# Tema 8: Herencia

Antonio J. Sierra

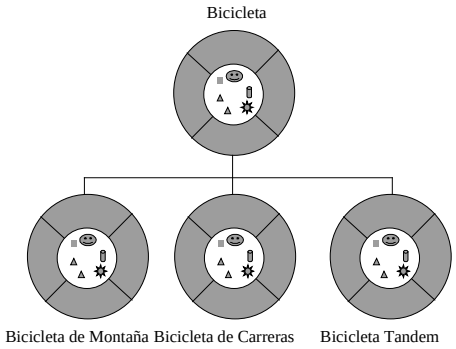
# Índice

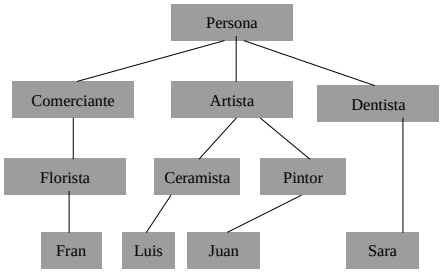
# Introducción. Herencia

- Las ‘Subclases’ son versiones más especializadas de una clase, que **heredan** atributos y **propiedades** de las clases **padres**, y pueden introducir las suyas propias.
- Por ejemplo, la clase **`Perro`** podría tener las subclases **`Collie`**, **`Chihuahua`** y **`GoldenRetriever`**.
- En este caso, **`Lassie`** debería ser una instancia de **`Collie`**.
- Suponga que la clase **`Perro`** define un método **`ladrar()`** y la propiedad **`colorPelo`**. Cada una de sus subclases también lo heredarán. El programador solo deberá escribir una sola vez el código.
- Las subclases pueden alterar las propiedades tratadas.

# Introducción. Herencia Múltiple.

- Es una herencia de más de una clase antecesora, con las antecesoras sin ser antecesoras de las otras.
- No siempre se puede realizar.
- Es difícil de implementar.
- Es difícil de usar.




# Introducción

- Clasificaciones jerárquicas.
- Reutilización de código.
- Componentes:
  - Clase más general: superclase (padre).
  - Clase más específica: subclase (hija).
- La subclase hereda las variable de instancia y métodos.

# Ejemplo

```
class SuperClaseA{
 int i;
 void muestraI(){
  System.out.println("Valor de i "
   +i);
    }
}
class SubClaseB extends SuperClaseA{
  int j;
  SubClaseB (int par){
     this.j = par;
     i = par + 20;
  }
  void muestraIJ(){
    System.out.println("Valor de i "
        + i + "\nValor de j "+ j);
    }
}
```

```
class prueba{
  public static void main(String args[]){
   SubClaseB obj = new SubClaseB(5);
   obj.muestraIJ();
   System.out.println();
   obj.muestraI();
 }
}
```

# Variable de la superclase referenciado a un objeto de la subclase

```
class prueba{
  public static void main( String args[]){
   SubClaseB objSub = new SubClaseB(5);
   SuperClaseA objSuper = null;
   objSuper = objSub ;
  }
}
```

## Acceso a miembros y herencia

- No se accede a los miembros de la **`private`** de la superclase

```
class SuperClaseA{
    int i;
    private int k; // atributo privado
    void muestraI(){
        System.out.println("Valor de i "+i);
    }
}
class SubClaseB extends SuperClaseA{
    int j;
    SubClaseB (int par){
        this.j = par;
        i = par + 20;
    }
    void muestraIJ(){
        System.out.println("Valor de i " + i + "\nValor de j "+ j);
    }
}
```

## super

- Una subclase puede referirse a su superclase inmediata, mediante la palabra clave **`super`**
- Se puede utilizar de dos formas:
  1. para llamar al constructor de la superclase.
  2. para acceder a un miembro de la superclase que ha sido ocultado por un miembro de la subclase.

## `super`: Llamada a un constructor de la superclase

- Se usa en los constructores
- Debe ser la primera línea
- Sintaxis: `super(ListaDeParametros);`
- `ListaDeParametros` especifica los parámetros del constructor de la superclase.

## Jerarquía multinivel

- Se pueden construir jerarquías que contengan tantos niveles de herencia como se desee.
- No existe límite en la profundidad.

# Ejemplo (I)

```
class Caja {
    private double altura;
    private double anchura;
    private double profundidad;

    // Construye un duplicado de un objeto
    //pasa un objeto al constructor
    Caja(Caja ob) {
          altura = ob.altura;
          anchura = ob.anchura;
          profundidad = ob.profundidad;
    }

    // Este constructor se usa cuando
    // se dan todas las dim.
    Caja(double h, double a, double p) {
          altura = h;
          anchura = a;
          profundidad = p;
    }

    // Si no se especifica ninguna dimension
    Caja() {
          altura = -1;
          anchura = -1;
          profundidad = -1;
    }

    // constructor para el cubo
    Caja(double len) {
          altura = anchura = profundidad = len;
    }

    //calcula y devuelve el volumen
    double volumen(){
          return altura*anchura*profundidad;
    }
}
```

# Ejemplo (II)

```
// PesoCaja ahora implementa todos los constructores
class PesoCaja extends Caja {
    double peso;

    //construye un duplicado de un objeto
    PesoCaja(PesoCaja ob) { //Pasa un objeto al constructor
           super(ob);
           peso = ob.peso;
    }

    //constructor si se especifican todos los parámetros
    PesoCaja(double h, double a, double p, double pes) {
           super(h,a,p);
           peso = pes;
    }

    //constructor por defecto
    PesoCaja(){
           super();
           peso = -1;
    }

    //constructor para el cubo
    PesoCaja(double lon, double pes) {
           super(lon);
           peso = pes;
    }
}
```

# Ejemplo (III)

```
//añade los costes del transporte
class Envio extends PesoCaja {
    double coste;

    //Construye un duplicado de un objeto
    Envio(Envio ob){//pasa un objeto al constructor
           super(ob);
           coste = ob.coste;
    }

    //Este constructor con todos los parametros
    Envio(double p, double h, double a, double pp, double c){
           super(p,h,a,pp);
           coste = c;
    }

    //constructor por defecto
    Envio() {
           super();
           coste = -1;
    }

    //constructor para el cubo
    Envio(double lon, double m, double c){
           super(lon,m);
           coste = c;
    }
}
```

# Ejemplo (y IV)

```
class Ejem3 {
 public static void main(String args[]){
  Envio envio1 = new Envio(10,20,15,10,34.3);
  Envio envio2 = new Envio(2,3,4,0.076,1.28);
  double vol;

  vol = envio1.volumen();
  System.out.println("El volumen de la envio1 es " +vol);
  System.out.println("El peso de envio1 es " +envio1.peso);
  System.out.println("El coste de envio1 es "
    +envio1.coste);
  System.out.println();

  vol = envio2.volumen();
  System.out.println("El volumen de la envio2 es " +vol);
  System.out.println("El peso de envio2 es " +envio2.peso);
  System.out.println("El coste de envio2 es "
    +envio2.coste);
  System.out.println();
    }
}
```

```
/*
El volumen de la envio1 es 3000.0
El peso de envio1 es 10.0
El coste de envio1 es 34.3

El volumen de la envio2 es 24.0
El peso de envio2 es 0.076
El coste de envio2 es 1.28
*/
```

## **super**: Acceso a un miembro de la superclase

- **super** es similar a **this**, excepto que **super** siempre se refiere a la superclase de la subclase en la que utiliza.
- Su formato es

  **super.*miembro***

  donde *miembro* puede ser un método o una variable de instancia.
- **super** se utiliza en aquellos casos en los nombres de miembros de una subclase ocultan los miembros que tienen el mismo nombre en la superclase.

## Ejemplo

```
//uso de super para evitar ocultar nombres.
class A {
 int i;
}

//Crea una subclase extendiendo la clase A.
class B extends A{
 int i;   //esta i oculta a la i de A

 B(int a, int b) {
  super.i = a; //i de A
  i = b ;  //i de B
 }
 void show() {
  System.out.println("i de la superclase: " +
    super.i);
  System.out.println("i de la subclase: " +
    i);
    }
}
```

```
class Ejem2 {
 public static void
   main(String args[]){
  B subOb = new B(1,2);
  subOb.show();
 }
}
/*
i de la superclase: 1
i de la subclase: 2
*/
```

# Orden de ejecución de los constructores

- **Los constructores se ejecutan en orden de derivación**
  - desde la superclase a la subclase
- **`super()`** tiene que ser la primera sentencia que se ejecute dentro de constructor de la subclase,
  - este orden es el mismo tanto si se utiliza **`super()`** como si no
- Si no se utiliza **`super()`**, entonces se ejecuta el constructor por defecto o sin parámetros de cada superclase

# Ejemplo

```
class A {
   A(){System.out.println("En el constructor de A.");}
}
class B extends A{
   B(){ System.out.println("En el constructor de B.");}
}
class C extends B {
   C(){ System.out.println("En el constructor de C.");}
}
class Ejem3bis {
   public static void main(String args[]){
    C c = new C();
   }
}
/*
En el constructor de A.
En el constructor de B.
En el constructor de C.
*/
```

# Sobreescritura de un método

- Se dice que un método de la subclase **sobreescribe** al método de la superclase en una jerarquía de clases, cuando un método de una subclase tiene el mismo nombre y tipo que un método de su superclase.
- La invocación de un método sobreescrito de una subclase, se refiere a la versión del método definida por la subclase.
  - La versión del método definida por la superclase queda oculta.

# Ejemplo

```
class A {
     int i, j;
     A(int a, int b){
            i = a;
            j = b;
     }

     //imprime i y j
     void imprime(){
      System.out.println("i y j: " + i + "" + j);
     }
}
class B extends A{
     int k;
     B(int a, int b, int c){
            super(a, b);
            k=c;
     }

     //imprime k sobrescribe el método de A
     void imprime()
     {
            System.out.println("k: " + k);
     }
}
class Ejem4 {
     public static void main(String args[]){
      B subOb = new B(1, 2, 3);
      subOb.imprime();  //llama al método imprime() de B
     }
}
/*
k: 3
*/
```

# métodos sobrecargados

```
class A {
 int i, j;
 A(int a, int b){
  i = a;
  j = b;
 }

 //imprime i y j
 void imprime(){
  System.out.println("i y j: " + i + " " + j);
 }
}
class B extends A{
 int k;
 B(int a, int b, int c){
  super(a, b);
  k=c;
 }

 //sobrecarga el método imprime()
 void imprime(String msg) {
  System.out.println(msg + k);
 }
}
```

```
class Ejem5 {
 public static void main(String args[]){
  B subOb = new B(1, 2, 3);

    //llama al método imprime() de B
    subOb.imprime("Esto es k: ");
    //llama al métodoimprime() de A
    subOb.imprime();
 }
}

/*
Esto es k: 3
i y j: 1 2
*/
```

# Selección de método dinámica (I)

- La selección de método dinámica es el mecanismo mediante el cual una llamada a una función sobrescrita se **resuelve en tiempo de ejecución**, en lugar de durante la compilación.
- La selección de método dinámica es importante ya que es la forma que tiene Java de implementar el **polimorfismo durante la ejecución**.

# Selección de método dinámica (II)

- Utiliza dos cosas:
  - **Una variable de referencia de la superclase puede referirse a un objeto de la subclase**
  - **Sobreescritura de método**
- Es el **tipo del objeto que está siendo referenciado**, y no el tipo de la variable referencia, el que determina qué versión de un método sobrescrito será ejecutada.

# Ejemplo

```
class A {
 void imprime(){
  System.out.println("Se ejecuta el método imprime en A");
 }
}
class B extends A{
 void imprime(){ //sobrescribe imprime
  System.out.println("LLama al método imprime en B");
 }
}
class C extends A{
 void imprime(){//sobrescribe imprime
  System.out.println("LLama al método imprime en C");
 }
}
```

```
class Ejem7 {
     public static void main(String args[]){
      A a = new A();//objeto del tipo A
      B b = new B();//objeto del tipo B
      C c = new C();//objeto del tipo C
      A r; //obtiene una referencia de tipo A

      r = a;//r hace referencia a un objeto A
      r.imprime();//llama al metodo de A

      r = b; //r hace referencia a un objeto B
      r.imprime();//llama al metodo de B

      r = c;//r hace referencia a un objeto C
      r.imprime();//llama al metodo de C
     }
}
/*
Se ejecuta el método imprime en A
LLama al método imprime en B
LLama al método imprime en C
*/
```

# final

- Para atributos:

```
class Circulo {
    . . .
    public final static float PI = 3.141592;
    . . .
}
```

- Para clases: no se permite que sea superclase.

```
final class Ejecutivo {
//. . .
}
```

- Para métodos: no se permite sobreescritura.

```
class Empleado {
    . . .
 public final void aumentarSueldo(int porcentaje) {
        . . .
 }
    . . .
}
```

# Object

- Es la clase raíz de todo el árbol de la jerarquía de clases Java.
- Proporciona métodos de utilidad general que pueden utilizar todos los objetos.

```
public              boolean equals( Object obj );
public              String  toString();
public              int     hashCode();
protected           void    finalize();
protected           Object  clone()
public final        void    wait();
public final native void    wait( long timeout );
public final native void    wait( long timeout, int nanos );
public final native void    notify();
public final native void    notifyAll();
public final native Class   getClass();
```

## protected

- Modificador de acceso.
- `finalize()`
- Visible por todas las subclases.

## Clases Abstractas

- Define una superclase que declara la estructura de una abstracción sin proporcionar implementación completa de todos métodos.
- Deja a cada subclase la tarea de completar los detalles.
- La superclase determina la naturaleza de los métodos que las subclases deben implementar.
- No se puede instanciar clases abstractas.

# Sintaxis para las clases abstractas

- Sintaxis de método:

```
abstrac tipo nombre(ListaDeParametros);
```

- Sintaxis de clase :

```
abstrac class nombre{
 //
}
```

# Ejemplo (I)

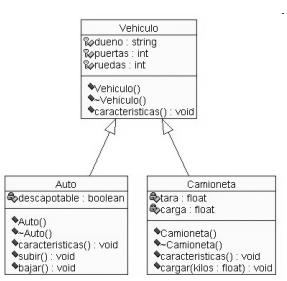
```
//Un ejemplo sencillo de abstract
abstract class A {
   abstract void Llamada();

   void OtraLlamada(){
          System.out.println("Este es un método concreto.");
   }
}

class B extends A{
   void Llamada(){
          System.out.println("Llamada en B.");
   }
}

class Abstract0 {
   public static void main(String args[]){
          B b = new B();
          b.Llamada();
          b.OtraLlamada();
   }
}
```

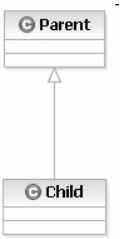
# Ejemplo (II)

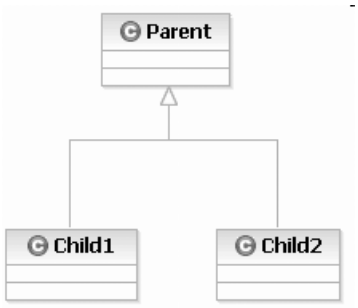
```
//Un ejemplo sencillo de abstract
abstract class Figura {
    double dim1;
    double dim2;

    Figura(double a, double b){
          dim1= a;
          dim2= b;
    }
    abstract double area();
}

class Rectangulo extends Figura {
    Rectangulo(double a, double b){
          super(a,b);
    }
    double area(){
          return dim1*dim2;
    }
}

class Triangulo extends Figura {
    Triangulo(double a, double b){
          super(a,b);
    }
    double area(){
          return dim1*dim2/2;
    }
}
```

```
class Abstract1 {
 public static void main(String args[]){
  Rectangulo r = new Rectangulo (10,10);
  Triangulo t = new Triangulo (10,10);
  Figura RefFig;
    //esto es correcto, no se crea objeto
    //No es correcto "Figura f = new Figura(10,10);"
   RefFig = r;
  System.out.println("El área es " + RefFig.area());

  RefFig = t;
  System.out.println("El área es " + RefFig.area());
 }
}
/*
El área es 100.0
El área es 50.0
*/
```

# Herencia en UML



# clases abstractas en UML