Tema 7. Mejorando la estructura de una aplicación: herencia y polimorfismo. Interfaces.

Actividad

Introducimos dos nuevos conceptos:

- **la herencia y**
- **el polimorfismo.**

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.

En esta unidad veremos también el uso de las **Interfaces.**

El uso de interfaces explota el polimorfismo en su grado máximo. Diseñando y codificando para especificaciones de interfaz proporciona un alto grado de flexibilidad y extensibilidad.

Número de sesiones

Se estiman un total de 32 horas.

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 clase base, superclase o clase padre.

La clase que se obtiene como extensión se llama subclase, 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.

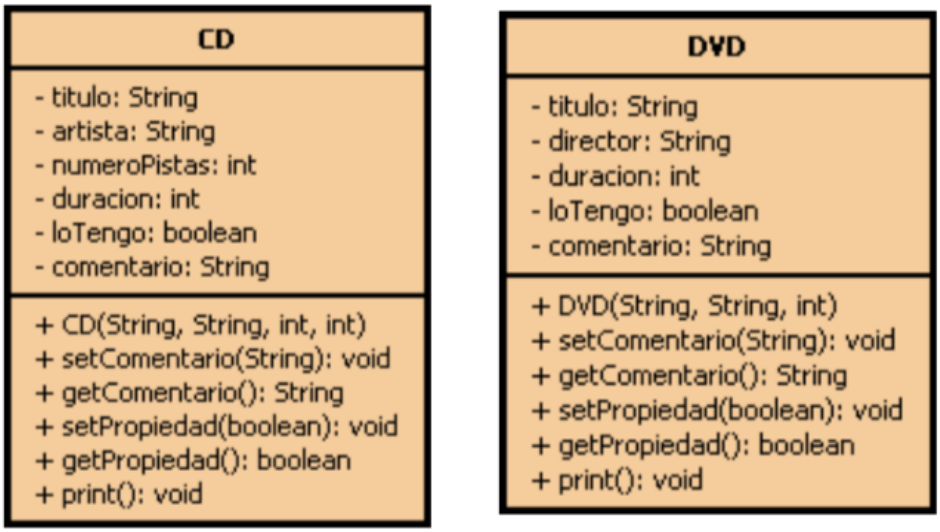
Imaginemos que queremos realizar una aplicación que cree un catálogo (una base de datos) con información de los CDs y DVDs que poseemos.

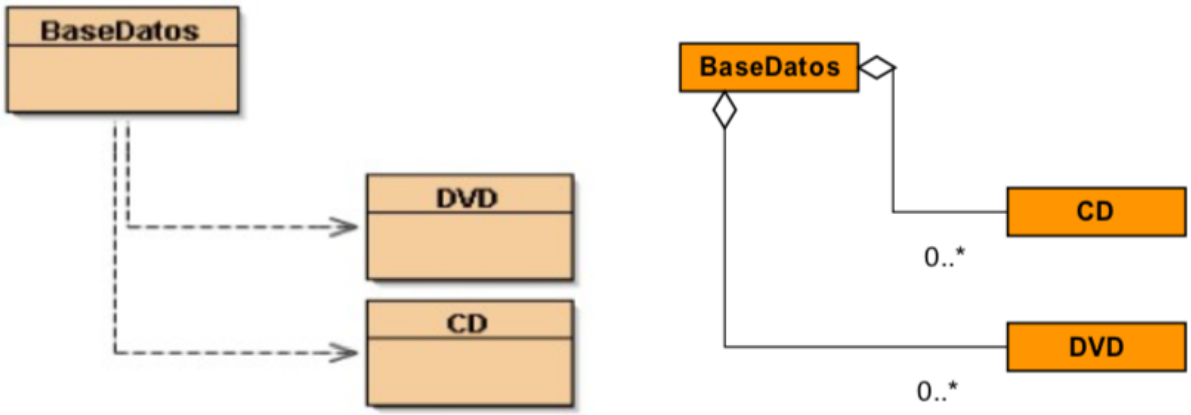
De cada CD queremos guardar:

- 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
- comentario

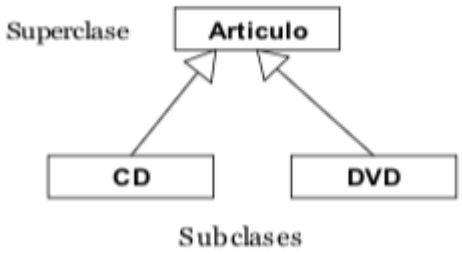




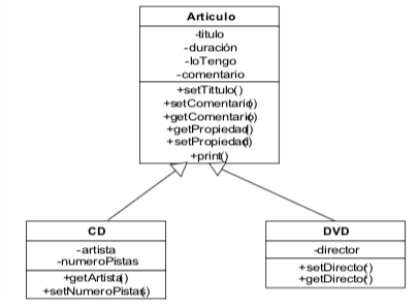

```
import java.util.ArrayList; public class BaseDatos {

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.

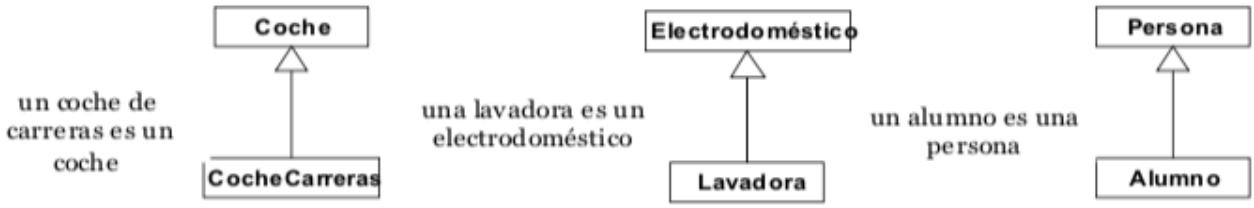


En UML la relación de herencia se representa con una línea continua que conecta la clase padre e hija y la punta de flecha apuntando a la clase padre



La herencia se denomina relación es-un ya que una subclase es una especialización (es un caso especial) de la superclase (un CD es un Artículo, un vídeo es un Artículo).

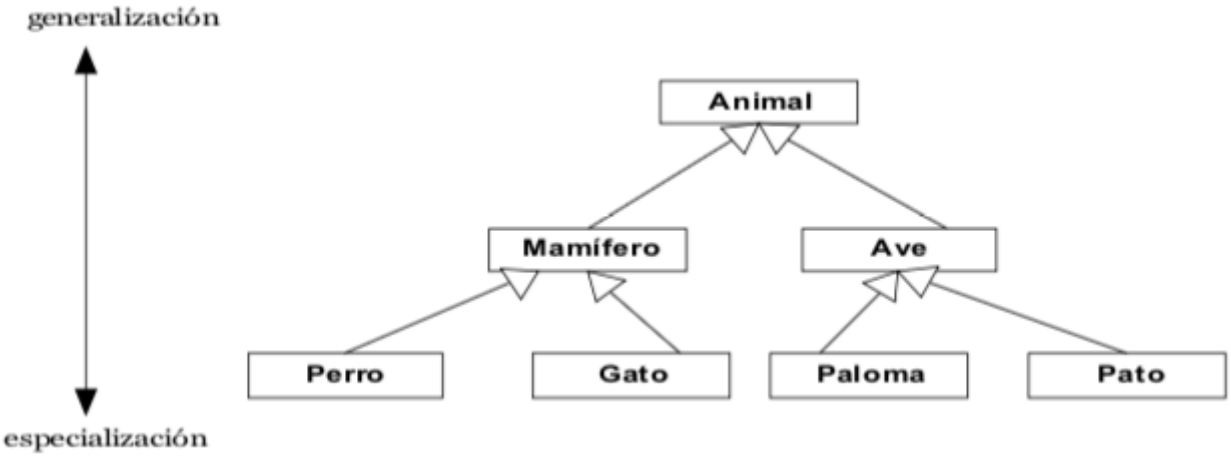
La superclase expresa la estructura y comportamiento comunes a las subclases.



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.

Ejercicio 7.1

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

Haz lo mismo clasificando los distintos tipos de vehículos que se pueden dar (los que se desplazan por tierra, por mar, ...)

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 a través 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 *Artículo* y añade dos atributos propios.

Ejercicio 7.2

Define la clase DVD (incluye solo los atributos).

Solución

```
public class DVD{
      private String titulo;
      private String director
      private int duracion;
      private boolean loTengo;
      private String comentario;
}
```

Ejercicio 7.3

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***?

Solución

Enlace con la solución del proyecto

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**.

En UML la visibilidad *protected* se expresa con el símbolo #.

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 a las subclases aunque no estén en el mismo paquete

Por defecto, si no se especifica visibilidad se asume *package*.

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 claúsula **super**. Es importante recordar que debe ser la primera sentencia en el constructor de la subclase.

La referencia 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).

Ejercicio 7.4

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

Solución

DVD

```
public DVD (String titulo, String director, int duracion){
    this.titulo=titulo;
    this.director=director;
    this.duracion=duracion;
}
```

PuntoTresDimensiones

```
public PuntoTresDimensiones(double x, double y, double z){
    super(x,y);
    this.z=z;
}
```

Ejercicio 7.5

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 la jerarquía. Además de los atributos, define los constructores.

Solución

Enlace con el proyecto solucionado

7.1.5 Ventajas de la herencia

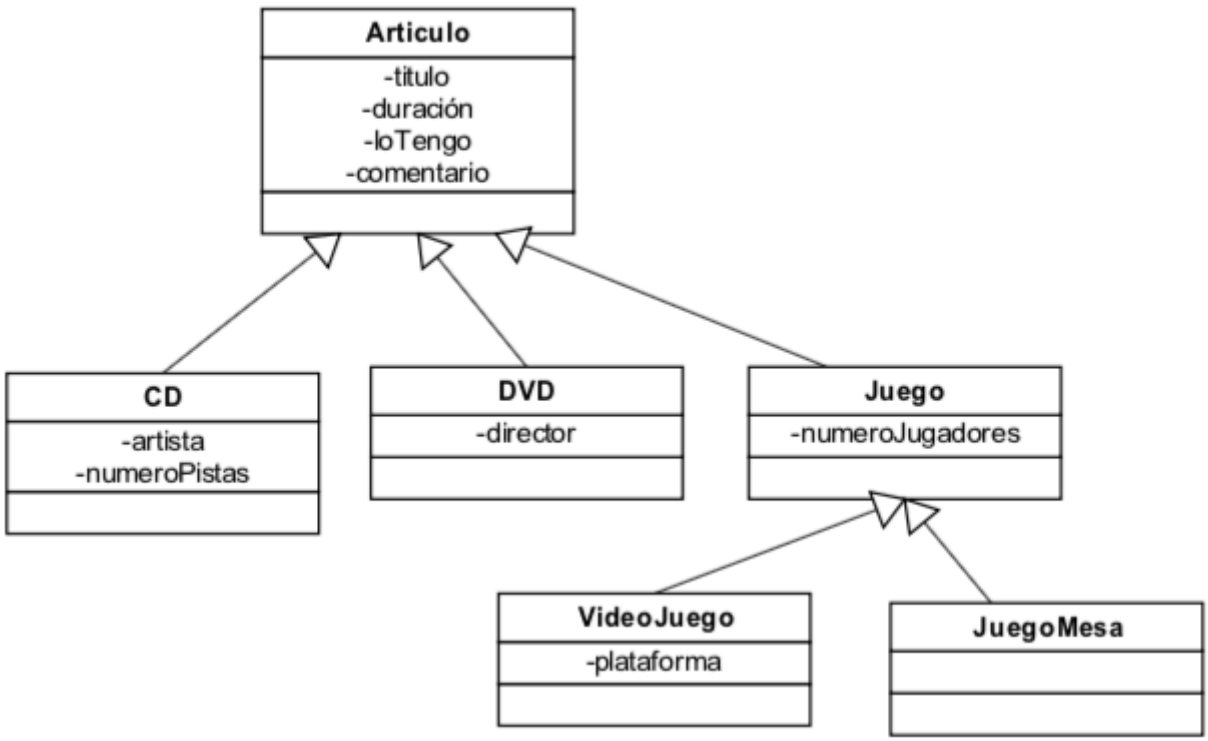
Las principales ventajas de la herencia en la POO son:

¦ 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



Ejercicio 7.6

- 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);
    }
    /**
    * Listar los artículos de la base de datos
    */
    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);

Solución

Enlace con el proyecto solucionado

Ejercicio 7.7

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().

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

Solución

Enlace con la solución del proyecto

Ejercicio 7.8

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.

Solución

Enlace con la solución

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.

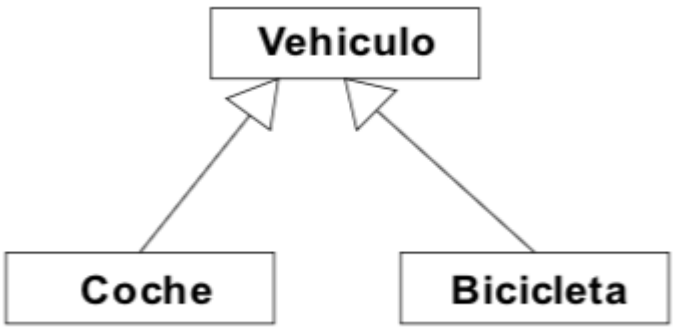
Una variable puede referenciar objetos de su tipo declarado y objetos de cualquier subtipo de su tipo declarado. El tipo declarado de una variable es su tipo en tiempo de compilación.

En los lenguajes orientados a objetos hay un principio, conocido como **principio de sustitución (de Liskov)** 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 declarado) 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 declarado *Vehiculo*, pero en ejecución, una vez instanciado el objeto, la variable apunta realmente a un objeto de tipo Coche. En compilación, la variable tiene un tipo, en ejecución puede tener otro tipo, 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.

Ejercicio 7.9

1- ¿Es correcto?

Coche unCoche = new Vehiculo();

2-Partiendo de la siguientes clases...

```
public class Gato {
}
```

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

¿Es correcto?

- *Gato g = new Gato();*
- *GatoConBotas felix = new GatoConBotas();*
- *GatoConBotas miGato = new Gato();*

Solución

1- ¿Es correcto?

Coche unCoche = new Vehiculo(); --> NO es correcto.

2-Partiendo de la siguientes clases...

¿Es correcto?

- *Gato g = new Gato(); --> CORRECTO*
- *GatoConBotas felix = new GatoConBotas(); --> CORRECTO*
- *GatoConBotas miGato = new Gato(); --> INCORRECTO*

Ejercicio 7.10

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

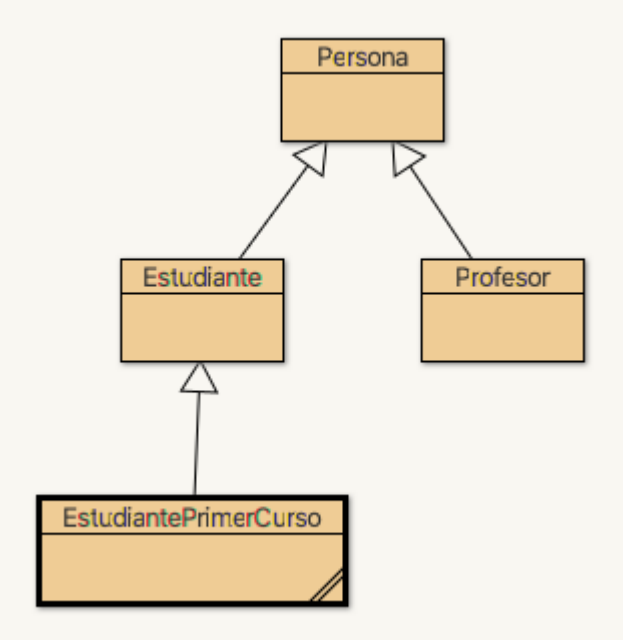
Indica cuál de las siguientes asignaciones es correcta y por qué?

```
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;

Solución



- *Persona p1 = new Estudiante(); --> OK*
- *Persona p2 = new EstudiantePrimerCurso(); --> OK*
- *EstudiantePrimerCurso estu1 = new Estudiante(); --> INCORRECTO*
- *Profesor prof1 = new Persona(); --> INCORRECTO*
- *Estudiante estu2 = new EstudiantePrimerCurso(); --> OK*

- *estu2 = p1; --> INCORRECTO*
- *estu1 = p2; --> INCORRECTO*
- *p1 = estu1; --> OK*
- *prof1 = estu1; --> INCORRECTO*
- *estu2 = estu1; --> OK*

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:

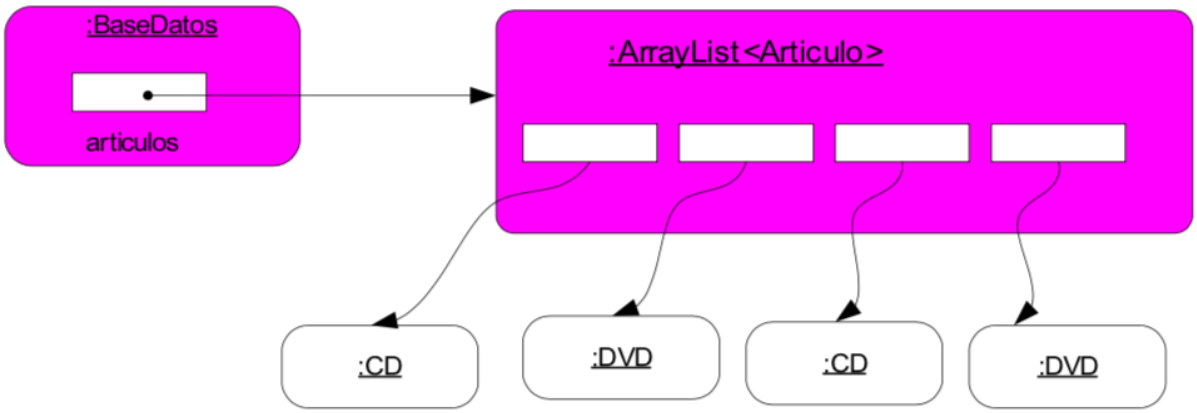
```
BaseDatos miBaseDatos = new BaseDatos();

DVD dvd1 = new DVD(.....);
```

miBaseDatos.addArticulo(dvd1);

CD cd1 = new CD(....);

miBaseDatos.addArticulo(cd1);

.....



Ejercicio 7.11

Sea el método: *public void limpiarGato(Gato g)*

y las declaraciones del ejercicio 7.9.b)

¿Es correcto?

limpiarGato(felix);

limpiarGato(g);

Solución

limpiarGato(felix); --> OK (Felix es una especialización de Gato)
limpiarGato(g); --> OK

7.2.2 Variables polimórficas

Las variables que referencian diferentes tipos de objetos se denominan **variables polimórficas** (de múltiples formas).

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 es 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.

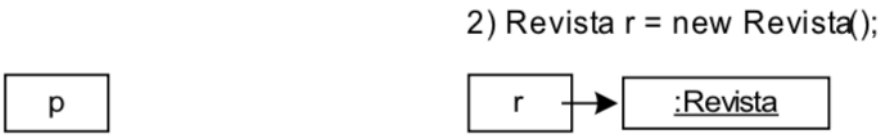
En la jerarquía dada si hacemos:

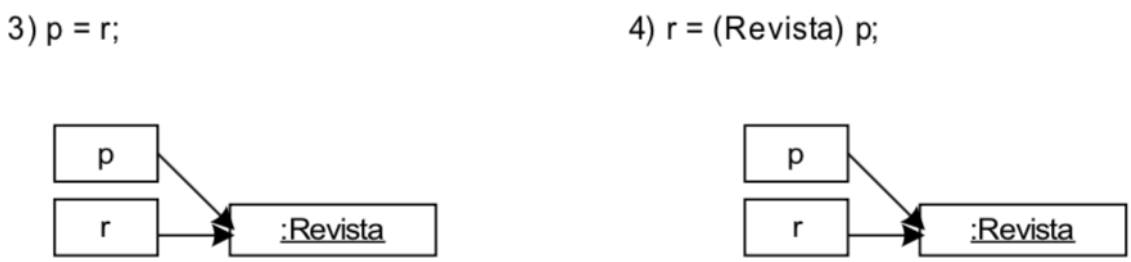
```
Publicacion p;
Revista r = new Revista();
p = r; //correcto
r = p; // error en compilación
```

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.

Ejercicio 7.12

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;

Solución

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; --> *Error de ejecución*
  - l = p; --> *Error de compilación*
  - r = l; --> *Error de compilación*
  - r = (Revista) p; --> *(realizamos un casting. No hay error de ejecución. Puede existir de ejecución)*

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

- Vehiculo v;
- Coche c = new Coche();
  - v = c; --> *Error de ejecución*
  - c = v; --> *Error de compilación*

Sección

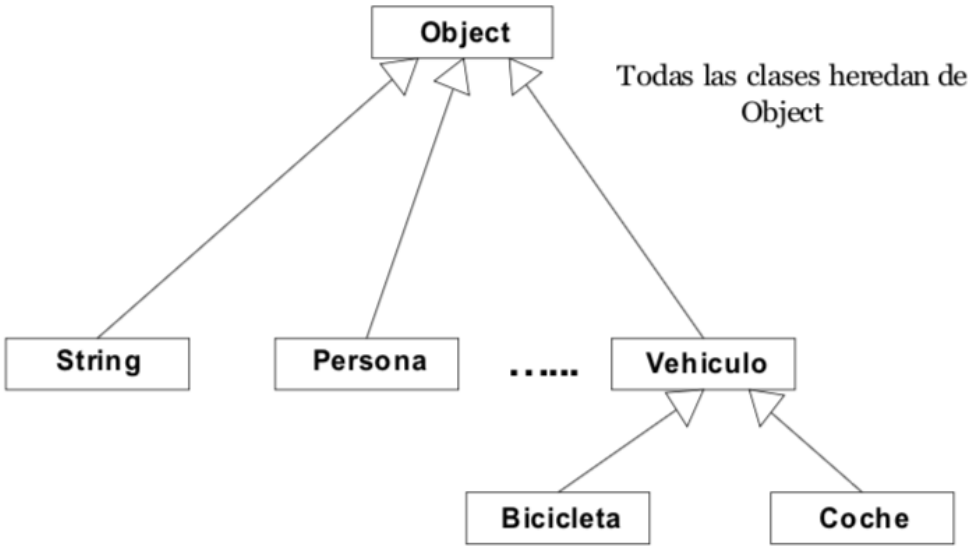
7.3 La clase Object

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

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 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:

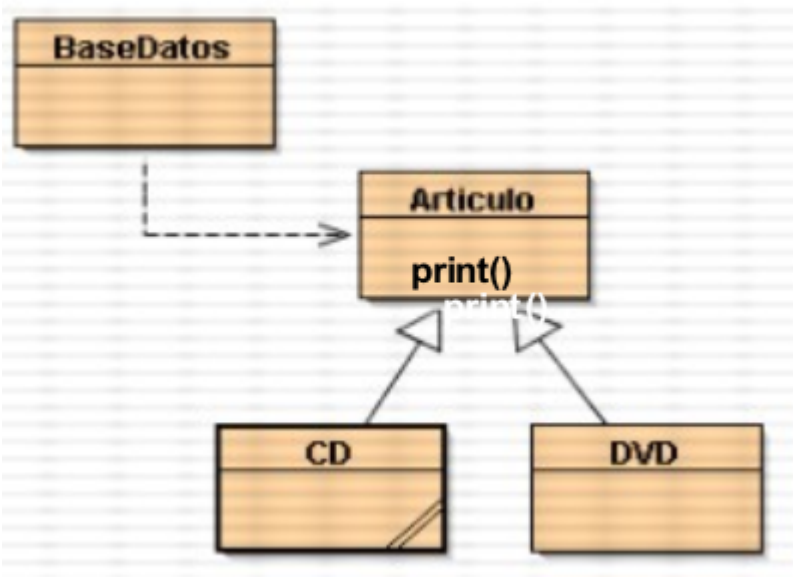
```
ArrayList lista = new ArrayList(); //definición y creación de una lista antes de Java 1.5
```

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

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
```

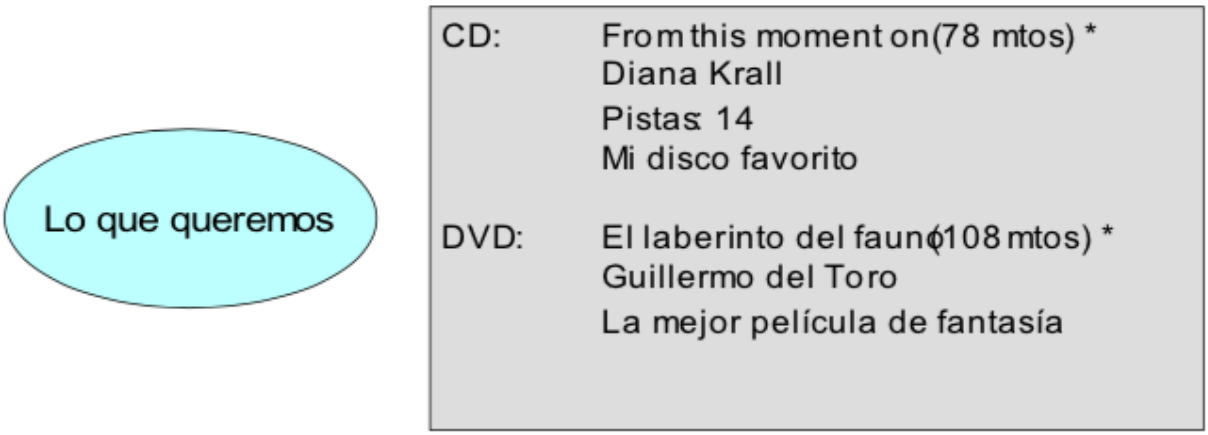
7.5 Polimorfismo y redefinición (overriding) de métodos

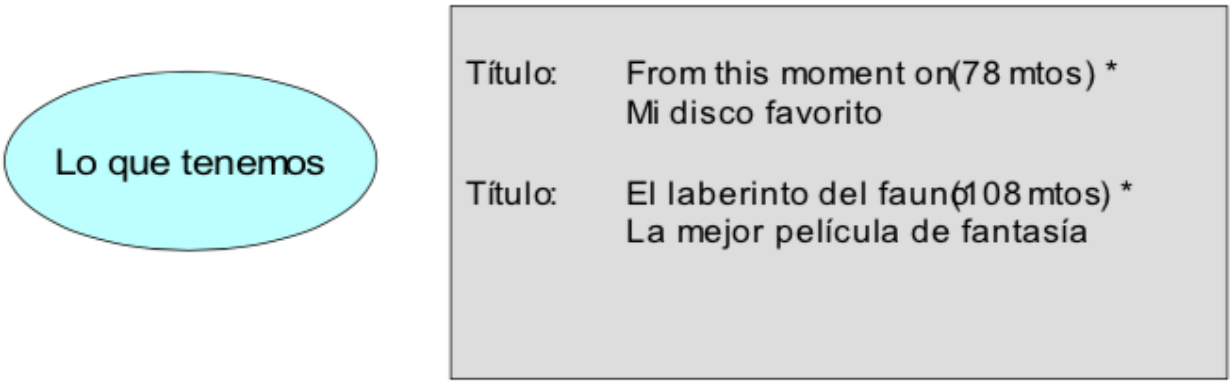


Si ejecutamos el método *listar()* de la clase BaseDatos observaremos que su funcionamiento no es correcto ya que cuando se invoca ***a.print()***, en realidad, no se escribe adecuadamente la información relativa a un CD o un DVD, sólo se escribe la información común a ambos artículos, pero no lo que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





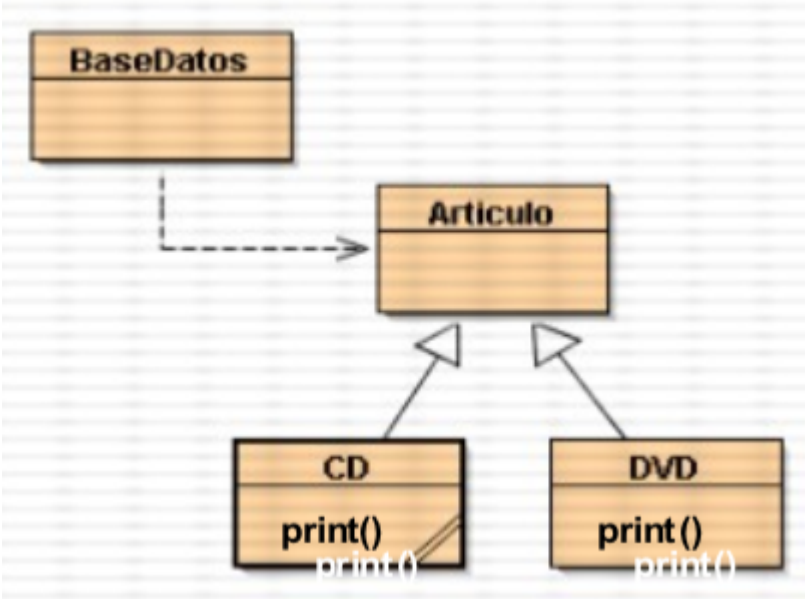

```
public class Articulo
{
        .....
        /**
         Escribir detalles sobre el artículo
        */
        public void print()
        {
                System.out.print("Título: " + titulo + " (" + duracionTotal + " mins)");
                if  (loTengo())
                        System.out.println("*");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

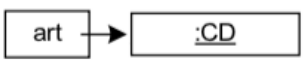
El **tipo estático** de una variable es el tipo declarado en la sentencia de declaración de dicha variable. Es su tipo en tiempo de compilación.

Articulo art; // tipo estático

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

Articulo art = new CD(); // tipo dinámico, apunta a un objeto CD

Articulo art = new CD();



tipo estático – Articulo
tipo dinámico - CD

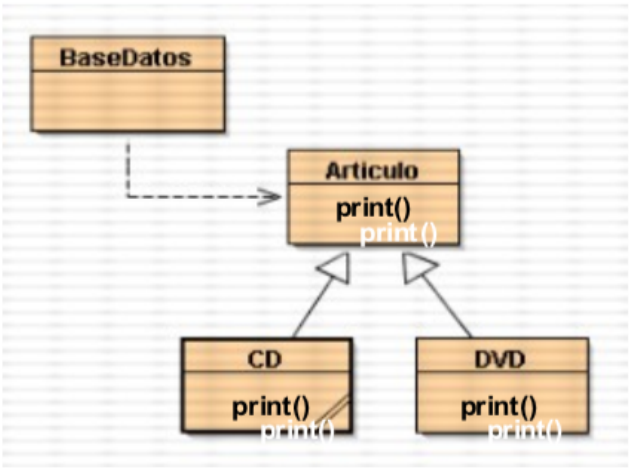
El trabajo del compilador es verificar la no violación de los tipos estáticos. Si en la clase BaseDatos hemos definido:

```
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
{
        ……..
        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.print();

  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

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, no y/o tipo de parámetros distintos (el valor de retorno no influye). 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 añadir(Articulo a)
        {
                articulos.add(a);
        }

}
```

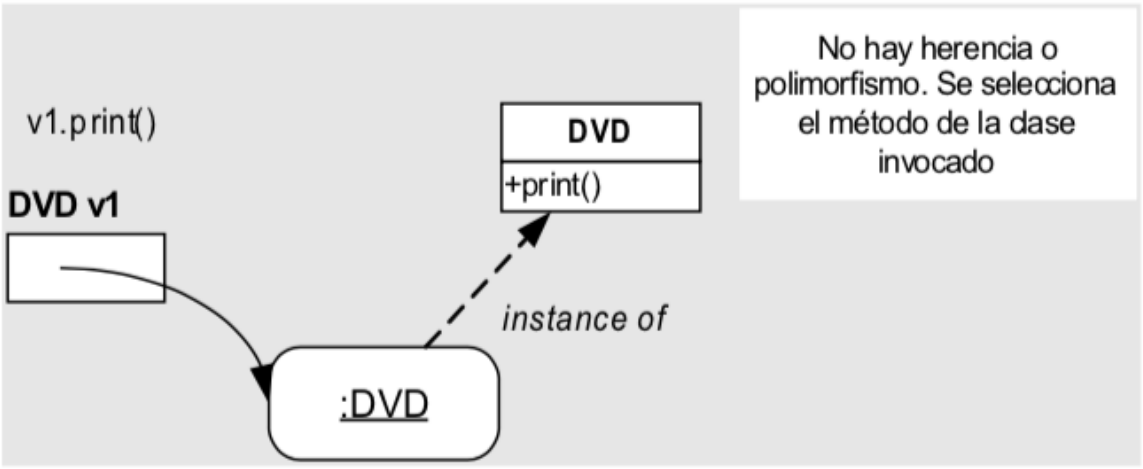
```
public void listar()
{
        for (Articulo arti: articulos)
        {
                arti.print();
                System.out.println();
        }

}
```

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

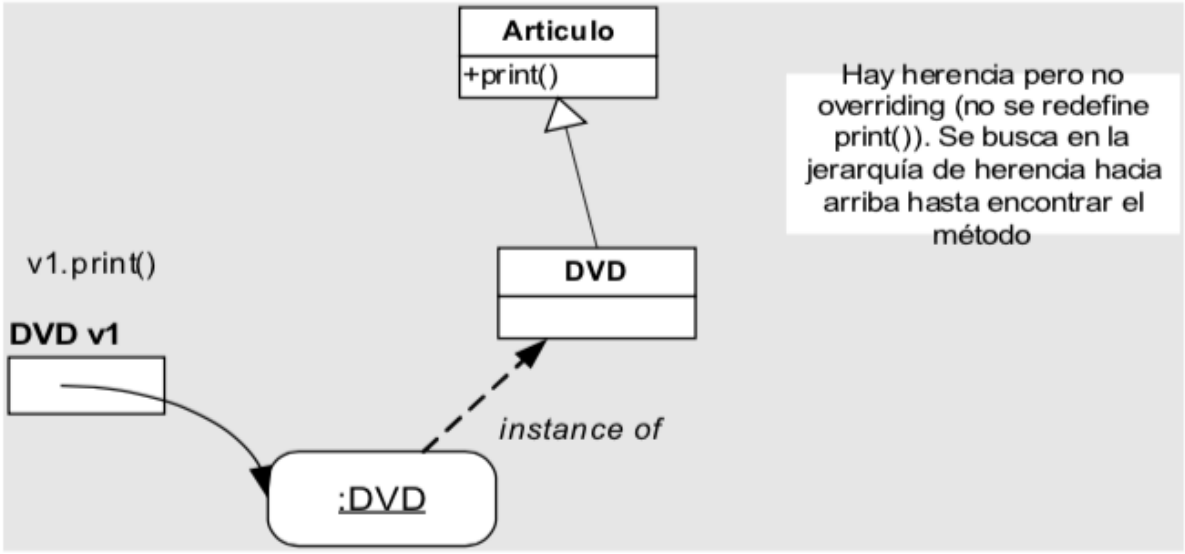
El método *print()* que se ejecuta es el determinado por el tipo dinámico, es decir, el redefinido en las clases *CD* o *DVD*.

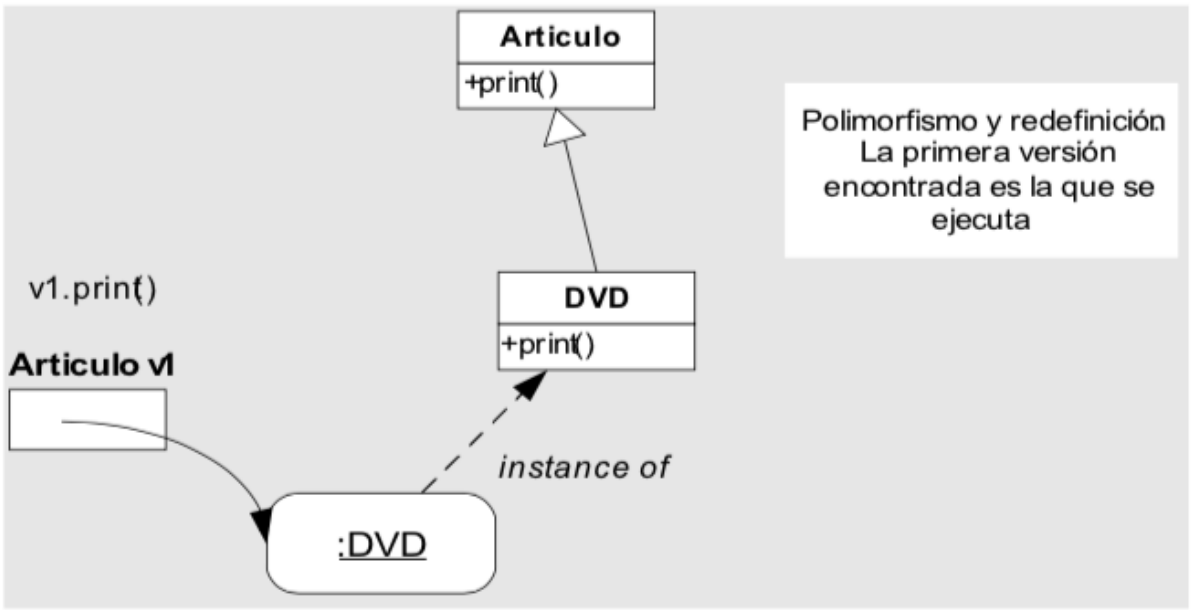
Cuando se redefinen métodos en las subclases tienen precedencia sobre los métodos de las superclases.

Repasemos todo esto con ejemplos gráficos:



La búsqueda de métodos estáticos se hace en tiempo de compilación





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 alguna derivada).

Ej.

Vehiculo v = new Coche(); //asumimos que *Coche* deriva de *Vehiculo Coche c = new Coche();*

Persona p = new Persona();

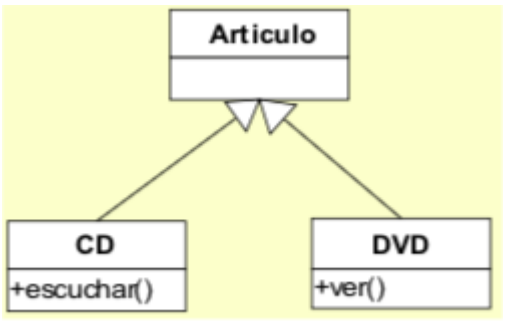
Vehiculo v2 = new Vehiculo();

Sea la jerarquía:

- *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

Ejercicio 7.13

Sea la jerarquía:



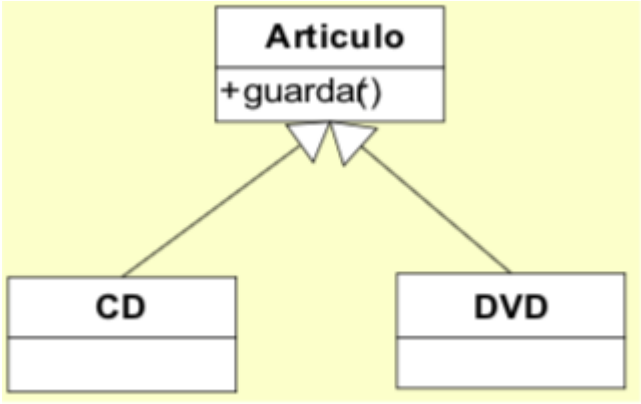
Hacemos:

CD c1 = new CD();

- ¿Tipo estático de ***c1***?
- ¿Tipo dinámico de ***c1***?
- Si hago ***c1.escuchar()*** , ¿qué método se ejecuta?
- ¿Hay polimorfismo?

Articulo arti = new DVD();

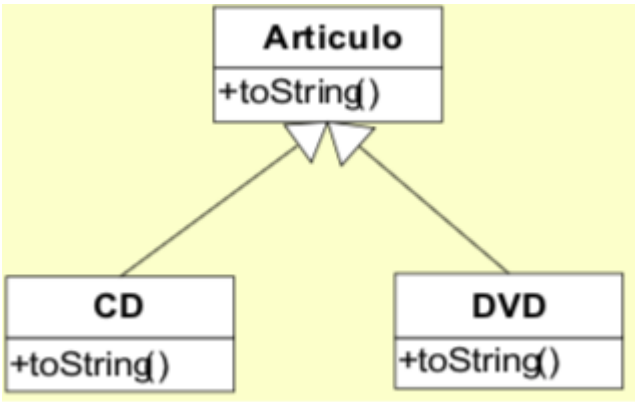
- ¿Tipo estático de ***arti***?
- ¿Tipo dinámico de ***arti***?
- Si hacemos ***arti.ver()***, ¿se produce algún error?
- Si es así, ¿cómo resolverlo?



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 ***a1***?
- ¿Tipo dinámico de ***a1***?
- ¿Qué método ***toString()*** se ejecuta?
- ¿Qué característica de la POO se está aplicando?

Solución

CD c1 = new CD();

- ¿Tipo estático de ***c1***? --> *CD*
- ¿Tipo dinámico de ***c1***? --> *CD*
- Si hago ***c1.escuchar( )*** , ¿qué método se ejecuta? --> *El método escuchar de la clase CD*
- ¿Hay polimorfismo? --> *NO*

Articulo arti = new DVD();

- ¿Tipo estático de ***arti***? --> *Articulo*

- ¿Tipo dinámico de *arti*? --> *DVD*
- Si hacemos *arti.ver()*, ¿se produce algún error? --> *Si. Porque el método ver( ) es propio de la clase DVD*
- Si es así, ¿cómo resolverlo? --> *Cambiando el tipo estático de Articulo a DVD (DVD arti = new DVD();)*

Articulo d1 = new DVD();

d1.guardar();

- ¿Tipo estático de *d1*? --> *Articulo*
- ¿Tipo dinámico de *d1*? --> *DVD*
- ¿Hay algún problema? --> *No. El método guardar( ) es propio de la clase Articulo*

Articulo a1 = new CD();

a1.toString();

- ¿Tipo estático de *a1*? --> *Articulo*
- ¿Tipo dinámico de *a1*? --> *CD*
- ¿Qué método *toString()* se ejecuta? --> *el de la clase CD*
- ¿Qué característica de la POO se está aplicando? --> *Polimorfismo*

---

* Enlace con el proyecto implementado

Ejercicio 7.14

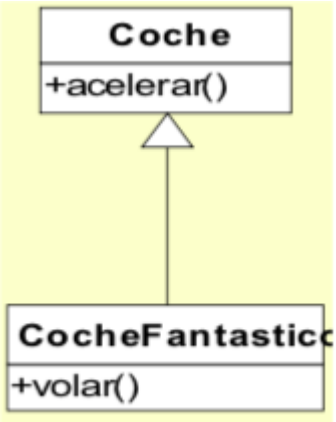
Responde a las siguientes cuestiones:

- 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?

- En el caso anterior, si ambas clases tienen una implementación de *getNombre()*, ¿cuál se ejecutará? ¿Qué tipo de método es *getNombre()*?
- Según el siguiente diagrama:



- *Coche miCoche = new CocheFantastico();*
- *Coche tuCoche = new Coche();*

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

- *miCoche.volar();*
- *tuCoche.acelerar();*
- *miCoche.acelerar();*
- *tuCoche.volar();*

Solución

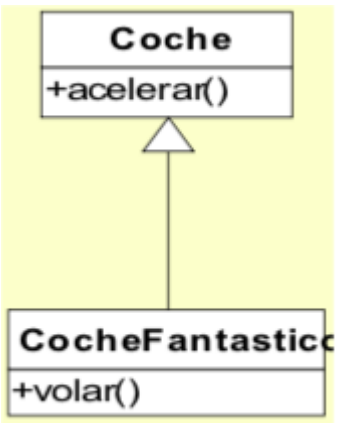
Responde a las siguientes cuestiones:

- 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? --> *La clase Dispositivo*

- En el caso anterior, si ambas clases tienen una implementación de getNombre(), ¿cuál se ejecutará? ¿Qué tipo de método es getNombre()? --> *La de la subclase*
- Según el siguiente diagrama:



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

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

- miCoche.volar(); --> *Se ejecuta volar( ) de la clase CocheFantástico*
- tuCoche.acelerar(); --> *Se ejecuta acelerar( ) de la clase Coche*
- miCoche.acelerar(); --> *Error de compilación*
- tuCoche.volar(); --> *Posible error de ejecución*

7.6 Algunos métodos de la clase Object

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

Método toString() -

Devuelve un 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 (en hexadecimal) 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. 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().***

El siguiente ejemplo muestra cómo se redefine habitualmente este método:

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

        public boolean equals(Object obj)
        {
                if ( ! (obj  instanceof  Punto) )
                        return false;
                Punto p = (Punto) obj;
                return p.getX() == x && p.getY() == y;
        }
}
```

```
public class Punto
{
  private int x;
  private int y;
  public boolean equals(Object obj)
  {
     if ( this == obj)
          return true;
     if ( obj == null)
          return false;
     if(this.getClass() !=  obj.getClass())
          return false;
      Punto p = (Punto) obj;
          return p.getX() == x && p.getY() == y;
  }
}
```

```
public class Punto
{
  private int x;
  private int y;
  public boolean equals(Object obj)
  {
    if ( ! (obj  instanceof  Punto) )
          return false;
   Punto p = (Punto) obj;
   if ( this.getClass() != p.getClass())
          return false;
   return p.getX() == x && p.getY() == y;
  }
}
```

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().*

```
public class Persona
{
        private String nombre
        private int edad;

        @override
        public int hashCode()
        {
                return edad + nombre.hashCode() * 11; //una posible implementación
                                                      // de hashCode()
        }
}
```

Si un objeto va a ser clave en un map se redefine ***equals()*** y ***hashCode()*** siempre. La clase ***String*** redefine ***hashCode()*** así como las clases ***Integer, Double, ..., Date.***

Método clone() -

Realiza una copia (clonación) de un objeto. Es una "shallow copy", es decir, crea un nuevo objeto del mismo tipo que el original y copia los valores de todos sus atributos. Si los atributos son referencias a objetos, el original y la copia comparten los objetos comunes.

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 la clase del objeto .

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  "
                        + this.getClass());  // también this.getClass().getName()
        }
}


DVD d = new DVD;
d.escribirClaseObjeto(); //    visualiza "El objeto pertenece a la clase class 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) pero no desde otras clases.

El acceso protected se puede aplicar tanto a atributos como a métodos aunque debería aplicarse únicamente a estos últimos. Si se definen atributos protected se está debilitando la encapsulación. Aunque, es cierto, que a veces interesa un acceso directo desde las subclases. La relación de herencia entre clases representa un grado de acoplamiento entre las clases mayor que en otras relaciones (agregación, composición , asociación).

Se puede evitar el uso de protected en atributos utilizando los accesores y mutadores. Al redefinir un método se puede ampliar su visibilidad pero no disminuirla.

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.

```
public final class EstudiantePrimerCiclo {

    ..............

}
```

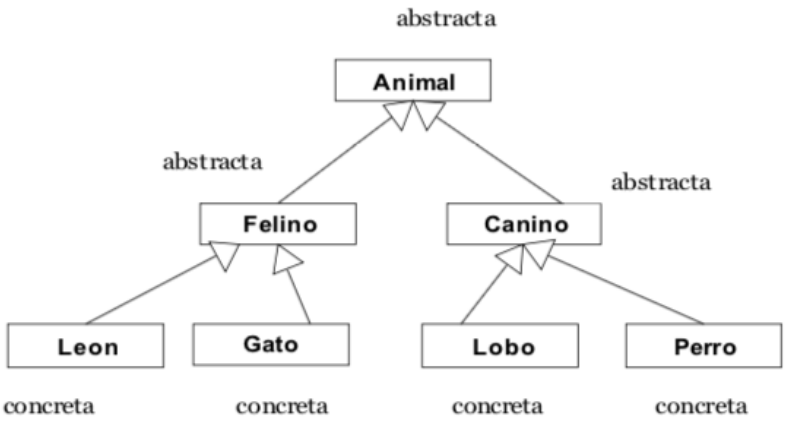
7.8 Clases abstractas

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();
}
```

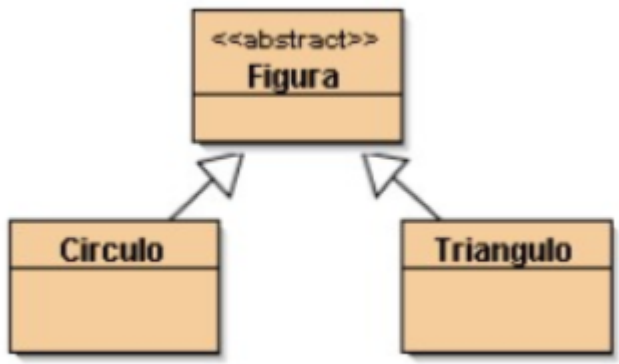
```
public  abstract class Canino extends Animal
{
        public abstract void comer();
        public abstract void correr();

}
```

```
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

Ejercicio 7.15

Define las clases mostradas en la jerarquía.

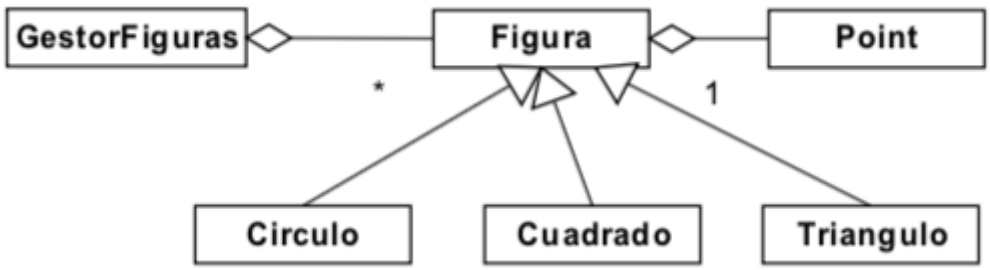


Figura es una clase de la que no se crearán instancias.

Toda figura tiene un centro (de tipo Point) y un color (de tipo Color) – consulta la API para estas clases.

Una figura incluye un constructor con dos parámetros de tipo int que marcan las coordenadas del centro y proporciona accesores y mutadores para sus atributos.

De una figura deseamos conocer su área y su perímetro. Incluye también un método toString(). Estudia qué métodos puedes implementar aquí y cuáles no.

Un círculo es una figura que tiene un radio. La clase Circulo proporciona accesores y mutadores para todos sus atributos e incluye un método toString(). Analiza qué otros métodos heredados de Figura (además de toString()) hay que redefinir.

Un cuadrado es una figura que tiene un lado. La clase Cuadrado proporciona accesores y mutadores para todos sus atributos e incluye un método toString(). Analiza qué otros métodos heredados de Figura (además de toString()) hay que redefinir.

Un triángulo es una figura definida por sus tres lados. La clase Triangulo proporciona accesores y mutadores para todos sus atributos e incluye un método toString(). Analiza qué otros métodos heredados de Figura (además de toString()) hay que redefinir.

La clase GestorFiguras guarda en un ArrayList una lista de figuras. La clase incluye, además del constructor:

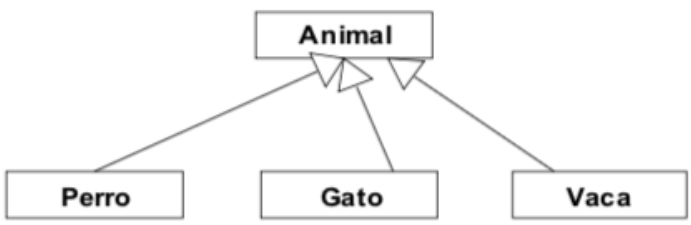
- *public void addFigura (Figura f)*
- *public void listarFiguras ()* – lista la información de todas las figuras
- *public Figura mayorArea ()* - figura con el área mayor
- *public void borrarDeColor (Color color)* – borra las figuras de un color determinado
- *public void borrarIgualesA(Figura f)* – borra las figuras iguales a f (tendrás que redefinir equals(), estudia en qué clases)

Solución

Enlace al proyecto solucionado

Ejercicio 7.16

Modela la siguiente jerarquía de animales:



La clase Animal es una clase genérica que captura las características comunes a todos los animales. Todo animal posee un nombre y no de patas y un dueño (un objeto de la clase Persona). Un animal puede emitir un sonido y comer. De un animal podemos además obtener una representación textual. Define un constructor parametrizado en esta clase.

Un perro, un gato y una vaca tienen todas las características de un animal pero su forma de emitir sonidos y comer es diferente para cada uno (“comer huesos”, “comer pescado”, “comer hierba”). Además un perro tiene un lugar favorito en el que tumbarse y un gato un juguete preferido. Modela adecuadamente estas clases incluyendo constructores, accesores y mutadores además de otros métodos que necesiten redefinición u otros nuevos.

Incluye una clase Granja que guarde en un array una serie de animales e incluya, además del constructor:

- *public void addAnimal (Animal a)*
- *public void mostrarAnimales ()*
- *public Animal getAnimal (int i)* – devuelve el animal de la posición i
- *public int cuantosPerros ()* – cuántos perros hay en la granja
- *public void borrarGatos ()* - elimina los gatos de la granja

Solución

Enlace con la solución del ejercicio

7.8.2 Clases abstractas y la API de Java

En la API de Java encontramos multitud de clases abstractas que proporcionan implementaciones parciales de los interfaces. 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, un JButton.

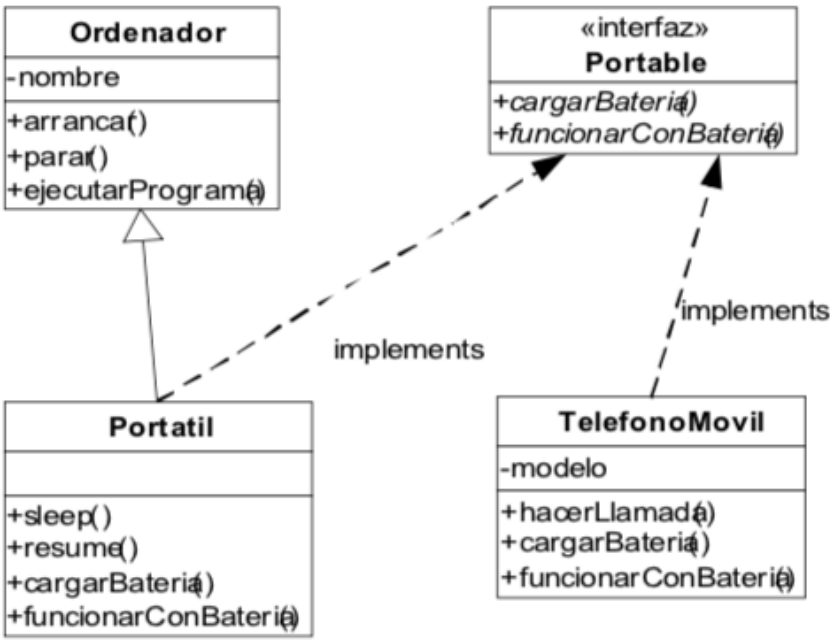
7.9 Interfaces

Un interfaz Java es la especificación de un tipo, nombre del tipo y conjunto de métodos, que no define ninguna implementación para los métodos. No tienen constructores.
Un interfaz define un conjunto de comportamientos que puede ser implementado por cualquier clase en cualquier punto de la jerarquía de clases.

Una clase que implemente un interfaz está de acuerdo ("firma un contrato") en proporcionar implementación para todos los métodos definidos en el interfaz.

Son similares a las clases abstractas en las que todos los métodos son abstractos pero también tienen diferencias con ellas:

- un interfaz no puede implementar ningún método, una clase abstracta sí
- una clase puede implementar cualquier no de interfaces pero solo tiene una superclase
- un interfaz no es parte de una jerarquía, clases que no tienen ninguna relación pueden implementar el mismo interfaz. Un interfaz puede ser usado 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 el mismo interfaz, Portable, que especifica los métodos que están disponibles para ambos. El interfaz Portable expresa el comportamiento que es común a los dispositivos portátiles. Define lo que hacen los objetos pero no cómo.

La definición siguiente es totalmente correcta:

```
ArrayList<Portable> elementos = new ArrayList<Portable> ();
elementos.add(new Portatil());
elementos.add(new TelefonoMovil());
elementos.add(new CamaraDigital()); // se ha definido la clase

                                    // public class CamaraDigital implements Portable
```

7.9.1 Definiendo un interfaz

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

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

cuerpo del interface

En UML un interface se denota con el estereotipo <<*interface*>>. En UML el nombre del interface se escribe en cursiva (Portable).

- Todos los métodos en un interfaz son abstractos (sin implementación). No se necesita especificar que lo son.
- No contienen métodos estáticos
- Los 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
- Pueden extender otros interfaces (en la API, el interface ***Set*** extiende el interface *Collection*)
- Se denotan habitualmente (no siempre) con adjetivos (*Comible, Movible, Editable, Observable, Comparable*)

7.9.2 Implementando un interfaz

Una clase puede heredar de un interfaz. Se dice que la clase **implementa** (implements) el interfaz. Cuando una clase implementa un interfaz debe **proporcionar implementación de todos los métodos** que define el interfaz.

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

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

}
```

Si una clase hereda de otra e implementa un interfaz la cláusula *extends* va en primer lugar.

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

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

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

        public  int   compareTo(Portatil otro)
        {
                ..........................
        }

        public  void transportar()
        {
                ..........................
        }

}
```

Si una clase implementa un interfaz cualquier objeto de esa clase es del tipo del interfaz.

- *Portable p;*
- *Portable miPortatil = new Portatil();*
- *Portatil miPortatil = new Portatil();*
- *p = miPortatil;*

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.

Los interfaces no proporcionan la primera ventaja porque no contienen implementación alguna de métodos. Pero sí la segunda.

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.

Los interfaces no tienen instancias directas pero sirven como supertipos para instancias de otras clases, las que implementan el interfaz.

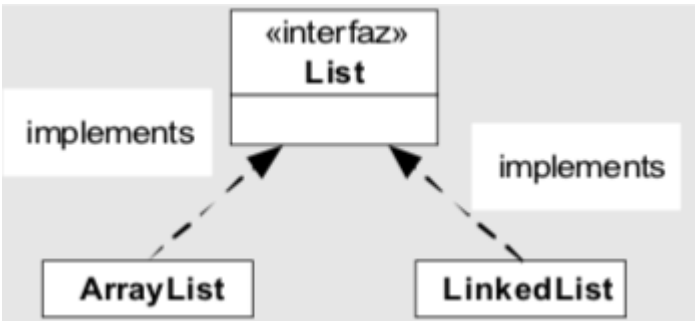
7.9.4 Interfaces como especificaciones

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

La característica más importante de un interfaz es que separa completamente la definición de la funcionalidad de una implementación.

Un interfaz realmente es un ***contrato*** entre el proveedor del comportamiento (el propio interfaz) y un usuario del mismo, la clase que implementa el interfaz. Esta se compromete, garantiza, que va a proporcionar la funcionalidad de todos los servicios que proporciona el interfaz 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.



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

Las clases *ArrayList* y *LinkedList* proporcionan diferentes implementaciones del mismo interfaz.

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 interfaz (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 ese interfaz.

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 es preferible utilizar interfaces ya que:

- permiten herencia múltiple
- proporcionan mayor flexibilidad

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

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

Los interfaces se utilizan como tipos referencia incluso si las clases que lo implementan forman parte de la misma jerarquía. La ventaja de utilizar un interfaz es que hablando en términos de interface el código no hace ninguna asunción acerca de la implementación de los métodos. Esto quiere decir que es muy fácil cambiar completamente la implementación que subyace sin que se vea afectada la aplicación.

Los interfaces se definen en sus propios ficheros *java*.

Ejercicio 7.17

Una serie de clases (en principio sin relación entre ellas) tienen en común dos métodos: ***getNombre() y print()***. ¿Cómo conseguimos un tipo común para todas ellas? Define el interfaz ***Nombrable***.

Indica que ***Contacto, Animal*** y ***Persona*** implementan el interfaz. Qué deben cumplir las clases ***Contacto***, ***Animal*** y ***Persona***?

¿Es correcto: *Nombrable n = new Contacto()*? Razona la respuesta.

La clase ***ContactoSocial*** hereda de ***Contacto***. ¿Es correcto:

Nombrable n = new ContactoSocial()? Razona la respuesta.

Sea List<Object> lista = new ArrayList<Object>() una lista de objetos de cualquier tipo algunos de los cuales implementan el interfaz ***Nombrable***. Escribe el método,

public void printNombrables(String nombre)

que visualiza aquellos objetos cuyo nombre coincide con el parámetro (utiliza un iterador para la lista).

Solución

Enlace con el proyecto

- Qué deben cumplir las clases Contacto, Animal y Persona? --> *Deben de tener implementados todos los métodos de la interfaz Nombrable*
- ¿Es correcto: Nombrable n = new Contacto()? Razona la respuesta. --> *Si. Las interfaces se pueden utilizar como tipos de referencia.*
- La clase ContactoSocial hereda de Contacto. ¿Es correcto: Nombrable n = new ContactoSocial()? Razona la respuesta. --> *Si. Es un tipo de la clase genérica. No dará error de compilación. Podría dar error de ejecución.*

Ejercicio 7.18

Las figuras de la jerarquía definida anteriormente pueden moverse horizontal y verticalmente y para ello cumplen el contrato definido por el interfaz Movible.

```
public interface Movible {

    public void moverHorizontal(int desp);

    public void moverVertical(int desp);

}
```

El movimiento horizontal y vertical se traduce en incrementar / decrementar la coordenada del centro de la figura en el valor del desplazamiento.

Dentro de la clase ***GestorFiguras*** define un método ***testMovibles()*** y define un array de objetos de tipo ***Movible***. Instancia círculos, cuadrados y triángulos y muestra sus características. Mueve los objetos horizontal y verticalmente y vuelve a mostrar sus datos.

Solución

Enlace con la solución

7.9.6 Usando interfaces y clases abstractas

Un interface puede ser usado para expresar el comportamiento común de un conjunto de clases. Cada clase debe proporcionar una implementación completa del comportamiento. Cualquier método declarado en el interface debe ser definido en cada clase que implemente dicho interface. Esto puede generar algunos problemas:

- código similar se escribe en varias clases (poca reutilización)
- falta de consistencia entre implementaciones de métodos similares

Una buena solución es usar un interface y una clase abstracta:

- el interface se usa como tipo referencia – es el contrato que indica lo que la clase debe poder hacer
- la clase abstracta proporciona implementaciones básicas de métodos comunes que pueden ser heredados o redefinidos por las clases concretas.

7.10 Interfaces Comparable, Cloneable, Comparator

Java define algunos interfaces que incluyen métodos que pueden ser utilizados por cualquier clase que implemente el interfaz. Todos se encuentran en el paquete *java.lang*.

7.10.1 Interface Comparable

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

- < 0 si es menor (el receptor) == 0 si son iguales
- > 0 si es mayor (el receptor)

```
public interface Comparable<T> {

    public int compareTo(<T> obj);

}
```

<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) {

        if (this.velocidad == otro.getVelocidad())

            return 0;

        if (this.velocidad < otro.getVelocidad())

            return -1;

        return 1;

    }

}
```

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().*

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

Ejercicio 7.19

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.

Solución

Enlace con la solución del ejercicio

7.10.2 Interface Cloneable

Para hacer una copia completa, “deep copy”, de un objeto hay que clonar el objeto, hacer una copia local del mismo. El método clone(), definido en la clase Object, permite realizar esta clonación.

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:

protected Object clone()
throws CloneNotSupportedException //así está definido en Object

La excepción se produce si el objeto a clonar no implementa el interface Cloneable o no es una instancia de la clase que lo implementa.

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). Hace una “shallow copy” (copia no profunda).

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

        @override
        public Coche  clone()
        {
                try
                {
                        return (Coche) super.clone();
                }
                catch (CloneNotSupportedException e)
                {
                        return null;
                }
        }
}

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

La excepción *CloneNotSupportedException* se lanza si el objeto a clonar no es una instancia de la clase que implementa el interfaz *Cloneable*.

7.10.3 Interface Comparator

Es posible ordenar un conjunto de objetos (en una colección, o un array, ...) que no implementen el interface Comparable o 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> {

    public int compare(T obj1, T obj2);

}
```

El método compara los dos argumentos devolviendo un resultado:

- < 0 si el primer objeto es menor que el segundo (según el criterio de ordenación 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 ComparadorCoches implements Comparator<Coche> {

    public int compare(Coche c1, Coche c2) {

        if (c1.getVelocidad() < c2.getVelocidad())

            return -1;

        if (c1.getVelocidad() == c2.getVelocidad())

            return 0;

        return 1;

    }

}
```

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

Podemos hacer lo mismo con una clase anónima:

```
Collections.sort(lista, new Comparator<Coche>() {

    public int compare(Coche c1, Coche c2) {
```

```
            .....
        }
} );
```

Ejercicio 7.20

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*.

Solución

Enlace con la solución del ejercicio