Scala

SCALA

Agenda

In this module, we are going to look at the following topics:

- ✓ Classes
- ✓ Access Modifiers
- ✓ Objects
- ✓ Traits
- ✓ Case Classes

Classes

- In Scala, a class is not declared as public.
- A Scala source file can contain multiple classes, and all of them have public visibility.

```
class Counter {
    private var value = 0           // You must initialize the field
    def increment() { value += 1 } // Methods are public by default
    def current() = value
}
val myCounter = new Counter
myCounter.increment()
println(myCounter.current)
```

Classes

- Methods are public by default

- You can call a parameter-less method with or without parentheses, but the general practice is to call *mutators* with () and *accessors* without.

```
myCounter.increment()       // Use () with mutator
println(myCounter.current) // Don't use () with accessor
```

Getters & Setters

- Scala provides getter and setter methods for every field.

- For example if we create `class Person { var age = 0 }` Scala generates a class for the JVM with getter & setter methods `(age & age_=)` that we can override if required

```
println(fred.age)      // Calls the method fred.age()
fred.age = 21          // Calls fred.age_=(21)
```

- If the field is *private*, the getter and setter are *private*. Otherwise *public*

- If the field is a `val`, only a getter is generated and a private final field is generated.

Access Modifiers

- Members of packages, classes or objects can be labeled with the access modifiers **`private`** and **`protected`**, and if we are not using either of these two keywords, then access will be assumed as **`public`**.

- These modifiers restrict accesses to the members to certain regions of code. To use an access modifier, you include its keyword in the definition of members of package, class or object

Access Modifiers

- **Private**
  - A private member is visible only inside the class or object that contains the member definition.
- **Protected**
  - A protected member is only accessible from subclasses of the class in which the member is defined.
- **Public**
  - There is no explicit modifier for public members. Such members can be accessed from anywhere.
- **Object-Private fields**
  - If you don’t want any getter or setter, declare the field as private[this]. The value may be mutated from within the class but not using class objects.

Scope of Protection

- Access modifiers in Scala can be augmented with qualifiers.
- A modifier of the form private[X] or protected[X] means that access is private or protected "up to" X, where X designates some enclosing package, class or singleton object.

```
package society {
   package professional {
      class Executive {
         private[professional] var workDetails = null
         private[society] var friends = null
         private[this] var secrets = null

         def help(another : Executive) {
            println(another.workDetails)
            println(another.secrets) //ERROR
         }
      }
   }
}
```

Constructors

- In Scala, every class should have one *primary constructor* and optionally additional *auxiliary constructors*.

- The auxiliary constructors are called `this`. Each auxiliary constructor *must start with a call to a previously defined auxiliary constructor* or the *primary constructor*.

- In Scala, every class has a primary constructor. The primary constructor is not defined with a `this` method. It is interwoven with the class definition.

- The parameters of the primary constructor are placed *immediately after the class name.* The primary constructor executes *all statements in the class definition.*

Objects

- An object defines a single instance of a class with the features that you want.
- Scala has no static methods or fields. Use objects for singletons and utility methods.
- The constructor of an object is executed when the object is first used. If an object is never used, its constructor is not executed.
- An object can have essentially all the features of a class except providing constructor parameters.

```
object Accounts {
    private var lastNumber = 0
    def newUniqueNumber() = { lastNumber += 1; lastNumber }
}
```

Companion Objects

- A class can have a companion object with the same name.
- The class and its companion object can access each other’s private features.
- They must be located in the *same source file.*

Object Inheritance

- An object can extend a class and/or one or more traits. The result is an object of a class that extends the given class and/or traits, and in addition has all of the features specified in the object definition.

```
abstract class Undoable(val desc: String) { def undo(): Unit }
object DoNothing extends Undoable("Do nothing“){ override def undo(){} }
val actions = Map("open"->DoNothingAction, "save"->DoNothingAction, ...)
```

Apply & App

- **`apply()` method:**
  - The apply method is called for expressions of the form *Object(arg1, arg2 ..) and* returns an object of the companion class.
- **`App`** trait :
  - Each Scala program must start with an object's `main` method of type `Array[String] => Unit`: Instead, you can extend the `App` trait and place the program code into the constructor body

Object Enumeration

• Define an object that extends Enumeration helper class to define enumerations. Initialize each vale in your enumeration with a call to Value method.

• Enumeration values have ID & Name that may be explicitly supplied or implicitly inferred. The ID of an enumeration value is returned by the id method, and its name by the toString method.

• You can look up an enumeration value by its ID or name.

```
object TrafficLightColor extends Enumeration {
     val Red = Value(0, "Stop")
     val Yellow = Value(10) // Name "Yellow"
     val Green = Value("Go") // ID 11
}
```

Inheritance

- You extend a class in Scala with the **extends** keyword.
- A class declared as **final** cannot be extended.
- Use **override** modifier when you override a method that isn’t abstract.
- You can override a **val** (or a parameterless **def**) with another **val** field of the same name. a **var** can override another abstract **var**.

Inheritance

- To invoke a superclass method use `super` keyword.
- To test whether an object belongs to a given class, use the `isInstanceOf` method. Use the `asInstanceOf` method to convert a reference to a subclass reference.
- The `classOf` method is defined in the `scala.Predef` object that is always imported.

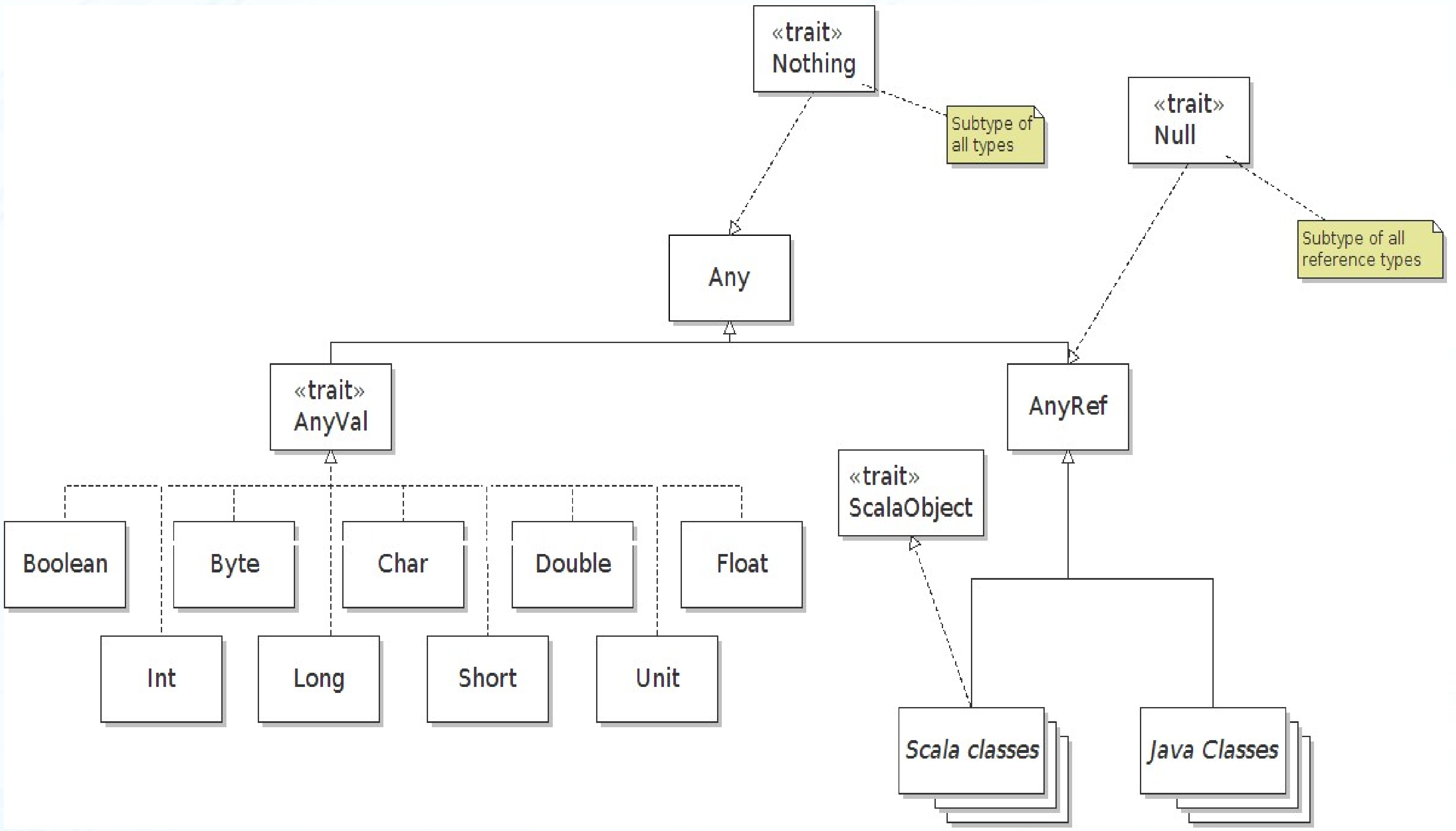
Abstract Class

- **Abstraction**: Use the abstract keyword to create an **abstract class** that cannot be instantiated, usually because one or more of its methods are not defined (called **abstract methods**). An **abstract field** is simply a field without an initial value.

```
abstract class Person (val name: String) {
    def id: Int // No method body — an abstract method
    val id: Int // abstract field with an abstract getter method
    var name: String // with abstract getter and setter methods
}
```

- As with methods, no **`override`** keyword is required in the subclass when you define a field that was **`abstract`** in the superclass.

Scala Inheritance Hierarchy



Packages

- Add items to a package as shown below:

```
package com {
    package tekcrux {
       package scala { some classes .. }
        package spark { some other classes .. }
    }
}
```

- In Scala, package names are relative, just like inner class names.

- A package clause can contain a “chain” or path segment. Such a clause limits the visible members.

```
package com.tekcrux.scala {
    // Members of com and com. tekcrux are not visible here
    package utilities { class Person ... }
}
```

Package Objects

- Package objects allow functions and variables to be defined at the package level. Behind the scenes, the package object gets compiled into a JVM class with static methods and fields, called package.class, inside the package.

```
package com.tekcrux.scala
package object people {
   val defaultName = "Kanakaraju Y"
}
package people {
   class Person {
      var name = defaultName   // A constant from the package
      def description = "A person with name " + name
   }
}
```

Imports

- In Scala, an import statement can be anywhere, not just at the top of a file. The scope of the import statement extends until the end of the enclosing block.

- If you want to import a few members from a package, use a *selector like this:* `import java.awt.{Color, Font}`

- Rename ➔ `import java.util.{HashMap => JavaHashMap}`

- Hide ➔ `import java.util.{HashMap => _}`

- Every Scala program implicitly starts with

```
import java.lang._
import scala._
import Predef._
```

Traits

- A trait can work similar to a Java interface. You need not declare the method as abstract. Any unimplemented method in a trait is automatically abstract.
- A subclass can provide implementation. Use **`extends`** keyword to implement. No need of override keyword when overriding an abstract method of a trait. For extending multiple traits use **`with`**

```
trait Logger { def log(msg: String) // An abstract method }
// Use ‘extends’, not implements. Use ‘with’ to add multiple traits
class ConsoleLogger extends Logger with Cloneable with Serializable {
  def log(msg: String) { println(msg) } // No override needed
}
```

Traits

- All java interfaces can be used as Scala traits.
- Method of a trait need not be always abstract. We can have concrete methods as well.

Objects with Traits & Layered Traits

- Mix-in You can add a trait to an object when you construct it. This feature allows us to *mix-in* different implementations of a trait during object instantiation.

```
class SavingsAccount extends Account with Logger
val acct1 = new SavingsAccount with ConsoleLogger
val acct2 = new SavingsAccount with FileLogger
```

- To a class or an object you can add multiple traits (layered traits) that invoke each other *starting with the last one* allowing you apply staged transformations.

Objects with Traits & Layered Traits

- To override an abstract method of a trait in a class that extends it, use **`abstract override`** keywords if you are calling the super class method.

- Traits can be used to create rich interfaces based on a few abstract methods. For ex: Scala **`Iterator`** provides many methods using **`next`** & **`hasNext`** abstracts.

- A field is a trait can be concrete or abstract. An initialized field is concrete. An abstract field must be overridden in a concrete subclass.

Traits

- Traits can have constructors made up of field initializations and other statements in the trait’s body.

- Traits can not have constructor parameters. Every trait has a single parameter-less constructor.

```
trait FileLogger extends Logger {
  val out = new PrintWriter("app.log") // Part of trait’s constructor
  out.println(s"# ${java.time.Instant.now()}") // Also in constructor
  def log(msg: String) { out.println(msg); out.flush() }
}
```

- A trait can extend a class. That class becomes a superclass of any class mixing on that trait.

Case Classes

- Case Classes are special kind of classes optimized for use in pattern matching.
- A minimal ‘case class’ requires the keywords case class, an identifier, and a parameter list (which may be empty)

```
case class Book(isbn: String)
        val somebook = Book("978-0486282114")
```

- Case classes have an `apply` method by default, so you don’t need `new`. It also has an `unapply` method that make pattern matching work. `unapply` takes an object and give back arguments for pattern matching.

Case Classes

- All the construction parameters becomes a `val` unless declared as `var`.
- Methods `toString`, `equals`, `hashCode` & `copy` are generated unless they are explicitly provided.
- Otherwise Case classes are just like other classes. You can add methods, fields, extend them etc.

Case Classes Infix Notation

- When an `unapply` method yields a pair, you can use infix notation in the case clause, particularly with a case class that has two parameters.

```
val amt = Currency (1000.0, "EUR")
amt match { case a Currency u => a + " " + u }    // instead of Currency(a, u)
```

- Infix notation is meant for matching sequences. For ex., every List object is either `Nil` or another object of the case class '**::**' .

```
val lst = List(1, 7, 2, 9)
lst match {
  case h :: t => println( h + ", " + t.length )    // 1, 3
  case _ => 0
}

List(1, 7, 2, 9) match {
  case f :: s :: rest => f + s + rest.length       // 1 7 2
  case _ => 0
}
```

Sealed Classes

• Sealed Classes allow a way to exhaust all the matches available for Case Class matching. While declaring Case Classes have them all extend a common super class which is declared as sealed class.

```
sealed abstract class Amount
case class Dollar (value: Double) extends Amount
case class Currency (value: Double, unit: String) extends Amount
```

• All the subclasses of a sealed class must be defined in the same file as the class itself. When a class is sealed, all of its subclasses are known at compile time, enabling the compiler to check pattern clauses for completeness.

• It’s a good idea for all Case Classes to extend a seal class or trait.

THANK YOU