



Traits, Clases y Objetos en Scala

Traits y Objetos en Scala

Objetivos del tema:

- Conocer la definición de objetos en Scala
- Conocer los objetos de compañía y la función que desempeñan en Scala
- Conocer lo que aportan los Traits

1 Clases, atributos y métodos



Clases, atributos y métodos

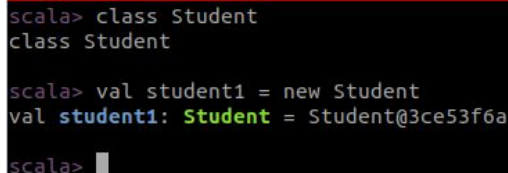
Las clases en Scala son similares a las de Java, son plantillas estáticas que pueden ser instanciadas por muchos objetos en tiempo de ejecución.

Las clases pueden contener métodos, atributos de tipo `val` o `var`, implementar Traits u objetos y extender otras clases.

Ejemplo básico donde la clase `Student` tiene un constructor por defecto que no toma parámetros.

Para instanciar una clase se hace como en Java, usando la palabra reservada `new`

```
class Student  
val student1 = new Student
```



```
scala> class Student  
class Student  
  
scala> val student1 = new Student  
val student1: Student = Student@3ce53f6a  
  
scala> 
```

Clases, atributos y métodos

Ahora veamos un ejemplo donde se muestra la definición de una clase a la que se le define un constructor, un método y sobrescribe el método toString

A diferencia de otros lenguajes de programación, el constructor de esta clase viene definido por los atributos que espera la clase: (var x: Int, var y: Int)

El método move toma dos parámetros enteros, pero devuelve un Unit ya que lo que hace es actualizar el estado del objeto actualizando sus atributos.

```
class Point(var x: Int, var y: Int) {  
  def move(dx: Int, dy: Int): Unit = {  
    x = x + dx  
    y = y + dy  
  }  
  override def toString: String =  
    s"($x, $y)"  
}
```

```
scala> class Point(var x: Int, var y: Int) {  
  |   def move(dx: Int, dy: Int): Unit = {  
  |     x = x + dx  
  |     y = y + dy  
  |   }  
  |   override def toString: String =  
  |     s"($x, $y)"  
  | }  
class Point  
  
scala> val point1 = new Point(2, 3)  
val point1: Point = (2, 3)  
  
scala> point1.x  
val res0: Int = 2  
  
scala> println(point1)  
(2, 3)  
  
scala> 
```



Clases, atributos y métodos

Al igual que en las funciones a las que se les puede definir un valor por defecto, a los constructores de las clases también se les puede establecer un valor por defecto con lo cual se haría que ese constructor tuviera valores opcionales,

```
class Point(var x: Int = 0, var y: Int = 0)

// x e y tiene el valor por defecto, 0
val origin = new Point

// el valor x de point1 será 5, debido a que el constructor
// lee los argumentos de izquierda a derecha.
val point1 = new Point(5)
```

```
scala> class Point(var x: Int = 0, var y: Int = 0)
class Point

scala> val origin = new Point
val origin: Point = Point@4404a6b

scala> val point1 = new Point(5)
val point1: Point = Point@77d95e5a

scala> origin
val origin: Point
scala> origin.x
val res4: Int = 0

scala> point1.x
val res5: Int = 5

scala>
```

2 Objetos singleton



Objetos singleton

Un objeto es una clase que tiene exactamente una instancia.

Se crea perezosamente cuando se hace referencia a él, como un lazy val.

Los objetos singleton son aquellos que contienen valores y métodos que no están asociados a ninguna clase.

Se definen con la palabra reservada `object` en lugar de `class`.

Si lo llevamos al ámbito de Java, se pueden ver a como la definición de métodos estáticos ya que en Scala éstos no existen.

Un objeto singleton se podría ver como un conjunto de funciones de utilidades.

Los objetos pueden extender clases o traits.



Objetos singleton

Ejemplo de definición de un objeto singleton que implementa una función y no está asociado a ninguna clase:

```
object Box {  
  def sum(l: List[Int]): Int = l.sum  
}
```

```
scala> object Box {  
  |   def sum(l: List[Int]): Int = l.sum  
  | }  
object Box  
  
scala> Box.sum(List(9,8,7,6,5))  
val res7: Int = 35  
  
scala> █
```



Objetos singleton - Companion Object

Un companion object es un singleton que está asociado a una clase con el mismo nombre.

Para ello, ambos (clase y objeto) tienen que estar definidos en el mismo fichero y se tienen que llamar igual.

Cuando una clase tiene definido un companion object ambos pueden acceder a todas las funciones y atributos de cada uno, incluso los privados.

Se usan los companion object para los métodos y valores que no son específicos de las instancias de la clase acompañante.



Objetos singleton - Companion Object

```
// se define la clase Student
class Student(val name: String) {
  // su atributo studentId se inicializa
  // llamando a la función de su companion object
  var studentId = Student.newStudentId()
}

object Student {
  var studentId = 0
  def newStudentId() = {
    studentId += 1
    studentId
  }
}
```

```
scala> class Student(val name: String) {
      |   var studentId = Student.newStudentId()
      |   }
      |   object Student {
      |     var studentId = 0
      |     def newStudentId() = {
      |       studentId += 1
      |       studentId
      |     }
      |   }
class Student
object Student

scala> val student1=new Student("charles")
val student1: Student = Student@392a40d9

scala> student1.studentId
val res8: Int = 1

scala> val student2=new Student("john")
val student2: Student = Student@6d487f2b

scala> student2.studentId
val res9: Int = 2
```



3 Case classes



Case classes

Una `case class` puede ser vista como objetos de datos planos e inmutables que deben depender exclusivamente de sus argumentos de construcción.

Para instanciar una `case class` no hace falta la palabra clave `new`.

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

```
scala> case class Book(isbn: String)
class Book

scala> val frankenstein = Book("978-0486282114")
val frankenstein: Book = Book(978-0486282114)

scala> frankenstein
val res11: Book = Book(978-0486282114)

scala>
```



Case classes

Los parámetros de un constructor de una `case class` son `val` y públicos.

Eso quiere decir que no se pueden modificar una vez instanciado y son accesibles desde fuera de la instancia de la clase.

Es posible definir los argumentos de una `case class` como `var`, pero es desaconsejado.

```
case class Message(sender: String, recipient: String, body: String)
val message = Message("luis@madrid.com", "jorge@catalonia.es", "Qué pasa, tron?")
```

```
scala> case class Message(sender: String, recipient: String, body: String)
class Message

scala> val message = Message("luis@madrid.com", "jorge@catalonia.es", "Qué pasa, tron?")
val message: Message = Message(luis@madrid.com,jorge@catalonia.es,Qué pasa, tron?)

scala> message.sender
val res12: String = luis@madrid.com

scala> message.body
val res13: String = Qué pasa, tron?

scala> message.recipient="pepe@granada.com"
      ^
error: reassignment to val

scala> █
```



Case classes

Las case classes son especialmente útiles para el reconocimiento de patrones: Pattern Matching

```
case class Email(sender: String, title: String, body: String)
val notification = Email("charles", "saludos", "qué pasa tron?")
notification match {
  case Email(sender, title, _) =>
    s"You got an email from $sender with title: $title"
  case _ =>
    s"You got an notification"
}
```

```
scala> case class Email(sender: String, title: String, body: String)
class Email

scala> val notification = Email("charles", "saludos", "qué pasa tron?")
val notification: Email = Email(charles,saludos,qué pasa tron?)

scala> notification match {
  |   case Email(sender, title, _) =>
  |     s"You got an email from $sender with title: $title"
  |   case _ =>
  |     s"You got an notification"
  | }
val res19: String = You got an email from charles with title: saludos

scala>
```



4 Traits



Traits

Los traits en Scala son similares a las interfaces de Java

Son usados para definir tipos de objetos con un comportamiento determinado por los métodos provistos por el trait.

Scala permite a los traits ser parcialmente implementados, es posible definir implementaciones por defecto para algunos métodos, principal diferencia con las interfaces de Java.

A diferencia con las clases, los traits no pueden tener parámetros de constructor ni tampoco se pueden instanciar, son abstractas, sólo pueden ser extendidas por una clase u objeto.



Traits

La definición es tan sencillo como usar la palabra clave `trait` seguido de un identificador.

Por ejemplo:

- definamos un trait sin atributos `Similarity`
- definamos el trait `Pet` que tenga sólo un atributo `name`

```
trait Similarity

trait Pet {
  val name: String
}
```

```
scala> trait Similarity
trait Similarity

scala> trait Pet {
  |   val name: String
  | }
trait Pet

scala> val similarity = new Similarity
                        ^
      error: trait Similarity is abstract; cannot be instantiated

scala> 
```



Traits

Los traits pueden ser extendidos/implementado por clases, esto hace que si hay alguna colección que requiere el tipo del trait, las clases que lo implementan sean aceptados.

```
import scala.collection.mutable.ArrayBuffer

trait Pet {
  val name: String
}

class Cat(val name: String) extends Pet
class Dog(val name: String) extends Pet

val dog = new Dog("Harry")
val cat = new Cat("Sally")

val animals = ArrayBuffer.empty[Pet]
animals.append(dog)
animals.append(cat)
// Prints Harry Sally
animals.foreach(pet => println(pet.name))
```

```
scala> trait Pet {
  |   val name: String
  | }
trait Pet

scala> class Cat(val name: String) extends Pet
  | class Dog(val name: String) extends Pet
class Cat
class Dog

scala> val dog = new Dog("Harry")
  | val cat = new Cat("Sally")
val dog: Dog = Dog@462a6cc1
val cat: Cat = Cat@5e4eb85b

scala> import scala.collection.mutable.ArrayBuffer
import scala.collection.mutable.ArrayBuffer

scala> val animals = ArrayBuffer.empty[Pet]
val animals: scala.collection.mutable.ArrayBuffer[Pet] = ArrayBuffer()

scala>
  | animals.append(dog)
  | animals.append(cat)
val res20: animals.type = ArrayBuffer(Dog@462a6cc1, Cat@5e4eb85b)

scala> animals.foreach(pet => println(pet.name))
Harry
Sally

scala> □
```



