



Módulo Profesional: Big Data Aplicado

Scala

Herencia

```
object Herencia extends App {  
  
  class Animal {  
    def comer = println("comer comer")  
  }  
  
  class Gato extends Animal  
  
  val gato = new Gato //instancia de la clase Cat  
  gato.comer //devuelve comer comer  
}
```

PROBLEMS

12

OUTPUT

DEBUG CONSOLE

TERMINAL

PORTS

comer comer

Herencia: Métodos Private

```
object Herencia extends App {  
  
  class Animal {  
    private def comer = println("comer comer")  
  }  
  
  class Gato extends Animal  
  
  val gato = new Gato //instancia de la clase Cat  
  gato.comer  
}
```

Cuando un método o una variable se declara como **private**, su visibilidad está restringida a la clase en la que se define.

Herencia: Métodos Protected

```
object Herencia extends App {  
  
  class Animal {  
    protected def comer = println("comer comer")  
  }  
  
  class Gato extends Animal {  
    def comerGato = {  
      comer //el método protegido sí se será accesible desde la subclase, pero no es accesible fuera de la clase  
      println("comida gato")  
    }  
  }  
  
  val gato = new Gato //instancia de la clase Cat  
  //gato.comer  
  gato.comerGato  
}
```

Cuando método o una variable se declara como **protected**, su visibilidad está limitada a la clase en la que se declara o las subclase que heredan esta clase.

Herencia

Acceso	Visibilidad	Ejemplo
private	Sólo dentro de la clase Animal	<code>private def comer= println("comer comer")</code>
protected	Dentro de Animal y su subclases (Gato)	<code>protected def comer= println("comer comer")</code>
Por defecto	Público accesible	<code>def comer =println("comer comer")</code>

Herencia

```
//constructores  
class Persona (nombre: String, edad: Int)  
class Adulto (nombre: String, edad: Int, carnetConducir: String) extends Persona
```

¿Por qué da error?

Herencia

```
//constructores  
class Persona (nombre: String, edad: Int)  
class Adulto (nombre: String, edad: Int, carnetConducir: String) extends Persona (nombre, edad)
```

Porque lo primero que se realiza es crear la instancia de la clase Persona que necesita dos parámetros nombre y edad.

Herencia: Override

```
//overriding --anulando
class Perro extends Animal {
  ↑ comer
  | override def comer = println("comer perro")
}
val perro = new Perro
perro.comer
```

Override se utiliza para anular un método con el mismo nombre de la clase padre.

La subclase puede necesitar implementar el método reemplazando el comportamiento definido en la clase padre.

Herencia: Override

```
class Animal {  
  val TipoCriatura = "Salvaje"  
  protected def comer = println("comer comer")  
}
```

```
//overriding --anulando  
class Perro extends Animal {  
  ↑ TipoCriatura  
  override val TipoCriatura = "Domestica"  
  ↑ comer  
  override def comer = println("comer perro")  
}
```

```
println(perro.TipoCriatura)
```

→ ¿Qué imprime por pantalla?

Herencia: Override

```
↑ TipoCriatura
class Perro (override val TipoCriatura: String) extends Animal {
  ↑ comer
  | override def comer = println("comer perro")
}

val perro = new Perro ("Indiferente")
println(perro.TipoCriatura)
```

Herencia: Polimorfismo de subtipos

```
//tipo sustitución: polimorfismo
class Animal {
    val TipoCriatura = "Salvaje"
    def comer = println("comer comer")
}

↑ TipoCriatura
class Perro (override val TipoCriatura: String) extends Animal {
    ↑ comer
    override def comer = println("comer perro")
}

val esAnimal: Animal = new Perro ("Ni idea")
esAnimal.comer
```

El polimorfismo de subtipos se basa en la herencia y permite trabajar con tipos derivados a través de una referencia al tipo base

¿Qué imprime por pantalla esAnimal.comer?

Herencia: Super

```
//super

class Animal {
    val TipoCriatura = "Salvaje"
    def comer = println("comer comer")
}
↑ TipoCriatura
class Perro2 (override val TipoCriatura: String) extends Animal {
    ↑ comer
    override def comer = {
        super.comer
        println ("comer perro 2")
    }
}
val perro2 = new Perro2 ("domestica")
perro2.comer
```

Esto asegura que el comportamiento original del método de la clase base se ejecute antes (o después, dependiendo del orden) de añadir el nuevo comportamiento en la clase derivada.

Herencia: Impedir la anulación

```
//impedir la anulación
//1- utilizar la palabra reservada final
class Animal {
    val TipoCriatura = "Salvaje"
    final def comer = println("comer comer")
}


class Perro (override val TipoCriatura: String) extends Animal {
    override def comer = {
        super.comer
        println("comer Perro")
    }
}
```

**ERROR PORQUE
NO PERMITE
QUE SEA
SOBREESCRITO**

Herencia: Impedir la anulación

```
//2- utilizar final en toda la clase
final class Animal {
    val TipoCriatura = "Salvaje"
    def comer = println("comer comer")
}

class Perro (override val TipoCriatura: String) extends Animal {
    override def comer = {
        super.comer
        println("comer Perro")
    }
}
```



FINAL:
impide que una clase
o un miembro
(método o campo)
sea sobrescrito o
extendido.

Herencia

//3- sellar la clase = extender clases en este fichero, para prevenir la extensión en otros ficheros

```
sealed class Animal {  
    val tipoCriatura = "Salvaje"  
    def comer = println ("comer comer")  
}
```

El compilador sabe todas las posibles subclases de una clase sealed porque están en el mismo archivo

```
↑ TipoCriatura  
class Perro (override val TipoCriatura: String) extends Animal {  
    ↑ comer  
    override def comer = {  
        super.comer  
        println("comer Perro")  
    }  
}
```


Ejercicio

```
/*
```

Contexto:

Se necesita modelar distintos tipos de vehículos utilizando herencia en Scala. La clase base debe definir un comportamiento y un atributo predeterminados, mientras que las clases hijas sobrescribirán estos elementos para agregar comportamientos específicos.

Clase Vehiculo

Crea una clase sellada llamada Vehiculo para restringir su extensión a un solo archivo.

Atributo: val TipoVehiculo con valor predeterminado "General".

Método de conducir, que imprima "Conduciendo vehículo".

Clase Hija: Coche

Crea una clase Coche que extienda la clase Vehiculo.

Sobrescribe el atributo TipoVehiculo para que sea personalizado al crear una instancia.

Sobrescribe el método conducir de la clase padre. El nuevo método debe:

Llamar al método original conducir de la clase padre utilizando super.conducir.

Imprimir "Conduciendo coche".

Llamadas:

Crea una instancia de la clase Coche indicando el tipo de vehículo (por ejemplo, "Sedán").

Imprime el valor de TipoVehiculo de la instancia.

Llama al método conducir para verificar el comportamiento sobrescrito.

```
*/
```