



UNIVERSIDAD
COMPLUTENSE
DE MADRID



ntic
master
School

Scala

Fundamentos Orientado a Objeto

Enero de 2023



Agenda

- **Clases**
- **Miembros de una Clase**
- **Operadores de notación Infix, Postfix y Prefix***
- **Comparación, Argumentos nombrados y por defecto**
- **Package e Imports**
- **Modificadores, Singleton y Companion Objects**
- **Predef**
- **Case Classes**



1 Classes



Orientado a objetos en Scala

- Clases y traits:
 - Los atributos determinan el estado de una instancia de la clase.
 - Los métodos dotan de operaciones (encapsulación).
 - Las funciones de acceso a los atributos determinan la visibilidad (protección de información).
- Los objetos Singleton son objetos de primera clase.
- Herencia
 - Herencia simple: sólo se extiende de una única superclase.
 - Múltiples traits pueden ser implementadas por una clase, mecanismo que emula “herencia múltiple”.

Define la clase más simple

```
scala> class Hello
defined class Hello

scala> val hello = new Hello
hello: Hello = Hello@6dd9d885

scala> hello.toString
res0: String = Hello@6dd9d885
```

- Se define con la palabra reservada **class**.
- Un objeto se crea haciendo uso de la palabra reservada **new**.
- **AnyRef** es la superclase de todas las clases si no se define una herencia explícitamente.

El constructor por defecto

```
scala> class Hello {  
  |   println("hello")  
  | }  
defined class Hello  
  
scala> val hello = new Hello  
hello  
hello: Hello = Hello@3d248797
```

- Todas las clases que se definen obtienen un constructor por defecto (o primario) de manera automática.
- La definición de la clase (nombre + atributos) determinan la firma del constructor.
- El cuerpo de la clase (salvo declaración de atributos y definición de métodos) es la implementación del constructor.

Caso de estudio

- Durante el curso se desarrollará una aplicación de Scala que para planificar viajes en tren: JourneyPlanner.
- Tendrá clases como Train, Stations, Times, JourneyPlanner y Schedules
- Los ejercicios serán propuestos según vayamos avanzando en el temario.

Ejercicio: Define una clase

- Crea una clase que se llame: Train.
- La clase debe estar en la carpeta src/main/scala.
- Una vez creada la clase, instancia objetos desde un worksheet.

Parámetros de clase (y del constructor)



```
scala> class Hello(message: String){
      | println(message)
      | }
defined class Hello

scala> val hello = new Hello("hello world")
hello world
hello: Hello = Hello@2c9c0e55

scala> new Hello
<console>:13: error: not enough arguments for constructor Hello: (message: String)Hello.
Unspecified value parameter message.
    new Hello
    ^
```

- Las clases pueden tener uno o más parámetros definidas con formato: ***name: type***.
- Los parámetros de las clases son a su vez parámetros del constructor primario.



Ejercicio: Define parámetros de clase



- A la clase Train, se le debe añadir un parámetro:
 - de nombre: ``number``
 - de tipo: ``Int``



2 Miembros de una clase

Constructores adicionales



```
scala> class Hello(message: String){
      |   def this()=this("hello")
      |   println(message)
      | }
defined class Hello

scala> new Hello
hello
res0: Hello = Hello@5df7d30d

scala> new Hello("Hello World!")
Hello World!
res1: Hello = Hello@5ad11dce
```

- El constructor primario es creado automáticamente.
- Un constructor adicional se crea con **def** seguido de **this**.
- Al final el constructor primario se debe llamar:
 - se asegura que todos los parámetros de la clase son inicializados.



Atributos inmutables



```
scala> class Hello {  
    | val message: String = "hello"  
    | }  
defined class Hello  
  
scala> val h = new Hello  
h: Hello = Hello@5a4c3dc8  
  
scala> h.message  
res2: String = hello  
  
scala> h.message = "hola mundo"  
<console>:12: error: reassignment to val  
      h.message = "hola mundo"  
        ^
```

- Los atributos son parte de la clase y cuyo valor determina el estado de la clase.
- Los atributos se definen como un valor inmutable, usando la palabra reservada **val**.
- Son públicos, de manera similar a como lo son en Python.



Atributos mutables



```
scala> class Hello {  
    | var message: String = "hola"  
    | }  
defined class Hello  
  
scala> val h = new Hello  
h: Hello = Hello@384c5d0b  
  
scala> h.message  
res3: String = hola  
  
scala> h.message = "hello world!"  
h.message: String = hello world!  
  
scala> h.message  
res4: String = hello world!
```

- Para definir un atributo mutable se debe usar la palabra reservada: **var**
- Se puede reasignar su valor una vez el objeto haya sido creado.



¿Mutables o inmutables?

- Scala nos permite elegir la herramienta apropiada que mejor encaje en nuestro caso de uso.
- También nos recomienda siempre a mantener la API inmutable:
 - Es más fácil pensar en un código que no tiene efectos secundarios.
 - Funciones más fáciles de testear.
 - La inmutabilidad de los objetos evita problemas de concurrencia.
- **Best practices:**
 - Preferencia de objetos inmutables (y valores inmutables).
 - Uso de *var* sólo si se tiene un caso específico que lo requiera.

Parámetros de clase != Atributos

```
scala> class Hello(message: String){  
  | println(message)  
  | }  
defined class Hello  
  
scala> val hello = new Hello("hello world")  
hello world  
hello: Hello = Hello@2c9c0e55  
  
scala> new Hello  
<console>:13: error: not enough arguments for constructor Hello: (message: String)Hello.  
Unspecified value parameter message.  
  new Hello  
    ^
```

- Los parámetros de clase son sólo parámetros del constructor.
 - Se pueden usar dentro del cuerpo de la clase.
 - No son accesibles desde fuera del cuerpo de la clase.

De parámetros de clase a atributos

```
scala> class Hello(val message: String)
defined class Hello

scala> val hello = new Hello("Hola Mundo!")
hello: Hello = Hello@5ffe7b9e

scala> hello.message
res5: String = Hola Mundo!

scala> class Hola(message: String)
defined class Hola

scala> val h = new Hola("hola")
h: Hola = Hola@d6cab9

scala> h.message
<console>:13: error: value message is not a member of Hola
    h.message
      ^
```

- Añadiendo **val** o **var** antes del nombre del parámetro de clase:
 - Crea un atributo de la clase.
 - Inicializa el atributo con el valor del parámetro.

Ejercicio: Promoción de parámetros de clase



- Desde un worksheet instancia un objeto de la clase Train e intenta acceder a su parámetro de clase `number`.
- Añade un parámetro de clase a la clase Train, pero que sea el primer parámetro:
 - nombre: `kind`
 - tipo: `String`
- Conviértelos en atributos inmutables.
- Repite el primer punto, crea un Train e intenta acceder a `number` y `kind`



Ejercicio: Promoción de parámetros de clase



- Crea la clase `Time` con dos parámetros de clase de tipo Int:
 - `hours`
 - `minutes`
- Conviértelos en atributos de la clase.
- Dentro del cuerpo del constructor de Time, añade comentarios de TODO:
 - TODO: verificar `hours` entre $0 < x < 23$
 - TODO: verificar `minutes` entre $0 < x < 59$



Ejercicio: Define un atributo

- Dentro de la clase ``Time`` se debe definir un atributo inmutable: ``asMinutes`` que debe devolver la representación de en minutos multiplicando su atributo ``hours``*60 y sumándole ``minutes``
 - ``hours``*60 + ``minutes``

Métodos (o funciones de una clase)



```
scala> def hello= "hello"  
hello: String  
  
scala> def echo(message: String): String = message  
echo: (message: String)String
```

- Los métodos son otros miembros de una clase que proveen de operaciones (o comportamiento).
- La palabra reservada **def** es usada para definir un método y le sigue:
 - Nombre.
 - Lista de parámetros (opcional).
 - El tipo del valor de retorno (opcional).
 - La expresión a evaluar (cuerpo del método) después del signo igual.



Ejercicio: Define un método

- A la clase ``Time`` se le debe añadir el método ``minus``:
 - con parámetro de entrada debe ser de tipo ``Time``
 - tipo de retorno ``Int``
 - el método debe devolver la diferencia entre ambas instancias de ``Time`` en minutos.

3 Operadores Infix, Postfix y Prefix

Operadores de notación Infix

```
scala> "Martin Odersky".split(" ")
res7: Array[String] = Array(Martin, Odersky)

scala> "Martin Odersky" split " "
res8: Array[String] = Array(Martin, Odersky)
```

- Los operadores son métodos que se usan en notación de operador.
- Notación de operador significa omitir puntos (.) y paréntesis (`(` `)`).
- Los métodos con un único parámetro se puede usar con la notación infija (infix).

Operadores de notación Postfix

```
scala> "Martin Odersky" split " " size  
warning: there was one feature warning; for details, enable `:setting -feature' or `:replay -feature'  
res9: Int = 2
```

- Los métodos que no esperan ningún parámetro pueden ser usados con la notación posfija (*postfix*)
- En general, se debe evitar el uso de la notación *postfix*.

Operadores de notación Prefix

```
scala> ! false  
res10: Boolean = true  
  
scala> false.unary_!  
res11: Boolean = true
```

- Los métodos llamados *unary_* seguido de +, -, !, o ~, se pueden usar con la notación prefija (*prefix*)

Convenciones de notación de operadores



- Usemos la notación infija para métodos con nombres de símbolos.
`1 + 1`
- Usemos la notación prefija para los métodos *unary_*.
`if (!true)`
- En los demás casos, siempre usemos la notación de punto.
`“Scala Mola”.split(“ ”).size`



Ejercicio: Define un operador

- A la clase `Time` añádele el método ``-``:
 - Funcionará como un alias de ``minus``: el cuerpo de ``-`` debe invocar a ``minus``

4 Comparación, argumentos nombrados y por defecto

Comparación (tediosa)



```
scala> 3 == 3
res3: Boolean = true

scala> new String("Scala") == new String("Scala")
res4: Boolean = true
```

- Se usa “==” para comprobar igualdad (y “!=” para la desigualdad)
- Se usan “==” y “!=” para comparar, si se quiere sobrescribir el método de comparación, se debe implementar el método “*equals*”.
- El tipo del argumento que espera la función “==”/*equals* es “*Any*”



Comparación (siempre tediosa)



```
scala> null == new String("Scala")
<console>:12: warning: comparing a fresh object using `==` will always yield false
      null == new String("Scala")
            ^
res5: Boolean = false

scala> new String("Scala") eq new String("Scala")
<console>:12: warning: comparing a fresh object using `eq` will always yield false
      new String("Scala") eq new String("Scala")
                        ^
res6: Boolean = false

scala> new String("Scala") == new String("Scala")
res7: Boolean = true

scala> null ne new String("Scala")
<console>:12: warning: comparing a fresh object using `ne` will always yield true
      null ne new String("Scala")
            ^
res8: Boolean = true

scala> new String("Scala") ne null
<console>:12: warning: comparing a fresh object using `ne` will always yield true
      new String("Scala") ne null
                        ^
res9: Boolean = true
```

- Se usa “eq” y “ne” para verificar identidad (referencias en memoria)
- Las comparaciones en Scala son seguras frente a los null (si sobreescribimos la función *equals*, debemos mantener esta funcionalidad).

Argumentos por defecto



```
scala> def name(first: String = "", last: String = ""): String = first + " " + last
name: (first: String, last: String)String

scala> name("Charles")
res10: String = "Charles "
```

- Permite hacer una llamada a la función sin parámetros.
- Nos permite omitir parámetros finales.



Argumentos nombrados



```
scala> name()
res11: String = " "

scala> name(last="Flores")
res12: String = " Flores"

scala> name(last="Flores", first="Charles")
res13: String = Charles Flores
```

- Nos permite omitir los parámetros iniciales cuando se le asigna valor a los argumentos finales por nombre.
- Siempre se pueden dar todos los argumentos por nombre y hacerlo en un orden distinto al definido en la firma.



Ejercicio: Argumentos por defecto



- Adapta el código para que los parámetros de la clase `Time` (`hours` y `minutes`) tengan el valor 0 como valor por defecto.
- Pruébalo desde un worksheet.



5 Packages e Imports

Packages



```
|package com.ntic.scala.training
```

- Los packages ayudan a tener el código organizado.
- **Best practice:** La estructura de ficheros debe reflejar la estructura de paquetes, esto es común la mayoría de lenguajes, Python también.



Imports



```
import com.ntic.scala.training.Hello  
import com.ntic.scala.training._
```

- Se usa Import si no se quiere referenciar a la clase con el nombre completo (paquete incluido)
- Se usa el guión bajo (_) para importar todos los miembros de un paquete.



Imports II



```
import scala.concurrent.{Future, Promise}
import java.sql.{Date => SqlDate}

def method: Hello = {
  import com.ntic.scala.training.Hello
  new Hello()
}
```

- Si se quiere importar más de un miembro de un package se usará {...}
- Se puede renombrar objetos importados
- Los métodos también pueden tener sus *imports* válidos sólo en el cuerpo del método.



Ejercicio: Uso de packages



- La estructura de directorios determina la jerarquía de paquetes
- creemos un paquete que se llame: ``com.ntic.clases.planner`` dentro de ``src/main/scala``
- Reubiquemos las clases ``Time`` y ``Train`` a ``com.ntic.clases.planner``:
 - se puede usar el asistente del IDE: “refactor > Move...”
 - o mover los ficheros dentro del paquete y luego añadir el indicador de paquetes
- Instancia la clase `Time` desde un worksheet



6

Modificadores, Singleton y Companion Objects

Modificadores de acceso



```
scala> class Hello {  
  | private val message = "Hello"  
  | }  
defined class Hello  
  
scala> class Welcome {  
  | protected val message = "Hello"  
  | }  
defined class Welcome
```

- Todos los atributos de una clase por defecto son públicos.
- Se usa `private` para restringir el acceso, sólo será accesible dentro (en el cuerpo) de la clase.
- Con `protected` el atributo será accesible dentro de la clase y desde sus subclases (todas las clases que la extiendan).



Modificadores de acceso condicionados



```
package hello
class Hello {
  private[hello] val message = "Hello!"
}
class Hello {
  private[this] val message = "Hello!"
  def messageEqual(that: Hello): Boolean = this.message == that.message
}
```

- Se relaja el acceso a un atributo dependiendo de un cualificador.
- Se usa *this* como cualificador, si queremos hacer acceso ese miembro sólo a la instancia.



Singleton Objects



```
scala> object Hello {  
  |   def message = "Hello!"  
  | }  
defined object Hello  
  
scala> Hello.message  
res0: String = Hello!
```

- Se usa la palabra reservada ***object***.
- Sólo se puede crear una instancia de la clase.
- Se accede al objeto singleton usando su nombre, no se almacena la referencia en un *val* o *var*.



Singleton Objects: ¿para qué sirven?



- Casos de uso:
 - Almacenamiento de constantes.
 - Métodos factoría de una clase. (Companion objects)
 - Almacenamiento de funciones de utilidades, no dependen del estado de un objeto.
 - Definir aplicaciones.



Aplicaciones y su método Main



```
object Hello {  
  def main (args: Array[String]): Unit =  
    println("Hello!")  
}
```

- Una aplicación en Scala se define con un singleton object que tenga la función **main** implementada.



Lanzar una aplicación



```
→ exercises scala -cp target/scala-2.13/classes/ com.ntic.scala.training.Hello  
Hello!
```

- La aplicación se puede ejecutar desde la línea de comandos y usando, como se puede observar, el nombre completo de la clase.
- Alternativa, desde el IDE también podemos ejecutar nuestro código y nuestra aplicación.



Companion Objects



```
object Hello {  
  private val defaultMessage = "Hello!"  
}  
  
class Hello(message: String = Hello.defaultMessage) {  
  println(message)  
}
```

- Si un singleton object y una clase o trait comparten el nombre y están en el mismo package y fichero, se les denomina **companions**.
- Los companion tiene completo acceso a los atributos privados de la clase o trait a la que acompañan.



Ejercicio: Define un companion object



- Crea un companion object para `Time`
- Crea un método `fromMinutes` que tome como parámetro `minutes` de tipo `Int` y devuelva una instancia de `Time`:
 - Creará una instancia de `Time` que tendrá normalizadas los valores para `hours` y `minutes`: `minutes` entre 0 y 59 y `hours` entre 0 y 23
- Ponlo a prueba desde el worksheet



7 Predef

Conociendo Predef



```
scala> require(1 == 2, "raruno")
java.lang.IllegalArgumentException: requirement failed: raruno
    at scala.Predef$.require(Predef.scala:281)
    ... 31 elided

scala> █
```

- Existe un singleton object de la librería estándar de Scala llamada *Predef*.
- Todos los miembros de *Predef* son importados de manera automática.
- Un ejemplo: *require*, método usado para establecer precondiciones.



Ejercicio: Verifica precondiciones



- En los TODO de `Time`, aplica `require` para asegurarnos que `hours` y `minutes` se establecen con valores válidos.



8 Case Classes

Case Classes



```
scala> case class Time(hours: Int = 0, minutes: Int = 0)
defined class Time

scala> val time = Time(10)
time: Time = Time(10,0)

scala> Time(15)
res2: Time = Time(15,0)

scala> Time.apply(6)
res3: Time = Time(6,0)
```

- La palabra clave **case** añade funcionalidades extras a una clase:
 - Se puede instanciar una clase sin usar la palabra **new**
- Cada *case class* tiene un companion object con un método *apply* definido.
- En este contexto, *apply* se usa como método factoría.



Apply y su azúcar sintáctico



```
scala> object Reverse{  
  |   def apply(s: String): String = s.reverse  
  | }  
defined object Reverse  
  
scala> Reverse("hello")  
res4: String = olleh
```

- La invocación de apply funciona para cualquier objeto, no sólo como método factoría de clases.
- Es muy común su uso en Scala, como veremos con las colecciones y funciones.



Case classes y sus features extras



```
scala> time
res5: Time = Time(10,0)

scala> time == Time(10)
res6: Boolean = true

scala> time == Time(9, 10)
res7: Boolean = false

scala> time.hours
res8: Int = 10
```

- El compilador crea las implementaciones de las funciones *toString*, *equals* y *hashCode* automáticamente.
- Los parámetros de clase son convertidos a atributos inmutables de manera transparente al desarrollador.



Case classes y sus otras features extras



```
scala> time.hours  
res8: Int = 10  
  
scala> time.copy(minutes = 15)  
res9: Time = Time(10,15)
```

- El método *copy* es dado ya implementado.
- Uso de case classes en el *pattern matching*.



Ejercicio: Define case classes



- Convierte en `case class` a `Time` y `Train`
 - Elimina `val` de los parámetros de clase, incluso si no molesta.
 - Elimina `new` de `Time.fromMinutes`, incluso si no molesta.
- Pruébalo desde un worksheet.

