Working with Scala - 2

Classes in Scala

→ Class in Scala is very much Similar to Java or C++

```
class Ctr {
        private var value = 0 // fields must be initialized
        def incr() { value +=1 }
        def curr() = value
}
```

```
scala> class Ctr {
     | private var value = 0
     | def incr() { value +=1 }
     | def curr() = value
     | }
defined class Ctr
```

→ In Scala a class is NOT declared as public

→ A source file can contain multiple classes

→ All of the classes could be public

Classes in Scala

→ Previous class could be used in usual way

```
val ctr1 = new Ctr // Or new Ctr() is also valid
ctr1.incr()
print(ctr1.curr)
```

```
scala> val ctr1 = new Ctr
ctr1: Ctr = Ctr@6689ef79

scala> ctr1.incr()

scala> print(ctr1.curr)
1
```

→ Parameter less method could be called with or without parentheses

→ Using any form is programmer's choice

→ However, as convention

- » Use () for mutator method
- » Use no parentheses for accessor method

Properties with Getters and Setters

→ Getters and Setters are better to expose class properties

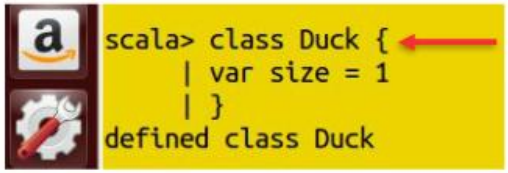
→ In Java, we typically keep the instance variables as private and expose the public getters and setters

```
public class Duck{
        private int size;
        public int getSize() { return size; }
        public void setSize(int size)
        { (if size > 0) this.size = size}
        }
```

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

→ We define a public field

```
class Duck {
        var size = 1
        }
```



Properties with Getters and Setters

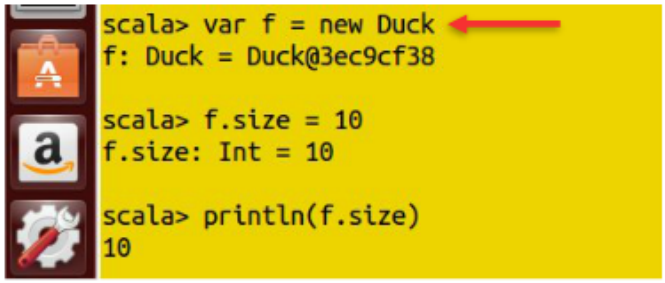
→ Scala generate a class for the JVM with a private size variable and public getter and setter methods

→ If the field is declared as private, the getters and setters would be private

→ The getters and setter methods in previous case would be:

- » size and size_=
- » Example:

```
var f = new Duck
f.size = 10 // It calls f.size_=(10)
println(f.size) // It calls f.size()
```



Properties with Getters and Setters

→ Getters and Setters can be redefined as explained below:

```
class Duck {
        private var privateAge = 0
        def age = privateAge   //getter
        def age_=(newAge: Int) {if (newAge > privateAge) privateAge = newAge; }  //setter
```

```
scala> class Duck {
     | private var privateAge = 0
     | def age = privateAge
     | def age_=(newAge: Int) {if (newAge > privateAge) privateAge = newAge; }
     | }
defined class Duck
```

Properties with Only Getters

→ Sometimes we need read-only properties

→ There are two possibilities:

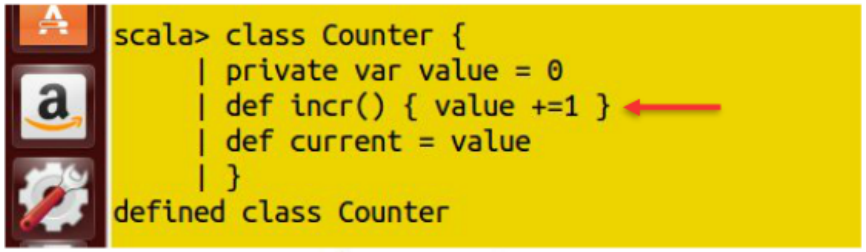
- » The property value never changes
- » The value is changed indirectly

→ For the first case, we declare the property as val. Scala treats it as final variable and thus generates only getter, no setter

→ In second case, you need to declare the field as private and provide the getter, as explained below:

→ Semicolons are optional in Scala

```
class Counter
{
        private var value = 0
        def incr() { value +=1 }
        def current = value
}
```



Object - Private Field

→ In Scala (and other languages as well), a method can access the private fields of its class

→ Example:

```
class Counter {
        private var value = 0
        def incr() { value +=1 }
        def current = value
        def isLess (otherVal: Counter) = value < otherVal.value
}
```

```
scala> class Counter {
     | private var value = 0
     | def incr() { value +=1 }
     | def current = value
     | def isLess (otherVal: Counter) = value < otherVal.value
     | }
defined class Counter
```

→ We can declare the variables as object-private by private[this] qualifier

→ Now the methods can only access the value field of current object

→ For the class-private field, private getter and setter are generated

→ For object-private field, NO getter and setter methods are generated

Summarizing Properties

→ In Scala, the getters and setters are generated for each property

→ For private properties, the getter and setter are private

→ For a val, only getters are generated

→ In Scala you can't have a read-only property (i.e. only getter, no setter)

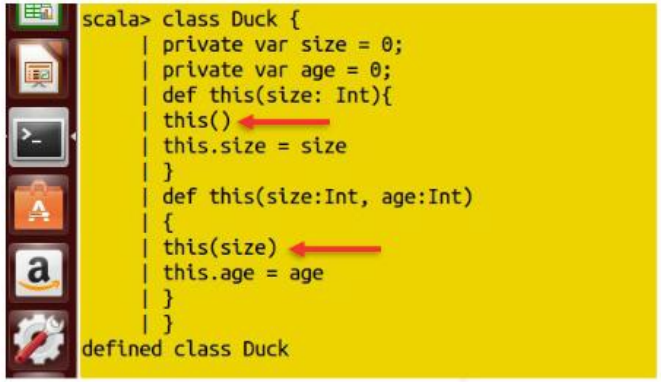
→ No getters and setters are generated for object-private fields

Auxiliary Constructor

→ We can have as many constructors as we need
→ There are two types of constructors in Scala:
  » Auxiliary Constructor
  » Primary Constructor

→ 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

```
class Duck {
        private var size = 0;
        private var age = 0;
        def this(size: Int){
                this()  // Calls the primary constructor
                this.size = size
                }
                def this(size:Int, age:Int)
                {
                        this(size)  // calls previous auxiliary constructor
                        this.age = age
                }
}
```



Primary Constructor

→ Every class in Scala has a primary constructor
→ Primary constructor isn't defined by this method
→ The parameters for primary constructor are placed immediately after the class name:

```
class Duck(val size:Int, val age:Int)
{
 // ... Code for intitialization
}
```

→ The primary constructor executes all the statements in the class definition, as explained below:

```
class Duck(val size:Int, val age:Int) {
        println("Inside duck constructor")
        def desc = "Duck of age " + age + " is of size " + size
}
```

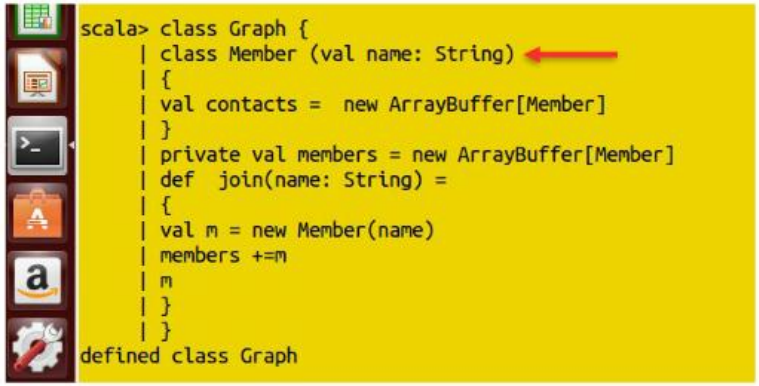
```
scala> class Duck(val size:Int, val age:Int) {
     | def desc = "Duck of age " + age + " is of size " + size
     | }
defined class Duck
```

→ The println statement is executed for every object creation

Nested Classes

→ Classes can be nested inside other classes
→ Example:

```
class Graph {
        class Member (val name: String)
        {
        val contacts =  new ArrayBuffer[Member]
        }
                private val members = new ArrayBuffer[Member]
                def  join(name: String) =
                {
                val m = new Member(name)
                        members +=m
                        m
                }
}
```



Nested Classes (Contd).

```
val chatter = new Graph
val mFace = new Graph
val fred = chatter.join("Fred")
val Wilma = chatter.join("Wilma")
fred.contacts += wilma
Val Barney = mFace.join("Barney")
Fred.contacts += barney //Error!
```

```
scala> val chatter = new Graph
chatter: Graph = Graph@75a52dc5

scala> val mFace = new Graph
mFace: Graph = Graph@7d554f59

scala> val fred = chatter.join("Fred")
fred: chatter.Member = Graph$Member@5c3e28e1

scala> val Wilma = chatter.join("Wilma")
Wilma: chatter.Member = Graph$Member@bed5379

scala> fred.contacts += Wilma
res38: fred.contacts.type = ArrayBuffer(Graph$Member@bed5379)

scala> Val Barney = mFace.join("Barney")
```

```
scala> fred.contacts += Barney
<console>:14: error: type mismatch;
 found   : mFace.Member
 required: chatter.Member
              fred.contacts += Barney
                               ^
```

Singletons

→ Scala doesn't have the concept of static methods or fields
→ It is supported by object construct
→ An object defines a single instance of a class
→ Example:

```
object Reservations {
        private var lastNum = 0;
        def newReservationNum () = { lastNum +=1; lastNum}
        }
```

```
scala> object Reservations {
     | private var lastNum = 0;
     | def newReservationNum () = { lastNum +=1; lastNum}
     | }
defined module Reservations
```

→ If we need new reservation number, we can call Reservations.newReservationNum()
→ Constructor of Singleton Object is executed when the object is first used
→ An object has all the features of a class
→ There is only one exception:- Parameters can't be provided to the constructor

Singleton – Use Cases

→ Singletons can be used in Scala as:

- » When a singleton instance is required for co-ordinating a service
- » When a single immutable instance could be shared for efficiency purposes
- » When an immutable instance is required for utility functions or constants

Companion Objects

→ In many programming languages, we typically have both instance methods and static methods in same class

→ In Scala, it is achieved by Companion Object of same name as of class

→ Example:

```
Class Account {
        val id = Account.newNum()
        private var bal = 0.0
                ....
        }
        object Account {
                private var lastNum = 0;
                private def newNum() = { lastNum +=1; lastNum }
        }
```

→ The class and it's companion objects need to be in same source file

→ The class and it's companion object can access each other's private features

→ The companion object of the class is accessible, but NOT in scope

Apply Method

→ Scala objects typically have an apply method

→ The general form of Apply method is: object(arg1)

→ This is same as object.apply(arg1)

→ Example:

```
class Account private (val id: Int, bal: Float) {
                    private var balance = bal
                    ....
          }

          object Account {
                    def apply(bal: Float) = new Account( newNum(), bal)
                    ...
          }
```

→ Now the new Account object can be created as follows:

Val acct = new Account(100.0) // where as the default constructor takes two parameters!

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, e.g.

```
package com {
                    package spark{
                      package edureka {
                        class Emp
                          ....
                      }
                    }
                  }
```

→ The class Emp can be accessed from anywhere as com.spark.edureka.Emp

→ Unlike, classes, a package can be defined in multiple files

→ Conversely, a single file can have more than one package

Scala : Scope Rules

→ Scope rules for packages in Scala are more consistent than Java

→ Scala packages nest just like all other scopes

→ Member names could be accessed fro enclosing scope, i.e.

```
package com {
        package test1 {
          object Utils {
                def sayHi = "Hi"
          }
         package test2 {
           class Test {
               def sayHello() {
                    Utils.sayHi
               }
           }
         } ....
```

→ Note that we just used Utils.sayHi, as the object Utils is defined in the parent package

Top Of File Notation

→ Instead of nested notation, we could have the package notation at the top of file also

→ Example:

```
package com {
                    package spark{
                      package edureka {
                        class Emp
                          ....
                    }
                  }
                }
```

→ is equivalent to:

```
package com.spark
      package edureka
       class Emp {
         ....
```

→ Note: In the above example, everything belongs to package com.spark.edureka, but the package com.spark is also opened up, so it's contents could also be referred

Package Visibility

→ In Java, we typically control the access of the class members by public, private or protected

→ In Scala, the same effect could be achieved through qualifiers

→ Example:

```
package com.spark.edureka
                class Employee {
                        private[edureka] def sayHi = "Hi !"
                }
```

→ The visibility could also be extended to enclosing package:

```
package com.spark.edureka
                class Employee {
                        private[spark] def sayHi = "Hi !"
                }
```

Imports and Implicit

→ 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._

Inheritence

→ Just like Java, classes can be extended using extends keyword

```
class Dog extends Animal {
                    var size: 1.0
                    ...
        }
```

```
scala> class Animal {
     | println("Animals")
     | }
defined class Animal

scala> class Dog extends Animal {
     | println("Animalsssssssss")
     | }
defined class Dog
```

→ Just like Java, new methods and fields can be introduced or superclass methods or fields could be overridden in subclasses

→ A class can be declared as final to avoid it being extended

→ Unlike Java, individual field or method could also be marked as final to avoid them being overridden

```
scala> final class Animal {
     | println("Animals")
     | }
defined class Animal

scala> class Dog extends Animal {
     | println("Animalsssssssss")
     | }
<console>:59: error: illegal inheritance from final class Animal
       class Dog extends Animal {
                         ^
```

Overriding Methods

→ override modifier must be used to override an abstract method:

```
public class Dog {
                    ......
                    override def bark = {"Woof!"}
                    ...
          }
```

→ The override modifier is useful in following scenarios:

- » When name of the method being overridden is misspelled
- » When a wrong parameter type is provided
- » When a new method is introduced in superclass which clashes with a subclass method

→ Invoking superclass method is same as in Java, by super keyword:

```
public class Employee extends Person{
                    ...
                    override def toString = super.toString + "Hi..!"
          }
```

Type Checking and Casting

→ isInstanceOf method is used to decide whether an object belongs to a class:

```
if (a.isInstanceOf(Employee)) {
                    val b = a.asInstanceOf(Employee)  // b has the type Employee
                    ...
          }
```

→ asInstanceOf method is used to convert a reference to a subclass reference

→ classOf method is used to determine the class of a given reference:

```
if(a.getClass == classOf[Employee]){
                    ...
          }
```

Superclass Construction

→ All the classes have a primary constructor and many auxiliary constructors, and all the auxiliary constructor should either call a primary constructor or previous auxiliary constructor

→ It means an auxiliary constructor can never invoke a superclass constructor directly

```
class Emp(name: String, dept: Int, salary: Double) extends
                Person(name, dept){
                                ...
                }
```

→ Putting the class and constructor together makes a very short code in Scala

→ The Emp class has three parameters, out of which, two are passed to its Superclass Person

Abstract Classes

→ Just like Java, you can use the abstract keyword for a class, which can't be instantiated:

```
abstract class Emp (name: String) {
                    def id: Int  //Only method declaration, no body
          }
```

→ Here we declared a method to generate id, but didn't provide implementation

→ Each concrete subclass of Emp should provide the implementation of id

→ In a subclass, we don't need to specify override while defining an abstract superclass method:

```
class EdurekaEmp(name:String) extends Person(name) {
                    def id = name.hashCode // override keyword not equired
          }
```

Question

As a convention, we should use parameterless methods for setters of classes

- True
- False

Answer

- False

Question

No getters and setters are generated for private members of the class

- True
- False

Answer

- False

Question

In Scala, you can override default getters and setters for class members

- True
- False

Answer

- True

Question

In Scala, the companion object members are always in scope

- True
- False

Answer

- False

Question

The package visibility of a class member of a package can be made as public, private or protected

- True
- False

Answer

- False

Thank You