UNIVERSIDAD COMPLUTENSE DE MADRID

ntic master School

Scala

Herencia

Noviembre de 2023

Agenda

- **Herencia: conceptos básicos**
- **Clases abstractas y sus miembros**
- **Traits**

1 Herencia: Conceptos básicos

Conceptos básicos

- La herencia en Scala nos permite:
  - reutilización de código.
  - especialización.
- Cada clase (a excepción de *Any*) puede tener como mucho una superclase
  - Los miembros (atributos) no privados (*private*) son heredados
  - Los miembros pueden ser sobrescritos cuando no sean declarados como constantes (*final*)
- Polimorfismo de los subtipos:
  - El *polimorfismo* indica que se puede tener múltiples formas.
  - El tipo de la subclase se ajusta al tipo de la superclase (la subclase es la superclase)

Subclases


```
class Animal
class Bird extends Animal
```

- *extends* marca una clase como subclase de la clase que extiende.
- *AnyRef* es la superclase de todas las clases que no extiendan explícitamente de otra (*extends*)
- Sólo se puede extender explícitamente una clase

Implementación de Herencia


```scala
class Animal {
 def eat(): Unit = println("yum yum!")
 private def secret(): Unit = println("top secret")
}

class Bird extends Animal {
 eat()     // <- Heredado
 secret()  // <- No heredado
}
```

- Todos los miembros de la clase que no sean privados (*private*) son heredados.
- El compilador nos devolvería un error: *not found: value secret*.

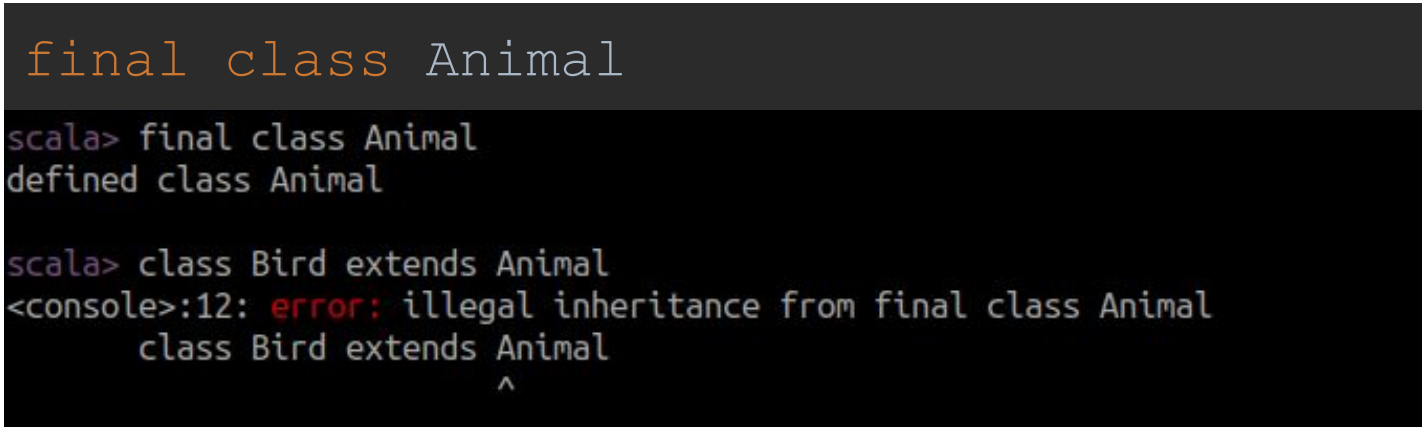
Constructor de la superclase

```
class Animal (val name: String)

class Bird(name: String) extends Animal(name)
```

```
scala> class Animal (val name: String)
defined class Animal

scala> class Bird(name: String) extends Animal
<console>:12: error: not enough arguments for constructor Animal: (name: String)Animal.
Unspecified value parameter name.
       class Bird(name: String) extends Animal
                                        ^
```

- La subclase debe invocar al constructor de la superclase tras *extends*.
- Todos los parámetros de clase deben ser inicializados.
- La superclase siempre se inicializa primero.

Evitando la herencia



- Si se quiere evitar que una clase sea extendida y se convierta en superclase, se debe usar el cualificador *final* antes de la definición de la clase.

Limitando la herencia


```
sealed class Animal
class Bird extends Animal
final class Fish extends Animal
```

- Las clases sealed sólo pueden ser extendidas en el mismo fichero fuente.
- Mecanismo de creación de ADT (*algebraic data type*).
- Las clases sealed restringen el número de posibles subclases.
- Ejemplo: *Option* con *Some* y *None*

Overriding


```
class Animal {
 def eat(): Unit = println("yum yum!")
}

class Bird extends Animal {
 override def eat(): Unit = println("beep!")
}
```

- La palabra reservada *override* determina el la sobrescritura del miembro:
  - Redefinición de la implementación de un método.
- Todos los miembros que no sean final pueden ser sobrescritos.
- Si la superclase tiene algún miembro con final, ésta no se puede sobrescribir o redefinir.

Acceso a miembros de la superclase


```
class Animal {
 def eat(): Unit = println("yum yum!")
}

class Bird extends Animal {
 override def eat(): Unit = {
   super.eat()
   println("beep!")
 }
}
```

- Se puede acceder a los miembros de la superclase accediendo a su referencia con la palabra clave: *super*

Principio de acceso uniforme


```
class Animal {
 def name: String = "Pepe"
 // val name: String = "Pepe"
}
```

- Recordatorio:
  - Principio de acceso uniforme: Desde el cliente el acceso a una “propiedad” se hace de la misma manera si la propiedad almacena el valor o el valor es resultado de un cálculo.
- Las propiedades pueden ser definidas usando *def* sin paréntesis o *val*.

Sobrescribir *def*s con *val*s


```
class Animal {
 def name: String =
scala.util.Random.shuffle("Pepe".toSeq).mkString
}

class Bird extends Animal {
 override val name = "Jose"
}
```

- Un *val* es estable no cambia su valor, pero un *def* sin parámetros podría devolver distintos resultados en diferentes llamadas.

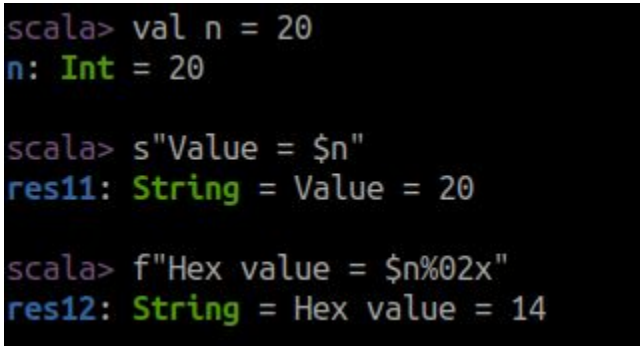
- Se puede decidir convertir a estable y sobrescribir un *def* con un *val*.

Valores perezosos

```
scala> lazy val lazySeven = {
     |     println("soy muy perezoso :)")
     |     7
     |   }
lazySeven: Int = <lazy>

scala> lazySeven
soy muy perezoso :)
res7: Int = 7
```

- Todos los atributos val se inicializan durante la construcción del objeto.
- Usando la palabra reservada lazy se posterga la inicialización del atributo hasta la primera vez que sea usado o invocado.

Interpolación de Strings



- Desde Scala 2.10 se dispone de esta feature.
- Anteponiendo *s* al String permite incluir expresiones usando *$id* (valor) o *${expr}* (expresión).
- Anteponiendo *f* al String permite dar formato a resultados de expresiones.

Ejercicio: override toString

- La función `toString` ya imprime en un formato legible la case class `Time`, pero se puede mejorar y que devuelva algo como: `10:30`
    - sobreescribe `toString` cualificándolo como `lazy val`
    - usa el formato `%02d` para `hours` y `minutes`
        - %02d hace que los enteros tengan 2 dígitos completando con 0 a la izquierda

2 Clases abstractas y sus miembros

Clases Abstractas


```
abstract class Animal {
 def name: String
 def eat(): Unit
}
```

- Las clases abstractas no pueden ser instanciadas
- Los miembros de estas clases definen únicamente la firma.
- Las clases abstractas pueden contener una mezcla de miembros concretos (implementados) o abstractos

Miembros abstractos


```
class Bird extends Animal{
 def name: String = "Pepe"
 override def eat(): Unit = println("beep!")
}
```

- La subclase debe implementar los miembros abstractos de la superclase.
- La palabra reservada *override* es opcional.
- ¿Se recomienda usar override? Depende:
  - Muy útil para indicar que un atributo viene declarado en la superclase.
  - Errores de compilación si se hace un refactor.

Ejercicio: Define un ADT

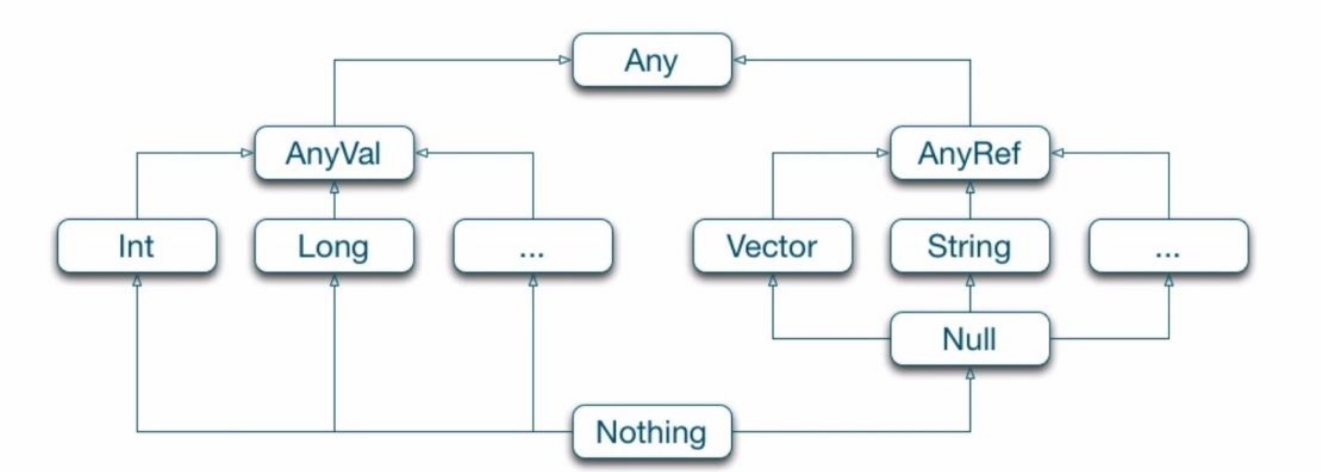
- Crea una clase `sealed abstract TrainInfo`
  - Define un método abstracto `number` que devuelva `Int`

- Crea las case clases `InterCityExpress`, `RegionalExpress` y `BavarianRegional` que extiendan `TrainInfo`
  - declara el parámetro de clase `hasWifi` de tipo `Boolean` con valor por defecto `false` en `InterCityExpress`

- Refactoriza `Train.kind` y `Train.number` con un parámetro de clase que se llame `info` de tipo `TrainInfo`

3 Traits

Jerarquía de tipos en Scala



- *AnyVal* es para los tipo de valor y *AnyRef* para los tipos de referencia.
- *Null* es el tipo límite por abajo para los tipos de referencia.
- *Nothing* es el límite por abajo genera.

Traits: síntesis

- La JVM sólo permite la herencia de una única clase:
  - Evita problemas de herencia múltiple: ambigüedad, complejidad, etc.,
  - Limitadas opciones: no hay herencia múltiple.

- Scala añade los traits para saltarse esta limitación:
  - Se puede heredar sólo una superclass.
  - Pero se puede mezclar (**mix-in**) múltiples *traits*.

¿Cuándo usar Traits?


```
abstract class Animal

class Bird extends Animal {
 def fly: String = "I am flying!"
}

class Fish extends Animal {
 def swim: String = "I am swimming!"
}

class Duck // no puede heredar de Bird y Fish
```

- Si los patos y peces pueden nadar, estaría bien que se herede el método swim y no externalizar o incluso duplicar código.

Creando un *trait*


```
trait Swimmer {
 def swim: String = "I am swimming!"
}
```

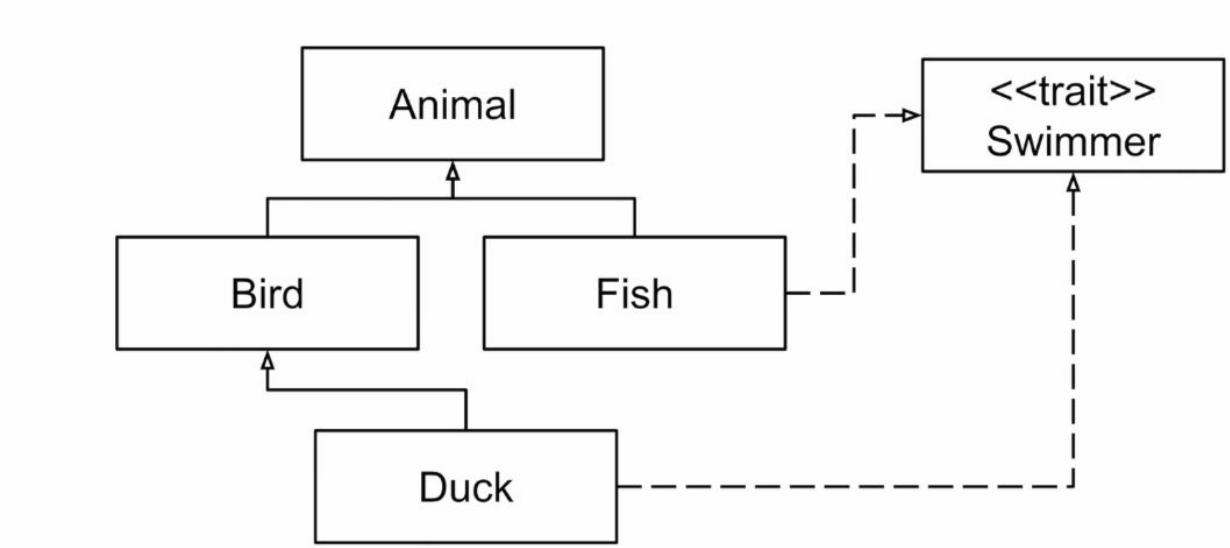
- El trait Swimmer contiene el método *swim*.

Combinando traits y clases


```
abstract class Animal

class Bird extends Animal {
 def fly: String = "I am flying!"
}

trait Swimmer {
 def swim: String = "I am swimming!"
}

class Fish extends Animal with Swimmer
class Duck extends Animal with Swimmer
```

- Combinar clase y trait usando la palabra reservada: *trait*

Combinando traits y clases II



- Fish y Duck ya pueden nadar (*swim*) usando el trait *Swimmer*.
- Y Duck a su vez puede volar (*fly*) mediante la clase *Bird*.

Características de los Traits


```
class Mobility

trait Swimmer extends Mobility {
 def swim: String = "I am swimming!"
 def float: String
}
```

- Los traits pueden contener miembros abstractos o concretos, sin o con implementación.
- Los traits son abstractos y no tienen el concepto de parámetros de clases (ni de traits).
- Al igual que las clases, los traits sólo pueden extender de una única superclase.

with vs extends I


```
trait Swimmer
trait Flyer
class Fish extends Swimmer
class Duck extends Swimmer with Flyer
```

- extends permite combinar el primer trait
- extends permite crear la subclase si lo que sigue es una clase o combinar con el primer trait
- with se usa con los siguientes traits

Reglas: Jerarquía de herencia


```
scala> class Omnivore; class Mobility
defined class Omnivore
defined class Mobility

scala> trait Swimmer extends Mobility
defined trait Swimmer

scala> class Duck extends Omnivore
defined class Duck

scala> class Duck extends Omnivore with Swimmer
<console>:13: error: illegal inheritance; superclass Omnivore
 is not a subclass of the superclass Mobility
 of the mixin trait Swimmer
       class Duck extends Omnivore with Swimmer
                                        ^
```

- Al combinar una clase con un trait, también se extiende de la superclase del trait.
- Los traits deben respetar la jerarquía de clases: una subclase no debe extender de dos superclases incompatibles.

Reglas: Miembros concretos


```
scala> trait Swimmer { def move = "swimmig" }
defined trait Swimmer

scala> trait Flyer { def move = "flying" }
defined trait Flyer

scala> class Duck extends Swimmer with Flyer
<console>:13: error: class Duck inherits conflicting members:
  method move in trait Swimmer of type => String  and
  method move in trait Flyer of type => String
(Note: this can be resolved by declaring an override in class Duck.)
       class Duck extends Swimmer with Flyer
             ^
```

- Si los traits que extendemos definen el mismo miembro, es obligatorio el uso de *override*.
- En el ejemplo: para poder usar override en Swimmer o Flyer, alguno de los traits debe extender el otro o crear un padre común con un miembro abstracto

Ejercicio: Usa un trait.

- Haz que `Time` extienda el trait `Ordered`:
  - `Ordered` es parte de la librería estándar de Scala; hay que revisarlo en las API docs
  - `Ordered` define operadores como <; <=; >
  - `Ordered` también declara un método abstracto que se debe definir e implementar por las clases que le extiendan.
  - `Ordered` da funcionalidad extra a `Time` y permitirá hacer comparaciones entre objetos de la clase `Time`.