Scala Hierarchy

The `Scala.Any` Class

- On top of the Scala classes hierarchy is the class `Any`. It defines the following methods:

```
final def ==(that: Any): Boolean
final def !=(that: Any): Boolean
def equals(that: Any): Boolean
def hashCode: Int
def toString: String
```

The `Scala.AnyRef` Class

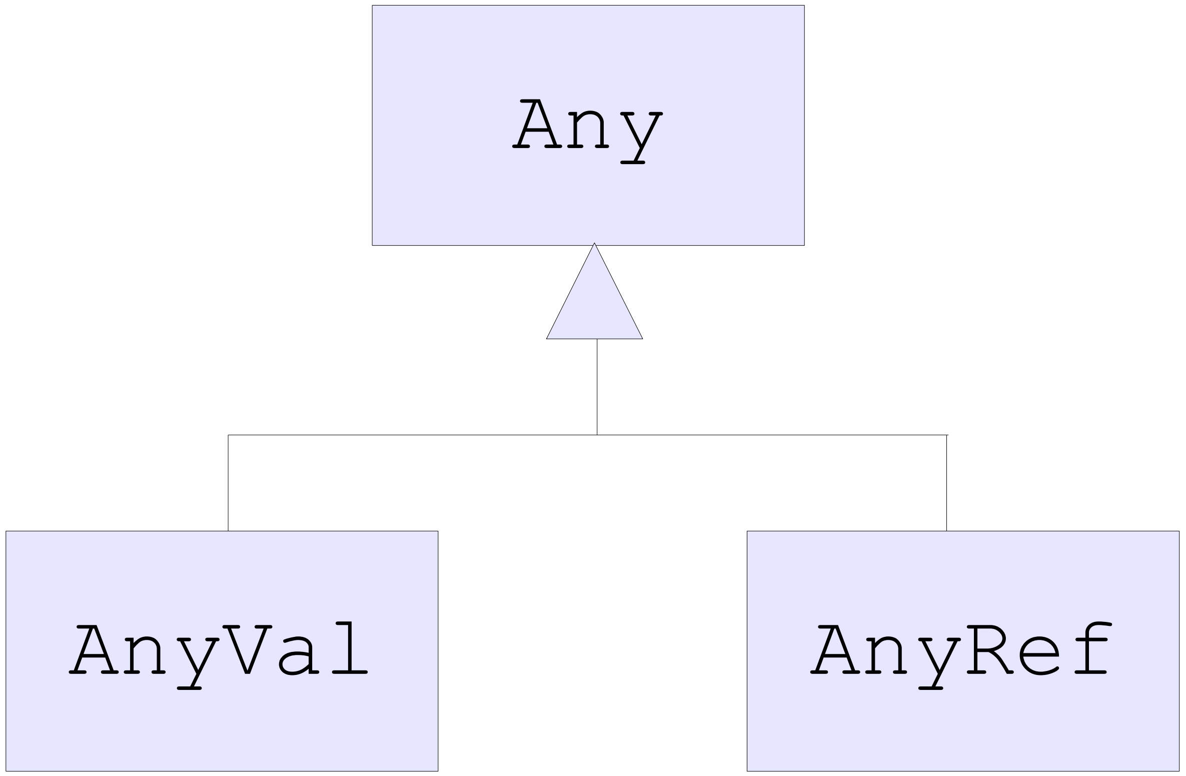
- The `scala.AnyRef` class extends `Scala.Any`. It is the base class for all reference type values.

The `Scala.AnyVal` Class

- The `scala.AnyVal` class extends `Scala.Any`. It is the base class for all value types.
- It has a fixed number subclasses, that describe the available value types.

`scala.Double`	`scala.Float`	`scala.Long`
`scala.Int`	`scala.Short`	`scala.Byte`
`scala.Char`	`scala.Boolean`	`scala.Unit`

The Hierarchy



Primitive Type Values

- Working with primitive type values, whenever there is a need in interoperability with code written in Java an object of the relevant Java class is created.

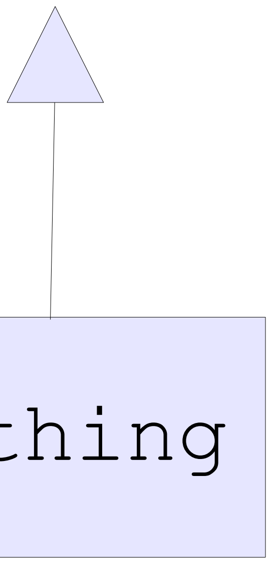
  One example is the transparent creation of objects of type `java.lang.Integer` whenever there is a need in working with integers within code written in Java.

The `Nothing` Type

- This type is a subtype of every other type. There is no value of this type.
- We use this type to signal an abnormal termination and as the element type of empty collections.

The `Nothing` Type

The Nothing Type is Subtype of Any Other Type



The Nothing Type

```
object HelloSample
{
  def main(args:Array[String]):Unit =
  {
    val ob = getMyStack[Int](10)
  }
  def getMyStack[T](num:Int) =
  {
    new EmptyMyStack[T]
  }
}

abstract class MyStack[T](size:Int)
{
  def data:T;
}

class EmptyMyStack[T] extends MyStack[T](0)
{
  def data:Nothing = throw new Exception("empty stack");
}
```

The `Nil` Object

- The `Nil` object extends `List[Nothing]`. It represents an empty list.
- Because `Nothing` is a sub type of any other type we can treat `Nil` as a `List` type no matter what is its elements type.

The `Nil` Object

```
object Demo {
  def main(args:Array[String]):Unit = {
    var obA:List[Rectangle] = Nil;
    var obB:List[String] = Nil;
   //...
  }
}
```

The `Null` Type

- The `Null` type is a final abstract class. There is only one instance of this type. It is `null`. We cannot instantiate this type for getting more objects. The `null` instance is been used for the same purpose we use it in Java.
- The `Null` type is a subtype of all other reference types. It isn't a subtype of all the value types.

The `Null` Type

```
object Demo {
  def main(args:Array[String]):Unit = {
    var obA:Rectangle = null
    var obB:Int = null  //doesn't compile
  }
}
```

The `Option` Type

- The Scala `Option[T]` is a container for zero or one element of a given type.
- The `Option[T]` can be either an object of the type `Some[T]` or of the type `None`. Object of the type `None` represents a missing value.

The `Option` Type

- The following code shows that we can assign a variable of the type `Option` either with a reference for a `Some` object or with a reference for `None` object.

```
object Demo
{
  def main(args:Array[String]):Unit =
  {
    var temp:Option[Int] = Some(5)
    temp = None
  }
}
```

The `None` Object

- The `None` object extends `Option[Nothing]`. When writing `None` we get the reference for the `None` object.

The `Nothing` Class

- The `Nothing` class is a subtype of every other type (including `Null`). There are no instances of this type. It is impossible to instantiate this class.

The `Nothing` Class

- The `Nothing` type is highly useful. One example is the `scala.collection.immutable.Nil` object that extends `List[Nothing]`.
- The lists in Scala are covariant, which means that for any T the `List[T]` type the `Nil` object can be treated as an object of the type `List[T]`.

The `Nothing` Class

```
object Demo {
  def main(args:Array[String]):Unit = {
    var obA:List[Rectangle] = Nil;
    var obB:List[String] = Nil;
   //...
  }
}
```

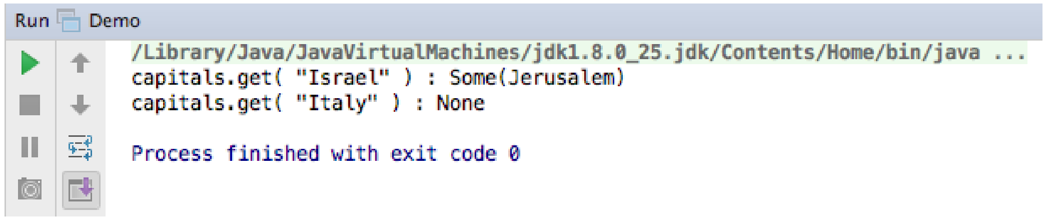
The `Some` Class

- Instantiating the `Some` class we will get an object that represents an existing value. Instantiating `Some[A]` we will get an object that represents a specific value of the type `A`.
- The `get` method of Scala's `Map` produces `Some(value)` if a value corresponding to a given key has been found, or `None` if it wasn't.

The Some Class

- The following code sample shows the use of `None` and `Some` as subtypes of `Option`.

```
object Demo {
  def main(args: Array[String]) {
    val map = Map("Israel" -> "Jerusalem", "England" -> "London")
    println("capitals.get( \"Israel\" ) : " +  map.get( "Israel" ))
    println("capitals.get( \"Italy\" ) : " +  map.get( "Italy" ))
    val temp1:Option[String] = map.get("Israel")
    val temp2:Option[String] = map.get("Italy")
    if(temp1!=None) println(temp1.get)
    if(temp2!=None) println(temp2.get)
  }
}
```



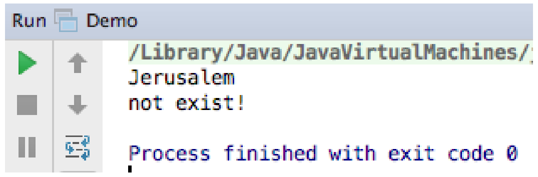
```
/Library/Java/JavaVirtualMachines/jdk1.8.0_25.jdk/Contents/Home/bin/java ...
capitals.get( "Israel" ) : Some(Jerusalem)
capitals.get( "Italy" ) : None

Process finished with exit code 0
```

The `Map` Collection

❖ The common way for taking optional values apart is through patterns matching.

```
object Demo {
  def main(args: Array[String]) {
    val map = Map("Israel" -> "Jerusalem", "England" -> "London")
    val temp1:Option[String] = map.get("Israel")
    val temp2:Option[String] = map.get("Italy")
    println(show(temp1))
    println(show(temp2))
  }
  def show(data:Option[String]):String = {
    data match {
      case Some(str) => str
      case None => "not exist!"
    }
  }
}
```



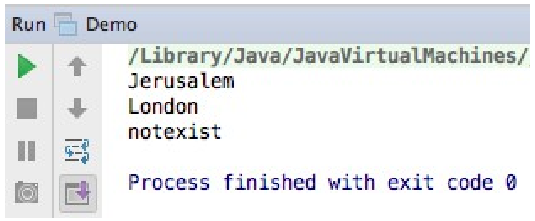
```
/Library/Java/JavaVirtualMachines/
Jerusalem
not exist!

Process finished with exit code 0
```

The `Map` Collection

- The `getOrElse` method assists us with getting the value we expect to get wrapped in a `Some` object.

```
object Demo {
  def main(args: Array[String]) {
    val map = Map("Israel" -> "Jerusalem", "England" -> "London")
    //val temp1:Option[String] = map.get("Israel")
    //val temp2:Option[String] = map.get("Italy")
    println(map.getOrElse("Israel","not exist"))
    println(map.getOrElse("England","not exist"))
    println(map.getOrElse("Italy","notexist"))
  }
}
```



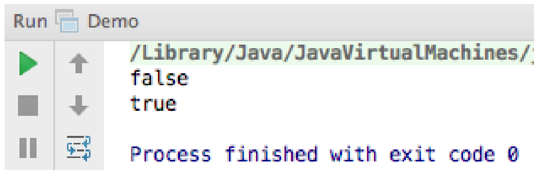
```
/Library/Java/JavaVirtualMachines/
Jerusalem
London
notexist

Process finished with exit code 0
```

The Map Collection

- We can use the isEmpty() for checking the value we hold whether it is None or not.

```
object Demo {
  def main(args: Array[String]) {
    val map = Map("Israel" -> "Jerusalem", "England" -> "London")
    val temp1:Option[String] = map.get("Israel")
    val temp2:Option[String] = map.get("Italy")
    println(temp1.isEmpty)
    println(temp2.isEmpty)
  }
}
```



```
/Library/Java/JavaVirtualMachines/
false
true

Process finished with exit code 0
```