PROGRAMACIÓN ORIENTADA A OBJETOS CON JAVA (II): HERENCIA.

6. HERENCIA

La herencia representa uno de los conceptos más importantes y potentes de la Programación Orientada a Objetos.

6.1. Concepto de herencia

Podemos definir la herencia como la capacidad de crear clases que adquieran de manera automática los miembros (atributos y métodos) de otras clases que ya existen, pudiendo al mismo tiempo añadir atributos y métodos propios.

6.2. Ventajas de la herencia

Entre las principales ventajas que ofrece la herencia en el desarrollo de aplicaciones, están:

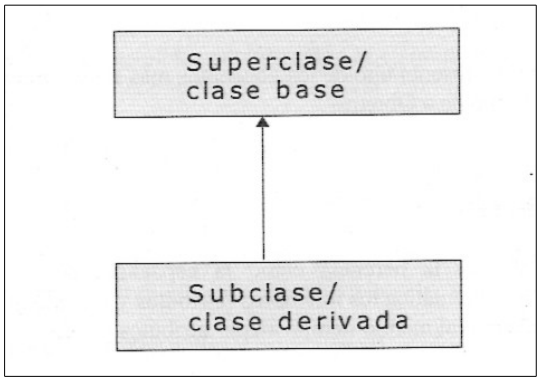
- **Reutilización de código.** En aquellos casos donde se necesite crear una clase que, además de otros propios, deba incluir los métodos definidos en otra, la herencia evita tener que reescribir todos esos métodos en la nueva clase.

- **Mantenimiento de aplicaciones existentes.** Utilizando la herencia, si tenemos una clase con una determinada funcionalidad y tenemos la necesidad de ampliar dicha funcionalidad, no necesitamos modificar la clase existente (la cual se puede seguir utilizando para el tipo de programa para la que fue diseñada) sino que podemos crear una clase que herede a la primera, adquiriendo toda su funcionalidad y añadiendo la suya propia.

  Por ejemplo, dada la clase `Punto` podríamos crear a través de la herencia una nueva clase, llamada `PuntoColor`, que adquiriese las coordenadas *x* e *y* como atributos propios y además pudiera añadir algunos adicionales, como el color.

6.3. Nomenclatura y reglas

Antes de ver cómo se crean clases en Java utilizando herencia, vamos a definir la nomenclatura básica y a conocer ciertas reglas a tener en cuenta sobre la herencia.

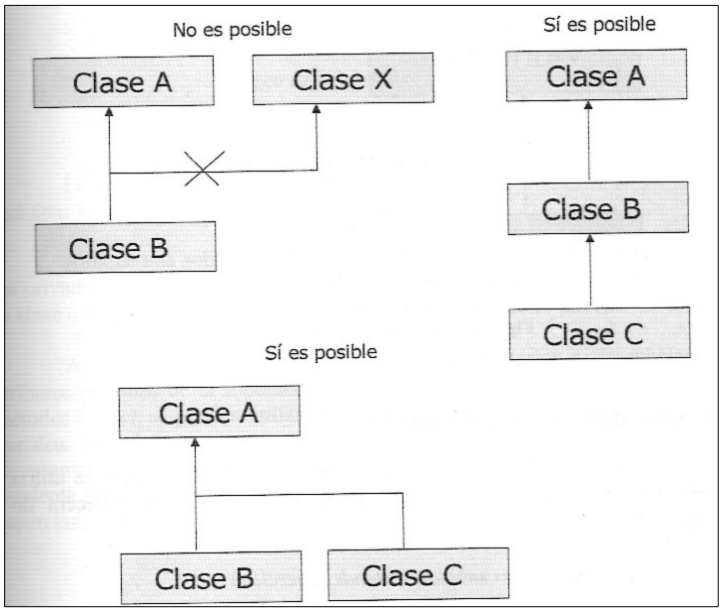


Relación de herencia

En POO, a la clase que va a ser heredada se la llama **superclase o clase base,** mientras que a la que hereda se la conoce como **subclase o clase derivada.** Gráficamente, la herencia entre dos clases se representa con una flecha saliendo de la subclase hacia la superclase.

Hay unas reglas básicas sobre la herencia en Java que hay que tener presentes y que quedan ilustradas en la figura que se presenta a continuación.

- En Java no está permitida la herencia múltiple, es decir, una subclase no puede heredar más de una clase.
- Sí es posible una herencia multinivel, es decir, A puede ser heredada por B y C puede heredar B.
- Una clase puede ser heredada por varias clases.



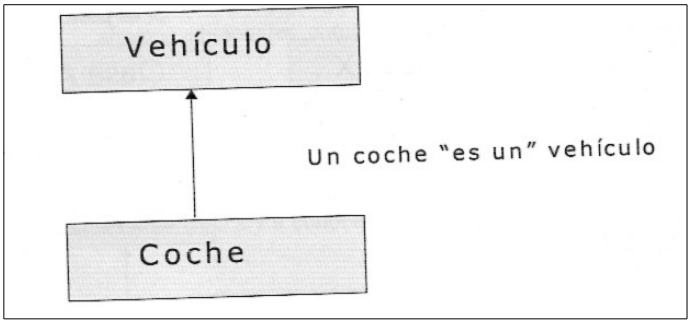
Relaciones de herencia posibles y no posibles.

6.4. Relación "Es un"

La herencia entre dos clases establece una relación entre las mismas de tipo "Es un", lo que significa que un objeto de una subclase también "es un" objeto de la superclase.

Por ejemplo, Vehículo es la superclase de Coche, por lo que un coche "es un" vehículo . De la misma forma, Animal es la superclase de Mamífero y ésta es a su vez superclase de León, esto nos lleva a que un León "es un" mamífero y "es un" animal.

Así pues, una forma de saber si una relación de herencia entre dos clases está bien planteada es comprobar si se cumple la relación "Es un" entre la subclase y la superclase. Por ejemplo, para crear una clase `Linea` podríamos intentar heredar `Punto` pensando que es una subclase de ésta, sin embargo ¿una línea "es un" punto? la respuesta es NO, por lo que la herencia está mal planteada.



Relación "es un" entre subclase y superclase.

6.5. Creación de herencia en Java

A la hora de definir una clase que va a heredar otra clase, se utiliza la palabra ***extends,*** seguida del nombre de la superclase en la cabecera de la declaración:

```
public class subclase extends superclase {
      // código de la subclase
}
```

La nueva clase, podrá incluir atributos y métodos propios para completar su función. Por ejemplo, la clase `PuntoColor` heredaría de la clase `Punto` para adquirir las coordenadas *x* e *y*, y además incluiría el atributo `color`:

```
public class PuntoColor extends Punto {
      private String color;
      // resto de la clase
}
```

Todas las clases de Java heredan alguna clase. En caso de que no se especifique mediante `extends` la clase que se va a heredar, implícitamente se heredará de la clase **`Object`**. Esta clase se encuentra en el paquete `java.lang` y proporciona el soporte básico para cualquier clase Java. Así pues, la definición de una clase que no herede explícitamente de otra, equivale a:

```
public class NombreClase extends Object {
      //código de la clase
}
```

La clase `Object` es, por tanto, la superclase de todas las clases de Java, incluidas las del API Java SE y Java EE.

Aunque una subclase hereda todos los miembros de la superclase, incluidos los privados, no tiene acceso directo a éstos, puesto que `private` significa **privado a la clase** o lo que es lo mismo, **solamente accesibles desde el interior de ésta.**

Así pues, ¿cómo se puede acceder desde el interior de la subclase a los atributos privados de la superclase? En el caso de que la superclase disponga de métodos ***set/get***, se pueden utilizar directamente desde la subclase al ser heredados por ésta. No obstante, de cara a la inicialización de atributos, los constructores representan la mejor opción para acceder a los datos miembro de la clase; en el siguiente apartado veremos cómo podemos hacer uso de los constructores de la superclase desde la subclase.

La clase Punto es:

```
public class Punto {
    private int x,y;

    //constructor por defecto
    public Punto(){
    }

    public Punto(int x, int y){
        this.x = x;
        this.y = y;
    }

    public Punto(int v){
        this.x = v;
        this.y = v;
    }

    public int getX(){
        return x;
    }

    public int getY() {
        return y;
    }

    public void setX(int x) {
        this.x = x;
    }

    public void setY(int y) {
        this.y = y;
    }

    public void dibujar() {
        System.out.println("Coordenadas: "+x+","+y);
    }
}
```

6.6. Ejecución de constructores con la herencia

Hay que hacer especial mención al comportamiento de los constructores de la superclase y subclase cuando se va a crear un objeto de esta última.

Como norma universal, cada vez que en Java se crea un objeto de una clase, **antes de ejecutarse el constructor de dicha clase se ejecutará primero el de su superclase.** Según esto, tras la ejecución del método `main()` del siguiente programa:

```
class Primera {
  public Primera() {
    System.out.println("Constructor de la superclase");
  }
}

class Segunda extends Primera {
  public Segunda() {
    System.out.println("Constructor de la subclase");
  }
}

public class Principal {
  public static void main(String[] args) {
    Segunda s = new Segunda();
  }
}
```

Aparecerá en pantalla lo siguiente:

```
Constructor de la superclase
Constructor de la subclase
```

La explicación a esta situación la tenemos en el hecho de que el compilador Java añade, como primera línea de código en todos los constructores de una clase, la siguiente instrucción:

```
super();
```

Instrucción que provoca una llamada al constructor sin parámetros de la superclase.

Los constructores por defecto también incluyen esta instrucción, así pues, el aspecto real de estos constructores es:

```
public NombreClase() {
      super();
}
```

Aquellas clases que no hereden de ninguna otra también incluirán esta instrucción como primera línea de código en sus constructores, puesto que, como hemos comentado anteriormente, toda clase que no herede explícitamente otra clase heredará implícitamente la clase `java.lang.Object`.

Si en vez de llamar al constructor sin parámetros quisiéramos invocar a cualquier otro constructor de la superclase, deberíamos hacerlo explícitamente, añadiendo como **primera línea** del constructor de la subclase la instrucción:

```
super(argumentos);
```

Los *argumentos* son los parámetros que necesita el constructor de la superclase que se desea invocar. Esto permite, entre otras cosas, que el constructor de la subclase pueda pasarle al constructor de la superclase los datos necesarios para la inicialización de los atributos privados y que no son accesibles desde la subclase, tal y como se muestra en el siguiente listado de la clase `PuntoColor`:

```
public class PuntoColor extends Punto {
  private String color;

  // contructor propio
  public PuntoColor(int x, int y, String c) {
    super(x, y); // constructor de Punto (superclase)
    color = c; // inicialización de atributos propios
  }
}
```

Es necesario volver a recalcar que la llamada al constructor de la superclase, **debe ser la primera línea de código del constructor de la subclase,** de no hacerse así se producirá un error de compilación.

Otra posibilidad es que, en vez de incluir una llamada al constructor de la superclase, se desee invocar a otro de los constructores de su clase. En este caso, se utilizará la palabra reservada *this* en vez de *super*, siendo el constructor invocado el que incluirá la llamada al constructor de la superclase:

```
public class PuntoColor extends Punto {
  private String color;

  public PuntoColor(int x, int y, String c) {
    super(x, y) ;
    color = c;
  }

  public PuntoColor (int cord, String c) {
    // otro constructor de PuntoColor
    this(cord, cord, c);
  }

  …
}
```

6.7. Métodos y atributos protegidos

Existe un modificador de acceso aplicable a atributos y métodos de una clase, pensado para ser utilizado con la herencia, el modificador `protected`.

Un miembro de una clase (atributo o método) definido como `protected`, será accesible desde cualquier subclase de ésta, independientemente de los paquetes en que estas clases se encuentren.

Por ejemplo, dada la siguiente definición de una clase `Persona`:

```
package basico;

public class Persona {

  private String fechaNacimiento;

  public Persona(String f) {
    fechaNacimiento = f;
  }

  protected int getEdad(){
    //implementación del método
  }
}
```

Una subclase de `Persona` definida en otro paquete podrá hacer uso del método `getEdad()`:

```
package varios;

import basico.Persona;

public class Empleado extends Persona {

  private int edad;
  private long nSeg;

  public Empleado(String fechaNacimiento, long nSeg){
    super(fechaNacimiento);
    this.nSeg = nSeg;
  }

  public void mostrarDatos(){
    // Acceso al método protegido
    System.out.println( "Edad :" + this.getEdad());
    System.out.println("NISS: " + nSeg);
  }
}
```

Importante: las subclases acceden a los miembros protegidos a través de la herencia, **no pudiendo utilizar una referencia a un objeto de la superclase para acceder al miembro protegido.** Por ejemplo, la siguiente versión del método `mostrarDatos()` en la clase `Empleado` no compilaría:

```
public void mostrarDatos(){
  Persona p = new Persona("3/11/05");

  // la siguiente instrucción produce error porque se
  // está intentando acceder a un miembro protegido
  // desde otro paquete sin usar herencia.
  System.out.println("Edad: " + p.getEdad());
  System.out.println("NISS: " + nSeg) ;
}
```

El modificador `protected`, indica que los elementos sólo pueden ser accedidos desde su mismo paquete (como el acceso por defecto) y desde cualquier clase que extienda la clase en que se encuentra, independientemente de si esta se encuentra en el mismo paquete o no.

6.8. Clases finales

Si se quiere evitar que una clase sea heredada por otra, deberá ser declarada con el modificador `final` delante de `class`:

```
public final class ClaseA {

}
```

Si otra clase intenta heredar una clase final se producirá un error de compilación:

```
// la siguiente clase no compilará
public class ClaseB extends ClaseA {

}
```

6.9. Sobrescritura de métodos

Cuando una clase hereda a otra, el comportamiento de los métodos que hereda no siempre se ajusta a las necesidades de la nueva clase. Por ejemplo, el método `dibujar()` que hereda la clase `PuntoColor` no se ajusta del todo a sus necesidades, ya que no tiene en cuenta el color a la hora de dibujar el punto.

En estos casos, la subclase puede optar por **volver a reescribir el método heredado,** es lo que se conoce como **sobrescritura** de un método.

A la hora de sobrescribir un método hay que tener en cuenta las siguientes reglas:

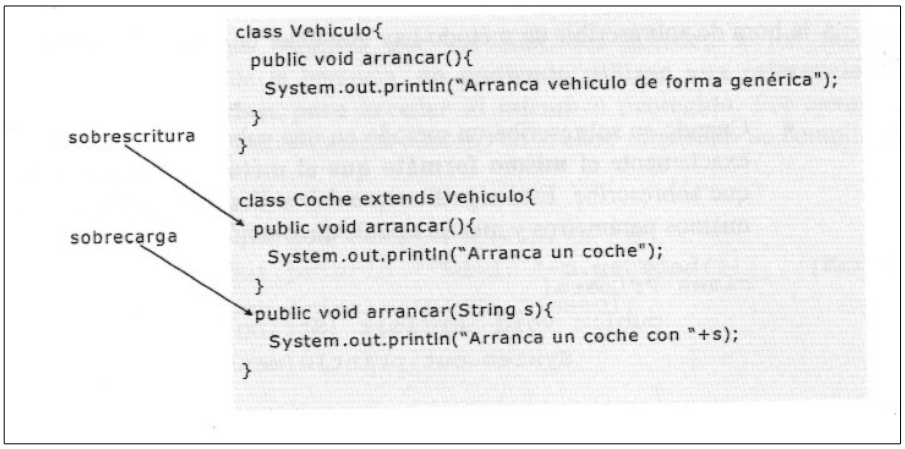
- **Cuando se sobrescribe un método en una subclase, éste debe tener exactamente el mismo formato que el método de la superclase que sobrescribe. Esto significa, que deben llamarse igual, tener los mismos parámetros y mismo tipo de devolución.**

```
class Primera {
  public void imprimir(String mensaje) {
    System.out.println(mensaje);
  }
}

class Segunda extends Primera {
  // Se mantiene la firma original del método
  public void imprimir(String mensaje) {
    // nuevo código del método imprimir()
    System.out.println("El mensaje es: ");
    System.out.println(mensaje);
  }
}
```

Hay que tener presente que si al intentar sobrescribir un método en una subclase se mantiene el mismo nombre y se modifican los parámetros, el nuevo método no sobrescribe al de la superclase pero tampoco se produce un error de compilación, ya que estaríamos ante un caso de **sobrecarga de métodos**: dos métodos con el mismo nombre y distintos parámetros.

La figura siguiente muestra la diferencia entre ambos conceptos, presentando un caso de Sobrescritura y sobrecarga de un mismo método en una subclase.



Sobrescritura y sobrecarga de un método

- El método sobrescrito puede tener un modificador de acceso menos restrictivo que el de la superclase. Por ejemplo, el método de la superclase puede ser `protected` y la versión sobrescrita en la subclase puede ser `public`, pero nunca uno más restrictivo.
- Para llamar desde el interior de la subclase a la versión original del método de la superclase, debe utilizarse la expresión:

super. *nombreMetodo (argumentos);*

- Si no se utiliza `super` delante del nombre del método, se llamará a la versión sobrescrita en la clase.

En el siguiente código se muestra la clase PuntoColor al completo. En ella se sobrescribe el método `dibujar()` de `Punto`, proporcionando una versión adaptada para los objetos `PuntoColor`:

```
public class PuntoColor extends Punto {
  private String color;
  public PuntoColor(int x, int y, String c) {
    super(x, y);
    color = c;
  }
  public PuntoColor(int cord, String c) {
    //invoca al otro constructor de la clase
    this(cord, cord, c);
  }
  public String getColor() {
    return color;
  }
  @Override
  public void dibujar() {
    super.dibujar(); // versión definida en Punto
    System.out.println("Color: " + this.color);
  }
}
```

La anotación `@Override` es opcional, pero es muy útil a la hora de evitar errores en tiempo de ejecución que son difíciles de detectar, puesto que fuerza al compilador a comprobar en tiempo de compilación que se está sobreescribiendo correctamente un método de la superclase, en caso contrario el compilador generará un error.

Si no se usa esta anotación, se puede caer en el fallo de sobrecargar el método en vez de sobreescribir el de la superclase (si se cambian los parámetros) o crear un método totalmente nuevo si por error se escribe su nombre de forma incorrecta.

Por otro lado, si se quiere evitar que un método pueda ser sobrescrito desde una subclase deberá declararse con el modificador `final`:

```
public final void metodo(){

}
```