Unidad 8

Herencia

Contenidos

Herencia

- Definición
- Finalidad
- Ejemplo de herencia
- Generalización y Especialización
- Diseño de la jerarquía de clases
- Tipos de Herencia
- Declaración de herencia en Java
- Acceso clase derivada o subclase
- Modificadores de acceso
- Constructores y Herencia
- Llamada a constructor de padre implicito
- Llamada al constructor del padre: super( )
- super(con parámetros)

Contenidos

Herencia

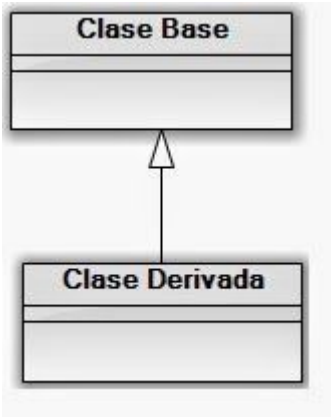
- Redefinición de métodos heredados
- Polimorfismo
- final
- Conversión de referencias (casting)
- Referencias a clases hijas y Visibilidad
- Selección dinámica de métodos
- getClass()
- getClass() para comparar
- instanceof
- Clases abstractas
- Métodos abstractos

Definición

La herencia es una de las características fundamentales de la Programación Orientada a Objetos.

Mediante la herencia podemos definir una clase a partir de otra ya existente, y, por tanto, se permite heredar las características (atributos y métodos) de otra clase.

La clase nueva se llama clase derivada o subclase (o hija) y la clase existente se llama clase base o superclase (o padre).



En UML la herencia se representa con una flecha apuntando desde la clase derivada a la clase base.

Finalidad

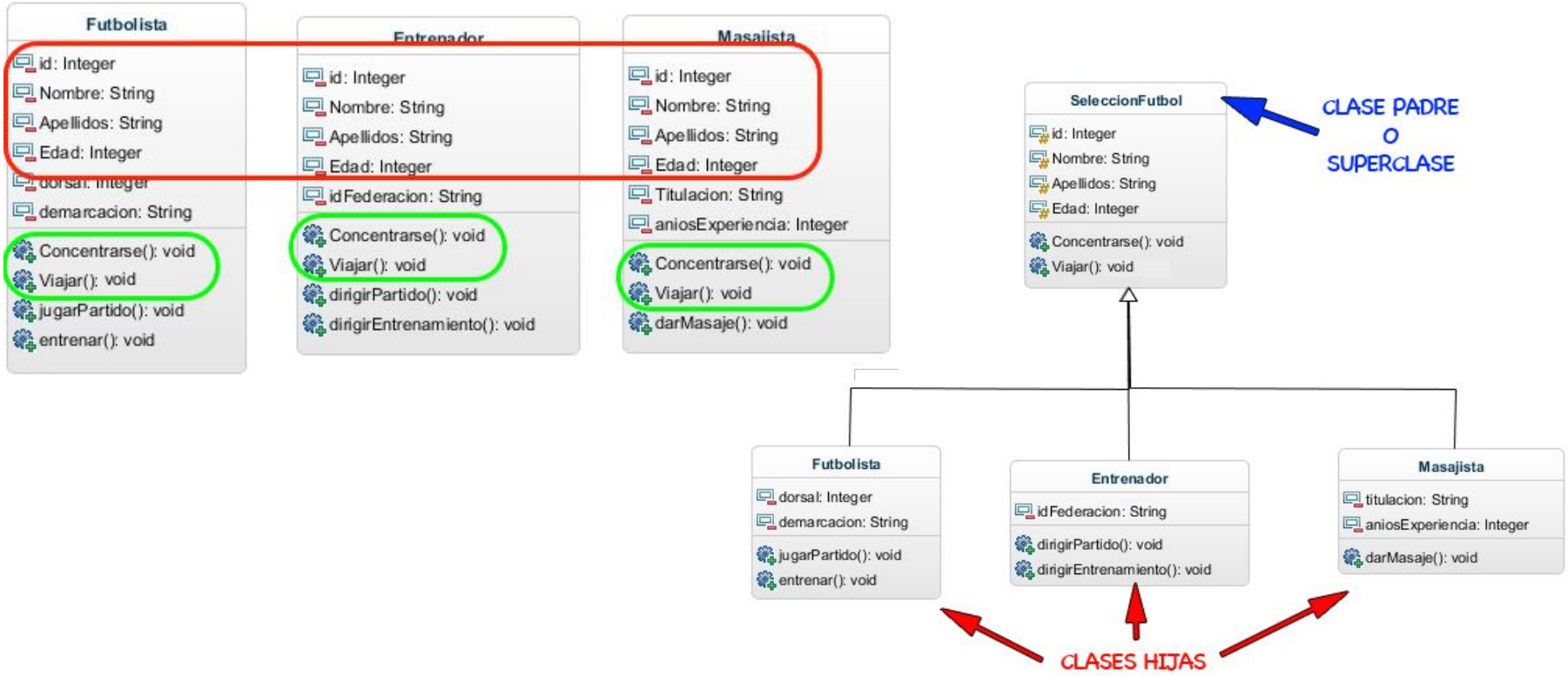
La clase derivada hereda los componentes (atributos y métodos) de la clase base.

La finalidad de la herencia es:

- **Extender la funcionalidad** de la clase base: en la clase derivada se pueden añadir atributos y métodos nuevos.
- **Especializar el comportamiento** de la clase base: en la clase derivada se pueden modificar (sobrescribir, override) los métodos heredados para adaptarlos a sus necesidades.
- **Reutilización de código**: El código se escribe una vez en la clase base y se utiliza en todas las clases derivadas.

La herencia permite la reutilización del código, ya que evita tener que reescribir de nuevo una clase existente cuando necesitamos ampliarla en cualquier sentido. Todas las clases derivadas pueden utilizar el código de la clase base sin tener que volver a definirlo en cada una de ellas.

Ejemplo de herencia



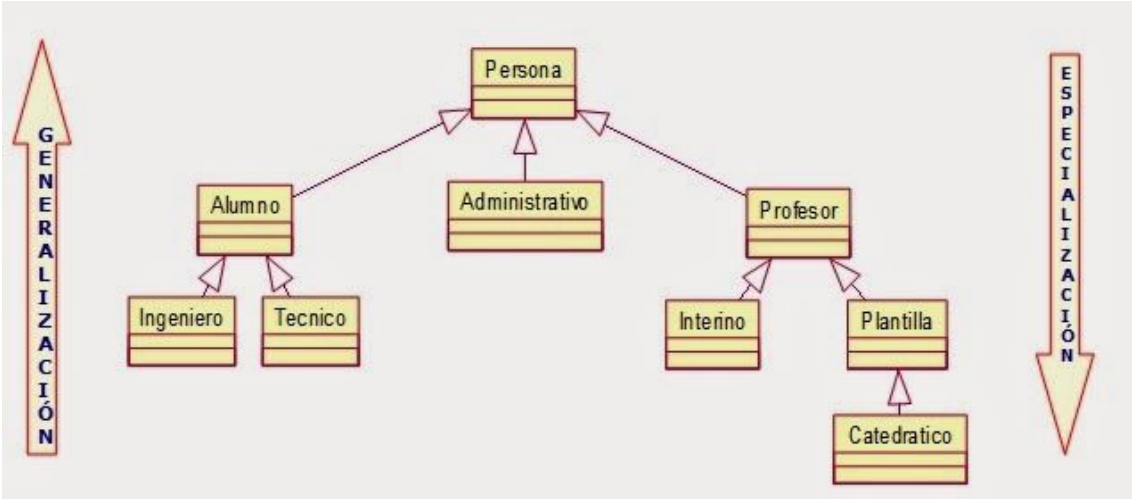
Generalización y Especialización

La **generalización** es la propiedad que permite compartir información entre dos entidades evitando la redundancia.

En el comportamiento de objetos existen con frecuencia propiedades que son comunes en diferentes objetos y esta propiedad se denomina **generalización**

El proceso inverso de la generalización por el cual se definen nuevas clases a partir de otras ya existentes se denomina **especialización**.

Diseño de la jerarquía de clases



Generalización (*Factorización*) Se detectan clases con un comportamiento común
Especialización (*Abstracción*) Se detecta que una clase es un caso especial de otra

No hay receta mágica para crear buenas jerarquías

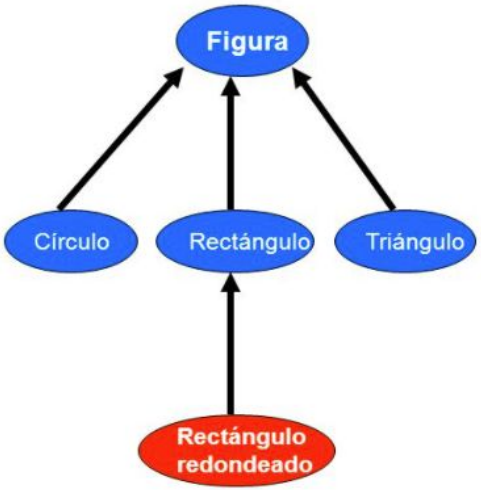
Tipos de herencia

Herencia simple

Una clase puede heredar de una única clase.

Se forma una estructura jerárquica.

Java, C#



Herencia múltiple

Una clase puede heredar de varias clase.

Se forma una estructura en forma de grafo acíclico.

Eiffel, C++

Java no permite herencia múltiple
Se simula con interfaces

Declaración de herencia en Java

```
public class ClasePadre {
    // Declaración de los atributos y métodos comunes
}


public class ClaseHija extends ClasePadre {
    // Declaración de los atributos y métodos específicos
    // de la ClaseHija
}
```

Declaración de herencia en Java

Ejemplo

```java
//Clase Persona. La clase Persona es la Clase Base

public class Persona {

    private String nif;
    private String nombre;

    public String getNif() {
        return nif;
    }

    public void setNif(String nif) {
        this.nif = nif;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
}
```

```java
// Clase Alumno. Es la clase derivada.
// La clase Alumno hereda de la clase Persona

public class Alumno extends Persona {

    private String curso;

    public String getCurso() {
        return curso;
    }

    public void setCurso(String curso) {
        this.curso = curso;
    }

}
```

Acceso clase derivada o subclase

Los constructores no se heredan. Las clases derivadas deberán implementar sus propios constructores.

Una clase derivada **puede acceder a los miembros públicos** y protegidos de la clase base como si fuesen miembros propios.

Una clase derivada **no tiene acceso a los miembros privados** de la clase base. Deberá acceder a través de métodos heredados de la clase base.

Si se necesita tener acceso directo a los miembros privados de la clase base se deben declarar **protected** en lugar de private en la clase base.

Una clase derivada puede **añadir** a los miembros heredados, sus propios **atributos y métodos** (extender la funcionalidad de la clase).

También puede **modificar** los **métodos heredados** (especializar el comportamiento de la clase base).

- Una clase derivada puede, a su vez, ser una clase base, dando lugar a una jerarquía de clases

Modificadores de acceso

	private	default	protected	public
Misma clase	**Si**	**Si**	**Si**	**Si**
Subclase (mismo paquete)	**No**	**Si**	**Si**	**Si**
No subclase (mismo paquete)	**No**	**Si**	**Si**	**Si**
Subclase (paquete diferente)	**No**	**No**	**Si**	**Si**
No subclase (paquete diferente)	**No**	**No**	**No**	**Si**

Constructores y Herencia

El **constructor** para la superclase construye la porción de la superclase del objeto, y el constructor para la subclase construye la parte de la subclase. Por lo tanto, su construcción debe estar separada.

En la práctica, la mayoría de las clases tendrán constructores explícitos (no predeterminados).

Cuando solo la subclase define un constructor, el proceso es sencillo: simplemente construye el objeto de la subclase. La porción de superclase del objeto se construye automáticamente utilizando su constructor predeterminado.

Llamada al constructor de padre implícito

La clase base es la encargada de inicializar sus atributos.

La clase derivada se encarga de inicializar solo los suyos.

Cuando se crea un objeto de una clase derivada se ejecutan los constructores en este orden:

1. Primero se ejecuta el constructor de la clase base

2. Después se ejecuta el constructor de la clase derivada.

Cuando se invoca al constructor de la clase Alumno se invoca automáticamente al constructor de la clase Persona y después continúa la ejecución del constructor de la clase Alumno.

El constructor por defecto de la clase derivada llama al constructor por defecto de la clase base.

```
public class Persona {

    private String nif;
    private String nombre;

    public Persona() {
        System.out.println("Ejecutando el constructor de Persona");
    }

    /////// Resto de métodos
}

public class Alumno extends Persona{

    private String curso;

    public Alumno() {
        System.out.println("Ejecutando el constructor de Alumno");
    }

    /////// Resto de métodos

}
```

Llamada al constructor del padre: super( )

La instrucción para invocar al constructor por defecto de la clase base es: ***super();***

Todos los constructores en las clases derivadas contienen de forma implícita la instrucción *super()* como primera instrucción.

```
public Alumno() {
        super(); //esta instrucción se ejecuta siempre. No es necesario escribirla
        System.out.println("Ejecutando el constructor de Alumno");
}
```

super (con parámetros)

Cuando se crea un objeto de la clase derivada y queremos asignarle valores a los atributos heredados de la clase base:

- La clase derivada debe tener un constructor con parámetros adecuado que reciba los valores a asignar a los atributos de la clase base.
- La clase base debe tener un constructor con parámetros adecuado.
- El constructor de la clase derivada invoca al constructor con parámetros de la clase base y le envía los valores iniciales de los atributos. Debe ser la primera instrucción.
- La clase base es la encargada de asignar valores iniciales a sus atributos.
- A continuación el constructor de la clase derivada asigna valores a los atributos de su clase.

super(con parámetros)

Ejemplo

```
public class Persona {

    private String nif;
    private String nombre;

    public Persona(String nif, String nombre) {
        this.nif = nif;
        this.nombre = nombre;
    }

    //Resto de métodos
}
```

```
public class Alumno extends Persona{

    private String curso;

    public Alumno(String nif, String nombre, String curso) {
        super(nif, nombre);
        this.curso = curso;
    }

    //Resto de métodos

}
```

Redefinición de métodos heredados

SOBRECARGA DE MÉTODOS

Un método se puede redefinir (volver a definir con el mismo nombre) en una subclase.

En estos casos, indicaremos nuestra intención de sobreescribir un método mediante la etiqueta `@Override`

Si no escribimos esta etiqueta, la sobreescritura del método se realizará de todas formas ya que @Override indica simplemente una intención. Si escribes `@Override` y luego te equivocas en el nombre del método entonces el compilador diría algo como:

"¡Cuidado! algo no está bien, me has dicho que ibas a sobreescribir un método de la superclase y sin embargo no está definido".

Polimorfismo

En Programación Orientada a Objetos, se llama polimorfismo a **la capacidad que tienen los objetos de distinto tipo (de distintas clases) de responder al mismo método**.

Mediante la sobrecarga de métodos, podemos llamar con el **mismo método** a diferentes subclases y realizarán **acciones diferentes**.

final

Cuando una **clase** tiene el modificador **final**, no se pueden crear subclases de ésta. Es decir, no puede haber clases hijas que hereden de ella.

Cuando un **método** se define como **final**, no se puede sobreescribir el método en subclases.

Cuando un **atributo** se define como **final**, no se puede modificar su valor. Es una constante.

Conversión de referencias (casting)

El casting o moldeo permite el uso de un objeto de una clase en lugar de otro de otra clase con el que haya una relación de herencia.

La conversión hacia arriba es implícita

```
SuperClase x = new SubClase();
```

Ejemplos:

```
Empleado e = new Empleado();

Persona p = e;
```

La conversión hacia abajo debe indicarse

```
SubClase x = (SubClase) variableSuperClase;
```

Ejemplos:

```
Persona p = new Empleado();

Empleado e = (Empleado) p;
```

Si el tipo del que queremos hacer el casting no es correcto, nos dará un error de conversión en tiempo de ejecución.

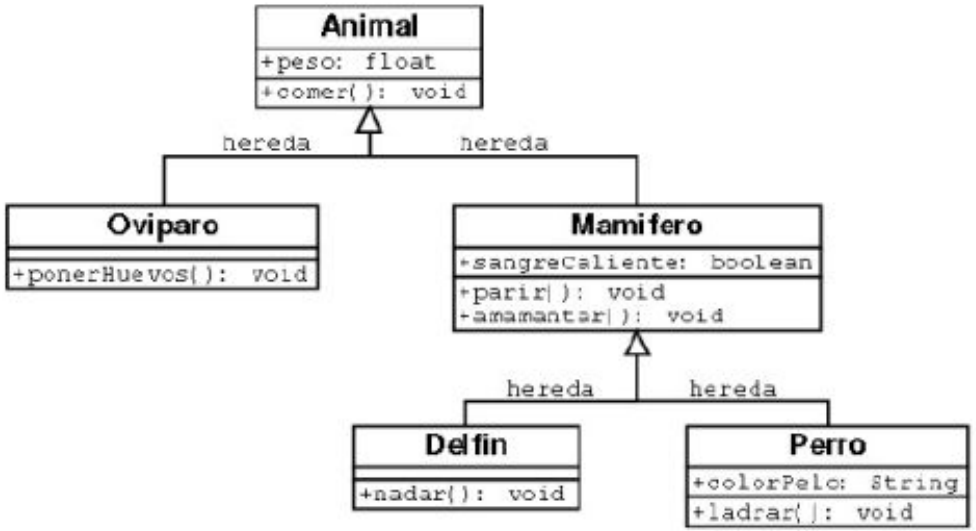
Referencias a clases hijas y visibilidad

Cuando definimos una clase como subclase de otra, los objetos de la subclase son también objetos de la superclase.

```
Perro p = new Perro();
. . .
Mamifero m = p;
Animal a = p;
```

Con variables de tipo más general podemos referenciar a clases hijas. Pero no al contrario.

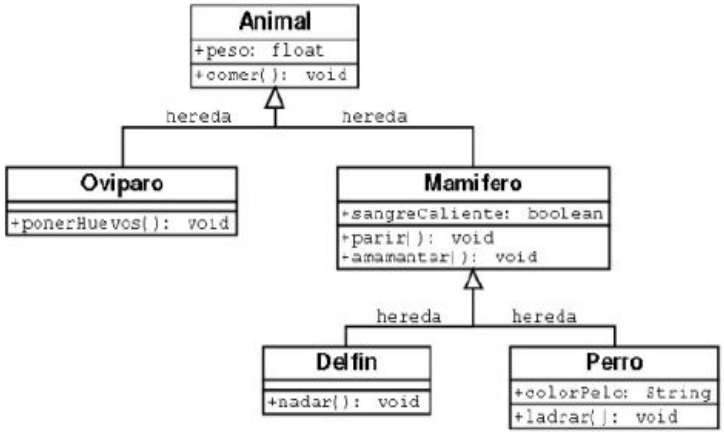
```
Animal a = new Animal();
. . .
Mamifero m = a;   // ERROR
Perro p = a;      // ERROR
```



Referencias a clases hijas y visibilidad

```
Perro p = new Perro();

Mamifero m = p;
Animal a = p;
```



Solo se podrán acceder a los **atributos o métodos** definidos en el tipo de la variable aunque esté apuntando a una subclase. Compruébalo con el ejemplo...

```
a.peso;
a.comer();
```

```
m.peso;
m.comer();
m.sangreCaliente;
m.parir();
m.amamantar();
```

```
p.peso;
p.comer();
p.sangreCaliente;
p.parir();
p.amamantar();
p.colorPelo;
p.ladrar();
```

Selección dinámica de métodos

En caso de que hubiera sobreescritura de métodos, se ejecutaría el código del método más especializado.

Por ejemplo, si el método `comer()` está definido en la clase Animal y se sobreescribe en la clase Perro.

```
Perro p = new Perro();
Mamifero m = p;
Animal a = p;

p.comer();     // Se ejecuta el código del método comer() de la clase Perro.
m.comer();     // Se ejecuta el código del método comer() de la clase Perro porque
               // aunque m es del tipo Mamifero, referencia a un objeto de la clase Perro
a.comer();     // Por el mismo motivo. Se ejecuta el código del método comer() de la clase Perro.
```

getClass( )

El método getClass() es un método final (no puede sobreescribirse) que devuelve una representación en tiempo de ejecución de la clase del objeto.

Este método **devuelve un objeto Class** al que se le puede pedir información sobre la clase, como su nombre, el nombre de su superclase y los nombres de los interfaces que implementa.

```
Object p = new Perro();
Class clase = p.getClass();
System.out.println(clase);            // class paquete.Perro
Class clase2 = clase.getSuperclass();
System.out.println(clase2);           // class paquete.Mamifero
Class clase3 = clase2.getSuperclass();
System.out.println(clase3);           // class paquete.Animal
Class clase4 = clase3.getSuperclass();
System.out.println(clase4);           // class java.lang.Object
```

getClass( ) para comparar

Se puede usar **`getClass()`** para comparar si una instancia es de una clase concreta.

```
Object o = new Perro();
if ( o.getClass().equals(Perro.class) ) {
      Perro p = (Perro) o;
      p.ladrar();
}
```

Pero tiene que ser igual a la clase exacta.

```
o.getClass().equals(Perro.class);          // true
o.getClass().equals(Mascota.class);        // false
o.getClass().equals(Animal.class);         // false
```

Si queremos saber si es un Animal, sea cual sea su especialización, tendremos que ir haciendo varias comprobaciones. En estos casos es mejor usar **`instanceof`**

instanceof

El operador **instanceof** toma como primer operando una variable referencia y como segundo un tipo, según la siguiente sintaxis:

```
referencia instanceof Tipo
```

Al lado izquierdo tenemos la instancia y el lado derecho el nombre de la clase y devuelve un resultado booleano. True si el objeto al que apunta una referencia dada es una instancia de una clase o hereda de la misma.

```
Animal[] listado = new Animal[10];
listado[0] = new Animal();
listado[1] = new Mascota();
listado[2] = new Perro();
…
listado[9] = new Delfin();

for (int i=0; i<listado.length; i++) {
   if (listado[i] instanceof Perro) {
       Perro p = (Perro) listado[i];
       p.ladrar();
    }
}
```

Clases Abstractas

Una clase abstracta es aquella que **no va a tener instancias de forma directa**, aunque sí habrá instancias de las subclases (siempre que esas subclases no sean también abstractas).

Por ejemplo, si se define la clase Animal como abstracta, no se podrán crear objetos de la clase Animal, es decir, no se podrá hacer

```
Animal animal = new Animal()
```

pero sí se podrán crear instancias de la clase Oviparo, Mascota, Delfin o Perro.

Métodos abstractos

Para que un método se considere abstracto ha de incluir la palabra clave abstract. Tiene estas peculiaridades:

- No tiene cuerpo (llaves): sólo consta de la definición con los parámetros, pero no el código.
- Sólo puede existir dentro de una clase abstracta. De esta forma se evita que haya métodos que no se puedan ejecutar dentro de clases concretas. Visto de otra manera, si una clase incluye un método abstracto, forzosamente la clase será una clase abstracta.
- Los métodos abstractos forzosamente habrán de estar sobreescritos en las subclases. Si una subclase no implementa un método abstracto de la superclase tiene un método no ejecutable, lo que la fuerza a ser una subclase abstracta. Para que la subclase sea concreta habrá de implementar métodos sobreescritos para todos los métodos abstractos de sus superclases.

Un método abstracto para Java es un método que nunca va a ser ejecutado porque no tiene cuerpo. Simplemente, un método abstracto referencia a otros métodos de las subclases.