Tema 7: Programación Orientada a Objetos

Contenidos

1. Historia y características

Nacimiento

- La Programación Orientada a Objetos es un paradigma de programación que explota en los 80 pero nace a partir de ideas a finales de los 60 y 70
- Primer lenguaje con las ideas fundamentales de POO: Simula
- Smalltalk como lenguaje paradigmática de POO
- Alan Kay es el creador del término “Object-Oriented”
- Artículo de Alan Kay: “The Early History of Smalltalk”, ACM SIGPLAN, March 1993

Alan Kay

> “I invented the term Object-Oriented and I can tell you I did not have C++ in mind.”
>
> “Smalltalk is not only NOT its syntax or the class library, it is not even about classes. I’m sorry that I long ago coined the term objects for this topic because it gets many people to focus on the lesser idea. The big idea is messaging.”
>
> “Smalltalk’s design–and existence–is due to the insight that everything we can describe can be represented by the recursive composition of a single kind of behavioral building block that hides its combination of state and process inside itself and can be dealt with only through the exchange of messages.”

¿Interesados en Smalltalk?

Visitar:

- http://www.squeak.org/
- http://swiki.agro.uba.ar/small_land
- http://www.squeakland.org

Lenguajes OO

- Smalltalk, Java, Scala, Ruby, Python, C#, C++ (si lo usamos con prudencia), …

Del paradigma imperativo al OO

- Programación procedural: estado abstracto (tipos de datos y barrera de abstracción) + funciones
- Siguiente paso: agrupar estado y funciones en una única entidad
- Los objetos son estas entidades

Objeto

- Un objeto contiene un estado (atributos o variables de instancia) y un conjunto de funciones (métodos) que implementan las funcionalidades soportadas
- Al ejecutar un método, el objeto modifica su estado
- Pedimos a un objeto que ejecute un método lanzándole un mensaje

Características de la POO

- Objetos (creados/instanciados en tiempo de ejecución) y clases (plantillas estáticas/tiempo de compilación)
- Los objetos agrupan estado y conducta (métodos)
- Los métodos se invocan mediante mensajes
- *Dispatch dinámico*: cuando una operación es invocada sobre un objeto, el propio objeto determina qué código se ejecuta. Dos objetos con la misma interfaz pueden tener implementaciones distintas.
- Herencia: las clases se pueden definir utilizando otras clases como plantillas y modificando sus métodos y/o variables de instancia.

Clase

- Una clase es la plantilla que sirve para definir los objetos
- En una clase se define los elementos que componen el objeto (sus atributos o campos) y sus métodos
- En algunos lenguajes se pueden definir también en las clases variables (variables de clase) compartidas por todos los objetos de esa clase

2. POO básica en Scala

Definición de clases

Definición de clases en Scala:

- Nombre de la clase
- Campos (variables de instancia) privados **que se deben inicializar** (a no ser que la

clase sea abstracta)
- Métodos, por defecto públicos

Veamos un primer ejemplo, en el que definimos una clase `Contador` que tiene un campo `valor` y dos métodos sin argumentos:

```
class Contador {
    private var valor: Int = 0
    def incrementa(): Unit = { valor += 1 }
    def actual(): Int = { valor }
}
```

Se puede simplificar la declaración dejando que Scala infiera los tipos de datos y quitando las llaves en los métodos que se definen en una línea:

```
class Contador {
    private var valor = 0
    def incrementa() = valor += 1
    def actual() = valor
}
```

También se pueden poner las llaves y quitar el `=` en las definiciones de métodos de tipo `Unit` (métodos imperativos que contienen sentencias de ejecución que cambian estado local y no devuelven nada):

```
class Contador {
    private var valor: Int = 0
    def incrementa() { valor += 1 }
    def actual() = { valor }
}
```

Para usarlo:

```
val c = new Contador
c.incrementa()
c.actual() ⇒ 1
c.actual ⇒ 1. Se puede llamar al método sin los paréntesis
```

Estamos creando un nuevo contador, incrementando su valor y devolviendo su valor actual.

Los métodos deben tener nombre distinto de los campos, el siguiente código daría error porque estamos definiendo el nombre del método con el mismo nombre que el campo privado:

```
class Contador {
    private var valor = 0
```

```
    def valor() = valor // Error
}
```

Una diferencia muy importante entre Scala y Java es que Scala permite invocar a los métodos sin argumentos sin poner los paréntesis. Es importante darse cuenta de que la segunda llamada `c.actual` no está accediendo a un campo, sino ejecutando un método. Es idéntica a la invcación anterior.

También sería posible declarar un método que no tiene argumentos sin escribir los paréntesis. Por ejemplo, el método `actual` lo podríamos escribir de la siguiente forma:

```
class Contador {
    private var valor = 0
    def incrementa() { valor += 1 }
    def actual = valor // Sin () en la declaración
}
```

En este caso, sí que es obligatorio invocar al método sin paréntesis:

```
c.actual
```

y sería erróneo hacerlo así:

```
c.actual() ⇒ error
```

Campos públicos

¿Qué pasa si no declaramos los campos de una clase como privados? Veámoslo con otro ejemplo:

```
class Persona {
    var edad = 0
}
```

Lo que sucede es que Scala genera automáticamente un campo privado `edad` y una pareja de métodos *getter* y *setter* que recuperan y definen el valor. Estos métodos son públicos y se llaman:

- `edad`
- `edad_=`

Por ejemplo:

```
    println(toni.edad) // Llama al método `toni.edad()`
```

```
    fred.edad = 21 // Llama al método `fred.age_=(21)`
```

La ventaja de este enfoque en el que Scala no deja acceder directamente a los campos es que en cualquier momento podemos redefinir los métodos *getter* y *setter* y actualizar todas las llamadas que acceden a esos valores.

Por ejemplo, podemos impedir que alguien haga más joven a una persona:

```
class Persona {
    private var edadPrivada = 0 // Hacemos el campo privado

    def edad = edadPrivada
    def edad_= (nueva: Int) {
        if (nueva > edadPrivada) edadPrivada = nueva; // No puede convertirse en má
    }
}
```

El usuario de la clase puede seguir accediendo a `fred.age`, pero ahora no puede convertirlo en más joven:

```
val fred = new Persona
fred.edad = 30
fred.edad = 21
println(fred.edad) // 30
```

- Si un campo es privado, su *getter* y su *setter* también lo son

Un campo también pueden ser de tipo `val`, en cuyo caso se trata de un valor inmutable que se proporciona en la inicialización del objeto y que no puede ser reasignados. Sólo se genera su *getter*. Por ejemplo:

```
class Persona(nom : String) {
    val nombre = nom
    private var edadPrivada = 0 // Hacemos el campo privado

    def edad = edadPrivada
    def edad_= (nueva: Int) {
        if (nueva > edadPrivada) edadPrivada = nueva; // No puede convertirse en má
    }
}
```

Si intentamos modificar el nombre se produce un error:

```
val fred = new Persona("Fred")
```

```
fred.nombre = "Pepe"
// error: reassignment to val
```

Constructor primario

El **constructor primario** puede tener parámetros, colocándolos justo después del nombre de la clase. Lo hemos visto antes con el ejemplo del nombre de la persona. Podemos obligar a inicializar también la edad:

```
class Persona(nom: String, años: Int) {
   private var edadPrivada = años
   ...
}
```

Código de inicialización

Podemos escribir código entre la declaración de la clase y la definición de sus métodos. Este código se ejecuta cada vez que se crea un nuevo objeto (una nueva instancia) de la clase. Por ejemplo:

```
class Persona(nom : String, años: Int) {
    val nombre = nom
    private var edadPrivada = años
    println("Se acaba de construir otra persona")  // Se ejecuta en el constructor

    def edad = edadPrivada
    def edad_= (nueva: Int) {
        if (nueva > edadPrivada) edadPrivada = nueva; // No puede convertirse en má
    }
}
```

Campos paramétricos en el constructor primario

Si precedemos el o los parámetros del constructor con un `val` o `var` se definen automáticamente campos en la clase. Se denominan **campos paramétricos**:

```
class Persona(val nombre: String, var edad: Int) {
   def descripcion = nombre + " tiene " + edad + " años"
}
```

Hemos definido los campos `nombre` y `edad` en la clase. El campo `nombre` es de tipo `val` y sólo se generará su *getter*. El campo edad es de tipo `var` y se generará su *getter* y su *setter*:

```
val fred = new Persona("Fred", 30)
fred.nombre ⇒ "Fred"
fred.edad ⇒ 30
fred.descripcion ⇒ "Fred tiene 30 años"
```

Es posible definir valores por defecto en los parámetros:

```
class Persona(val nombre: String, var edad: Int = 0)
```

Por ejemplo:

```
val fred = new Persona("Fred")
fred.edad ⇒ 0
fred.descripcion ⇒ "Fred tiene 0 años"
fred.edad = 30
fred.descripcion ⇒ "Fred tiene 30 años"
```

Constructores auxiliares

Es posible definir constructores auxiliares utilizando `this` como nombre:

```
class Persona(val nombre: String){
    private var edadPrivada = 0

    def edad = edadPrivada
    def edad_= (nueva: Int) {
        if (nueva > edadPrivada) edadPrivada = nueva; // No puede convertirse en má
    }

    def this(nombre: String, edad: Int) { // Constructor auxiliar
        this(nombre) // Llama al otro constructor auxiliar
        this.edad = edad
    }

    def descripcion = nombre + " tiene " + edad + " años"
}
```

Ahora podemos crear una `Persona` de dos formas:

```
val p = new Persona("Fred") // Constructor primario
val p2 = new Persona("Fred", 42) // Constructor auxiliar
```

Un ejemplo con objetos inmutables

Hemos visto que en Scala hay bastantes clases inmutables (listas, tuplas, etc.). Para definir un objeto inmutable basta con definir todos sus campos como `val` y utilizar *semántica de copia* en todos los métodos que deben devolver un objeto modificado. De esta forma, siempre se devolverá un objeto nuevo y se utilizará un estilo de programación más cercano al funcional. Este tipo de objetos se denominan también *objetos valor* (*value objects* en inglés), porque se comportan de una forma idéntica a los tipos valor (Int, Double, etc.)

Un ejemplo en el que definimos la clase `Punto2D` con una semántica de *objeto valor*:

```
class Punto2D(xc: Int, yc: Int) {
    val x: Int = xc
    val y: Int = yc

    def mover(incX: Int, incY: Int): Punto2D = new Punto2D(x + incX, y + incY)

    def +(otroPunto: Punto2D): Punto2D =
       new Punto2D(x + otroPunto.x, y + otroPunto.y)

    def +(inc: Int): Punto2D =
        new Punto2D( x+inc, y+inc)

    def descripcion = "(" + x + "," + y + ")"
}
```

Hemos definido dos métodos `+` cada uno que acepta un tipo de dato distinto. El primero acepta otro `Punto2D` y devuelve la suma de los dos puntos y el segundo acepta un `Int` y devuelve el resultado sumar el entero a las dos coordenadas.

Un ejemplo de ejecución:

```
val p1 = new Punto2D(10,10)
val p2 = new Punto2D(5,3)
val p3 = p1 + p2
p3.descripcion // ⇒ (15,13)
val p4 = p1 + 4
p4.descripcion // ⇒ (14,14)
```

Es importante recalcar que **en Scala no existen operadores**: las expresiones `p1 + p2` y `p1 + 4` las convierte el compilador en llamadas a métodos:

```
val p3 = p1.+(p2)
val p4 = p1.+(4)
```

Singletons

En Scala las clases no pueden tener métodos ni variables estáticas (de clase). En este

sentido, Scala es más orientado a objetos que Java, porque obliga a que cualquier llamada a un método se haga sobre un objeto.

Podemos definir *singletons* (instancias únicas) con la instrucción `object`. La definición es la misma que la de una clase, con la diferencia de que se usa la palabra `object` en lugar de `class`:

```
object Cuenta {
    private var ultimoId = 0
    def nuevoId() = { ultimoId += 1; ultimoId }
}

Cuenta.nuevoId()
```

La llamada a `nuevoId` se hace sobre el objeto `Cuentas`. En Java esto se haría definiendo el método `nuevoId` como un método estático.

En la definición del *singleton* se define al mismo tiempo el objeto y la clase. Al no utilizar la palabra `class` no es posible definir más objetos de esa clase.

La forma de definir en Scala métodos estáticos (de clase) es definiéndolos dentro de un `object` con el mismo nombre de la clase. A este `object` con el mismo nombre que la clase se le denomina *objeto compañero*. El objeto compañero y su clase pueden acceder mutuamente a sus miembros privados.

Por ejemplo, se podría hacer que el `object` definido anteriormente fuera el *objeto compañero* de una clase `Cuenta`:

```
object Cuenta {
    private var ultimoId = 0
    private def nuevoId() = { ultimoId += 1; ultimoId }
}

class Cuenta {
    val id = Cuenta.nuevoId()
    private var balance = 0.0
    def deposita(cantidad: Double) { balance += cantidad }
    def saldo() = balance
}
```

Ahora podemos crear distintas cuentas y trabajar con ellas:

```
val cuenta1 = new Cuenta
val cuenta2 = new Cuenta
cuenta1.deposita(100)
cuenta2.deposita(200)
```

```
println("La cuenta " + cuenta1.id + " tiene " + cuenta1.saldo + " euros")
println("La cuenta " + cuenta2.id + " tiene " + cuenta2.saldo + " euros")
```

Método `apply`

Hemos visto que en Scala es posible inicializar listas o arrays sin llamar a `new`:

```
val lista = List(12,3,4)
val array = Array("hola","adios")
```

¿Qué está pasando aquí? ¿Cómo se está creando un objeto sin llamar a `new`? Realmente, el compilador está expandiendo las definiciones anteriores en una llamada a un método `apply` definido en el objeto compañero de la clase `List` y `Array`:

```
val lista = List.apply(12,3,4)
val array = Array.apply("hola","adios")
```

Nosotros podríamos hacer lo mismo con nuestras cuentas y permitir crear las cuentas de esta forma:

```
object Cuenta {
    private var ultimoId = 0
    private def nuevoId() = { ultimoId += 1; ultimoId }
    def apply(balanceInicial: Double) =  new Cuenta(balanceInicial)
}

class Cuenta {
    val id = Cuenta.nuevoId()
    private var balance = 0.0

    private def this(cantidad: Double) {
        this()
        balance = cantidad
    }

    def deposita(cantidad: Double) { balance += cantidad }
    def saldo() = balance
}
```

Ahora podemos crear una nueva cuenta de la siguiente forma:

```
val c1 = Cuenta(1000.0)
```

Herencia

Resumen: diferencias con Java

- Las palabras reservadas `extends` y `final` funcionan como en Java
- Debes usar `override` cuando se sobreescribe un método de la clase padre
- Sólo el constructor primario puede llamar al constructor primario de la clase padre
- Puedes sobreescribir campos

Por ejemplo podemos definir una clase `Empleado` que hereda de `Persona`, añadiéndole el campo `sueldo`:

```
class Empleado(nombre: String) extends Persona(nombre) {
    var sueldo = 0.0
}
```

Relación "IS-A"

En una relación de herencia entre clases se debe cumplir la relación ES-UN (*IS-A* en inglés):

> Si una clase B es una subclase de una clase A la especificación de B implica la especificación de A, o, dicho de otra forma, B IS-A A

Por ejemplo:

- Un Empleado *es-una* Persona
- Un Gato *es-un* Mamífero

La subclase hereda todos los campos y métodos de la clase padre. Es posible asignar (o pasar como parámetro) un objeto de la subclase en una variable de la clase padre (al revés no).

Constructores

Hemos visto que se debe invocar siempre al constructor primario en el `extends`. También se pueden invocar a constructores auxiliares:

```
class Empleado(nombre: String, edad: Int, var sueldo : Double) extends Persona(nomb
```

Aquí estamos obligando a que todos los objetos empleados deben crearse indicando el nombre (obligatorio por ser personas), la edad y el sueldo.

Una vez definida la nueva clase, podemos crear instancias y llamar a métodos de la clase propia o de la clase padre:

```
val emp = new Empleado("Ana García", 25, 23000)
emp.sueldo ⇒ Double = 23000.0
```

```
emp.descripcion ⇒ String = Ana García tiene 25 años
```

Se puede definir una subclase de una clase Java:

```
class Square(x: Int, y: Int, width: Int) extends java.awt.Rectangle(x, y, width, wi
```

Sobreescritura de campos

Se puede sobreescribir un campo `val` o un `def` sin parámetros con otro `val` con el mismo nombre. Se puede acceder al valor de la clase padre con la variable `super`.

Ejemplo:

```
class Vehiculo {
    val numeroRuedas = 0
    def descripcion = numeroRuedas + " ruedas"
}

class Bicicleta extends Vehiculo {
    override val numeroRuedas = 2
    override def descripcion = "Soy una bicicleta y tengo " + super.descripcion
}
```

Nuevos métodos

La nueva clase puede añadir nuevos métodos a los definidos en la superclase:

```
class Coche extends Vehiculo {
    var velocidad = 0.0
    override val numeroRuedas = 4

    def acelera(incVelocidad: Double): Double = {
        velocidad += incVelocidad
        velocidad
    }

    def frena(): Double = {
        velocidad = 0.0
        velocidad
    }

    override def descripcion = "Soy un automóvil, tengo " + super.descripcion + " y
}
```

Clases abstractas

Las clases padres de una relación de herencia pueden ser *abstractas*. No es posible crear objetos de una clase abstracta.

```
abstract class Mamifero(val nombre: String) {
   // Campo inicializado en la superclase
   var peso: Double = 0.0

   // Campo abstracto
   var inteligencia: Int; // de 1 a 100

   // Método abstracto sin cuerpo; lo deben implementar las subclases
   def hazSonido(): String
}

class Gato(nombre: String) extends Mamifero(nombre) {
   // Inicialización de campos abstractos de la superclase
   var inteligencia = 30;

   // Métodos propios
   def maulla() = "Miauuuu.. me llamo " + nombre
   def araña() = "Me has enfadado y te he arañado"

   // Implementación de los métodos abstractos de la clase padre
   // No se requiere override
   def hazSonido() = maulla()
}

class Perro(nombre: String) extends Mamifero(nombre) {
   // Inicialización de campos abstractos de la superclase
   var inteligencia = 40;

   // Métodos propios
   def ladra() = "Gauuuu.. me llamo " + nombre
   def alegrate() = "Estoy contento y muevo el rabo"

   // Implementación de los métodos abstractos de la clase padre
   // No se requiere override
   def hazSonido() = ladra()
}
```

- En una subclase no hay que utilizar `override` para definir el método abstracto
- Podemos definir campos en la clase abstracta, e incluso inicializarlos
- Las subclases deben inicializar los campos no inicializados en la superclase

Ejemplos:

```
val gato = new Gato("Misifú")
```

```
val perro = new Perro("Boby")
gato.inteligencia // ⇒ 30
gato.hazSonido() // ⇒ "Miauuuu.. me llamo Misifú"
gato.araña() // ⇒ "Me has enfadado y te he arañado"
perro.inteligencia // ⇒ 40
perro.hazSonido() // ⇒ "Gauuuu.. me llamo Boby"
perro.alegrate() // ⇒ "Estoy contento y muevo el rabo"
```

Como hemos visto antes, podemos utilizar métodos definidos en la superclase llamando a `super`:

```
class Minino(nombre: String) extends Gato(nombre) {
   peso  = 1.0
   override def maulla() = super.maulla() + " y soy un minino"
}
val gato = new Minino("Kitty")
gato.maulla() // ⇒  "Miauuuu.. me llamo Kitty y soy un minino"
```

Resumen modificadores

- `override`: necesario en aquellos miembros que sobreescriben un miembro no abstracto de la superclase
- `final`: cuando queremos asegurarnos de que una subclase no pueda sobreescribir un miembro
- `private`: igual que en Java, los miembros privados sólo son visibles dentro de la clase u objeto que contienen la definición del miembro. Por defecto todos los miembros son públicos.
- También se pueden añadir modificadores a los campos paramétricos.

Downcasting

- Es posible asignar un objeto de una subclase a una variable de un tipo de la superclase
- Sólo se pueden acceder a los métodos y campos definidos en la superclase

```
val animal: Mamifero = new Gato("Misifu")
animal.peso // ⇒ 0.0
animal.hazSonido // ⇒ "Miauuuu.. me llamo Misifú"
animal.inteligencia // ⇒ 30
animal.araña // ⇒ error
```

- Para poder utilizar los métodos específicos de la subclase, debemos hacer un *downcasting*, asignar el objeto a una variable de la subclase y utilizar el método `asInstanceOf`:

```
val gato: Gato = animal.asInstanceOf[Gato]
```

```
gato.araña // ⇒ "Me has enfadado y te he arañado"
```

Comprobando el tipo de una variable

Para comprobar el tipo de una variable `p`:

```
if (p.isInstanceOf[Empleado]) {
    val s = p.asInstanceOf[Empleado] // s tiene el tipo Empleado
    ...
}
```

En Scala se suele prefir hacerlo usando *pattern matching* de la siguiente forma:

```
p match {
        case s: Empleado => ... // Procesa s como un Empleado
        case _ => ... // p no es un Empleado
}
```

Un último ejemplo:

```
val animales: List[Mamifero] = new Gato("Misifú") :: new Perro("Boby") :: new Perro
animales.foreach(x => println(x.hazSonido()))
animales.foreach(x => x match {
   case s: Gato => println(s.araña())
   case s: Perro => println(s.alegrate())
   }
)
```

Ejemplo BinaryTree

Podemos usar lo que hemos aprendido hasta ahora para definir un árbol binario de enteros en Scala:

```
abstract class BinaryTree
object Vacio extends BinaryTree
class NodoBinaryTree(val value: Int,
                     val left: BinaryTree,
                     val right: BinaryTree) extends BinaryTree
```

Un ejemplo de utilización podría ser la siguiente función `sumaBTree` que toma un árbol binario y devuelve la suma de todos suns nodos:

```
def sumaBTree(tree:BinaryTree): Int = {
```

```
    tree match {
            case t : NodoBinaryTree =>  t.value + sumaBTree(t.left) + sumaBTree(t.r
        case Vacio => 0
    }
}
```

Creamos los nodos hojas usando árboles vacíos como hijos (el objeto `Vacio`). El árbol padre lo construimos a partir de los nodos hojas:

```
val t1 = new NodoBinaryTree(2,Vacio,Vacio)
val t2 = new NodoBinaryTree(5,Vacio,Vacio)
val t3 = new NodoBinaryTree(4,t1,t2)
sumaBTree(t3) // ⇒ Int = 11
```

Clases genéricas

Al igual que hemos visto con las funciones, las clases pueden tener parámetros de tipo. En Scala hay que definirlos entre corchetes:

```
class Pareja[T, S](val primero: T, val segundo: S)
```

Esto define una clase con dos parámetros de tipo: `T` y `S`. Se pueden usar estos parámetros de tipo para definir los tipos de las variables, de los parámetros de los métodos y de los valores devueltos.

Una clase con uno o más parámetros de tipo se denomina genérica. Cuando se substituye los parámetros de tipo por tipos reales, obtenemos una clase normal, como por ejemplo `Pair[Int, String]`.

Scala infiere los parámetros reales a partir de los parámetros del constructor:

```
val p = new Pareja(42, "Pepe") // Crea una instancia de una clase Pareja[Int, Strir
```

También los puedes especificar tu mismo:

```
val p2 = new Pair[Double, String](42, "Pepe")
```

Métodos genéricos

Los métodos también pueden ser genéricos. Un ejemplo:

```
class Pareja[T, S](val primero: T, val segundo: S) {
```

```
    def devuelveTupla: (T,S) = (primero, segundo)
}
```

Un ejemplo completo: árbol binario genérico

Podemos definir la clase genérica `BinaryTree` en el que podemos guardar un tipo cualquiera `T` y en la que definimos el método `suma` genérico, al que hay que pasar una función y un elemento neutro que devuelven los nodos vacíos:

```
abstract class BinaryTree[T] {
    // método suma abstracto
    def suma(neutro: T, sumaDatos: (T,T) => T): T
}

class Vacio[T] extends BinaryTree[T] {
    // el árbol binario vacío devuelve el elemento neutro de la suma
    def suma(neutro: T, sumaDatos: (T,T) => T): T = neutro
}

class NodoBinaryTree[T](val value: T,
                        val left: BinaryTree[T],
                        val right: BinaryTree[T]) extends BinaryTree[T] {
    // el nodo implementa la suma usando una recursión
    def suma(neutro: T, sumaDatos: (T,T) => T): T = {
        val sumaLeft: T = this.left.suma(neutro, sumaDatos)
        val sumaRight: T = this.right.suma(neutro, sumaDatos)
        sumaDatos(sumaDatos(this.value, sumaLeft), sumaRight)
    }
}
```

Ejemplo de funcionamiento:

```
val vacio = new Vacio[Int]
val bt1 = new NodoBinaryTree[Int](10,vacio,vacio)
val bt2 = new NodoBinaryTree[Int](25,vacio,vacio)
val bt3 = new NodoBinaryTree[Int](15,bt1,bt2)
bt3.suma(0, (a,b) => a+b) // ⇒ Int = 50
bt3.suma(0, _+_) // Expresión equivalente a la anterior ⇒ Int = 50
```

3. Traits

- Un *trait* encapsula definiciones (abstractas o concretas) de métodos y campos, que se pueden reutilizar en otras clases para **ampliar** su especificación. Es idéntico a una **interfaz** de Java, con la posibilidad adicional de incluir implementaciones de los métodos y campos adicionales.
- Decimos que una clase **se mezcla** con un *trait*. Cuando una clase se mezcla con un trait

debe implementar todos los métodos abstractos del mismo. A diferencia de la herencia, donde una clase sólo puede heredar de una superclase, una clase puede mezclar cualquier número de *traits*.
- Se definen como una clase sustituyendo la palabra clave `class` por `trait`

Desde el punto de vista de diseño orientado a objetos los *traits* representan funcionalidades o propiedades que se pueden aplicar **a distintas clases**. Cuando una clase se mezcla con un *trait*, no se está definiendo una relación ES-UN, sino que se está **ampliando el comportamiento** de una clase ya existente con una nueva funcionalidad. No es posible crear objetos a partir de un *trait*.

Un *trait* define un nuevo tipo en nuestro programa. Podemos definir variables y parámetro de ese nuevo tipo y asignar a esas variables objetos creados con *otras clases* que extienden el trait. En la variable definida podremos llamar sólo a métodos del trait.

Ejemplos de traits con métodos abstractos y métodos implementados

Supongamos el *trait* `Volador` en el que se declara (sin implementar) el método `vuela()`:

```
trait Volador {
   def vuela()
}
```

Podemos entonces definir una clase `Murcielago` que hereda de `Mamifero` y *se mezcla* con `Volador`. La clase debe implementar el método `vuela` si queremos hacerla no abstracta:

```
class Murcielago(nombre: String) extends Mamifero(nombre) with Volador {
  var inteligencia = 30
  def hazSonido() = "Cri, cri"
  def vuela() = println("Vuelo con mis alas de murciélago")
}
```

Si la clase que se mezcla con el *trait* no usa `extends` para extender otra clase, debemos escribir `extends` en lugar de `with`:

```
class Mosquito extends Volador {
   def vuela() = println("Vuelo más rápido que tu mano")
}
```

Supongamos el *trait* `Parlanchin` en el que se define el método `habla`:

```
trait Parlanchin {
   def habla() = println("Hablo por los codos")
```

```
}
```

Para que una clase utilice (mezcle) el *trait* se puede escribir de dos formas:

- Si la clase no hereda de ninguna otra clase se debe utilizar la palabra `extends`:

```
class Rana extends Parlanchin {
   override def toString = "Rana"
}

val ranita = new Rana
ranita.habla()
⇒ Hablo por los codos
```

- Si la clase que se mezcla con el *trait* se define extendiendo una superclase, se utiliza la palabra `with`:

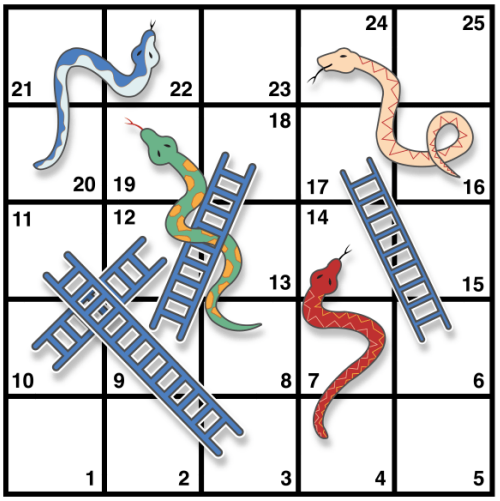
```
class Animal
class Rana extends Animal with Parlanchin {
   override def toString = "Rana"
}
```

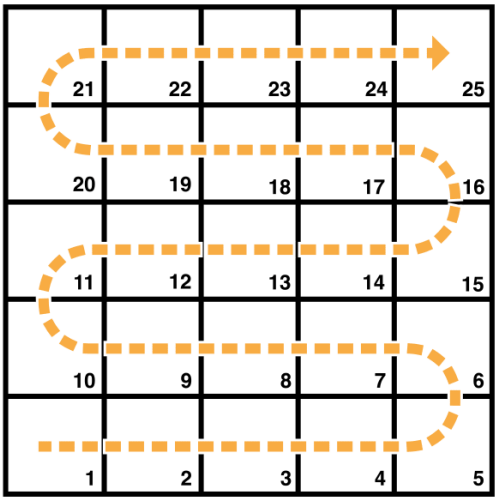
Se cual sea la forma definir el primer *trait* cuando se añade más de un `trait` los siguientes hay que declararlos con `with`:

```
trait TienePatas {
   def tienePatas = true;
}

class Rana extends Parlanchin with TienePatas {
   override def toString = "Rana"
}

val ranita = new Rana
ranita.habla()
ranita.tienePatas
```

Un ejemplo de traits como interfaces: el juego de serpientes y escaleras

Vamos a poner un ejemplo algo más largo de lo habitual, implementando un juego de tablero, el juego de *serpientes y escaleras*. En el juego se comienza en la casilla inicial y se avanza con el resultado de la tirada de un dado. Si se llega a una casilla donde hay una escalera **se avanza** automáticamente hasta el final de la escalera. Si se llega a una casilla donde hay una serpiente **se retrocede** hasta el final de la serpiente. El juego termina cuando se llega **exactamente** a la casilla final.





Para implementarlo definimos una clase abstracta `JuegoTablero` y una clase `SerpientesEscaleras` que la exitiende. El método `jugar` de la clase `SerpientesEscaleras` realiza el bucle que implementa el comportamiento del juego para un único jugador. En este método no mostraremos nada por pantalla porque queremos definir en esta clase sólo el núcleo del juego y dejar que sea otra clase la que defina cómo mostrar el juego. Así podemos crear distintas formas de mostrar el comportamiento del juego **sin cambiar la clase que realiza el juego**. Utilizamos un *trait* en el que se definen los métodos debe implementar la clase que muestre el juego. Lo llamaremos `PresentadorJuegoTablero`:

```
trait PresentadorJuegoTablero {
    def comienzoJuego(juego: JuegoTablero): Unit
    def juego(juego: JuegoTablero, tiradaDado: Int): Unit
    def finalJuego(juego: JuegoTablero): Unit
}
```

Definimos una clase concreta que implementa este *trait*, haciendo que los métodos muestren por pantalla con `println` la evolución del juego:

```
class PresentadorTexto extends PresentadorJuegoTablero {
    var numeroTurnos = 0
    def comienzoJuego(juego: JuegoTablero) {
        numeroTurnos = 0
        println("Ha empezado el juego")
        println("El juego está usando un dado de " + juego.dado.caras + " caras")
    }

    def juego(juego: JuegoTablero, tiradaDado: Int) {
        numeroTurnos += 1
        println("Nueva tirada: " + tiradaDado)
        println("Casilla: " + juego.casillaActual)
    }

    def finalJuego(juego: JuegoTablero) {
```

```
        println("Final del juego")
        println("El juego ha durado " + numeroTurnos + " turnos")
    }
}
```

En el método `jugar` de la clase `SerpientesEscaleras` se llama al objeto presentador en cada paso del juego.

El código completo es el siguiente:

```
class Dado(val caras: Int) {
    private val rnd = new scala.util.Random
    def tirada(): Int = rnd.nextInt(caras) + 1
}

abstract class JuegoTablero {
    val dado: Dado
    var casillaActual: Int
    def jugar(): Unit
}

trait PresentadorJuegoTablero {
    def comienzoJuego(juego: JuegoTablero): Unit
    def juego(juego: JuegoTablero, tiradaDado: Int): Unit
    def finalJuego(juego: JuegoTablero): Unit
}

class PresentadorTexto extends PresentadorJuegoTablero {
    var numeroTurnos = 0
    def comienzoJuego(juego: JuegoTablero) {
        numeroTurnos = 0
        println("Ha empezado el juego")
        println("El juego está usando un dado de " + juego.dado.caras + " caras")
    }

    def juego(juego: JuegoTablero, tiradaDado: Int) {
        numeroTurnos += 1
        println("Nueva tirada: " + tiradaDado)
        println("Casilla: " + juego.casillaActual)
    }

    def finalJuego(juego: JuegoTablero) {
        println("Final del juego")
        println("El juego ha durado " + numeroTurnos + " turnos")
    }
}

class SerpientesEscaleras(unDado: Dado) extends JuegoTablero {
    val casillaFinal = 25
    val dado = unDado
```

```
    val tablero: Array[Int] = Array.fill(casillaFinal+1)(0)
    tablero(3) = 8; tablero(6) = 11; tablero(9) = 9; tablero(10) = 2;
    tablero(14) = -10; tablero(19) = -11; tablero(22) = -2; tablero(24) = -8
    var presentador: PresentadorJuegoTablero = null
    var casillaActual = 0

    def jugar() {
        casillaActual = 0
        var finalJuego = false
        // llamamos al objeto presentador
        // para informarle que comienza el juego
        if (presentador != null) presentador.comienzoJuego(this)
        do {
            val tirada = dado.tirada()
            val nuevaCasilla = tirada + casillaActual
            if (nuevaCasilla == casillaFinal) {
                casillaActual = nuevaCasilla
                finalJuego = true
            }
            if (nuevaCasilla < casillaFinal) {
                casillaActual = nuevaCasilla
                casillaActual += tablero(casillaActual)
            }
            // llamamos al objeto presentador
            // para informarle que se ha hecho una tirada
            if (presentador != null) presentador.juego(this, tirada)
        } while (!finalJuego)
        // llamamos al objeto presentador
        // para informarle que el juego ha terminado
        if (presentador != null) presentador.finalJuego(this)
    }
}

val dado = new Dado(6)
val juegoSerpientes = new SerpientesEscaleras(dado)
val presentador = new PresentadorTexto
juegoSerpientes.presentador = presentador
juegoSerpientes.jugar
```

La utilización del *trait* para definir el presentador del juego permite que podamos cambiar la forma de mostrar el juego sin tener que modificar la propia clase que realiza el juego. Podemos definir distintas formas de mostrar el juego, definiendo distintas implementaciones del *trait*. Por ejemplo, una presentación por la salida de texto, haciendo *println*. O una presentación gráfica en pantalla. O un robot que simule el juego…

Tiempo de ejecución y tiempo de compilación

Es posible mezclar con un trait un objeto concreto, al realizar su creación. De esta forma, los traits se pueden utilizar en tiempo de ejecución (es una característica *dinámica* de Scala que

se define en *runtime*) a diferencia de la herencia de clases, que sólo se puede utilizar en tiempo de compilación (una característica *estática* que se define en tiempo de compilación).

Por ejemplo, en el caso anterior, podemos hacer que sólo una rana en concreto sea parlanchina:

```
class Animal
trait TienePatas {
   def tienePatas = true;
}
trait Parlanchin {
   def habla() = println("Hablo por los codos")
}
class Rana extends Animal with TienePatas {
   override def toString = "Rana"
}

val ranita = new Rana
val ranaParlanchina = new Rana with Parlanchin
```

Hemos creado dos objetos de la clase `Rana` y sólo el segundo de ellos se mezcla con el *trait* `Parlanchin`. Notad que en este caso tenemos que utilizar `with`.

Sólo la segunda rana puede hablar:

```
ranita.habla
⇒ <console>:12: error: value habla is not a member of Rana
ranaParlanchina.habla
⇒ Hablo por los codos
```

Traits para modificar el comportamiento

Es posible utilizar los traits para modificar un comportamiento predefinido de una clase, sobreescribiendo en el trait un método ya definido en la clase.

Vamos a verlo con un ejemplo, en el que implementamos una cola.

Definimos una sencilla cola de enteros, con los métodos `put` y `get` que añaden y sacan elementos. Primero definimos la clase abstracta `IntQueue` que define los métodos sin implementar, y después definimos la clase `BasicIntQueue` que los implementa

```
import scala.collection.mutable.ListBuffer

abstract class IntQueue {
  def get(): Int
  def put(x: Int)
}
```

```
class BasicIntQueue extends IntQueue {
  private val buf = new ListBuffer[Int]
  def get() = buf.remove(0)
  def put(x:Int) = buf += x
}
```

Lo probamos:

```
scala> val cola = new BasicIntQueue
scala> cola.put(10)
scala> cola.put(20)
scala> cola.get()
res2: Int = 10
scala> cola.get()
res3: Int = 20
```

Vamos a modificar el comportamiento de `put` mediante un *trait*:

```
trait Doubling extends IntQueue {
  abstract override def put(x:Int) = super.put(2 * x)
}
```

El trait `Doubling` hace dos cosas muy interesantes:

- Declara como superclase `IntQueue`. Esta declaración hace que el *trait* sólo pueda ser mezclado en una clase que también extienda `IntQueue`.
- Define un método también llamado `put` que sobreescribe el método `put` de cualquier clase con la que se mezcle. El modificador `abstract override` indica que el método sobreescribe el método abstracto `put` definido en la clase `IntQueue`. Realmente va a sobreescribir el método `put` concreto definido en la clase hija con la que se mezcla el *trait*. Cualquier clase hija de `IntQueue` tendrá que definir ese método, por haber sido declarado abstracto en la clase padre.
- El *trait* tiene una llamada `super` que invocará al método `put` definido en la clase que se mezcla con el *trait*.

Vamos a probarlo. Creamos una nueva clase `MiCola`:

```
class MiCola extends BasicIntQueue with Doubling
```

Lo probamos y vemos que el *trait* inserta en la cola el doble del parámetro:

```
val cola = new MiCola
cola.put(10)
cola.get()
```

```
/Int = 20/
```

La clase `MiCola` no define código nuevo, sólo mezcla una clase con un trait.

Se podría hacer el `new` directamente:

```
val cola = new BasicIntQueue with Doubling
cola.put(10)
cola.get()
/res2: Int = 20/
```

Modificaciones apilables

Podemos añadir nuevos trait que también sobreescriben el método y permiten implementar **modificaciones apilables**:

```
trait Incrementing extends IntQueue {
  abstract override def put(x:Int) = super.put(x + 1)
}
trait Filtering extends IntQueue {
  abstract override def put(x:Int) = if (x >= 0) super.put(x)
}
```

Ahora, podemos elegir aquellas modificaciones que queramos para una determinada cola. Por ejemplo, definimos una cola que filtra números negativos y añade uno a todos los números que almacena:

```
val cola = new BasicIntQueue with Incrementing with Filtering
cola.put(-1)
cola.put(0)
cola.put(1)
cola.get()
/res2: Int = 1/
cola.get()
/res3: Int = 2/
```

El orden de los traits es significativo. Van de **derecha a izquierda**. Cuando se llama a un método definido en varios traits se llama primero al método del trait más a la derecha. Si ese método a su vez llama a super, invocará al método del siguiente trait a su izquierda.

En el ejemplo anterior, primero se invoca al put de `Filtering`, de forma que se eliminan los enteros negativos y se añade uno a aquellos que no se han eliminado. Si invertimos el orden primero se incrementarán los enteros y entonces los enteros que todavía son negativos se descartarán:

```
val cola = new BasicIntQueue with Filtering with Incrementing
cola.put(-1)
cola.put(0)
cola.put(1)
cola.get()
/Int = 0/
cola.get()
/res3: Int = 1/
cola.get()
/res3: Int = 2/
```

4. Actores

Concurrencia en Java

- Concurrencia: sucede cuando distintos procesos se ejecutan simultáneamente y pueden interactuar unos con otros
- El modelo de Java para manejo de concurrencia está basado en threads (hilos de ejecución), mediante el bloqueo de datos compartidos
- Para usar este modelo, el programador decide qué datos se compartirán por los threads y los marca como synchronized
- El mecanismo de bloqueo asegura que sólo un hilo en un momento determinado entra en las secciones sincronizadas y tiene acceso a los datos compartidos
- Complicado crear aplicaciones robustas multi-hilos

Actores y paso de mensajes

- Scala proporciona una alternativa basada en un modelo de paso de mensajes, más fácil de programar ya que no hay datos compartidos (y por tanto no tenemos el problema de los interbloqueos ni de las condiciones de carrera)
- Un actor es una entidad similar a un thread que dispone de un buzón para recibir mensajes
- Para implementar un actor, se debe extender la clase scala.actors.Actor e implementar el método `act`

Ejemplo básico de un actor

```
import scala.actors._

class Perro extends Actor {
  def act() {
    for (i <- 1 to 5) {
      println("Guau!")
      Thread.sleep(1000)
    }
```

```
  }
}
```

- Activamos un actor invocando al método start proporcionado por la clase `Actor`
- Se activa un hilo que se ejecuta en paralelo al shell
- La llamada `Thread.sleep(1000)` provoca que el actor (hilo) se detenga durante 1 segundo

Ejecución:

```
scala> val p = new Perro
scala> p.start()
scala> Guau!
Guau!
Guau!
Guau!
Guau!
```

La ejecución de los actores es independiente de unos a otros también. Ejemplo:

```
import scala.actors._

class Gato extends Actor {
  def act() {
    for (i <- 1 to 5) {
      println("Miau!")
      Thread.sleep(1000)
    }
  }
}
```

Ejecutamos actores de ambas clases al mismo tiempo:

```
val a = new Perro
val b = new Gato

a.start; b.start
scala>  Miau!
Guau!
Miau!
Guau!
Miau!
Guau!
Miau!
Guau!
Guau!
Miau!
```

Creando actores con actor

Se pueden crear objetos actores sin haber definido ninguna clase con la función `actor` a la que se le pasa el código a ejecutar:

```
import scala.actors.Actor._

val seriousActor2 = actor {
  for (i <- 1 to 5) {
    println("Miau!")
    Thread.sleep(1000)
  }
}
```

Comunicación entre actores: mensajes

- Ya hemos creado actores que se ejecutan independientemente unos de otros.
- ¿Cómo pueden trabajar juntos? ¿Cómo se pueden comunicar sin compartir memoria y sin bloqueos?
- Los actores se comunican enviándose mensajes.
- Se puede enviar un mensaje usando el método `!`
- Para recibir los mensajes los actores deben llamar a la *función parcial* `receive` (implementa algo similar a un match)
- La llamada a `receive` deja el actor a la espera del siguiente mensaje que llega a su buzón

Ejemplo:

```
import scala.actors.Actor._

val echoActor = actor {
  while(true) {
    receive {
      case msg => println("Mensaje recibido: "+ msg)
    }
  }
}

echoActor ! "hola"
```

El envío de mensajes es asíncrono

- Cuando un actor envía un mensaje no se queda esperando el resultado, sino que continúa su proceso
- Esto se denomina comunicación asíncrona

- Una invocación a una función o a un método es síncrona: el flujo de ejecución del objeto llamador se detiene hasta que la función devuelve un resultado
- Los mensajes se almacenan en una cola en el actor receptor
- La recepción de los mensajes es *síncrona*: el `receive` se queda esperando a que llegue un mensaje al buzón

Mensajes con tipo

- Un actor puede esperar mensajes de un tipo en concreto, especificándolo en el tipo del `case`.

Por ejemplo, un actor que sólo espera mensajes de tipo `Int`:

```
import scala.actors.Actor._

val intActor = actor {
  receive {
      case x: Int => println("Tengo un Int: "+ x)
  }
}
```

Si recibe mensajes de otro tipo los ignora

```
scala> intActor ! "hola"
scala> intActor ! scala.math.Pi
scala> intActor ! 12
scala> Tengo un Int: 12
scala> intActor ! 30
// no devuelve nada, porque el actor ya ha ejecutado el único receive
```

Self

La variable `self` referencia el actor (hilo) actual. Un ejemplo de utilización para enviar mensajes al hilo actual:

```
import scala.actors.Actor._

def pruebaSelf() = {
    val caller = self
    for(i <- (1 to 4)) {
        println("Lanzando el actor " + i)
        actor {
            Thread.sleep(1000*i)
            println("Lanzando el mensaje")
            caller ! ("hola", i)
        }
```

```
    }
    println("Estoy escuchando")
    for(i <- 1 to 4) {
        receive {
            case (msg, v) => println(msg + ": " + v)
        }
    }
}
```

Explicamos rápidamente qué pasa cuando hacemos una llamada a `pruebaSelf()`:

1. Se guarda el hilo actual en la variable `caller`.
2. Se hace un bucle en el que se crean cuatro actores distintos. Cada actor se queda en espera durante i segundos, y cuando termina esa espera se lanza un mensaje al hilo inicial. El mensaje es una tupla formada por la cadena `"hola"` y el entero `i`.
3. Mientras tanto el hilo inicial sigue ejecutándose, escribe `"Estoy escuchando"` y se queda a la espera de recibir los mensajes anteriores.
4. Cuando se recibe cada mensaje se escribe en la salida estándar

El resultado de la salida es:

```
scala> pruebaSelf
Lanzando el actor 1
Lanzando el actor 2
Lanzando el actor 3
Lanzando el actor 4
Estoy escuchando
Lanzando el mensaje
hola: 1
Lanzando el mensaje
hola: 2
Lanzando el mensaje
hola: 3
Lanzando el mensaje
hola: 4
```

Ejemplo completo

- Definimos dos actores que intercambian una serie de mensajes y luego finalizan. El primer actor envía mensajes `ping` al segundo actor, que como respuesta envía mensajes `pong` (por cada mensaje `ping` recibido, un mensaje `pong`)
- El actor `ping` comienza el intercambio de mensajes. Cuando el actor `ping` ha enviado un determinado número de mensajes, envía un mensaje `Stop` al actor `pong`
- Ambos actores los definimos como subclases de `Actor` porque queremos pasarles argumentos a sus constructores
- Para terminar la ejecución de un actor invocamos a `exit()`

Código:

```
import scala.actors.Actor
import scala.actors.Actor._

class Ping(count: Int, pong: Actor) extends Actor {
  def act() {
    var pingsQuedan = count - 1
    println("Ping envía Ping")
    pong ! "Ping"
    while (true) {
      receive {
        case "Pong" =>
          if (pingsQuedan > 0) {
            println("Ping envía Ping")
            pong ! "Ping"
            pingsQuedan -= 1
          } else {
            println("Ping envía Stop")
            pong ! "Stop"
            println("Ping stops")
            exit()
          }
      }
    }
  }
}

class Pong extends Actor {
  def act() {
    while (true) {
      receive {
        case "Ping" =>
          println("Pong envía Pong")
          sender ! "Pong"
        case "Stop" =>
          println("Pong stops")
          exit()
      }
    }
  }
}

object PingPong extends App {
  val pong = new Pong
  val ping = new Ping(5, pong)
  ping.start
  pong.start
}
```

Lo probamos:

```
> scalac PingPong.scala
> scala PingPong
Pong: Pong
Ping: Ping
Pong: Pong
Ping: Ping
Pong: Pong
Ping: Ping
Pong: Pong
Ping: Ping
Pong: Pong
Ping: stop
Pong: stop
```

5. Otras características de Scala

Igualdad

Scala define los métodos `equals` y `==` para comparar la igualdad de dos valores. En las clases de Scala se definen uno en función del otro. En las clases definidas por nosotros debemos sobreescribir ambos métodos.

En Scala se definen dos subclases de `Any` : `AnyVal` y `AnyRef` . La clase `AnyVal` es la clase padre de nueve *clases valor*: `Byte` , `Short` , `Char` , `Int` , `Long` , `Float` , `Double` , `Boolean` y `Unit` . En todas ellas se utilizan *literales* y se crean objetos sin necesidad de hacer `new` . Por ejemplo, `42` es una instancia de `Int` o `false` es una instancia de `Boolean` . En todas estas clases (y en las que se define una semántica de *objeto valor*) se define el `equals` como una *igualdad de valor*: dos objetos son iguales sin tienen los mismos atributos.

La clase `AnyRef` es la clase base de todas las *clases referencia* en Scala. Cualquier clase que creamos nueva se define automáticamente como una subclase de `AnyRef` . En estas clases el funcionamiento por defecto del `equals` (y del `==`) es realizar una *comparación por referencia*.

Podemos hacer como en Java, redefinir el `equals` para una comparación de valor, y dejar el `==` para una comparación de referencia.

Si redefinimos el método `equals` también hay que redefinir el método `hashMap` para que si dos objetos tienen el mismo `hashMap()` deben también ser `equals` (al revés no es necesario).

Un ejemplo de cómo redefinir el `equals` en la clase `Rational` :

```
class Rational(n: Int, d: Int) {

    require(d != 0)

    private val g = gcd(n.abs, d.abs)
    val numer = (if (d < 0) -n else n) / g
    val denom = d.abs / g

    private def gcd(a: Int, b: Int): Int =
      if (b == 0) a else gcd(b, a % b)

    override def equals(other: Any): Boolean =
      other match {

        case that: Rational =>
          numer == that.numer &&
          denom == that.denom

        case _ => false
      }

    override def hashCode: Int =
      41 * (
        41 + numer
      ) + denom

    override def toString =
      if (denom == 1) numer.toString else numer +"/"+ denom
  }
```

El tipo `Option`

Scala define el tipo `Option[A]` para poder definir valores que pueden ser del tipo `A` o nulos. De esta forma las funciones siempre devuelven un valor, nunca se devuelve `null`. Los dos posibles valores de un tipo `Option` son `Some(<valor>)` y `None`.

Por ejemplo:

```
def devuelvoSiMayorQue0(x: Int): Option[Int] = {
    if (x>0) Some(x)
    else None
}

devuelvoSiMayorQue0(10) //=> Option[Int] = Some(10)
devuelvoSiMayorQue0(-10) //=> Option[Int] = None
```

El método `isDefined` de un `Option` devuelve un `Booleano` indicando si el opcional contiene un valor:

```
class Persona(val nombre: String)
val p1: Option[Persona] = Some(new Persona("Sara"))
p1.isDefined //=> Boolean = true
val p2: Option[Persona] = None
p2.isDefined //=> Boolean = false
```

Para recuperar el valor de un `Some(valor)`, si sabemos que contiene un valor, podemos usar el método `get`:

```
val o = devuelvoSiMayorQue0(x)
if (o.isDefined) {
   println(o.get) // Imprime el valor de x
}
```

También podemos procesar el valor que hay en el `Option` usando un `match`:

```
val a: Option[Int] = devuelvoSiMayorQue0(10)

a match {
    case Some(i) => {println(i)}
    case None => {println("No hay valor")}
}
```

Por último, podemos utilizar el método `a.getOrElse(valor)` que devuelve el valor del dato `a`, si lo tiene, o el valor que pasamos como parámetro en caso de que `a` sea `None`:

```
val a: Option[Int] = Some(20)
val b: Option[Int] = None
a.getOrElse(-1)
//=> 20
b.getOrElse(-1)
//=> -1
```