Advanced Functional Thinking

Dr Surbhi Saraswat
School of Computer Science

SCALA CLASSES AND OBJECTS

Anonymous Object: An object which has no reference name is called anonymous object. It is good to create anonymous object when you don't want to reuse it further.

```
class Arithmetic{
    def add(a:Int, b:Int){
        var add = a+b;
        println("sum = "+add);
    }
}
object MainObject{
    def main(args:Array[String]){
        new Arithmetic().add(10,10);
    }
}
```

SCALA CLASSES AND OBJECTS

Singleton Object: It is an object which is declared by using **object keyword instead by class**. No object is required to call methods declared inside singleton object.

```
object Example{
   def main(args:Array[String]){
      SingletonObject.hello()        // No need to create object.
   }
}
object SingletonObject{
   def hello(){
      println("Hello, This is Singleton Object")
   }
}
```

SCALA CLASSES AND OBJECTS

Singleton Object: It is an object which is declared by using object keyword instead by class. No object is required to call methods declared inside singleton object.

- In Scala, there is no static concept. So, Scala creates a singleton object to provide entry point for your program execution.
- The method in the singleton object is globally accessible.
- You are not allowed to create an instance of singleton object.
- You are not allowed to pass parameter in the primary constructor of singleton object.
- In Scala, a singleton object can extend class and traits.
- In Scala, a main method is always present in singleton object.
- The method in the singleton object is accessed with the name of the object(just like calling static method in Java), so there is no need to create an object to access this method.

SCALA CLASSES AND OBJECTS

Companion Object: It is an object whose name is same as the name of the class.
In other words, when an object and a class have the same name, then that object is known as the companion object and the class is known as companion class.

- A companion object is defined in the same source file in which the class is defined.
- This has several benefits.
  - First, a companion object and its class can access each other's private methods and private fields.
  - The **apply** method of companion object allows creating new instances of companion class without having to use the **new** keyword.
  - It allows creating multiple constructors using multiple apply method in companion object (Polymorphism).
  - A companion objects **unapply** method lets you de-construct an instance of a class into its individual components

SCALA CLASSES AND OBJECTS

Companion Object:

A companion object and its class can access each other's private methods and private fields.

The printFilename method in this class will work because it can access the HiddenFilename field in its companion object.

```
class SomeClass {
  def printFilename() = {
    println(SomeClass.HiddenFilename)
  }
}
object SomeClass {
  private val HiddenFilename = "/tmp/foo.bar"
}
```

SCALA CLASSES AND OBJECTS

Companion Object:

The **apply** method of companion object allows creating new instances of companion class without having to use the **new** keyword.

```
class Person {
  var name = ""
}
object Person {
  def apply(name: String): Person = {
    var p = new Person
    p.name = name
    p
  }
}
```

You can create a new instance of the person class by:

```
val p = Person.apply("Fred Flinstone")
```

Or

```
val p = Person("Fred Flinstone")
```

SCALA CLASSES AND OBJECTS

You can create a new instance of the person class by:

```
val p = Person.apply("Fred Flinstone")
```

Or

```
val p = Person("Fred Flinstone")
```

Or

```
val zenMasters = List( Person("Nansen"), Person("Joshu") )
```

To be clear, what happens in this process is:

- You type something like val p = Person("Fred")
- The Scala compiler sees that there is no new keyword before Person
- The compiler looks for an apply method in the companion object of the Person class that matches the type signature you entered
- If it finds an apply method, it uses it; if it doesn’t, you get a compiler error

SCALA CLASSES AND OBJECTS

Companion Object: multiple constructors

```
class Person {
  var name =   " "
  var age = 0
}
object Person {
// a one-arg constructor
  def apply(name: String): Person = {
    var p = new Person
    p.name = name
    p }
  // a two-arg constructor
  def apply(name: String, age: Int): Person = {
    var p = new Person
    p.name = name
    p.age = age
    p }
  }
```

SCALA CLASSES AND OBJECTS

- Scala provides a **helper class**, called **App**, that provides the main method. Instead of writing your own main method, classes can extend the App class to produce concise and executable applications in Scala.

```
class Person {
  var name =  " "
  var age = 0
}
object Person {
// a one-arg constructor
  def apply(name: String): Person = {
    var p = new Person
    p.name = name
    p }
  // a two-arg constructor
  def apply(name: String, age: Int):Person ={
    var p = new Person
    p.name = name
    p.age = age
    p }
  }
```

```
object Main extends App {
  val p = Person.apply("Fred Flinstone")
  val q = Person("John", 30)

  println(s"Name: ${p.name}")
  println(s"Name: ${q.name}, Age: ${q.age}")
}
```

```
Output:
Name: Fred Flinstone
Name: John, Age: 30
```