:Int=10
var y:Int=20
def add(a:Int,b:Int)
{
a=x+1
println(“Value of a after modification :”+a);
}
}
Var p=new add()
p.add(5,10);
```

a.5
b.11
c.16
d.Error

Properties with Getters and Setters

- Getters and Setters are better to expose class properties
- In Java, we typically keep the instance variables as private and expose the public getters and setters

- Scala provides the getters and setters for every field by default

Constructor

- Constructors in Scala are a bit different than in Java
- Scala has 2 types of constructors

- Primary Constructors
- Auxiliary Constructors

- The auxiliary constructors in Scala are called this. This is different from other languages, where constructors have the same name as the class
- Each auxiliary constructor must start with a call to either a previously defined auxiliary constructor or the primary constructor

Check your Understanding

Syntax of primary constructor is:

a)Greet(message : String)
{
// …code
}

b. class Greet
{
//…code
}

c.class Greet(message : String)
{
// … code
}

d.public Greet(message : String)
{
//…code
}

Singletons

- Scala doesn't have the concept of static methods or fields
- Instead a Scala class can have what is called a singleton object, or sometime a companion object
- A singleton object definition looks like a class definition, except instead of the keyword class you use the keyword object
- An object defines a single instance of a class

Apply Methos

Packages

- A Packageis a special object which defines a set of member classes, objects and packages
- In Scala, packages serve the same purpose as in Java: to manage the names in a large program
- To add the items to a package, they can be included in package statements

```
package Skillspeed {
package Courses{
package Scala{
class HelloScala
}} }
```

Imports and Implicit Imports

- Packages/ classes can be imported in Scala
- Import serve the same purpose as in Java:

To use short names instead of long ones

- All the members of a package can be imported as:
  - **import java.awt._**
  - Note that "_" is used instead of "*"
- In Scala, imports can be anywhere, instead of being at the top of the file, unlike Java
- We can use selectors to import only few members of a package like:
  - **import java.awt.{Color, Font}**
- Every Scala program implicitly starts with:
  - **import java.lang._**
  - **import scala._**
  - **import Predef._**