



Herencia y Polimorfismo en Scala

Herencia y Polimorfismo en Scala

Objetivos del tema:

- Conocer qué es la herencia y el polimorfismo.
- Conocer el aporte del polimorfismo para reutilizar código existente.
- Conocer que el polimorfismo es extensible a las funciones.



1 Herencia



Herencia

La herencia es una de las características principales de la programación orientada a objetos.

Es el mecanismo por el cual una clase hereda características y atributos de otra.

Esto permite poder definir clases que basándose en otras ya existentes.

Si una clase deriva de otra, hereda sus atributos y métodos, pero también puede añadir atributos nuevos, métodos y redefinir métodos que haya heredado. A esto se le llama especialización.



Herencia

Una clase que se hereda se denomina superclase.

La clase que hereda se llama subclase.

Una subclase es una versión especializada de una superclase, ya que hereda atributos y métodos de la superclase y añade sus elementos propios.

Con la herencia nos permite reutilizar código general y centrarnos en la especialización (particularidades) de cada clase.

Para definir que una clase hereda de otra se usa la palabra clave 'extends '



Herencia

En el ejemplo definimos dos clases, una superclase Person y una subclase Student, en el ejemplo veremos que instanciamos dos objetos de tipo Student y creamos una lista de Person con las instancias de Student:

```
class Person{
  var SSN:String="999-32-7869"
}

class Student extends Person {
  var enrolmentNumber:String="0812CS141028"
  println("SSN: "+SSN)
  println("Enrolment Number: "+enrolmentNumber)
}
```

Vemos que la clase Student extiende (hereda) de la clase Person, eso quiere decir que cualquier instancia de Student, es también una instancia de Person.

```
scala> class Person{
  |   var SSN:String="999-32-7869"
  | }
class Person

scala> class Student extends Person {
  |   var enrolmentNumber:String="0812CS141028"
  |   println("SSN: "+SSN)
  |   println("Enrolment Number: "+enrolmentNumber)
  | }
class Student

scala> val st = new Student
SSN: 999-32-7869
Enrolment Number: 0812CS141028
val st: Student = Student@28bdbc88

scala> val st2 = new Student
SSN: 999-32-7869
Enrolment Number: 0812CS141028
val st2: Student = Student@3a17b2e3

scala> val lista: Seq[Person] = Seq(st, st2)
val lista: Seq[Person] = List(Student@28bdbc88, Student@3a17b2e3)

scala> □
```



Herencia

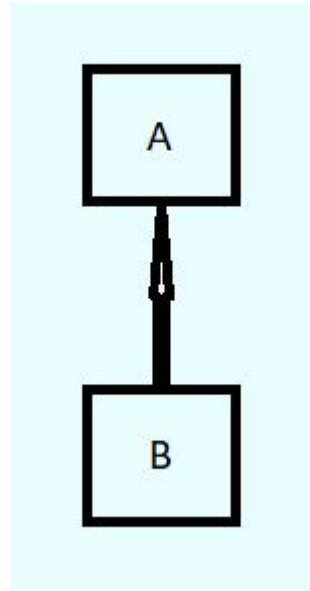
En Scala se soporta 5 tipos de herencia:

- Herencia simple:
 - Cuando una clase hereda de otra clase, como en el ejemplo anterior.
- Herencia múltiple:
 - En general, esto sucede cuando una clase hereda de múltiples clases base, es un caso de múltiples herencias.
 - En Scala una clase sólo puede heredar (extends) de una única clase, pero la herencia múltiple se consigue con traits, esto en scala se conoce como: mixins, cuando una clase u objeto extiende de una clase, pero implementa varios traits.
- Herencia jerárquica:
 - Cuando más de una clase hereda de una clase base, se dice que es una herencia jerárquica.
- Herencia híbrida:
 - Cuando existe una combinación de al menos dos tipos de herencia.



Herencia Simple

B extiende o hereda de A



Herencia Simple

Herencia multinivel

- En el ejemplo vemos que la clase A es extendida por B que a su vez, es extendida por C.
- Cuando se instancia C, se ejecuta las instrucción de A, B y C, en ese orden, debido a la precedencia de clases, ya que antes de poder instanciar C, se debe instanciar A

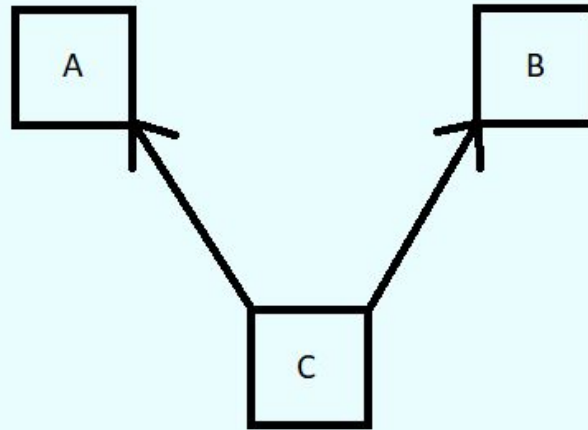
```
class A{  
    println("A")  
}  
class B extends A{  
    println("B")  
}  
class C extends B{  
    println("C")  
}
```

```
scala> class A{  
|   println("A")  
| }  
| class B extends A{  
|   println("B")  
| }  
| class C extends B{  
|   println("C")  
| }  
|  
class A  
class B  
class C  
  
scala> new C()  
A  
B  
C  
val res0: C = C@60a01cb  
scala> 
```



Herencia Múltiple

La clase C extiende A y B



Herencia Múltiple

Herencia múltiple, como se ha comentado, la herencia múltiple se consigue conjugando Clases con Traits:

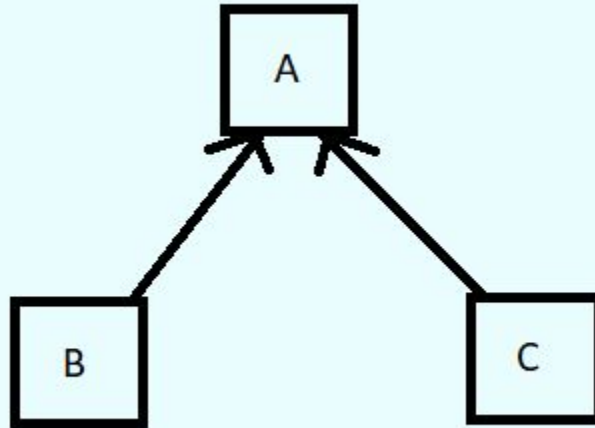
```
trait A{
  var length:Int= _
  def action={
    length += 1
  }
}
trait B{
  var height:Int = _
  def action={
    height += 1
  }
}
class C extends A with B{
  length=3;
  height+=6;
  override def action={
    super[A].action
    super[B].action
  }
}
var c=new C
c.action
// imprimirá por pantalla 7
println(c.height)
// imprimirá por pantalla 4
println(c.length)
```

```
scala> trait A{
  |   var length:Int= _
  |   def action={
  |     length += 1
  |   }
  | }
  | trait A
scala> trait B{
  |   var height:Int = _
  |   def action={
  |     height += 1
  |   }
  | }
  | trait B
scala> class C extends A with B{
  |   length=3;
  |   height+=6;
  |   override def action={
  |     super[A].action
  |     super[B].action
  |   }
  | }
  | class C
scala> var c=new C
var c: C = C@58835bba
scala> c.action
scala> println(c.height)
7
scala> println(c.length)
4
scala>
```



Herencia Jerárquica

Las clases B y C heredan de la clase A



Herencia Jerárquica

Herencia jerárquica, cuando más de una clase heredan de una misma superclase.

Aunque en Scala una clase sólo puede extender de una única clase. Una clase puede ser extendida por un número arbitrario de clases.

```
class A{
  println("A")
}
class B extends A{
  println("B")
}
class C extends A{
  println("C")
}

// imprimirá por pantalla
// A
// B
new B()

// imprimirá por pantalla
// A
// B
new C()
```

```
scala> class A{
|   println("A")
| }
| class B extends A{
|   println("B")
| }
| class C extends A{
|   println("C")
| }
class A
class B
class C

scala> new B()
A
B
val res0: B = B@109a2025

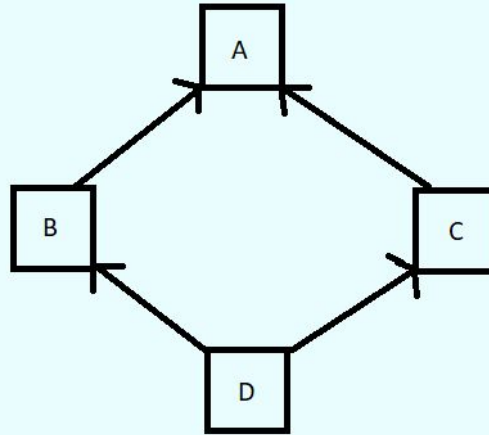
scala> new C()
A
C
val res1: C = C@516462cc

scala> 
```



Herencia Híbrida

D extiende B y C (herencia múltiple) y a su vez, B y C extienden A (herencia jerárquica)



2 Polimorfismo



Polimorfismo

El polimorfismo es un concepto clave en la Programación Orientación a Objetos.

El polimorfismo consiste en proporcionar una única interfaz a entidades de diferentes tipos o en utilizar un único símbolo para representar múltiples tipos diferentes.

En Scala, el polimorfismo significa que una variable o propiedad local `val` o `var` puede realmente hacer **referencia** a una instancia de un **tipo particular o cualquier subtipo de ese tipo**.



Polimorfismo

En el siguiente ejemplo, definimos una clase abstracta: Animal y dos clases que la extienden: Gato y Perro. Vemos que la superclase, define una función hacerRuido que no implementa, pero sí las clases que la extienden.

```
abstract class Animal(val especie: String, val nombre: String) {
  def hacerRuido: Unit
  def imprime(): Unit = {
    println(s"$nombre es un $especie y suena:")
    hacerRuido
  }
}

class Gato(n: String) extends Animal(especie="gato", nombre=n) {
  def hacerRuido(): Unit = println(s"meow")
}

class Perro(n: String) extends Animal(especie="perro", nombre=n) {
  def hacerRuido(): Unit = println(s"woof")
}
```

```
scala> abstract class Animal(val especie: String, val nombre: String) {
  |   def hacerRuido: Unit
  |   def imprime(): Unit = {
  |     println(s"$nombre es un $especie y suena:")
  |     hacerRuido
  |   }
  | }
  | }
class Animal

scala> class Gato(n: String) extends Animal(especie="gato", nombre=n) {
  |   def hacerRuido(): Unit = println(s"$nombre -> meow")
  | }
class Gato

scala> class Perro(n: String) extends Animal(especie="perro", nombre=n) {
  |   override def imprime(): Unit = {
  |     println(s"mi nombre es $nombre y soy un perro")
  |     println("woof! woof! woof!")
  |   }
  |   def hacerRuido(): Unit = println(s"$nombre -> woof")
  | }
class Perro

scala> var animal: Animal = new Perro("doggy")
var animal: Animal = Perro@7238072e

scala> animal.hacerRuido
doggy -> woof

scala> animal.imprime
mi nombre es doggy y soy un perro
woof! woof! woof!

scala> animal = new Gato("kitty")
// mutated animal

scala> animal.hacerRuido
kitty -> meow

scala> animal.imprime
kitty es un gato y suena:
kitty -> meow

scala>
```



Polimorfismo

Siguiendo con ejemplo anterior: el tipo de la variable `var animal` es `Animal`.

Eso quiere decir que puede hacer **referencia a `Animal`, `Gato` o `Perro`** o cualquier otra clase que extienda de `Animal`.

Primero almacenamos en la variable `animal` una referencia a `Perro` y se llama a la funciones de la variable `animal`: `hacerRuido` y luego a `imprime`.

Ambas funciones están disponibles en la instancia que referencia `animal` ya que están definidas en la clase `Animal` e implementadas en `Perro`. Si hubiera una función definida en la clase `Perro` que no se haya definido en `Animal`, no sería visible via `animal` (estaría presente, pero no sería accesible en la referencia que tiene `animal`).

Después creamos una instancia de la clase `Gato` y almacenamos su referencia en la variable `animal` y llamamos a `hacerRuido` e `imprime` de nuevo, sin embargo el comportamiento ejecutado será el implementado (hacer ruido) y el heredado (`imprime`).

```
scala> abstract class Animal(val especie: String, val nombre: String) {
  |   def hacerRuido: Unit
  |   def imprime(): Unit = {
  |     println(s"$nombre es un $especie y suena:")
  |     hacerRuido
  |   }
  | }
class Animal

scala> class Gato(n: String) extends Animal(especie="gato", nombre=n) {
  |   def hacerRuido(): Unit = println(s"$nombre -> meow")
  | }
class Gato

scala> class Perro(n: String) extends Animal(especie="perro", nombre=n) {
  |   override def imprime(): Unit = {
  |     println(s"mi nombre es $nombre y soy un perro")
  |     println("woof! woof! woof!")
  |   }
  |   def hacerRuido(): Unit = println(s"$nombre -> woof")
  | }
class Perro

scala> var animal: Animal = new Perro("doggy")
var animal: Animal = Perro@7238072e

scala> animal.hacerRuido
doggy -> woof

scala> animal.imprime
mi nombre es doggy y soy un perro
woof! woof! woof!

scala> animal = new Gato("kitty")
// mutated animal

scala> animal.hacerRuido
kitty -> meow

scala> animal.imprime
kitty es un gato y suena:
kitty -> meow

scala>
```



Polimorfismo

En resumen, el tipo de la variable `animal` actúa como filtro, asegurándose de que sólo las funciones y métodos comunes son accesibles.

En tiempo de ejecución la definición de la función `imprime`, que está definida tanto en `Animal` como en `Perro`, está ligada dinámicamente, esto quiere decir que la versión del método que se ejecutará viene dada por la clase de la instancia a la que se referencia en la variable. Primero se ejecuta la versión de la subclase, en caso de que no se encuentre en la subclase el método, se usará la implementación de la superclase.

```
scala> abstract class Animal(val especie: String, val nombre: String) {  
  |   def hacerRuido: Unit  
  |   def imprime(): Unit = {  
  |     println(s"$nombre es un $especie y suena:")  
  |     hacerRuido  
  |   }  
  | }  
class Animal  
  
scala> class Gato(n: String) extends Animal(especie="gato", nombre=n) {  
  |   def hacerRuido(): Unit = println(s"$nombre -> meow")  
  | }  
class Gato  
  
scala> class Perro(n: String) extends Animal(especie="perro", nombre=n) {  
  |   override def imprime(): Unit = {  
  |     println(s"mi nombre es $nombre y soy un perro")  
  |     println("woof! woof! woof!")  
  |   }  
  |   def hacerRuido(): Unit = println(s"$nombre -> woof")  
  | }  
class Perro  
  
scala> var animal: Animal = new Perro("doggy")  
var animal: Animal = Perro@7238072e  
  
scala> animal.hacerRuido  
doggy -> woof  
  
scala> animal.imprime  
mi nombre es doggy y soy un perro  
woof! woof! woof!  
  
scala> animal = new Gato("kitty")  
// mutated animal  
  
scala> animal.hacerRuido  
kitty -> meow  
  
scala> animal.imprime  
kitty es un gato y suena:  
kitty -> meow  
  
scala>
```

3 Clases y funciones genéricas



Clases Genéricas

En Scala, al igual que en Java, se permite definir clases cuyo tipo esté parametrizado:
`generic class`

Las `generic class` toman el tipo para el cual se instancian de un parámetro que se define cuando se declara la clase.

Para declarar el tipo para el que se implementa una clase, se hace, siguiendo al identificador de la clase entre corchetes [].

En Scala, existe la convención de identificar el parámetro de tipo con la letra A cuando se define la clase genérica



Clases Genéricas

Definimos la clase Pila, pero la definición de pila toma cualquier tipo [A] como parámetro.

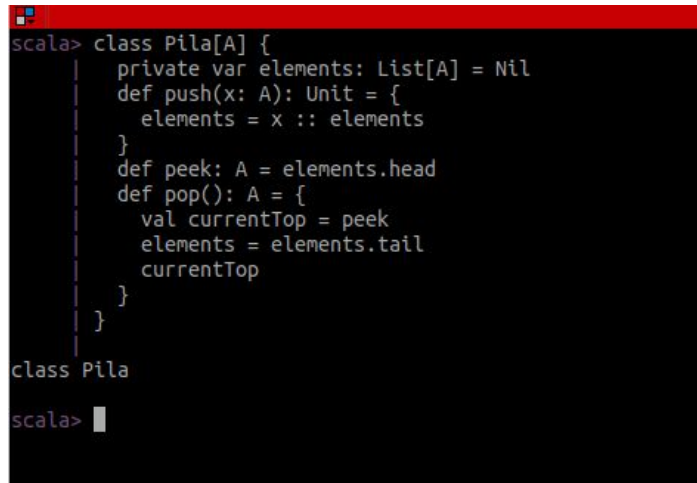
Esto implica que el campo elements será una lista que contendrá elementos del tipo A.

El método push, sólo aceptará argumentos que sean del tipo A

Las funciones peek y pop van a devolver como resultado una referencia a una instancia de la clase A.

Esto permite definir el comportamiento deseado de Pila para cualquier tipo de datos que necesitemos.

```
class Pila[A] {  
  private var elements: List[A] = Nil  
  def push(x: A): Unit = {  
    elements = x :: elements  
  }  
  def peek: A = elements.head  
  def pop(): A = {  
    val currentTop = peek  
    elements = elements.tail  
    currentTop  
  }  
}
```



```
scala> class Pila[A] {  
  |   private var elements: List[A] = Nil  
  |   def push(x: A): Unit = {  
  |     elements = x :: elements  
  |   }  
  |   def peek: A = elements.head  
  |   def pop(): A = {  
  |     val currentTop = peek  
  |     elements = elements.tail  
  |     currentTop  
  |   }  
  | }  
class Pila  
  
scala> |
```



Clases Genéricas

Teniendo definida la clase Pila del ejemplo anterior, vamos a definir una Pila de Fruta, teniendo dos tipos de Fruta, Manzana y Banana.

Las variables manzana y banana las instancias de las clases Manzana y Banana correspondientemente y ambos extienden de Fruta con lo cual ambos tienen como clase padre a Fruta, es por eso que es posible añadir a ambos a la variable miPilaFruta que es una instancia de Pila[Fruta]

```
class Fruta
class Manzana extends Fruta
class Banana extends Fruta

val miPilaFruta = new Pila[Fruta]
val manzana = new Manzana
val banana = new Banana

miPilaFruta.push(manzana)
miPilaFruta.push(banana)
```

```
scala> class Fruta
class Fruta

scala> class Manzana extends Fruta
| class Banana extends Fruta
class Manzana
class Banana

scala> val miPilaFruta = new Pila[Fruta]
| val manzana = new Manzana
| val banana = new Banana
val miPilaFruta: Pila[Fruta] = Pila@72168258
val manzana: Manzana = Manzana@4af84a76
val banana: Banana = Banana@b5ff70b

scala> miPilaFruta.push(manzana)

scala> miPilaFruta.push(banana)

scala> miPilaFruta.peak
val res4: Fruta = Banana@b5ff70b

scala> miPilaFruta.pop
val res5: Fruta = Banana@b5ff70b

scala> miPilaFruta.peak
val res6: Fruta = Manzana@4af84a76

scala> miPilaFruta.pop
val res7: Fruta = Manzana@4af84a76

scala> |
```



Clases Genéricas

Veamos cómo podemos usar la clase Pila definida anteriormente pero con distintos tipos de datos.

```
val miPilaEnteros = new Pila[Int]
miPilaEnteros.push(1)

val miPilaString = new Pila[String]
miPilaString.push("esto es un string")
```

```
scala> val miPilaString = new Pila[String]
val miPilaString: Pila[String] = Pila@4ac2d69c

scala> miPilaString.push("esto es un string")

scala> miPilaString.pop
val res23: String = esto es un string

scala> 
```

```
scala> val miPilaEnteros = new Pila[Int]
val miPilaEnteros: Pila[Int] = Pila@6edifdb7

scala> miPilaEnteros.push(1)

scala> miPilaEnteros.push(2)

scala> miPilaEnteros.push(3)

scala> miPilaEnteros.peak
val res18: Int = 3

scala> miPilaEnteros.pop
val res19: Int = 3

scala> miPilaEnteros.pop
val res20: Int = 2

scala> miPilaEnteros.pust("hola")
      ^
error: value pust is not a member of Pila[Int]
did you mean push?

scala> 
```



Funciones Genéricas

Los métodos, al igual que las clases pueden ser parametrizados con un tipo de datos específico.

Su sintaxis es similar a la definición de clase genérica: parámetros de valores son encerrados en un par de paréntesis, mientras que los parámetros de tipo son declarados dentro de un par de corchetes: [A].



Funciones Genéricas

Veamos un ejemplo, queremos definir una lista que repita el valor de uno de sus parámetros tantas veces como lo determine su segundo parámetro y sea una función a la que podamos llamar para tipos `Int` como `String`:

Como vemos en la definición de la función `listOfDuplicates`, ésta toma como parámetro de tipo `A` y parámetros como valores `x` y `length`, donde `x` tiene que ser del tipo `A`.

Esta es una función recursiva cuya condición de parada es `length < 1` y al cumplirse devuelve una lista vacía `Nil`.

En caso de no cumplirse la condición de parada de la función, se construye la lista con `x` (de tipo `A`) y el resultado de llamarse recursivamente a sí misma

```
def listOfDuplicates[A](x: A, length: Int): List[A] = {  
  if (length < 1)  
    Nil  
  else  
    x :: listOfDuplicates(x, length - 1)  
}
```



Funciones Genéricas

Si observamos en las ejecuciones:

- en la primera se especifica el tipo `[Int]` cuando llamamos `listOfDuplicates`. Además su primer argumento tiene que ser `Int` y devuelve una lista de `List[Int]`.
- en la segunda ejecución, vemos que no se define el parámetro de tipo, pero al ejecutarlo devuelve una instancia de `List[String]`, esto se debe a que el compilador ha sido capaz de inferir el tipo basándose en el contexto, en este ejemplo "La" es `String` así que el compilador sabe que `A` debe ser `String`.

```
def listOfDuplicates[A](x: A, length: Int): List[A] = {  
  if (length < 1)  
    Nil  
  else  
    x :: listOfDuplicates(x, length - 1)  
}
```

```
scala> def listOfDuplicates[A](x: A, length: Int): List[A] = {  
  |   if (length < 1)  
  |     Nil  
  |   else  
  |     x :: listOfDuplicates(x, length - 1)  
  | }  
def listOfDuplicates[A](x: A, length: Int): List[A]  
  
scala> listOfDuplicates[Int](3, 4)  
val res0: List[Int] = List(3, 3, 3, 3)  
  
scala> println(listOfDuplicates("La", 8))  
List(La, La, La, La, La, La, La, La)  
  
scala> 
```



