U7 Herencia y polimorfismo. Interfaces (v1.6.1)

Introducimos en este tema dos nuevos conceptos fundamentales en la POO, además de la abstracción y la encapsulación, como son la herencia y el polimorfismo.

Recordemos que la abstracción es el elemento que permite combatir la complejidad, extrae los aspectos esenciales de un objeto (de una entidad) ignorando los detalles.

El encapsulamiento agrupa los elementos de una abstracción que constituyen su estructura y comportamiento. La clase "firma un contrato con el exterior" comprometiéndose a exhibir un determinado comportamiento pero cómo lo consigue queda oculto o encapsulado en el interior de la clase.

La herencia y el polimorfismo permiten mejorar la estructura general de los programas. La herencia es el mecanismo fundamental de la POO que ayuda a fomentar y facilitar la reutilización del software. El polimorfismo, que nace de la interacción de la herencia, el enlace dinámico (propio de los lenguajes orientados a objetos) y la compatibilidad de tipos, permiten la escritura de código genérico.

Por último, el uso de interfaces explota el polimorfismo en su grado máximo. Diseñando y codificando para especificaciones de interfaces conseguimos un alto grado de flexibilidad y extensibilidad.

7.1.- La herencia. Características.

La ***herencia*** es el mecanismo de la POO que nos permite definir una clase como extensión de otra, de tal forma que la nueva clase *hereda* toda la estructura de la clase inicial y su comportamiento.

La clase de la que se hereda recibe el nombre de c*lase base, superclase o clase padre*.

La clase que se obtiene como extensión se llama s*ubclase, clase derivada o clase hija.*

El objetivo fundamental de la herencia es la reutilización del código. Se utilizan los componentes software que ya existen para construir los nuevos, ahorrando esfuerzo de diseño, implementación y test del software que ya existe.

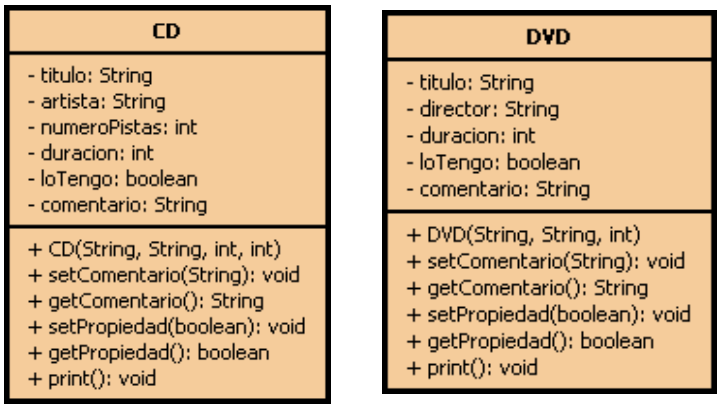
Imaginemos que queremos realizar una aplicación que cree un catálogo (una base de datos) con información de los CDs y DVDs que poseemos.

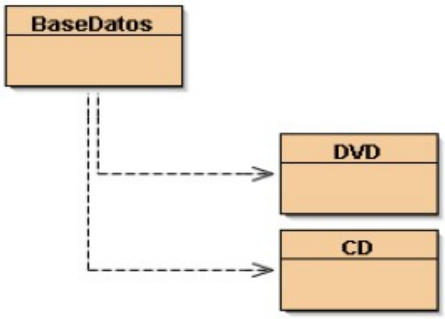
De cada CD queremos guardar:

- título
- el artista
- nº pistas del CD
- duración
- un indicador que marca si lo tenemos o no
- un comentario

De cada DVD se almacenará:

- título
- nombre del director
- duración
- un indicador que marca si tenemos una copia o no
- un comentario



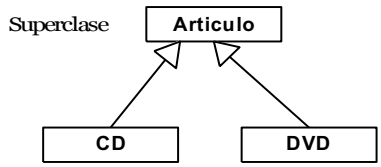



```
public class BaseDatos
        private ArrayList<CD> cds;
        private ArrayList<DVD> dvds;

        public BaseDatos()
        {
            cds = new ArrayList<CD>();
            dvds = new ArrayList<DVD>();
        }

        public void addCD(CD elCD)
        {
            cds.add(elCD);
        }

        public void addDVD(DVD elDVD)
        {
            dvds.add(elDVD);
        }
```

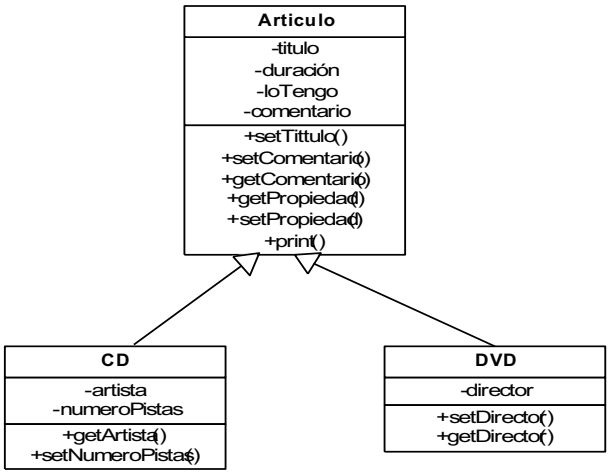
```
            public void listar()
            {

                for (CD cd : cds)
                {
                      cd.print();
                      System.out.println();
                }


                for (DVD dvd : dvds)
                {
                      dvd.print();
                      System.out.println();
                }
            }
}
```

El código fuente de las clases CD y DVD es muy similar puesto que las dos contienen atributos idénticos. Esto hace el mantenimiento más difícil y trabajoso y con el peligro de introducir errores. La clase BaseDatos además, que guarda una colección de CDs y DVDs, repite dos veces el código, una para el CD y otra para el DVD.

En lugar de escribir dos clases independientes, CD y DVD, cada una con sus propios atributos y métodos, puesto que ambas poseen características comunes se puede construir una superclase Articulo que incluya lo que es común a las dos y a partir de ésta, derivar dos subclases, CD y DVD, que heredarán todo lo que incluya la clase Articulo y añadirán cada una lo que es propio de ellas.

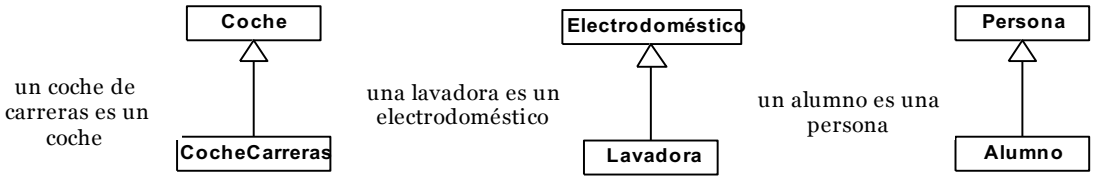


En UML la relación de herencia se representa con una línea continua que conecta la clase padre e hija y la punta de flecha apuntando a la clase padre



Una subclase es una especialización (es un caso especial) de la superclase (un CD es un Artículo, un DVD es un Artículo).

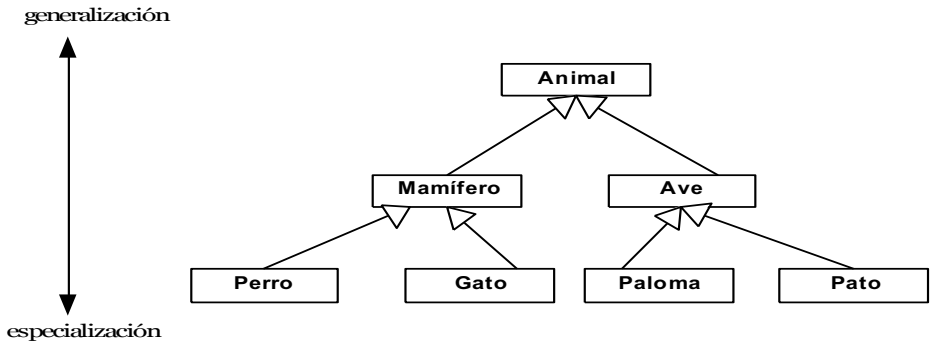
La superclase expresa la estructura y comportamiento comunes a las subclases.



Habitualmente una subclase aumenta o redefine la estructura y el comportamiento de la superclase (añade o modifica lo que hereda de su clase padre). Por ejemplo, una superclase Reloj tiene como atributo la hora y métodos para poner la hora y obtenerla. La clase RelojDespertador aumenta el comportamiento de la clase Reloj ya que es capaz de emitir una alarma a una hora prefijada. Esta clase aumenta la estructura con un atributo adicional, la hora de alarma, otro atributo para indicar si está o no activada y los métodos que permiten fijar la alarma y activarla / desactivarla.

Las instancias de las subclases tienen todos los atributos y métodos de la superclase y los nuevos que añaden o que hayan redefinido.

A través de las relaciones de herencia se pueden formar jerarquías de *generalización/ especialización.*



Cuanto más arriba en la jerarquía las clases son más generales, más abstractas. Cuanto más abajo en la jerarquía las clases se especializan, son más concretas. Cuanto más arriba menor nivel de detalle, cuanto más abajo más detalle.

La herencia es una técnica de abstracción que nos permite categorizar las clases de objetos bajo cierto criterio.

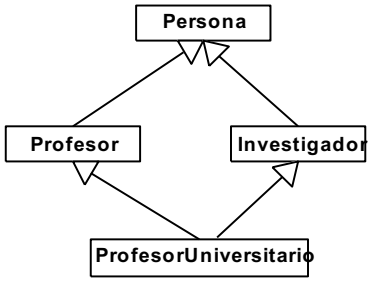
Dibuja una jerarquía de herencia que permita clasificar al personal que trabaja en el instituto (estudiantes, profesores, administrativos, ...)

7.1.1.- Herencia simple y múltiple.

Se pueden distinguir distintos tipos de herencia teniendo en cuenta el nº de superclases (padres) de una subclase:

a) *herencia simple* – una subclase hereda solo de una superclase

b) *herencia múltiple* – una clase hereda de más de una superclase



Java no permite la herencia múltiple, sólo herencia simple.

7.1.2.- Implementación de la herencia en Java.

En Java la relación de herencia se expresa mediante de la palabra clave ***extends***:

```
public class Articulo
{
   private String titulo;
   private int duracion;
     ..............................

}
```

```
public class CD extends Articulo
{
   private String artista;
   private int numeroPistas;
     ..............................

}
```

La clase Articulo define los atributos comunes, la clase CD *extiende* (hereda) todo lo que tiene la clase Articulo y añade dos atributos propios.

Define la clase DVD (incluye solo los atributos).

Define una clase Punto cuyos atributos son *x* e *y* que marcan las coordenadas del punto. Incluye el constructor con parámetros, accesores y mutadores y un método *toString()*.

La clase PuntoTresDimensiones deriva de la clase Punto y añade un nuevo atributo *z*. Escribe en Java el esqueleto de la clase, solo los atributos. Dibuja el diagrama UML que muestra la jerarquía. ¿Cuántos atributos tiene la clase PuntoTresDimensiones?

7.1.3.- Herencia y derechos de acceso.

Los miembros de una clase (atributos y métodos) definidos como públicos son accesibles a los objetos de otras clases. No ocurre así con los miembros privados.

Esta regla de privacidad se aplica también entre una subclase y su superclase: una subclase no puede acceder a los miembros privados de su superclase. Si un método de la subclase necesita acceder a los atributos privados heredados de la superclase, ésta debe proporcionar accesores y/o mutadores o bien definir el atributo (o método) como ***protected***.

El siguiente cuadro muestra el nivel de acceso permitido por cada especificador de acceso:

Especificador	clase	subclase	paquete	el resto (el mundo)
private	x			
protected	x	x(*)	x	
public	x	x	x	x
package	x		x	

(*) acceso desde las subclases aunque no estén en el mismo paquete

7.1.4.- Herencia y constructores.

Los constructores en Java no se heredan.

En Java, el constructor de una subclase debe llamar siempre al constructor de la superclase y lo hace incluyendo como su primera sentencia una llamada al constructor de su clase padre. Esto se hace con la palabra clave ***super***. Es importante recordar que *debe ser la primera sentenci*a en el constructor de la subclase.

La palabra clave *super* referencia a la superclase de la clase en la cual aparece *super*. Uno de los usos de *super* es para llamar al constructor de la superclase, otro para llamar a un método de la superclase.

Si no hay una llamada explícita al constructor de la superclase, el compilador Java inserta automáticamente una llamada para asegurar que los atributos heredados de la superclase se inicializan adecuadamente. **Esta inserción automática sólo funciona si la superclase tiene un constructor sin parámetros.** Es conveniente incluir llamadas explícitas incluso en este último caso.

```
public class Articulo
{
            private String titulo;
            private int duracion;
            private boolen loTengo;
            private String comentario;

            public Articulo(String titulo, int duracion)
            {
                  this.titulo = titulo;
                  this.duracion = duracion;
                  loTengo =  false;
                  comentario = "";
          }
}
public class CD extends Articulo
{
            private String artista;
            private int numeroPistas;

            public CD(String titulo, int duracion, String artista, int pistas )
            {
                  super(titulo, duracion);
```

```
            this.artista = artista;
            numeroPistas = pistas;
        }
    }
```

El constructor de la clase CD recibe los parámetros necesarios para inicializar los atributos heredados de Articulo y los nuevos que incorpora CD. La primera línea del constructor: ***super(titulo, duracion);*** es la llamada al constructor de la superclase. Se pasan tantos parámetros a *super()* como se hayan definido en el constructor de la clase padre.

A través de *super()* se pueden invocar otros métodos de la clase padre, no solo los constructores (lo veremos luego al redefinir los métodos).

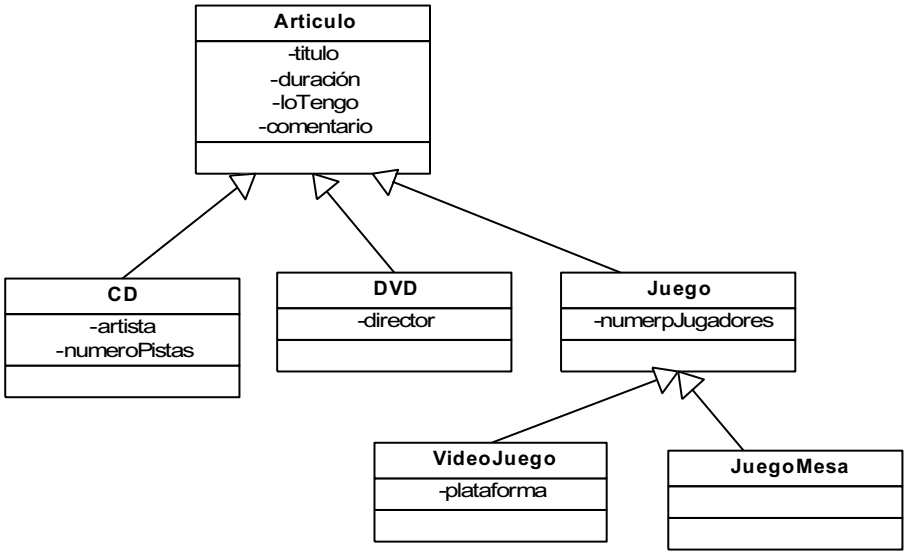
Escribe el constructor de la clase DVD y el de la clase PuntoTresDimensiones.

Un libro se caracteriza por su título, fecha de publicación, autor e ISBN. Una revista, como un libro, tiene un título y una fecha de publicación pero se caracteriza además por la periodicidad con la que se publica. Define una relación de herencia entre clases que modelan objetos como los descritos y una clase adicional Publicación que captura las características comunes de ambas publicaciones. Dibuja el diagrama UML y define el esqueleto de las clases que intervienen en le jerarquía. Además de los atributos, define los constructores.

7.1.5.- Ventajas de la herencia

Las principales ventajas de la herencia en la POO son:

- *evitar la duplicación de código* – el uso de la herencia evita la necesidad de escribir clases parecidas
- *reutilizar el código* – el código existente puede ser reutilizado. Si una clase similar a otra que necesitamos ya existe podemos "*extender*" de la clase existente la nueva clase en lugar de implementar todo otra vez.
- *hacer más fácil el mantenimiento* – un cambio, por ejemplo, en un atributo o método compartido en una relación de herencia solo se hace una vez
- *extensibilidad* – utilizando la herencia es más fácil extender una aplicación existente



Escribe la clase Juego (atributos y constructor). Idem con la clase VideoJuego y JuegoMesa.

Evitando la duplicación de código que se producía cuando no habíamos definido la superclase Artículo, el nuevo código de la clase BaseDatos que guarda la colección, tanto de CDs como de DVDs, quedaría:

```
import java.util.ArrayList;

public class BaseDatos
{
        private ArrayList<Articulo> articulos;

        /**
         * Construir una base de datos vacía
         */
        public BaseDatos()
        {
           articulos = new ArrayList<Articulo>();
        }

        /**
         * Añadir un artículo a la base da datos
         */
        public void addArticulo(Articulo
                                elArticulo)
        {
           articulos.add(elArticulo);
        }

        public void listar()
        {
                for (Articulo a: articulos)
                {
                   a.print();
                   System.out.println();
                }
        }

}
```

Antes teníamos:

```
public void addCD(CD elCD)
public void addDVD(DVD elDVD)
```

Ahora tenemos:

```
public void addArticulo(Articulo elArticulo)
```

Los artículos se añadirán a la base de datos, miBaseDatos por ej, así:

```
DVD miDVD = new DVD();
miBaseDatos.addArticulo(miDVD);
```

Define una clase Cuenta que modela una cuenta bancaria. Una cuenta bancaria se caracteriza por el nombre del cliente que posee la cuenta y el importe de la misma. Define constructor con parámetros, accesores y los mutadores *ingresar()* y *reintegrar()* que permiten añadir y sacar una determinada cantidad. Incluye un método *toString()*.

Una cuenta de ahorro es una cuenta bancaria con un tipo de interés aplicado. Define la clase CuentaAhorro. Incluye constructor con parámetros y el método *aplicarInteres()* que devuelve la cantidad que representa el interés de la cuenta.

Una cuenta corriente es una cuenta bancaria a la que se le aplica un recargo si el importe es menor que el importe mínimo exigido. Define la clase CuentaCorriente. Incluye constructor. accesor para el recargo (será 0 el recargo si la cuenta mantiene el importe mínimo).

Define *miCuenta* de tipo CuentaAhorro. Añade y reintegra alguna cantidad y calcular el interés. Visualiza los datos de la cuenta . ¿Qué se mostraría en pantalla?

Define tu*Cuenta* de tipo CuentaCorriente. Añade y reintegra alguna cantidad y calcula el recargo aplicado. Visualiza los datos de la cuenta . ¿Qué se mostraría en pantalla?

Define la relación "*un aula taller es un aula"* teniendo en cuenta que un aula se caracteriza por su nombre y el nº de pupitres que hay en ella y un aula taller además por un nº de ordenadores. Incluye en las clases constructor, accesores y mutadores. La clase padre de la jerarquía además proporciona un método *mostrar()*. La clase hija otro método *visualizar()* que muestra los datos del aula.

7.2.- Herencia y subtipos.

Las clases definen tipos.

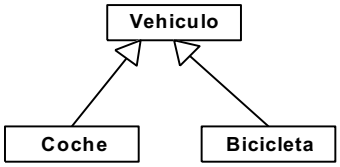
Cuando existe una relación de herencia, de la misma forma que hay una jerarquía de clases, existe una jerarquía de tipos. El tipo definido por una subclase es un *subtipo* del tipo definido por su superclase.

Una variable puede referenciar objetos de su tipo declarado y objetos de cualquier subtipo de su tipo declarado. El tipo declarado de una variable es su *tipo en tiempo de compilación*.

En los lenguajes orientados a objetos hay un principio, conocido como ***principio de sustitución*** que dice: *allí dónde se espere un objeto de la superclase puede aparecer (ser sustituido por) un objeto de la subclase.*

A la inversa no es válido.

Supongamos la siguiente jerarquía de herencia:



Sabemos que es correcto: `Coche miCoche = new Coche();`

La variable *miCoche* es de tipo Coche (éste es el tipo estático) y está referenciando a un objeto de tipo Coche.

Pero también es correcto,

```
Vehiculo unVehiculo, miVehiculo, tuVehiculo;
unVehiculo = new Vehiculo();
miVehiculo = new Coche();
tuVehiculo = new Bicicleta();
```

La variable *miVehiculo* tiene un tipo estático Vehiculo, pero en ejecución, una vez instanciado el objeto, la variable apunta realmente a un objeto de tipo Coche. En compilación las variables tienen un tipo declarado (tipo estático), en ejecución puede tener otro tipo (tipo dinámico), el tipo del objeto al que referencia, que será un objeto de alguna subclase.

El tipo de la variable declara lo que puede almacenar. Si se declara de tipo Vehiculo la variable podrá referenciar vehículos. Ya que un coche *es un* vehículo es correcto que pueda referenciar a un objeto de tipo Coche, por ejemplo.

Responder a las siguientes cuestiones:

a) Es correcto? Coche unCoche = new Vehiculo();
b)

```
public class Gato          public class GatoConBotas extends Gato
{                          {


}                          }
```

Es correcto? Gato g = new Gato();
GatoConBotas felix = new GatoConBotas();
GatoConBotas miGato = new Gato();

Representa a través de un diagrama de clases UML la relación de herencia entre las clases: Profesor, Persona, EstudiantePrimerCurso, Estudiante.

Indica cuál de las siguientes asignaciones es correcta y por qué?

```
Persona p1 = new Estudiante();
Persona p2 = new EstudiantePrimerCurso();
EstudiantePrimerCurso  estu1 = new Estudiante();
Profesor prof1 = new Persona();
Estudiante estu2 = new EstudiantePrimerCurso();

estu2 = p1;
estu1 = p2;
p1 = estu1;
prof1 = estu1;
estu2 = estu1;
```

7.2.1.- Subtipos y paso de parámetros. Subtipos y valores de retorno.

El principio de sustitución es aplicable cuando se efectúa un paso de parámetros, cuando se asocia un parámetro actual con uno formal. El comportamiento es el mismo que en una asignación. Se puede pasar un objeto del tipo de una subclase a un método que tiene como parámetro un objeto del tipo de la superclase.

Ocurre lo mismo con los tipos de retorno, por el principio de sustitución es posible devolver una valor de un tipo de una subclase de la clase indicada en el tipo de retorno.

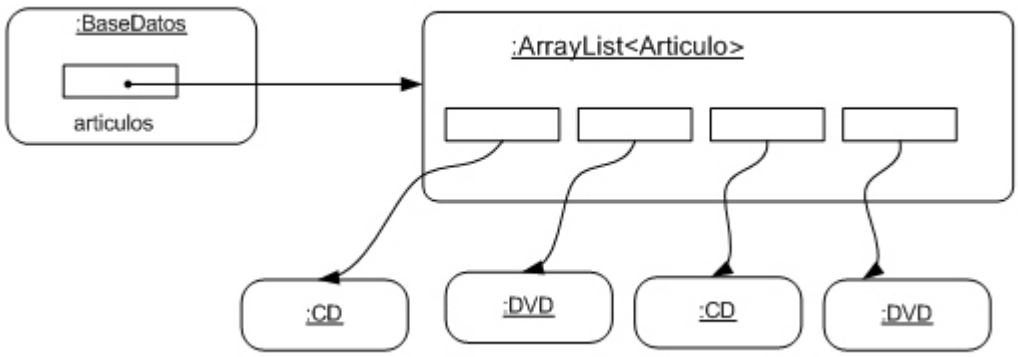
```
public  Articulo  getArticulo(int i)
{
        if (i >= 0 && i < articulos.size())
                return articulos.get(i);        // Se devolverá un CD o un DVD
        return nulll;
}
```

Ej. La clase BaseDatos almacena una relación de artículos, de tipo CD y DVD, y permite obtener una lista de todos ellos, así como añadir un nuevo artículo.

```
import java.uyil.ArrayList;

public class BDMultimedia
{
        private ArrayList<Articulo> articulos;

        .....

        public void addArticulo(Articulo elArticulo)
        {
                articulos.add(elArticulo);
        }

        public void listar()
        {
                for (Articulo a: articulos)
                {
                        a.print();
                        System.out.println();
                }
        }
```

Es posible hacer:

```
BaseDatos  miBaseDatos = new BaseDatos();
DVD  dvd11 = new DVD(.....);
miBaseDatos.addArticulo(dvd1);
CD cd1 = new CD(....);
miBaseDatos.addArticulo(cd1);
.....
```



Sea el método: public void limpiarGato(Gato g) y las declaraciones del ejercicio anterior del gato. ¿Es correcto ?

a) limpiarGato(felix);

b) limpiarGato(g);

7.2.2.- Variables polimórficas

El polimorfismo en los lenguajes orientados a objetos aparece en diferentes contextos. Las variables polimórficas es un primer ejemplo.

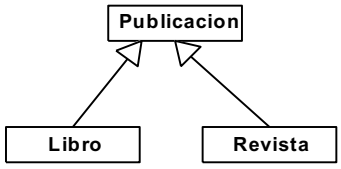
En Java las variables que referencian objetos son polimórficas, pueden apuntar a un objeto del tipo declarado, o a un objeto de un subtipo del tipo declarado.

```
public void listar()
{
        for (Articulo a: articulos)
        {
                a.print();
                System.out.println();
        }
}
```

La variable a *e*s un ejemplo de variable polimórfica. Está declarada de tipo Articulo pero al recorrer el ArrayList va apuntando bien a un CD o a un DVD.

7.2.3.- Casting.

¿Es posible asignar un valor de un supertipo (de una superclase) a una variable declarada de un subtipo (de una subclase)? La respuesta es no. No podemos asignar un supertipo a un subtipo.



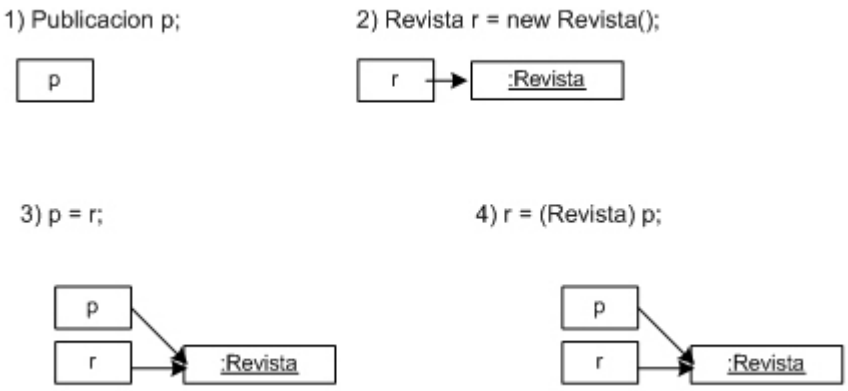
En la jerarquía dada si hacemos:

```
Publicacion p;
Revista r = new Revista();
p = r;                  // Correcto
r = p;                  // Error en compilación
```

Aunque p en la última asignación referencia a una revista el compilador, que verifica todos los tipos (Java es un lenguaje fuertemente tipado), no lo sabe por eso se queja.

Para poder efectuar la asignación (para que el compilador no emita error) hay que realizar un *casting* y eso sólo si sabemos seguro que la publicación p en ejecución es una revista. Si no es así en ejecución se lanza la excepción *ClassCastException*.

```
Publicacion p;
Revista r = new Revista();
p = r;                  // Correcto
r = (Revista) p;        // Bien, efectuamos un casting
```



Hay que tener cuidado con el *casting* y utilizarlo moderadamente porque puede causar errores de ejecución. La mayoría de código con *casting* puede reestructurarse para eliminar su necesidad, normalmente reemplazándolo por una llamada a un método polimórfico.

Responder a las siguientes cuestiones:

a) ¿Qué ocurre si hacemos? Indica si es error de compilación o ejecución.

```
Publicacion p = new Publicacion();
Revista r = new Revista();
Libro l = new Libro();
p = l; ??
l = p;  ??
r = l; ??
r = (Revista) p;  ??
```

b) ¿Qué ocurre si hacemos? Indica si es error de compilación o ejecución.

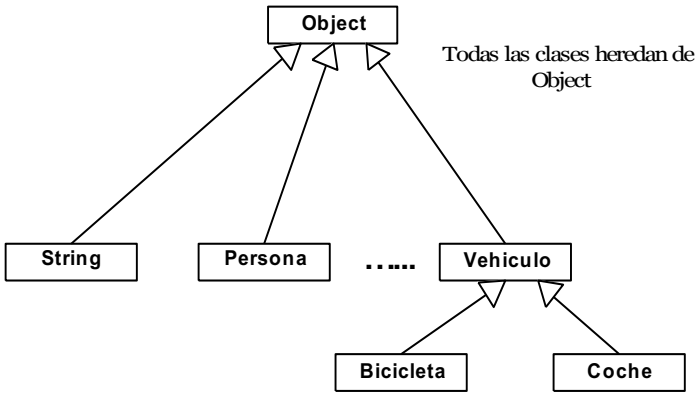
```
Vehiculo v;
Coche c = new Coche();
v = c;
c = v;
```

7.3.- La clase Object

La clase ***Object*** es una clase Java, incluida en el paquete *java.lang,* que sirve como superclase para todos los clases.

Todas las clases derivan de *Object*. Cuando no se indica explícitamente de quién deriva una clase se asume que lo hace de *Object*.

Object es la raíz de todas las clases en la jerarquía de clases Java.



Toda clase deriva, directa o indirectamente de *Object*. Las dos definiciones siguientes son equivalentes.

```
public class Persona                public class Persona extends Object
{                                   {

}                                   }
```

¿Por qué los objetos heredan de Object?

Tener una superclase común para todos los objetos tiene dos propósitos :

- la clase *Object* define algunos métodos que automáticamente están disponibles para cualquier objeto existente y
- podemos declarar variables polimórficas de tipo *Object* para referenciar cualquier objeto. Esto es lo que ocurre con la librería de colecciones de Java

7.4.- Colecciones polimórficas.

Las colecciones, a partir de Java 5, son genéricas parametrizadas, es decir, al definirlas hay que especificar el tipo de objeto que van a almacenar. Es posible, sin embargo, definir colecciones que guarden objetos de cualquier tipo, por ejemplo:

```
ArrayList<Object> lista = new ArrayList<Object>();
```

define una colección que puede guardar cualquier tipo de objeto, es una colección *polimórfica*.

Así sería posible añadir:

```
lista.add("pepe");
lista.add(new Integer(7));
```

De hecho, hasta la versión 1.5 las colecciones en Java eran todas polimórficas:

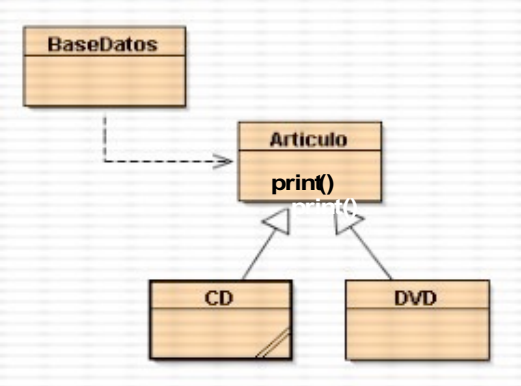
```
ArrayList lista = new ArrayList();      // Definición y creación de una lista antes de Java 1.5
```

Al recuperar los valores de la colección había que hacer un *cast* porque los objetos eran todos del tipo Object.

En la versión actual podemos encontrar entre las colecciones métodos cuyas signaturas incluyen parámetros de tipo Object. En la clase HashSet, por ejemplo, ya hemos visto:

```
boolean contains(Object o)        // El objeto que se recibe puede ser de cualquier tipo
boolean remove(Object o)
Object[] toArray()                // En Set  y que hereda HashSet
```

7.5.- Polimorfismo y redefinición (*overriding*) de métodos.

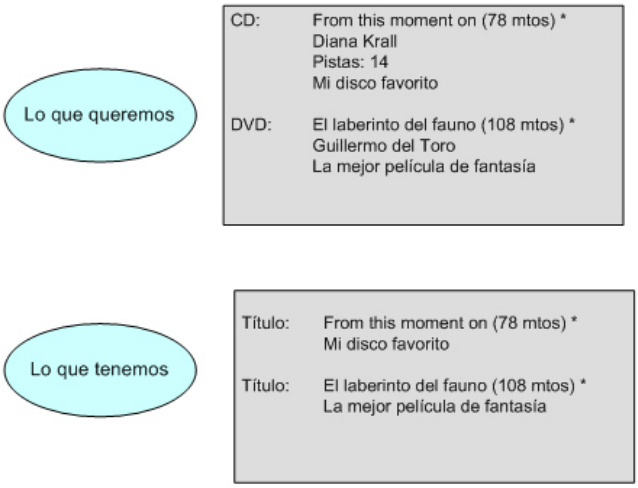


Si ejecutamos el método *listar()* de la clase BaseDatos observaremos que su funcionamiento no es correcto ya que cuando se invoca *articulo.print(),* en realidad, no se escribe adecuadamente la información relativa a un CD o un DVD, sólo se escribe la información común a ambos artículos, pero no lo que es específico de cada uno.

La herencia es un camino de "*una dirección*", una subclase hereda los atributos de la superclase pero la superclase no conoce nada acerca de los atributos de la subclase.

Cuando se establecen relaciones de herencia algunos métodos que se heredan es preciso adaptarlos en la nueva clase (clase hija), hay que ***redefinirlos (overriding)***.

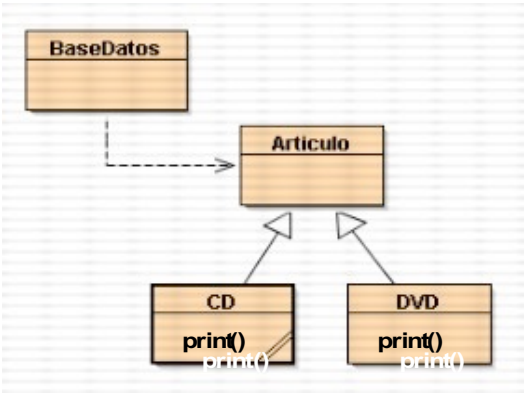
Estructura estática



```
public class Articulo
{
        .....
        /**
         Escribir detalles sobre el artículo
        */
        public void print()
        {
                System.out.print("Título: " + titulo + " (" + duracionTotal + " mins)");
                if  (loTengo())
                        System.out.println("*");
                else
                        System.out.println();

               System.out.println("    " + comentario);
        }
        .....
}
```

Antes de llegar a la solución definitiva (la redefinición del método *print()* en las subclases) intentaremos algunas alternativas que no funcionarán y estudiaremos algunos conceptos nuevos.

7.5.1.- Tipo estático y tipo dinámico.



Una posible solución sería colocar el método *print()* en las subclases, allí donde tenga acceso a toda la información que necesita. Cada clase podría tener su propia versión de *print()*. Sin embargo, esto provoca dos tipos de error:

- *print()* no puede acceder a los atributos de Articulo puesto que son privados y
- la clase BaseDatos no puede encontrar el método *print()*

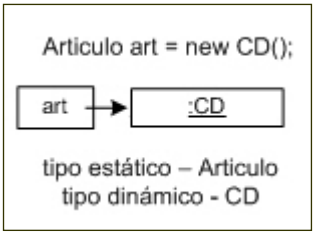
Por el principio de sustitución (subtipos) sabemos que una variable puede declararse de un tipo pero referenciar realmente a un objeto de otro tipo, de un tipo de clase derivada.

El ***tipo estático*** de una variable es el tipo declarado en la sentencia de declaración de dicha variable. Es su tipo en tiempo de compilación.

```
Articulo   art;      // Tipo estático
```

El ***tipo dinámico*** de una variable es el tipo del objeto al que referencia la variable en ejecución. Es su tipo en tiempo de ejecución.

```
Articulo   art = new CD();          // Tipo dinámico=CD porque apunta a un objeto CD
```



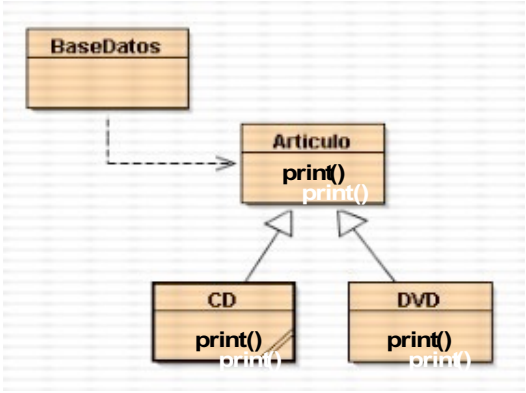
El trabajo del compilador es verificar la no violación de los tipos estáticos. Si en la clase BaseDatos hemos definido:

```
private ArrayList<Articulo> articulos;

public void listar()
{
        for (Articulo a: articulos)
            {
            a.print();
            System.out.println();
            }
}
```

La llamada al método a.print() genera un error en compilación.

7.5.2.- Redefinición de métodos (*overriding*).

La solución al problema anterior es la redefinición del método *print()*.



Las clases CD y DVD *redefinen* (modifican, adaptan) el método *print()* que heredan:

```
public class CD extends Articulo
{
        .....
        public void print()
        {
                super.print();
                System.out.println("    " + artista);
                System.out.println(" pistas   " +
                                        numeroPistas);
        }
        .....
}
```

Consideraciones al *redefinir* un método:

- para que una subclase pueda redefinir un método tiene que declarar un nuevo método con el mismo nombre y la misma signatura que el de la superclase (el heredado) pero con una implementación diferente
- los objetos de la subclase tienen en realidad dos métodos con el mismo nombre y la misma signatura: el heredado de la superclase y el nuevo que ha redefinido
- en principio, el método que se redefine en la subclase tiene preferencia sobre el heredado
- al implementar el método redefinido es posible llamar al heredado a través de ***super***:
  `super.escribir();`

  Al contrario que en los constructores el nombre del método de la superclase hay que ponerlo explícitamente.

  La llamada con *super* a los métodos que no son constructores puede hacerse en cualquier lugar del método, no tienen por qué ser la primera línea (en los constructores es obligatorio)
- si un método se declara con el modificador *final* no puede redefinirse

Uso de la palabra clave *final* al declarar una clase

Si declaramos una clase como *final* significa que no es posible crear subclases que hereden de ella. Ver por ejemplo en la API de Java como está declarada la clase String:

public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence

7.5.3.- Sobrecarga (*overloading*) y redefinición (*overriding*).

No hay que confundir la sobrecarga de métodos con la redefinición de métodos.

La *sobrecarga* significa que varios métodos en la misma clase tienen el mismo nombre pero distinta signatura (la sobrecarga permite definir la misma operación de diferentes formas para diferentes datos).

La redefinición permite tener un método en la clase padre y otro en la clase hija con el mismo nombre y la misma signatura (misma operación de diferentes formas para diferentes tipos de objetos).

7.5.4.- Métodos polimórficos. Búsqueda de métodos a través de la herencia

El polimorfismo es, junto con la encapsulación y la herencia, otra de las características que definen la POO. Polimorfismo significa "múltiples formas".

En terminología de objetos, existe polimorfismo cuando un mismo mensaje enviado a diferentes objetos de diferentes clases que forman parte de una jerarquía producen comportamientos diferentes.

El *polimorfismo de métodos* significa que una llamada a un mismo método produce ejecuciones diferentes dependiendo del tipo dinámico de la variable utilizada en la llamada. Se da en una relación de herencia cuando un método de una clase padre se redefine en la clase hija.

En nuestro ejemplo *print()* es un *método polimórfico*. Veamos por qué y qué versión se ejecutará cuando desde la clase BaseDatos se invoca al método *print()* sobre una referencia de Articulo.

```
import java.util.ArrayList;

public class BaseDatos
{
        private ArrayList<Articulo> articulos;

        .....
        public void anadir(Articulo a)
        {
                articulos.add(a);
        }

}
```

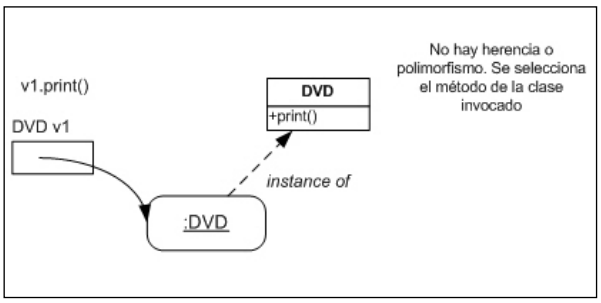
```
public void listar()
{
        {
            arti.print();
            System.out.println();
        }

 }
```

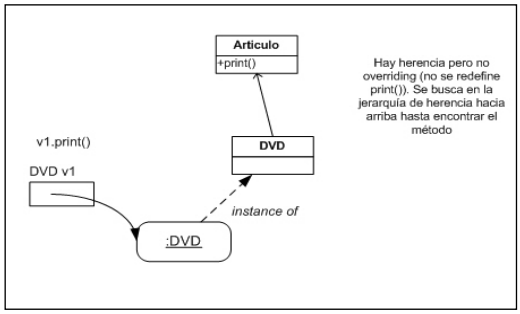
El tipo estático de la variable *arti* es Articulo. En ejecución el tipo dinámico puede ser DVD o CD.

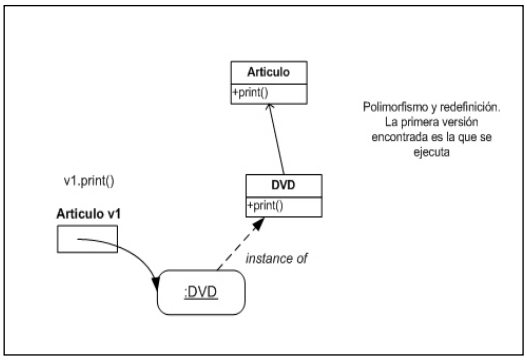
El método *print()* que se ejecuta es el determinado por el tipo dinámico, es decir, el redefinido en las clases CD o DVD.

Cuando se redefinen métodos en las subclases tienen precedencia sobre los métodos de las superclases.

Repasemos todo esto con ejemplos gráficos:





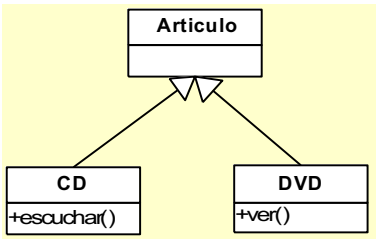


El operador ***instanceof*** verifica si un objeto dado es, directa o indirectamente, una instancia de una clase dada:

objeto *instanceof* ***clase*** devuelve *true* si el tipo dinámico de *objeto* es ***clase***
(o si **objeto** es un subtipo de **clase**).

Ej.
```
Vehiculo v = new Coche();          // Asumimos que Coche deriva de Vehiculo
Coche c = new Coche();
Persona p = new Persona();
Vehiculo v2 = new Vehiculo();

if (c  instanceof  Coche)          // Devuelve true
if (v  instanceof  Coche)          // Devuelve true
if (c  instanceof  Persona)        // Devuelve false
if (c  instanceof  Vehiculo)       // Devuelve true
if (v2  instanceof  Coche)         // Devuelve false
```

Sea la jerarquía:



Hacemos:

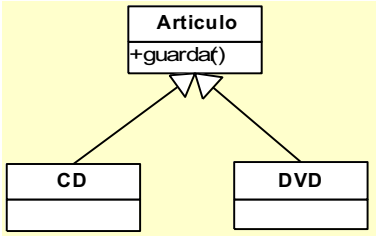
CD c1 = new CD();

¿Tipo estático de *c1 ?*
¿Tipo dinámico de *c1* ?

Si hago c*1.escuchar() ,* ¿qué método se ejecuta? ¿Hay polimorfismo?

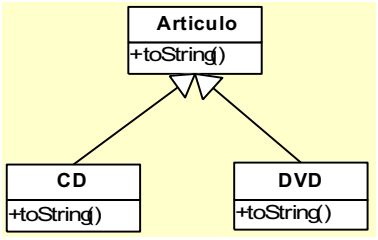
Articulo arti = new DVD();

¿Tipo estático de *arti*?
¿Tipo dinámico de *arti*?
Si hacemos *arti.ver(),* ¿se produce algún error? Si es así, ¿cómo resolverlo?



```
Articulo d1 = new DVD();
d1.guardar();
```

¿Tipo estático de *d1?*
¿Tipo dinámico de *d1*?
¿Hay algún problema?




```
Articulo a1 = new CD();
a1.toString();
```

¿Tipo estático de a*1?*
¿Tipo dinámico de a*1*?
¿Qué método *toString()* se ejecuta? ¿Qué característica de la POO se está aplicando?

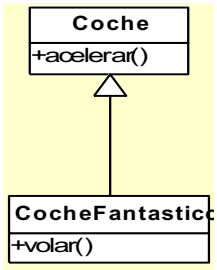
Responde a las siguientes cuestiones:

1. Supongamos el siguiente código:

   ```
   Dispositivo d1 = new Impresora();
   d1.getNombre();
   ```

   Impresora es una subclase de Dispositivo. ¿Cuál de estas dos clases debe tener una definición del método *getNombre()* para que el código compile?

2. En el caso anterior, si ambas clases tienen una implementación de *getNombre(),* ¿cuál se ejecutará? ¿Qué tipo de método es *getNombre()* ?

3. 


   ```
   Coche miCoche  = new CocheFantastico();
   Coche tuCoche = new Coche();
   ```

   Indica qué ocurre en cada caso y cómo arreglarlo si se da algún error:

   ```
   miCoche.volar();
   miCoche.acelerar();
   tuCoche.acelerar();
   tuCoche.volar);
   ```

7.6.- Algunos métodos de la clase Object.

La clase Object incluye algunos métodos que heredan todas las clases, puestos que todas heredan de ella directa o indirectamente.

Método toString()

Devuelve una cadena (String) conteniendo una representación del objeto. Por defecto, este método devuelve una cadena con el nombre de la clase a la cual pertenece el objeto instanciado, una @ y un nº hexadecimal que contiene el código hash del objeto. ***Ej***. Telefono@162b91.

Para hacer realmente útil este método se redefine en cada clase (es lo que hemos estado haciendo habitualmente).

Los métodos *print()* y *println()* si no reciben un objeto de tipo String invocan automáticamente al método *toString()* del objeto (el que haya heredado o redefinido) que aparezca como parámetro en *print()* o *println()*.

System.out.println(articulo);

es lo mismo que

System.out.uprintln(articulo.toString());

Método equals()

Dos objetos son *iguales* si son del mismo tipo y tienen el mismo valor, es decir, si los valores de sus atributos son iguales.

Dos objetos son *idénticos* si y solo si son el mismo objeto (dos referencias que apuntan al mismo objeto). Las dos referencias son "*alias*".

```
Alumno a = new Alumno("Alberto", 7.5);
Alumno b = new Alumno("Alberto", 7.5);
```

a y b son dos objetos iguales pero no idénticos. a == b es *false* en este ejemplo. El operador == denota identidad no igualdad.

El método *equals()* devuelve *true* si dos referencias constituyen un alias (es decir, si los dos objetos son el mismo objeto, si son IDENTICOS).
La implementación por defecto que proporciona Object para este método es:

public boolean equals(Object obj) { return this == obj; }

La signatura del método *equals()* es: **public boolean equals(Object obj)**

Podemos redefinir este método en nuestras clases para definir la relación de igualdad entre dos objetos de la forma en que nos sea más apropiada.

Ya vimos como la clase String definía un método *equals()* que devolvía *true* si dos objetos String contenían los mismos caracteres. La clase String redefine el método *equals()* heredado de Object y proporciona el suyo propio. Las clases Integer, Double, ..., Date, también redefinen *equals()* (también redefinen *toString()*).

Muchas de las colecciones del framework de Java (ArrayList, Set, HashMap, ...) utilizan *equals()* para comparar objetos. Es por ello que habitualmente tendremos que redefinir este método para efectuar correctamente las operaciones con estas colecciones. Por ejemplo, si queremos que una clave en un *map* sea un objeto Articulo habremos de redefinir *equals()* para que el *map* funcione correctamente al localizar una clave.

Si se redefine *equals()* obligatoriamente hay que redefinir *hashCode()*.

```
1   /**
2    * Compares this string to the specified object.  The result is {@code
3    * true} if and only if the argument is not {@code null} and is a {@code
4    * String} object that represents the same sequence of characters as this
5    * object.
6    *
7    * @param  anObject
8    *         The object to compare this {@code String} against
9    *
10   * @return  {@code true} if the given object represents a {@code String}
11   *          equivalent to this string, {@code false} otherwise
12   *
13   * @see  #compareTo(String)
14   * @see  #equalsIgnoreCase(String)
15   */
16  public boolean equals(Object anObject) {
17      if (this == anObject) {
18          return true;
19      }
20      if (anObject instanceof String) {
21          String anotherString = (String)anObject;
22          int n = value.length;
23          if (n == anotherString.value.length) {
24              char v1[] = value;
25              char v2[] = anotherString.value;
26              int i = 0;
27              while (n-- != 0) {
28                  if (v1[i] != v2[i])
29                      return false;
30                  i++;
31              }
32              return true;
33          }
34      }
35      return false;
36  }
```

Ilustración 1. Clase String - equals()

Método hashCode()

Este método invocado sobre un objeto devuelve un entero que es el valor que se utiliza como código hash para guardar el objeto en una tabla hash. La implementación por defecto de Object devuelve la dirección del objeto en el *heap* en hexadecimal.

Por contrato, si dos objetos son iguales (detectado con *equals()*) entonces su código hash ha de ser el mismo. Por eso hay que redefinir *hashCode()* en aquellas clases que redefinan *equals()*.

La clase String redefine *hashCode()* así como las clases Integer, Double, ..., Date.

```
1       /**
2        * Returns a hash code for this string. The hash code for a
3        * {@code String} object is computed as
4        * <blockquote><pre>
5        * s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]
6        * </pre></blockquote>
7        * using {@code int} arithmetic, where {@code s[i]} is the
8        * <i>i</i>th character of the string, {@code n} is the length of
9        * the string, and {@code ^} indicates exponentiation.
10       * (The hash value of the empty string is zero.)
11       *
12       * @return  a hash code value for this object.
13       */
14      public int hashCode() {
15          int h = hash;
16          if (h == 0 && value.length > 0) {
17              char val[] = value;
18
19              for (int i = 0; i < value.length; i++) {
20                  h = 31 * h + val[i];
21              }
22              hash = h;
23          }
24          return h;
25      }
```

Una posible implementación de hashCode para nuestras clases podría estar basada en invocar al método hashCode de la clase String por ejemplo:

```
public class Persona
{
	private String nombre;
	…

	public int hashCode()
	{
		return edad() + nombre.hashCode() * 11;
	}
}
```

Método clone()

Realiza una copia profunda (clonación, DEEP COPY) de un objeto. Para que un objeto se pueda clonar ha de implementar el interface Cloneable.

Este método es *protected* en Object por lo que se suele sobreescribir haciéndolo público (hay que ser cuidadoso a la hora de definir este método).

Método getClass()

Devuelve una instancia de *java.lang.Class* que contiene información sobre el tipo dinámico del objeto referenciado por la variable.

Antes de que un objeto de cualquier tipo sea creado, su clase es cargada y la JVM automáticamente crea una instancia de *java.lang.Class,* esta instancia nos proporciona información sobre la clase en tiempo de ejecución.

A esta instancia se le denomina *metaobjeto (*contiene información sobre una clase*).*

```
public class DVD
{
        public void escribirClaseObjeto()
        {
                System.out.println("El objeto pertenece a la clase  "
                        + getClass().getSimpleName());
        }
}

Articulo a = new DVD();
a.escribirClaseObjeto();            // Visualiza “El objeto pertenece a la clase DVD”
```

El tipo dinámico de la variable a es DVD.

7.7.- Modificadores *protected* y *final*

Declarar un atributo o método con visibilidad ***protected*** permite el acceso directo (o indirecto) a él desde las subclases (estén o no en el mismo paquete) y desde las clases que estén en el mismo paquete.

El acceso ***protected*** se puede aplicar tanto a atributos como a métodos aunque debería aplicarse únicamente a estos últimos. La relación de herencia entre clases representa un grado de acoplamiento entre las clases mayor que en otras relaciones (agregación, composición, asociación).

Se DEBE evitar el uso de *protected* en atributos utilizando accesores y mutadores para minimizar el acoplamiento entre clases y subclases, y reforzar la ocultación de datos.

Los atributos son PRIVADOS SIEMPRE. Los métodos los declararemos con visibilidad private, protected o public dependiendo del caso (o visibilidad de paquete).

Si un método se declara como ***final*** no puede redefinirse.

```
public final void escribir()
```

Si una clase se declara como ***final*** no se puede heredar de ella (por ejemplo la clase String de la API de Java).

```
public final class String
```

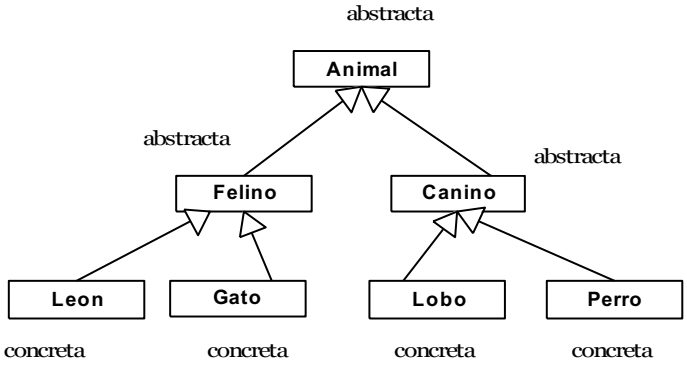
7.8.- Clases abstractas.

Una *clase abstracta* es una clase, en la jerarquía de clases, de la que no se van a crear instancias.

Modelan conceptos abstractos (genéricos) y sirven como superclase para otras clases. Una clase abstracta captura las características y comportamientos comunes a otras clases. Se usan como plantillas de clase y supertipos.

Se utilizan cuando deseamos definir una abstracción que agrupe objetos de distintos tipos y queremos hacer uso del polimorfismo.

Se declaran con el modificador ***abstract***.




```
public abstract class Animal
{

}

Animal a = new Animal();  // Error
```

Las clases que no son abstractas se denominan *clases concretas*.

Una clase abstracta puede contener métodos abstractos, es decir, métodos sin implementación, sólo con la signatura. Se declaran con el modificador *abstract.*

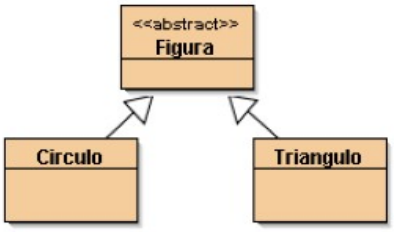
Si definimos una clase abstracta sabemos que deberá ser extendida. Si un método se define como abstracto debe ser redefinido.

Cuando se define una clase muy general, como la clase Animal no tiene sentido proporcionar implementación para algunos de sus métodos, por ejemplo, *comer()*, *correr(),* porque estas acciones dependen de un tipo de animal concreto, un perro, o un león. La razón para definir métodos abstractos es proporcionar un protocolo para todos los subtipos (subclases) y así utilizar el polimorfismo y poder manipular animales de forma genérica.

```
public abstract class Animal {
	public abstract void comer();
	public abstract void correr();
}

public abstract class Canino extends Animal {
	// public abstract void comer();
	// public abstract void correr();
}

public class Perro extends Canino {
	public void comer() {
		…
	}

	public void correr() {
		…
	}
}
```

Ya que no contienen implementación los métodos abstractos no pueden ejecutarse.

La implementación de los métodos abstractos se proporciona en las subclases, por ej, en Leon, Perro.



En UML una clase abstracta se escribe o bien en cursiva o con el estereotipo <<*abstract*>> (de ésta última forma lo muestra BlueJ).

7.8.1.- Consideraciones sobre las clases abstractas

1. No se pueden crear instancias
2. Sólo las clases abstractas pueden contener métodos abstractos. Esto asegura que todos los métodos de las clases concretas puedan ser ejecutados siempre
3. Las clases abstractas con métodos abstractos fuerzan a las subclases a redefinir e implementar los métodos declarados como abstractos. Si una subclase no proporciona implementación para un método abstracto heredado, la subclase será abstracta y no podrá instanciarse. Para que una subclase sea concreta debe implementar todos los métodos abstractos que herede
4. Una clase abstracta se puede utilizar como tipo aunque no se instancie
5. Pueden contener atributos y métodos no abstractos
6. Un método abstracto no puede ser definido como *final* porque ha de ser redefinido
7. Virtualmente su único uso es para ser extendidas (proporcionar un protocolo), sin embargo, hay una excepción, si tienen métodos estáticos pueden ser invocados.

Las clases abstractas proporcionan:

- mayor flexibilidad
- más abstracción
- más extensibilidad
- menor acoplamiento

7.8.2.- Clases abstractas y la API de Java

En la API de Java encontramos multitud de clases abstractas. Ya vimos entre las colecciones que existen algunas clases abstractas como AbstractCollection y AbstractList. De ésta última deriva ArrayList que es una clase concreta. HashSet es una clase concreta que deriva de AbstractSet.

En el paquete Swing (la librería gráfica de Java) también encontramos clases abstractas. JComponent es una clase abstracta de la que derivan todos los componentes de una GUI, JLabel, JList, JComboBox, ... Nunca se instancia un JComponent sino un JLabel o un JButton.

7.8.3.- Clases ≪utility≫ y constructores privados

Una clase solo debería ser declarada como abstracta si nuestra intención es que se puedan crear subclases para completar la implementación. Si lo que queremos es simplemente evitar la creación de objetos de esa clase, la forma apropiada de hacerlo es declarar un constructor **privado** sin parámetros, no invocarlo y no declarar otros constructores. Una clase de este tipo contiene normalmente únicamente miembros de clase (static), es decir, sería una clase ≪utility≫.

La clase Math es un ejemplo de una clase ≪utility≫ que no puede ser instanciada. Su declaración sería:

```
public final class Math {
     private Math() { }   // Nunca instanciar esta clase

     // Miembros de clase (atributos y métodos static)
     ……
}
```

7.9.- Interfaces

Una ***interface*** *en* Java es la especificación de un tipo, nombre del tipo y conjunto de métodos, que no define ninguna implementación para los métodos (excepto en los métodos static y default). También puede contener constantes.

Una interface define un conjunto de comportamientos que puede ser implementado por cualquier clase en cualquier punto de la jerarquía de clases.

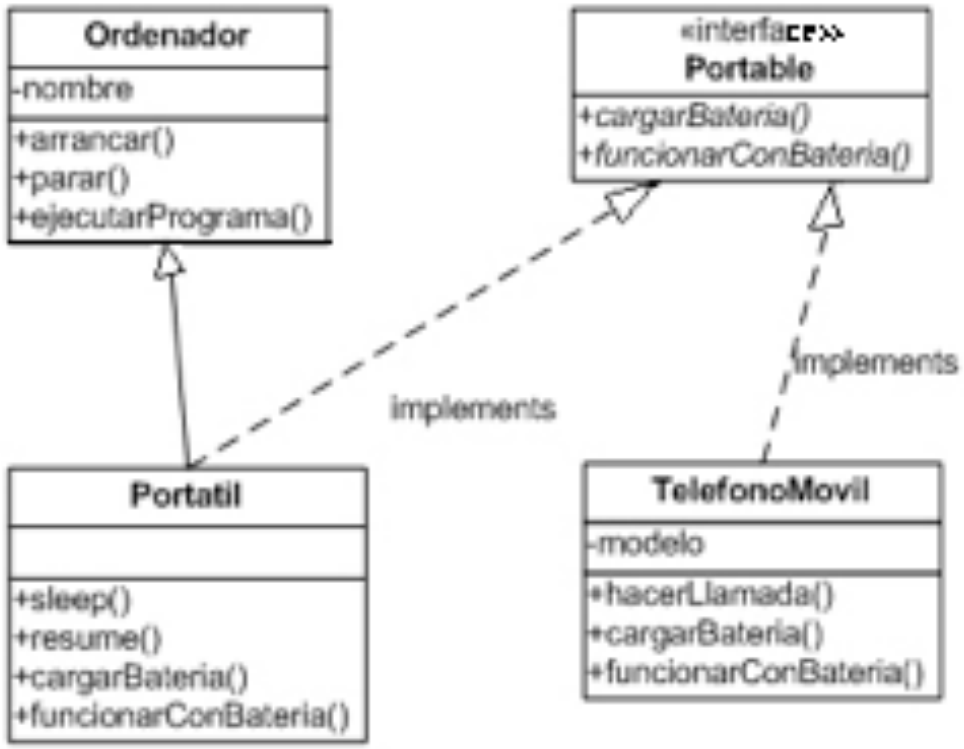
Una clase que implemente una interface está de acuerdo (*"firma un contrato"*) en proporcionar implementación **para todos los métodos** declarados en la interface.

Son similares a las clases abstractas pero también tienen diferencias con ellas:

- Una interface no puede implementar ningún método (aunque[1] en JDK 8, una interface ya puede definir métodos static y métodos default) , una clase abstracta sí
- Una clase puede implementar cualquier nº de interfaces
- Una interface no es parte de una jerarquía, clases que no tienen ninguna relación pueden implementar la misma interface. Una interface puede ser usada para expresar un comportamiento común entre clases que no tienen ninguna relación en una jerarquía.

La clase TelefonoMovil no está relacionada con la clase Portatil, sin embargo, tiene algunos métodos comunes con ella, *cargarBateria()*, *funcionarConBateria()*.

Portatil y TelefonoMovil implementan la misma interface, Portable, que especifica los métodos que están disponibles para ambos. La interface Portable expresa el comportamiento que es común a los dispositivos portátiles. Define lo que hacen los objetos pero no cómo.



[1] Ver http://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html - static

7.9.1.- Definiendo una interface

Se define a través de la palabra clave *interface* (en lugar de *class*) en la cabecera de declaración:

```
public interface Portable
{
        void cargarBateria();
        void funcionarConBateria();
}
```

En UML una interface se denota con el estereotipo <<*interface*>>

- Todos los métodos en una interface son abstractos (sin implementación), excepto los métodos static y default a partir de JDK 8
- ~~No contienen métodos estáticos~~ ← A partir de JDK 8 un interface ya puede incluir métodos estáticos (static)
- Las interfaces no contienen constructores
- Todos los métodos tienen visibilidad public y no es necesario indicarlo
- Además de métodos, solo se permiten atributos constantes que son public, static, final y no es preciso indicarlo
- Una interface puede heredar (extends) de una o varias interfaces (en la API, el interface Set extiende el interface Collection – herencia simple). En el caso de las interfaces se da la herencia múltiple (a diferencia de las clases que sólo pueden tener una clase padre, herencia simple).
- En general, el nombre de las interfaces se denota con adjetivos (Comible, Movible, Editable, Observable, Comparable)

7.9.2.- Implementando una interface

Una clase puede IMPLEMENTAR una interface (o varias). Se dice que la clase ***implementa*** (*implements*) la interface. Cuando una clase implementa una interface debe **proporcionar implementación de todos** los métodos que se declaran en la interface.

```
public  class Portatil
extends Ordenador
implements Portable
{
        public void cargarBateria()
        {
                System.out.println("Cargando batería");
        }

        public void funcionarConBateria()
        {
                System.out.println("Fucionando con batería");
        }

}
```

Si una clase hereda de otra e implementa una interface la palabra clave *extends* va en primer lugar.

Una clase puede implementar más de una interface. Esto permite la IMPLEMENTACION múltiple de interfaces (recordemos que Java solo permite herencia simple entre clases).

```
public  class Portatil
extends Ordenador
implements Portable, Comparable<Portatil>
{
        // Método correspodiente a la interface Portable
        public void cargarBateria()
        {
                System.out.println("Cargando batería");
        }

        // Método correspodiente a la interface Portable
        public void funcionarConBateria()
        {
                System.out.println("Fucionando con batería");
        }

        // Método correspodiente a la interface Comparable
        public int compareTo(Portatil otro)
        {
            if (x == otro.getX())
                return 0;
            else if (x < otro.getX())
                return -1;
            else
                return 1;
        }
}
```

7.9.3.- Interfaces como tipos

La herencia de clases (herencia de estructura) proporcionaba dos grandes beneficios:

- las subclases heredaban los atributos y el código (métodos) de la superclase. Esto permitía la reutilización.
- las subclases eran subtipos de las superclases. Esto permitía la existencia de variables polimórficas y las llamadas a métodos polimórficos.

Un interface define un tipo. Esto quiere decir que las variables pueden ser declaradas de un tipo *interface* aunque no haya objetos de ese tipo, sólo subtipos del interface.

Las interfaces no tienen instancias directas pero sirven como tipos para instancias de otras clases, las que implementan esa interface.

Si una clase implementa una interface cualquier objeto de esa clase es un subtipo de la interface.

En donde pueda aparecer un tipo, puede aparecer el nombre de una interface (ya que son tipos de datos).

```
                                                Portable miPortatil = new Portatil();
Portable p;
Portatil miPortatil = new Portatil();
p = miPortatil;
```

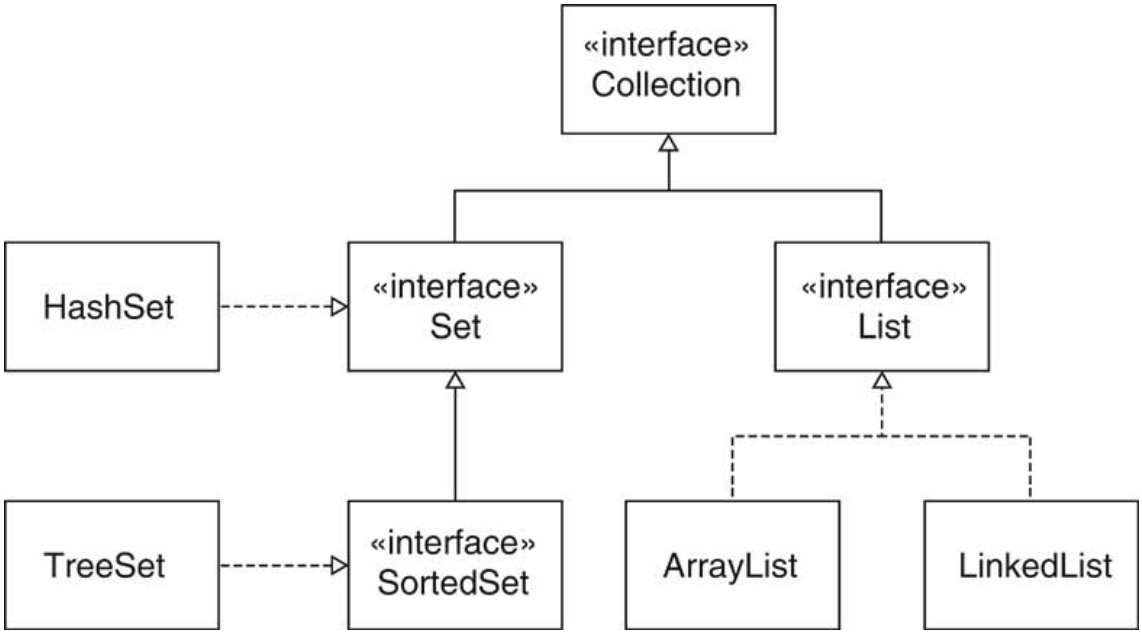
7.9.4.- Interfaces como especificaciones.

A través de interfaces, Java permite la herencia múltiple. Sin embargo, la gran ventaja de las interfaces no es esa.

La característica más importante de una interface es que separa completamente la especificación de funcionalidad de la implementación.

Una interface realmente es un ***contrato*** entre la interface y la clase que implementa dicha interface. La clase se compromete, garantiza, que va a proporcionar la funcionalidad de todos los servicios que proporciona la interface que implementa. Definen un papel que las clases que lo implementan van a jugar.

Un buen ejemplo de todo esto es el conjunto de colecciones de la API de Java. En ella hay una clara distinción y separación entre funcionalidad (comportamiento) e implementación, lo que proporciona un mejor y más flexible diseño.



(*) La clase TreeSet realmente implementa la interface NavigableSet que es una interface que hereda de la interface SortedSet

El interface List especifica la funcionalidad completa de una lista sin proporcionar implementación.

Las clases ArrayList y LinkedList proporcionan diferentes implementaciones de la misma interface.

Ej.

```
List<Persona> lista = new ArrayList<Persona>();   ó
List<Persona> lista = new LinkedList<Persona>();
lista.add(new Persona());

public void escribirLista(List<Persona>  lista)    // Vale tanto si lista es un ArrayList como
                                                   // si es LinkedList
```

Se declara la colección lista de un tipo interface (el que ofrece la funcionalidad). La implementación de lista será su tipo actual (tipo dinámico, el que se elija para conseguir mejor rendimiento).

Tenemos más ejemplos, el interface Map y las clases concretas HashMap y TreeMap, el interface Set y las clases HashSet y TreeSet.

Los interfaces también pueden formas jerarquías. Collection es uno de los interfaces raíz del *framework* de Java. List y Set heredan de esa interface.

7.9.5.- Clases abstractas o interfaces.

A menos que una clase vaya a contener algún método con implementación en cuyo caso tendría que ser una clase abstracta (excepto en el caso de interfaces con métodos static y defautl en Java 8) es preferible utilizar interfaces ya que:

- permiten herencia múltiple
- proporcionan mayor flexibilidad

Realmente una interface permite usar el polimorfismo en su máxima expresión. Son lo último en flexibilidad. Si se utilizan como tipo en argumentos y valores de retorno se puede pasar cualquier cosa que implemente la interface.

Con los interfaces una clase no tiene que pertenecer a ninguna jerarquía. Una clase puede extender otra clase e implementar una (o varias) interfaces. Pero otra clase puede implementar la misma interface y proceder de una jerarquía de clases totalmente distinta. En realidad, así conseguimos tratar objetos por el papel que juegan más que por el tipo de clase al que pertenecen.

7.10.- Interfaces Comparable, Cloneable, Comparator.

Java define algunos interfaces que incluyen métodos que pueden ser utilizados por cualquier clase que implemente la interface.

7.10.1.- Interface Comparable

Este interface define un único método que compara el objeto actual con el que recibe como parámetro y devuelve un entero:

```
c1.compareTo(c2)
c1-c2
5-10
-5

< 0     si c1 es menor que c2 → c1 precede a c2
== 0    si c1 es igual a c2
> 0     si c1 es mayor que c2 → c1 sucede a c2
```

```
public  interface Comparable<T>
{
        int compareTo(T o);
}
```

<T> - tipo de los objetos a comparar

Los objetos de las clases que implementan el interface Comparable pueden ser comparados en términos de orden.

Ej.

```
public  class Coche implements Comparable<Coche>
{
        private int velocidad;

        …

        public int compareTo(Coche otro)
        {
                return velocidad-otro.velocidad;
        }
}
```

Las colecciones de la API basan las operaciones de ordenación y algunas búsquedas en el método *compareTo()* de este interface. Así para poder ordenar una colección de objetos Persona, por ej., si queremos utilizar el método estático *Collections.sort()* la clase Persona debe implementar el interface Comparable. Lo mismo ocurre con *Arrays.sort()*.

```
List<Coche> lista = new ArrayList<Coche>();

// Añadir coches a la lista
…

// Ordenación de la lista de coches
Collections.sort(lista);
```

La clase Coche debe implementar la interface Comparable (con el método compareTo(…)).

Para poder realizar la ordenación con Collections.sort(…), todos los elementos de la lista deben ser comparables.

La clase String y las clases envolventes Integer, Double, Character, etc. implementan ya el interface Comparable.

Ejer. 7.1. Un objeto Casa tiene entre otros atributos superficie. Define la clase Casa de tal forma que sea posible establecer un orden entre dos objetos del tipo Casa en base a su superficie. Usar la interface Comparable.

7.10.2.- Interface Cloneable

Para hacer una copia completa, "*deep copy"*, de un objeto hay que clonar el objeto, hacer una copia del mismo y de todos los objetos de los que esté compuesto. Por tanto hay que definir el método clone y hacer la copia a mano en el caso de que el objeto esté formado por otros objetos.

El método clone() definido en la clase Object permite realizar únicamente una copia superficial ("shallow copy").

La implementación de *clone()* en Object copia cada atributo del objeto original en el objeto destino (si es un tipo primitivo se copia el valor, si es un objeto se copia la referencia).

```
protected Object clone()
throws CloneNotSupportedException          // Así está definido en Object
```

Este interface realmente está vacío, no tiene ningún método. Se utiliza para indicar a Java que la clase que lo implementa puede utilizar el método *clone()*.

Este método se declara *protected* (protegido) para evitar que se clonen objetos de una clase si no se desea. Si queremos dejar que nuestros objetos puedan clonarse redefiniremos *clone()* y lo haremos público.

El método *clone()* devuelve Object como resultado por lo que habrá que hacer un *casting* después de clonar:

```
public  class Coche implements Cloneable
{
        private int velocidad;

        @override
        public Object clone()
        {
                try
                {
                        // En este caso se realiza una copia superficial
                        // ("shallow copy") ya que se usa el método clone
                        // de la clase padre que es Object
                        return super.clone();
                }
                catch (CloneNotSupportedException e)
                {
                        return null;
                }
        }
}

Coche co1 = new Coche(180);
Coche co2= (Coche) c1.clone();
```

Ejer. 7.2. Crear un ejemplo donde un coche contenga 4 ruedas (objetos). Implementar el método clone() usando la interface Cloneable en la clase Coche realizando una copia profunda ("deep copy")

7.10.3- Interface Comparator

Es posible ordenar un conjunto de objetos (en una colección, o un array, ...) que no implementen el interface Comparable por algún otro criterio de ordenación además del que indique el método *compareTo()* de Comparable. Para ello necesitamos un objeto Comparator que encapsula un criterio de orden.

El interface Comparator consiste en un único método:

```
public interface Comparator<T>
{
   int compare(T o1, T o2)
}
```

El método compara los dos argumentos devolviendo un resultado:

< 0 si el primer objeto es menor que el segundo (según el criterio de comparación que se establezca)
= 0 si el primer objeto es igual que el segundo
> 0 si el primer objeto es mayor que el segundo

El interface Comparator está en *java.util.*

Ej.

```
import java.util.Comparator;
public  class ComparadorCoches1 implements Comparator<Coche>
{
        public int compare(Coche c1, Coche c2)
        {
                return c1.getVelocidad() - c2.getVelocidad();
        }
}

public class ComparadorCoches2 implements Comparator<Coche>
{
  // compare (con otro criterio de comparación distinto)
}

List<Coche> lista = new ArrayList<Coche>();
.....
Collections.sort(lista, new ComparadorCoches1());
Collections.sort(lista, new ComparadorCoches2());
```

De esta manera puedo ordenar la lista con distintos criterios de comparación.

~~**Ejer. 7.xx** Ordena la colección de figuras anterior en base al valor de su área. Utiliza el método *sort()* de la clase Collections. Añade el código necesario para poder ordenar las figuras también por el valor de su perímetro. Prueba lo que has hecho dentro de una clase DemoFiguras.~~