

UT7

Herencia y polimorfismo. Interfaces.

Módulo - Programación (1º)

Ciclos - Desarrollo de Aplicaciones Multiplataforma | Desarrollo de Aplicaciones Web

CI María Ana Sanz

Contenidos

- Herencia
 - Herencia simple y múltiple
 - Herencia y derechos de acceso
 - Herencia y constructores
 - Ventajas de la herencia
- Herencia y subtipos
 - Polimorfismo
- Colecciones polimórficas
- Polimorfismo y redefinición de métodos
 - Tipo estático y dinámico
 - Overriding

Contenidos

- Sobrecarga y overriding
 - Métodos polimórficos
- La clase Object
 - Métodos de la clase Object
- protected / final
- Clases abstractas
- Interfaces
 - Definición
 - Implementación
 - Interfaces como tipos
- Interfaces Comparable, Comparator, Cloneable
- Interfaces en Java 8

Características de la P00

- **Abstracción**

- manejar la complejidad
- extraer los aspectos esenciales ignorando detalles

- **Encapsulamiento**

- agrupar los elementos de una abstracción que constituyen su estructura y comportamiento
- mantener oculta la implementación de la abstracción

- **Modularización**

- dividir el programa en módulos para facilitar su diseño, mantenimiento y reutilización

- **Jerarquía**

- ordenar las abstracciones, organizarlas, simplificando así su desarrollo
- permite la implementación de la herencia y el polimorfismo

Relaciones entre clases vistas hasta ahora

- Asociación



- Agregación



- Composición



- Cualquier ejemplo con colecciones se ajustaría a una relación de asociación, agregación o composición
 - las tres se implementan de la misma manera

Qué es la herencia

■ Herencia

- mecanismo de la POO que nos permite definir una clase como extensión de otra
- la nueva clase hereda toda la estructura y comportamiento de la clase inicial

■ Objetivo

- reutilizar el código existente

Qué es la herencia

Herencia

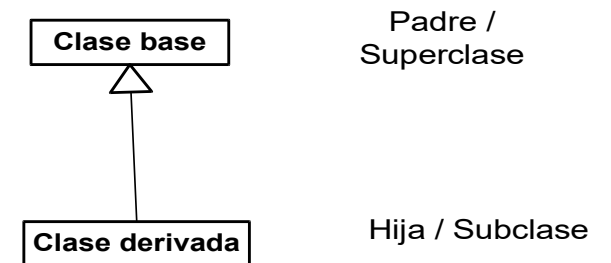
de estructura (por extensión)
extends

de interfaces (de subtipos)
implements

Qué es la herencia

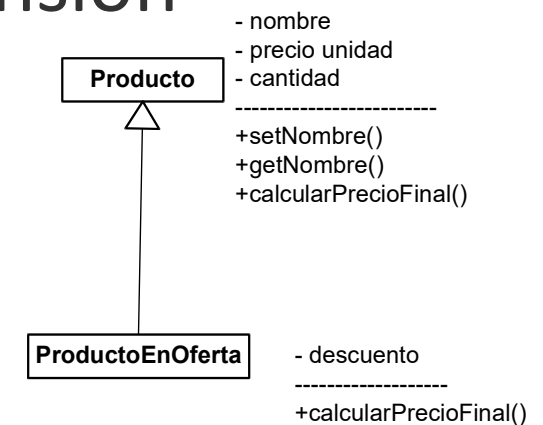
- **Clase base, superclase, clase padre**

- la clase de la que se hereda

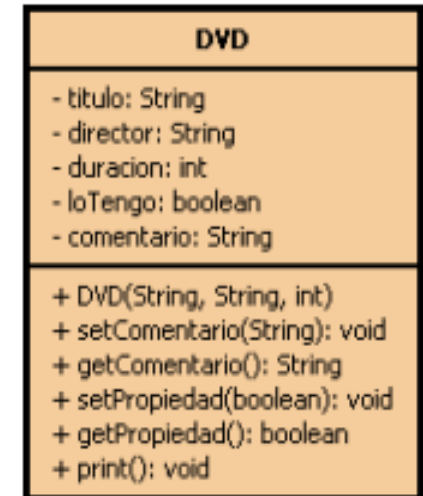
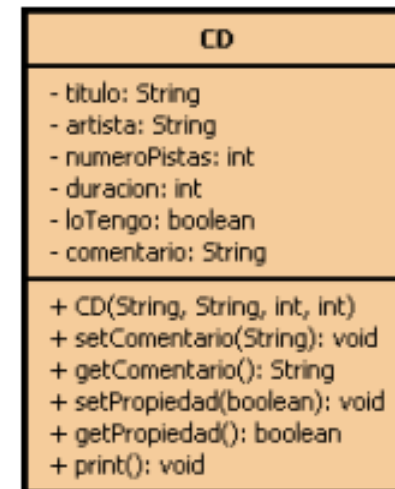
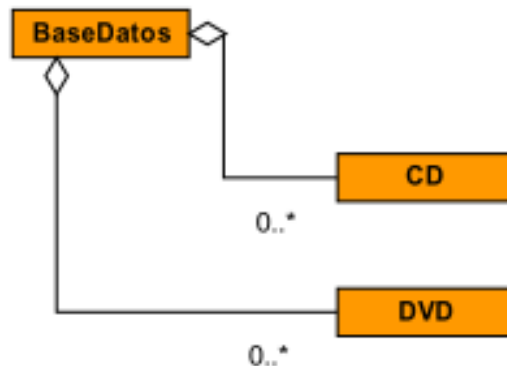
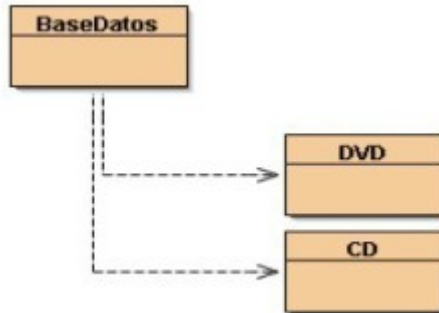


- **Clase derivada, subclase, clase hija**

- la clase que se obtiene como extensión



Ejemplo - Proyecto BD de CD y DVD



código fuente muy similar

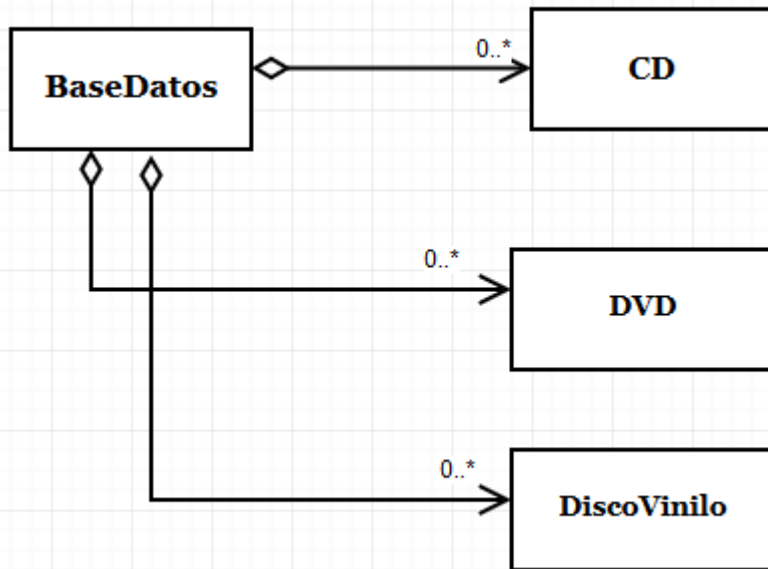
Ejemplo – Proyecto BD de CD y DVD

```
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();
    }
}
```

Ejemplo – Proyecto BD de CD y DVD – Y si añadimos una nueva clase DiscoVinilo?



más repetición de código

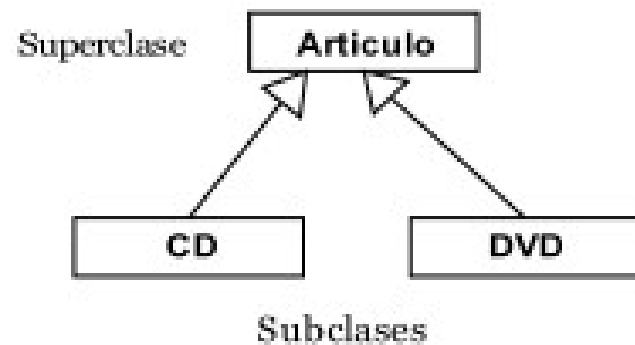
```
import java.util.ArrayList;
public class BaseDatos
{
    private ArrayList<CD> cds;
    private ArrayList<DVD> dvds;
    private ArrayList<DiscoVinilo> vinilos;

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

    public void addVinilo(DiscoVinilo elVinilo)
    {
        vinilos.add(elVinilo);
    }

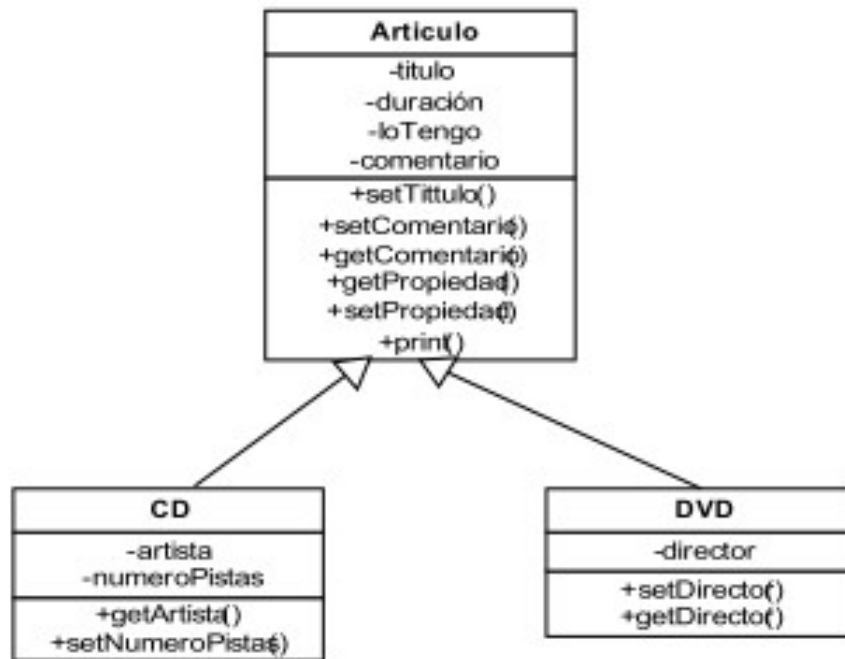
    .....
}
```

Ejemplo - Proyecto BD de CD y DVD - Solución



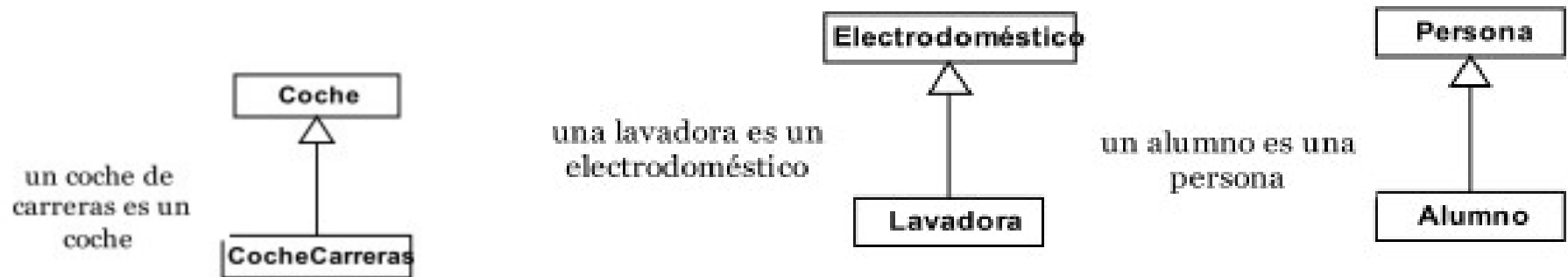
- CD y DVD poseen características comunes
 - se puede construir una superclase **Articulo** que incluya lo común a las dos
 - a partir de ésta, derivar dos subclases, **CD** y **DVD**
 - **CD** y **DVD** heredarán todo lo que incluya la clase **Articulo** y añadirán cada una lo que es propio de ellas

Ejemplo - Proyecto BD de CD y DVD - Solución



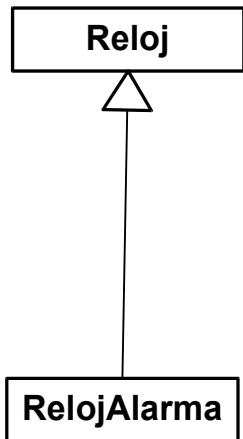
- La herencia se denomina relación **es-un**
 - una subclase es una especialización de la superclase
 - un CD es un Artículo,
 - un vídeo es un Artículo
- La superclase expresa la estructura y comportamiento comunes a las subclases

Más ejemplos



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

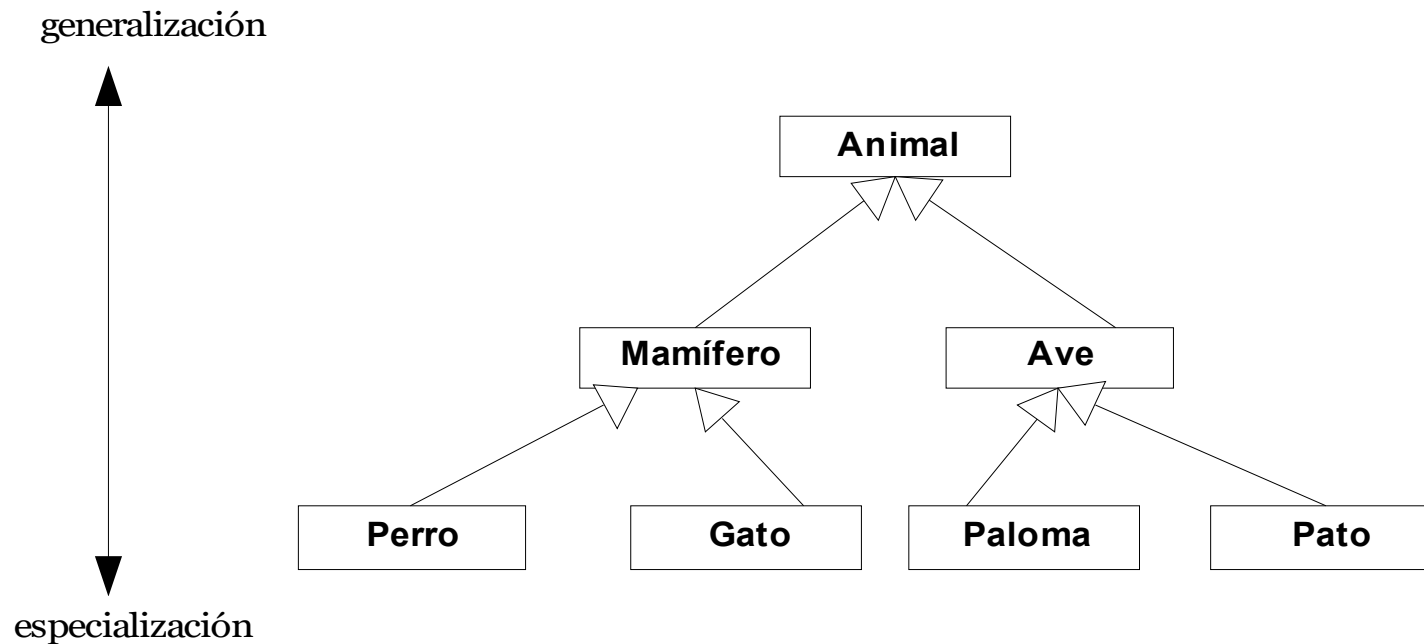
Más ejemplos



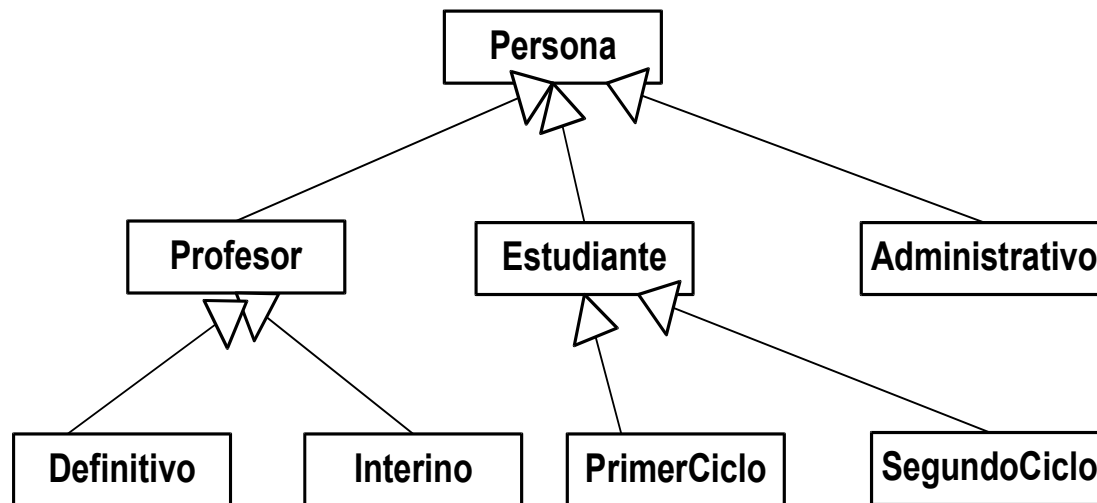
- una subclase aumenta y/o redefine la estructura y el comportamiento de la superclase
 - añade o modifica lo que hereda de su clase padre
- la superclase Relej
 - atributo hora
 - métodos para poner la hora y obtenerla
- la subclase RelejAlarma aumenta el comportamiento de la clase Relej
 - emite una alarma a una hora prefijada
 - añade un atributo adicional, la hora de alarma, otro para indicar si está o no activada y los métodos para fijar la alarma y activarla / desactivarla.

Más ejemplos

jerarquías de generalización/
especialización



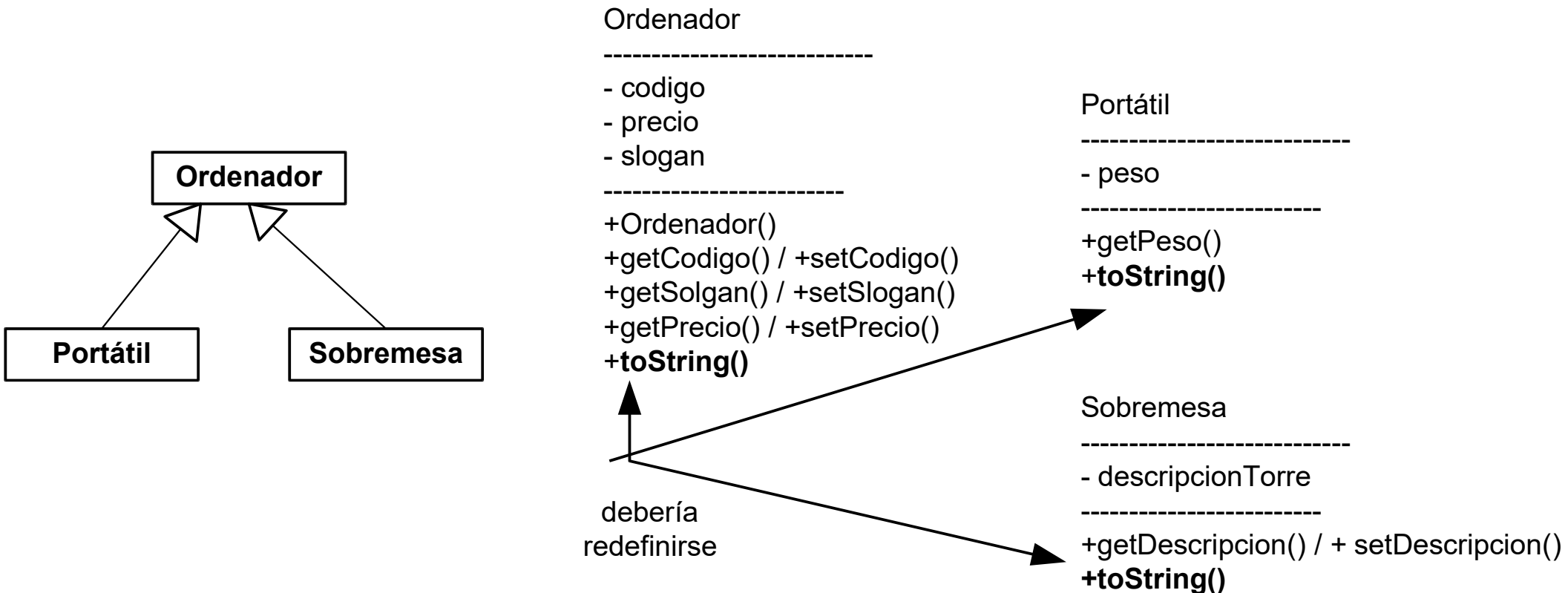
Ejer 7.1



Ejercicio en papel

- En una tienda se venden dos tipos de ordenadores: portátiles y de sobremesa
- Ambos tipos se caracterizan por su código y su precio
- Los dos tienen un slogan, *“Ideal para viajes”*, *“el que más pesa pero el que menos cuesta”*
- Los portátiles tienen como característica su peso y los de sobremesa la descripción de la torre
- Dibuja el diagrama (UML) que representa la relación de herencia entre las clases
- Detalla lo que es común y lo que es propio de cada clase

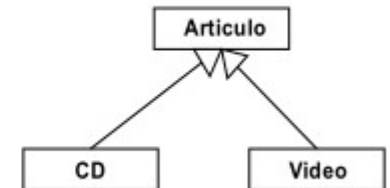
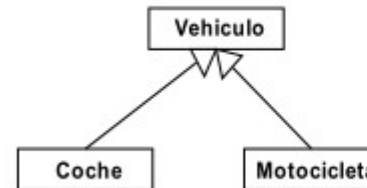
Ejercicio en papel



Herencia simple y múltiple

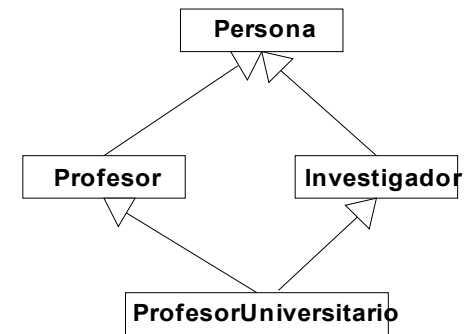
■ Simple

- una subclase hereda solo de una superclase
- Java, C#, VisualBasic.Net



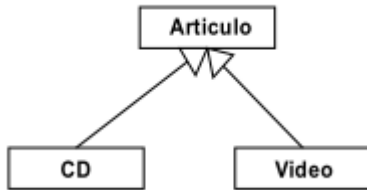
■ Múltiple

- una clase hereda de más de una superclase
- C++
- Java no lo permite, lo simula con interfaces



Implementación de la herencia en Java

- extends -



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

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

hereda todo lo que tiene
la clase Artículo y añade
dos atributos propios

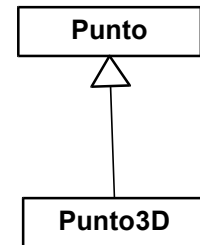
Ejer 7.2 (en papel)

```
public class DVD extends Articulo
{
    private String director;
}
```

Ejer 7.3 (en papel)

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

```
public class Punto3D extends Punto
{
    private int z;
}
```



Ejer Ordenador (en papel)

```
public class Ordenador
{
    private String codigo;
    private double precio;
    private String slogan;

    .....
}
```

```
public class Portatil
    extends Ordenador
{
    private double peso;

}

public class Sobremesa
    extends Ordenador
{
    private String descripcion;

}
```


Herencia y derechos de acceso

- Miembros public (atributos y métodos)
 - accesibles desde objetos de otras clases
- Miembros private (atributos y métodos)
 - accesibles desde dentro de la clase
 - no accesibles desde fuera
- ¿Qué ocurre con los miembros private en relaciones extends?

Visibilidad `protected` (#)

- Una subclase no puede acceder a los miembros privados de su superclase
 - a) un método de la subclase podrá acceder a los atributos privados heredados de la superclase a través de sus accesores / mutadores o
 - b) definir el atributo (o método) como **protected**

+ **public**
- **private**
~ **package**
protected

!!!OJO con protected!!! - De alguna manera viola la encapsulación. Elegiremos opción a) para atributos y opción b) para métodos.

Ejemplo

```
public class Punto3D extends Punto
{
    private int z;
    .....
    public void escribir()
    {
        System.out.println("(x, y, z)" +
    }
}
```

error, x e y private
en la superclase

x + " " + y + " " + z)

- utilizaremos los accesores / mutadores de la superclase
- si hay un método private en la superclase para acceder desde la subclase hay que definirlo como protected

Ejemplo

```
public class Punto3D extends Punto
{
    private int z;
    .....
    public void escribir()
    {
        System.out.println("(x, y, z)" +
                           getX() + " " + getY() + " " + z);
    }
}
```

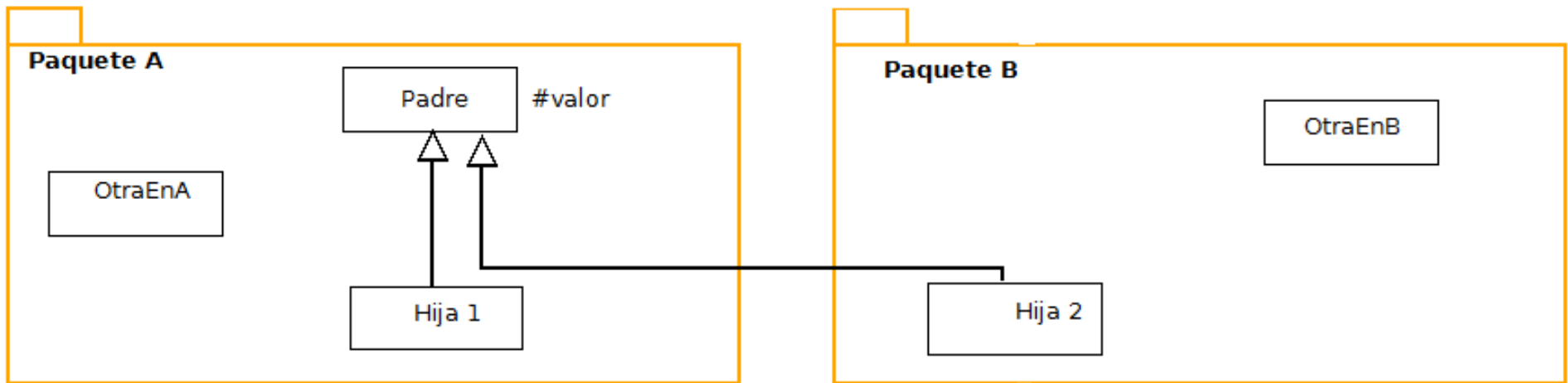
Herencia y derechos de acceso

<i>Especificador</i>	<i>clase</i>	<i>subclase</i>	<i>paquete</i>	<i>el resto (el mundo)</i>
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.

Herencia y derechos de acceso



Herencia y derechos de acceso

	Padre	Hija 1	Hija 2	OtraEnA	OtraEnB
valor					
-					
+					
#					
~					

Herencia y derechos de acceso

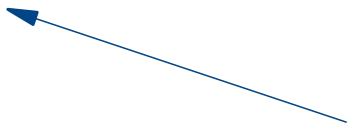
	Padre	Hija 1	Hija 2	OtraEnA	OtraEnB
valor					
-	X	-	-	-	-
+	X	X	X	X	X
#	X	X	X	X*	-
~	X	X	-	X	-

* viola la encapsulación

Herencia y constructores – comentarios previos

- Comentarios acerca de los constructores
 - Java incluye un constructor sin parámetros si no se especifica uno
 - es el constructor por defecto
 - (ver ejemplo)

```
public class PersonaSinConstructor
{
    private String nombre;
    private int edad;
    private double sueldo;
}
```

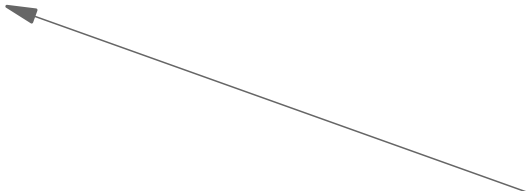


incluye un constructor
por defecto

Herencia y constructores – comentarios previos

```
public class PersonaConConstructoresSobrecargados
{
    private String nombre;
    private int edad;
    private double sueldo;

    /**
     * Constructor por defecto, es el constructor sin parámetros
     */
    public PersonaConConstructoresSobrecargados()
    {
        this ("Ana", 18, 1200);
    }
}
```



se utiliza la referencia **this**
para llamar a otro constructor
de la misma clase.
this siempre como primera sentencia

Herencia y constructores – comentarios previos

```
public class PersonaConConstructoresSobrecargados
{
    .....

    public PersonaConConstructoresSobrecargados(String nombre, int edad)
    {
        this (nombre, edad, 1200);
    }

    public PersonaConConstructoresSobrecargados(String nombre, int edad,
                                                double sueldo)
    {
        this.nombre = nombre;
        this.edad = edad;
        this.sueldo = sueldo;
    }
}
```

Herencia y constructores

- los constructores no se heredan
- hay que especificar un constructor para la subclase
- el constructor de una subclase debe llamar siempre al constructor de la superclase
 - incluyendo una llamada al constructor de su clase padre con la cláusula **super**
 - **debe ser la primera sentencia** en el constructor de la subclase.

Herencia y constructores

```
public class Articulo
{
    private String titulo;
    private int duracion;
    private boolean 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;
    }
}
```

llamada al constructor de la superclase

this y super no pueden ir juntos

¿Qué ocurre si no llamo a `super()` ?

- el compilador inserta automáticamente una llamada para asegurar que los atributos heredados de la superclase se inicializan adecuadamente
- sólo funciona si la superclase tiene un constructor sin parámetros
- buena práctica incluir llamadas explícitas incluso en este último caso

Ejer 7.4.

```
public class DVD extends Artículo
{
    private String director;
    public DVD(String titulo, int duracion,
               boolean loTengo, String comentario,
               String director)
    {
        super(titulo, duracion, loTengo, comentario);
        this.director = director;
    }
}
```

Ejer 7.4.

```
public class Punto3D extends Punto
{
    private int z;
    public Punto3D(int x, int y, int z)
    {
        super(x, y);
        this.z = z;
    }
}
```


Ejer Portátil

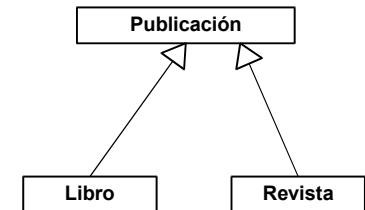
```
public class Portatil extends Ordenador
{
    private double peso;
    public Portatil(String codigo, double precio,
                    String slogan, double peso)
    {
        super(codigo, precio, slogan);
        this.peso = peso;
    }
}
```

- Define un portátil e instáncialo

```
Portatil miPortatil = new Portatil("45AX", 1200, "mmmmm", 2.5);
```

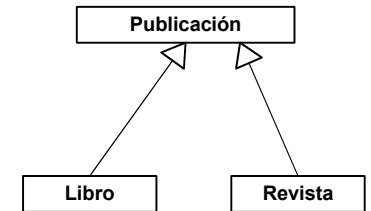
Ejer 7.5.

```
public class Publicación
{
    private String titulo;
    private LocalDate fecha;
    public Publicación(String titulo, LocalDate fecha)
    {
        this.titulo = titulo;
        this.fecha = fecha;
    }
}
```



Ejer 7.5.

```
public class Libro extends Publicación
{
    private String autor;
    private String ISBN;
    public Libro(String titulo, LocalDate fecha, String autor,
                  String ISBN)
    {
        super(titulo, fecha);
        this.autor = autor;
        this.ISBN = ISBN;
    }
}
```

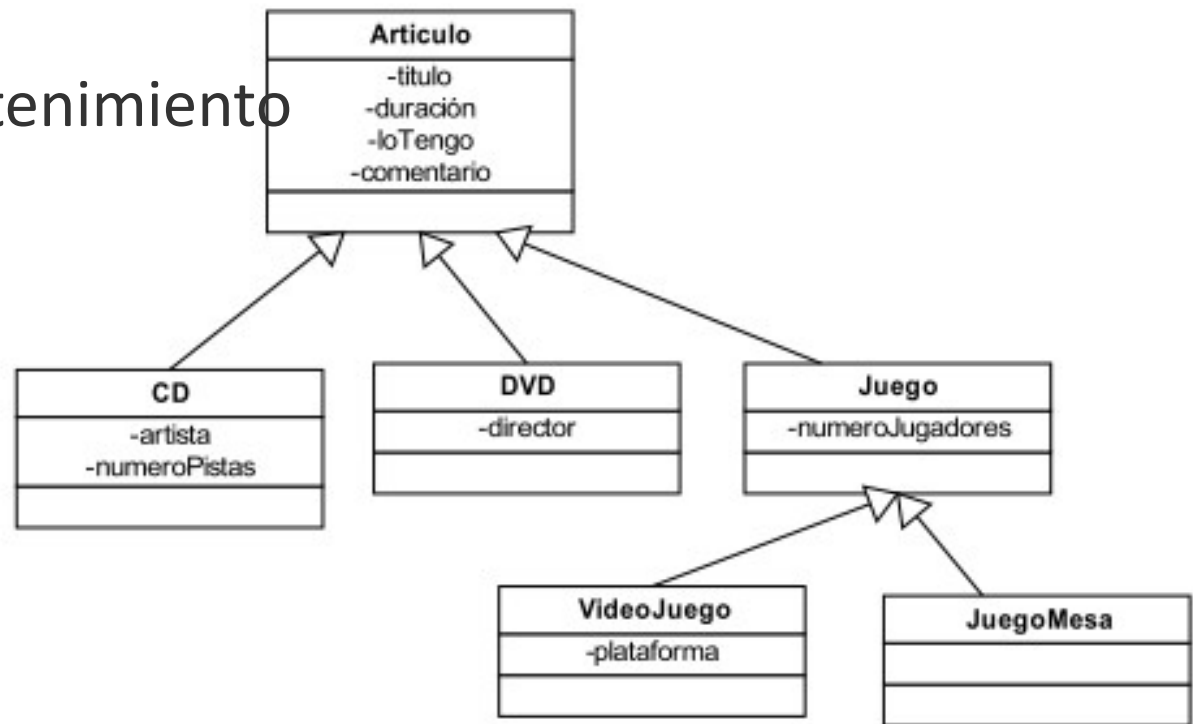


- Define un libro e instáncialo

```
Libro libro = new Libro("La catedral del mar",
                        fechaPublicacion, "Ildefonso Falcones", "48T57-B");
```

Ventajas de la herencia

- evitar la duplicación de código
- reutilizar el código
- hacer más fácil el mantenimiento
- extensibilidad



Proyecto BD Artículos

```
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 de datos
     */
    public void addArticulo(Articulo elArticulo)
    {
        articulos.add(elArticulo);
    }
}
```

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

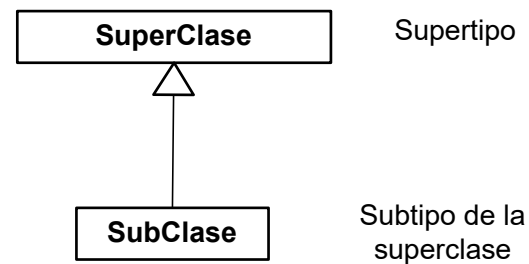
Proyecto BD Artículos

- Antes teníamos
 - `public void addCD(CD elCD)`
 - `public void addDVD(DVD elDVD)`
- Ahora tenemos:
 - `public void addArticulo(Articulo elArticulo)`
- Ahora añadiremos
 - `DVD miDVD = new DVD();`
 - `miBaseDatos.addArticulo(miDVD);`

Ejer 7.7 y 7.8 (en el ordenador)

Herencia y subtipos

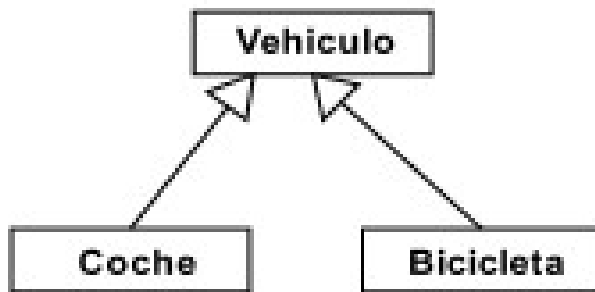
- Las clases definen tipos
- En una relación de herencia
 - jerarquía de clases y
 - jerarquía de tipos
- El tipo definido por una subclase es un subtipo del tipo definido por su superclase



Herencia y subtipos

- 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**

Herencia y subtipos



- `Coche miCoche = new Coche();`
 - `Coche` es el tipo declarado, tipo en tiempo de compilación de *miCoche*
- `Vehiculo unVehiculo = new Vehiculo();`

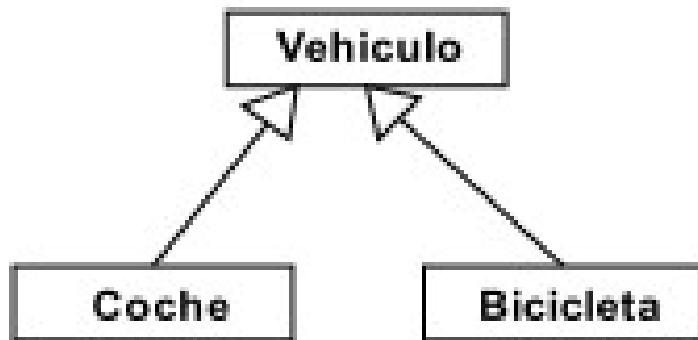
- `Vehiculo miVehiculo = new Coche();`
 - en ejecución *miVehiculo* apunta a un objeto de tipo `Coche`

tipo declarado,
tipo en tiempo de compilación

tipo en tiempo de ejecución, apunta
a un objeto de una subclase

Herencia y subtipos

- `Vehiculo tuVehiculo = new Bicicleta();`



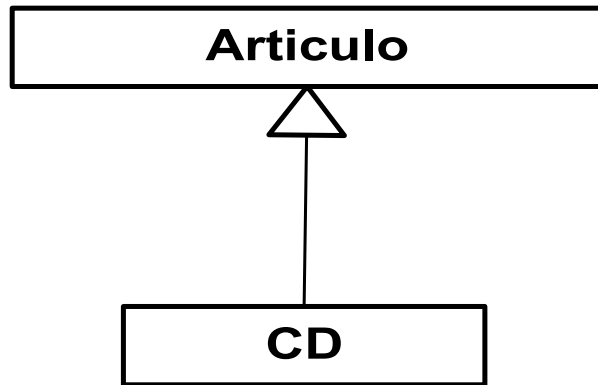
- El tipo de la variable declara lo que puede almacenar
 - Si se declara de tipo Vehiculo la variable podrá referenciar vehículos
 - un coche es un vehículo, es correcto que pueda referenciar a un objeto de tipo Coche

Principio de sustitución de Liskov

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

Principio de sustitución de Liskov



Todo CD es un artículo
pero todo artículo no es
un CD

```
CD unCd = new CD(.....);
Articulo art = new Articulo();
Articulo art2 = new CD();
```

tipo en tiempo
de compilación

tipo en tiempo
de ejecución,
es un CD

- `public void addArticulo(Articulo a)`
 - el parámetro `a` puede recibir un CD, un DVD

Ejer 7.9

- Es correcto? `Coche unCoche = new Vehiculo();`

- **NO**

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

- Es correcto? `Gato g = new Gato();`

- **SI**

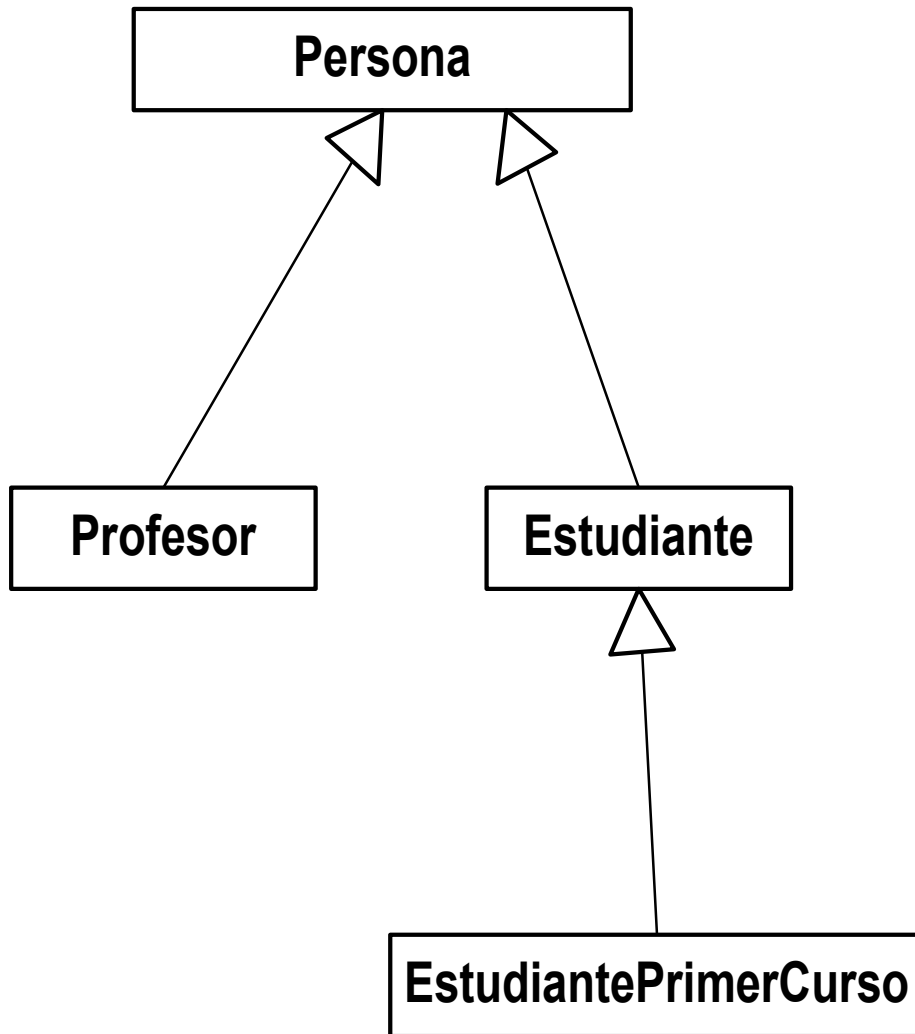
- `GatoConBotas felix = new GatoConBotas();`

- **SI**

- `GatoConBotas miGato = new Gato();`

- **NO**

Ejer 7.10



1. SI
2. SI
3. NO
4. NO
5. SI
6. NO
7. NO
8. SI, asumiendo una
instanciación correcta de
estu1
9. NO
10. SI

Subtipos y paso de parámetros.

Subtipos y valores de retorno.

- El principio de sustitución se aplica
 - al efectuar un paso de parámetros
 - al devolver un valor
 - puedo devolver un valor de un tipo de una subclase de la clase indicada en el tipo de retorno.

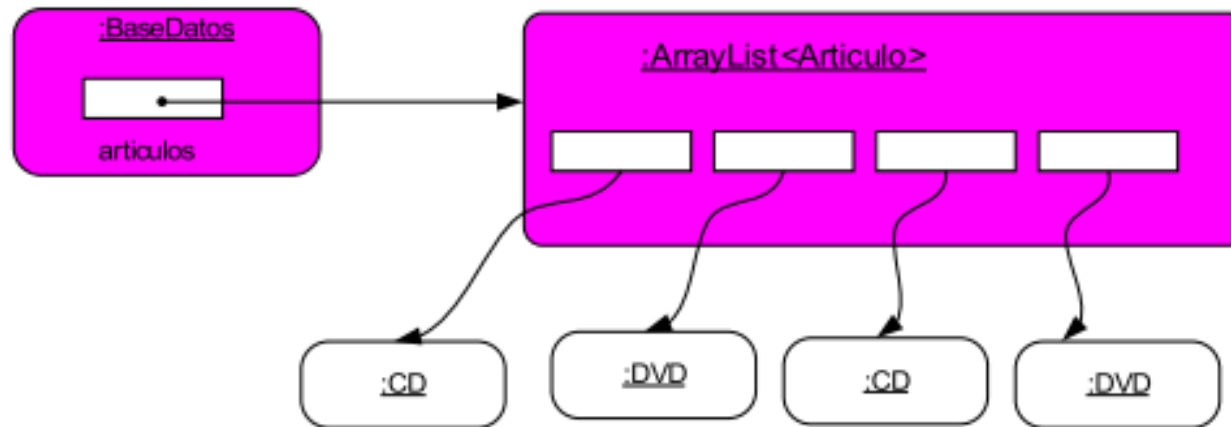
```
public void addArticulo(Articulo elArticulo) // se añadirá un CD o un DVD
{
    articulos.add(elArticulo);
}
```

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


Subtipos y paso de parámetros.

Subtipos y valores de retorno.

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



Ejer 7.11.

```
public void limpiarGato(Gato g)
{
}
}
```

- Con las declaraciones de 7.9.b)
 - ¿Es correcto ?
limpiarGato(felix);
 - SI
 - limpiarGato(g);
 - SI

Variables polimórficas

- **Polimórficas** – múltiples formas
 - variables que referencian diferentes tipos de objetos
- 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.

Variables polimórficas

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

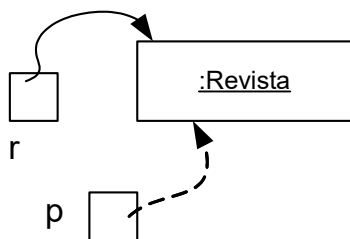
variable polimórfica,
declarada de tipo Articulo pero al
recorrer el ArrayList va apuntando
bien a un CD o a un DVD.

Casting

- ¿Es posible asignar un valor de un supertipo (de una superclase) a una variable declarada de un subtipo (de una subclase)?
 - NO, no podemos asignar un supertipo a un subtipo.



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



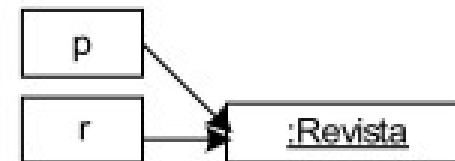
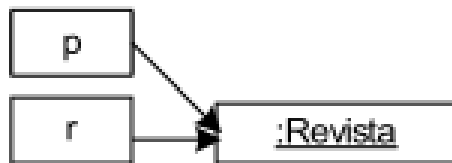
Publicacion p;

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

Casting

- Para poder efectuar la asignación
 - **casting**
 - sólo funcionará si la publicación p en ejecución es una revista. Si no es así en ejecución se lanza la excepción `ClassCastException`

4) `r = (Revista) p;`

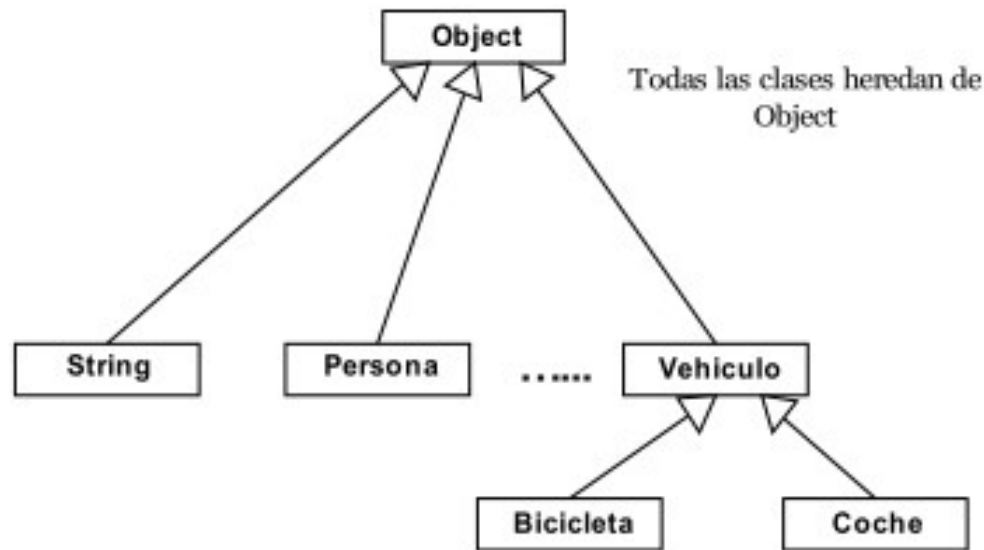


- Cuidado con el casting, no abusar, utilizar métodos polimórficos

Ejer 7.12.

- `Publicacion p = new Publicacion();`
 - `Revista r = new Revista();`
 - `Libro l = new Libro();`
 - `p = l; ?? Sin error`
 - `l = p; ?? Error en compilación (puedo hacer cast)`
 - `r = l; ?? Error en compilación (sin posibilidad de hacer cast)`
 - `r = (Revista) p; ?? Error en ejecución, p apunta a un libro`
-
- `Vehiculo v;`
 - `Coche c = new Coche();`
 - `v = c; // sin error`
 - `c = v; // Error en compilación (puedo hacer cast)`

Clase Object



- en *java.lang*
- raíz de todas las clases en la jerarquía de Java
- todas las clases derivan de **Object**
 - cuando no se indica explícitamente de quién deriva una clase se asume que lo hace de **Object**

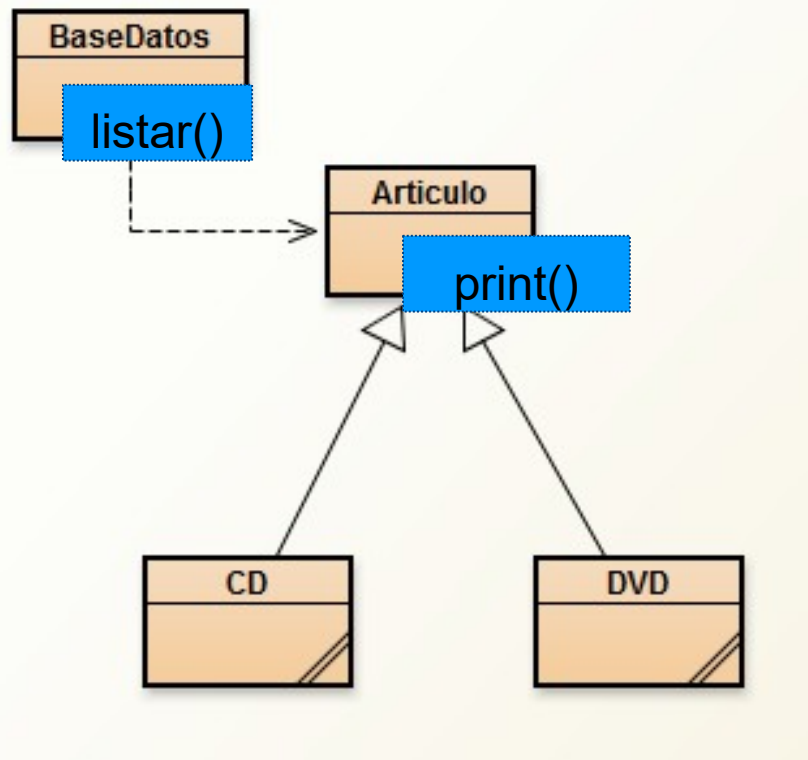
Por qué tener Object?

- 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.

Colecciones polimórficas

- A partir de Java 1.5
 - todas las colecciones genéricas parametrizadas
 - `ArrayList<Integer> col = new ArrayList<Integer>();`
- Podríamos definir
 - `ArrayList<Object> col = new ArrayList<Object>();`
 - una colección así podría contener objetos de cualquier tipo
- En la API encontramos
 - `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

Polimorfismo y redefinición de métodos



Lo que queremos

CD: From this moment on(78 mtos) *
Diana Krall
Pistas: 14
Mi disco favorito

DVD: El laberinto del faun(108 mtos) *
Guillermo del Toro
La mejor película de fantasía

Lo que tenemos

Titulo: From this moment on(78 mtos) *
Mi disco favorito

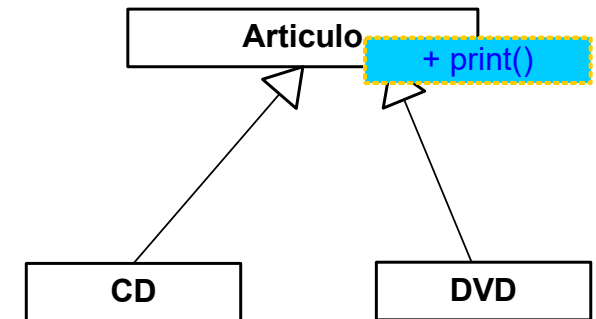
Titulo: El laberinto del faun(108 mtos) *
La mejor película de fantasía

- Al ejecutar **listar()** se muestra lo común pero no lo específico de cada artículo
- En la herencia hay métodos que debemos redefinir en las clases hijas **overriding**

Polimorfismo y redefinición de métodos

```
public class BaseDatos
{
    private ArrayList<Articulo> articulos;
    .....
    public void addArticulo(Articulo elArticulo)
    {
        articulos.add(elArticulo);
    }

    public void listar()
    {
        for(Articulo articulo: articulos)
        {
            articulo.print();
        }
    }
}
```



unas veces CD y
otras DVD, por el
principio de sustitución
de subtipos

se ejecuta el de la
clase padre, el heredado

Polimorfismo y redefinición de métodos

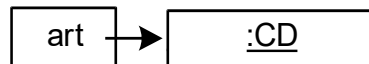
```
public class Artículo
{
    public void print()
    {
        System.out.print("Título: " + titulo + " (" + duracionTotal + " mins)");
        if (loTengo()) {
            System.out.println("*");
        }
        else {
            System.out.println();
        }

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

Tipo estático y tipo dinámico

- **Tipo estático** de una variable
 - el tipo declarado, su tipo en tiempo de compilación.
 - `Articulo art; // tipo estático`
- **Tipo dinámico** de una variable
 - el tipo real del objeto al que referencia la variable en ejecución, su tipo en tiempo de ejecución
 - `Articulo art = new CD(); // tipo dinámico, apunta a un objeto`

`Articulo art = new CD();`



tipo estático – `Articulo`

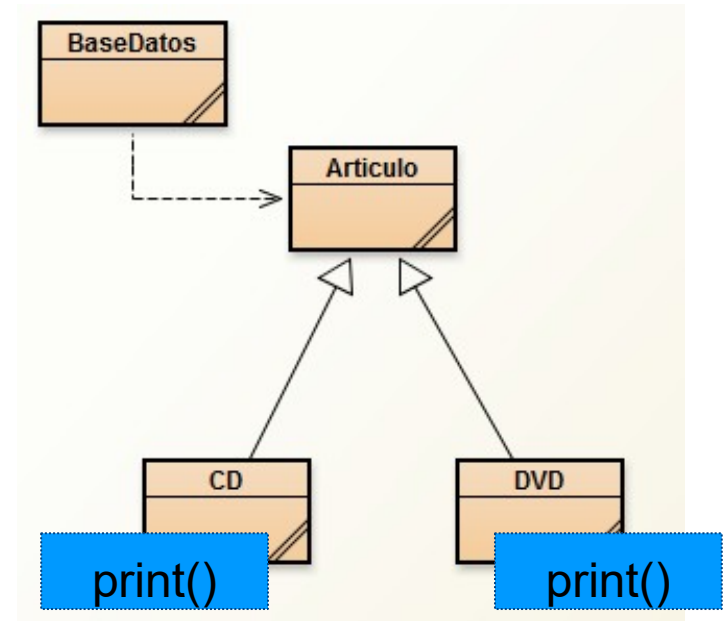
tipo dinámico - `CD`

Tipo estático y tipo dinámico

```
public class BaseDatos
{
    private ArrayList<Articulo> articulos;

    public void listar()
    {
        for(Articulo articulo: articulos)
        {
            articulo.print();
        }
    }
}
```

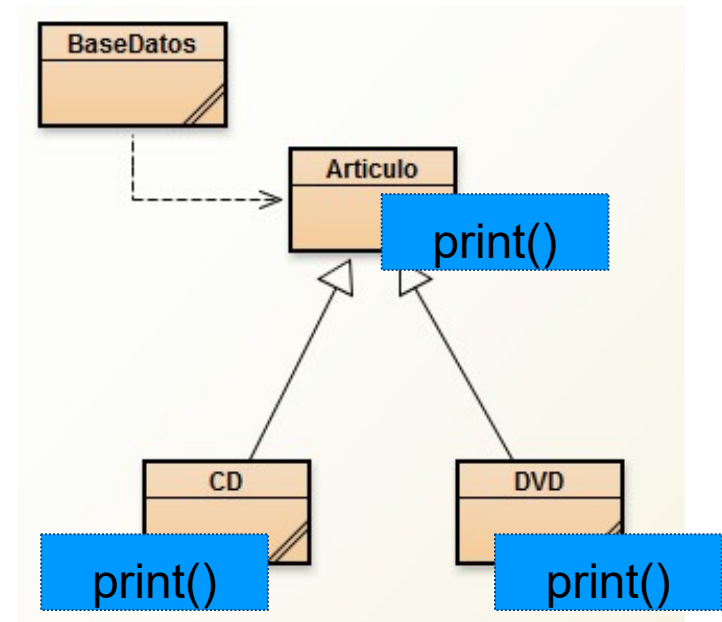
error, a es de tipo Articulo en compilación. El compilador, en tiempo de compilación, va a la clase Articulo a buscar print() para enlazar la llamada



Llevamos print() a las subclases como alternativa

Redefinición de métodos (overriding)

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



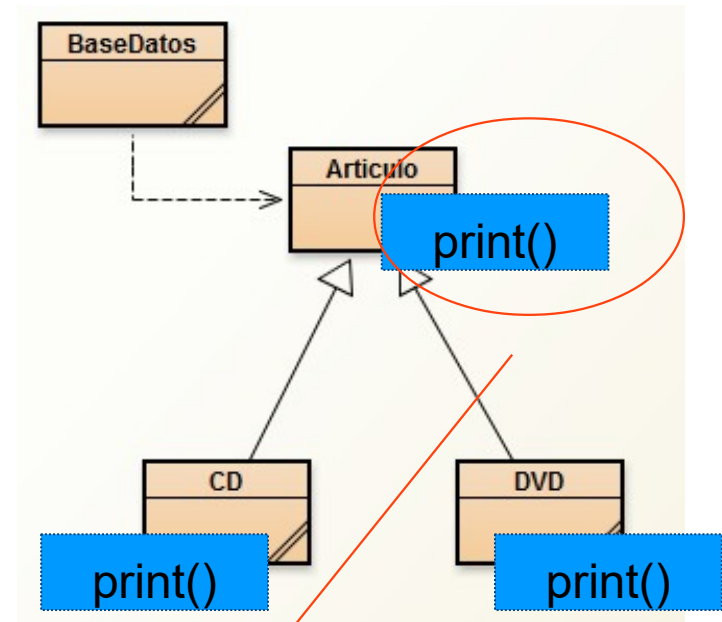
Se mantiene `print()` en la clase padre pero se redefine en las subclases.

CD y DVD dan sus propias versiones de `print()`

Redefinición de métodos (overriding)

```
public class BaseDatos
{
    private ArrayList<Articulo> articulos;

    public void listar()
    {
        for(Articulo articulo: articulos)
        {
            articulo.print();
        }
    }
}
```



Qué print() se ejecuta,
el de CD, el de DVD, el de Articulo?

Hasta la ejecución no se sabe,
el enlace del método se pospone hasta
la ejecución

el compilador no
da error,
hay un print() en
la clase padre

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 redefinido
- el método que se redefine en la subclase tiene preferencia sobre el heredado

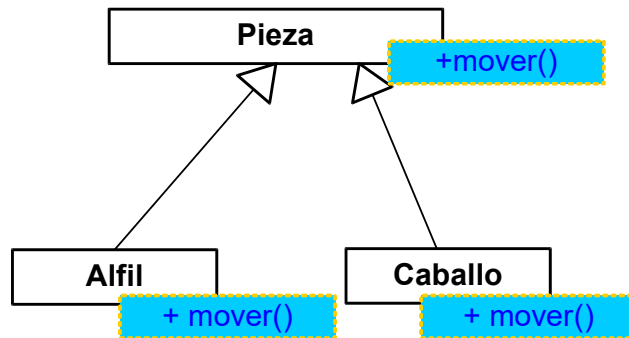
Consideraciones al redefinir un método

- al implementar el método redefinido es posible llamar al heredado a través de super
 - `super.print();`
 - 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

Consideraciones al redefinir un método

- Si hago, `CD unCd = new CD();`
 - `unCd.print();` // se ejecuta el de CD, tiene prioridad
- Si hago, `Articulo a = new CD();`
 - La resolución del método a ejecutar queda pendiente hasta la ejecución
 - si es un CD se ejecuta `print()` de CD
 - si es un DVD `print()` de DVD
 - si es Articulo `print()` de Articulo

Consideraciones al redefinir un método



- `Pieza p =;`
- `p.mover();`

Diferencia sobrecarga y redefinición

■ Sobrecarga

```
public class Demo
{
    public void calcular(int x, int y)
    public void calcular(int x, int y,
                        int z)
}
```

**varios métodos con el mismo nombre
pero distinta signatura**

■ Redefinición

```
public class Pieza
{
    public void mover()
}
public class Alfil extends Pieza
{
    public void mover()
}
```

**un método en la clase padre y otro
en la clase hija con el mismo nombre y
la misma signatura**

Polimorfismo

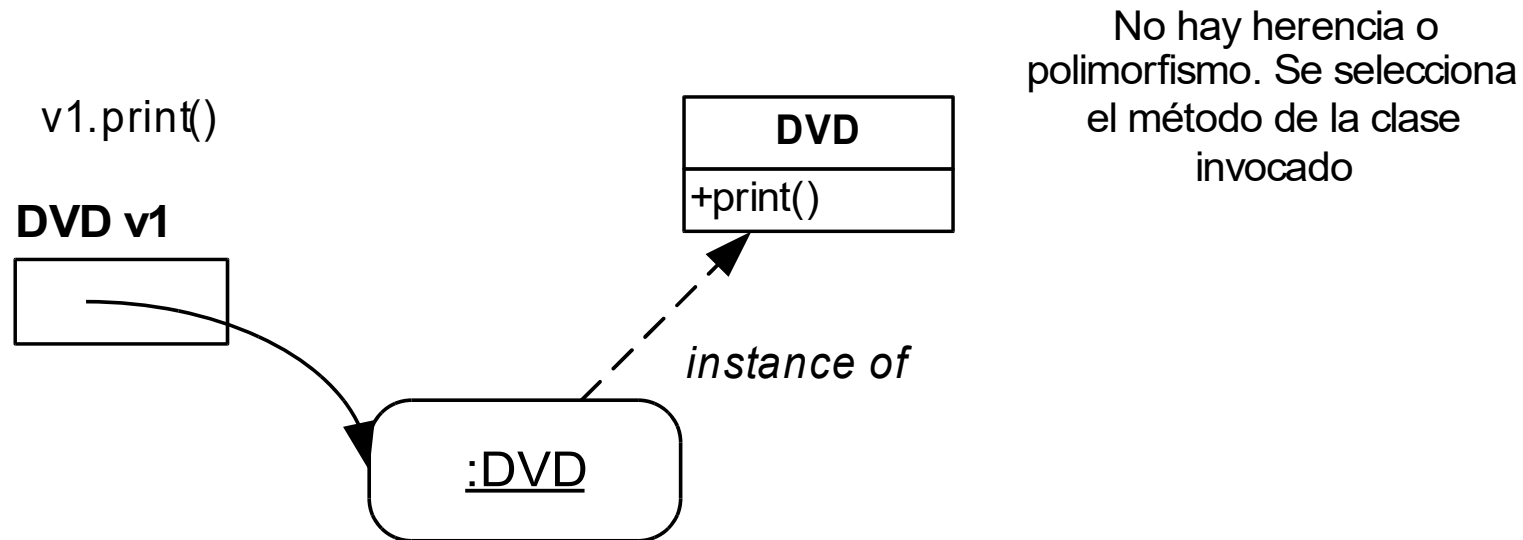
■ Polimorfismo

- múltiples formas
 - característica de la POO (junto con encapsulación y herencia)
-
- 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**

Métodos polimórficos

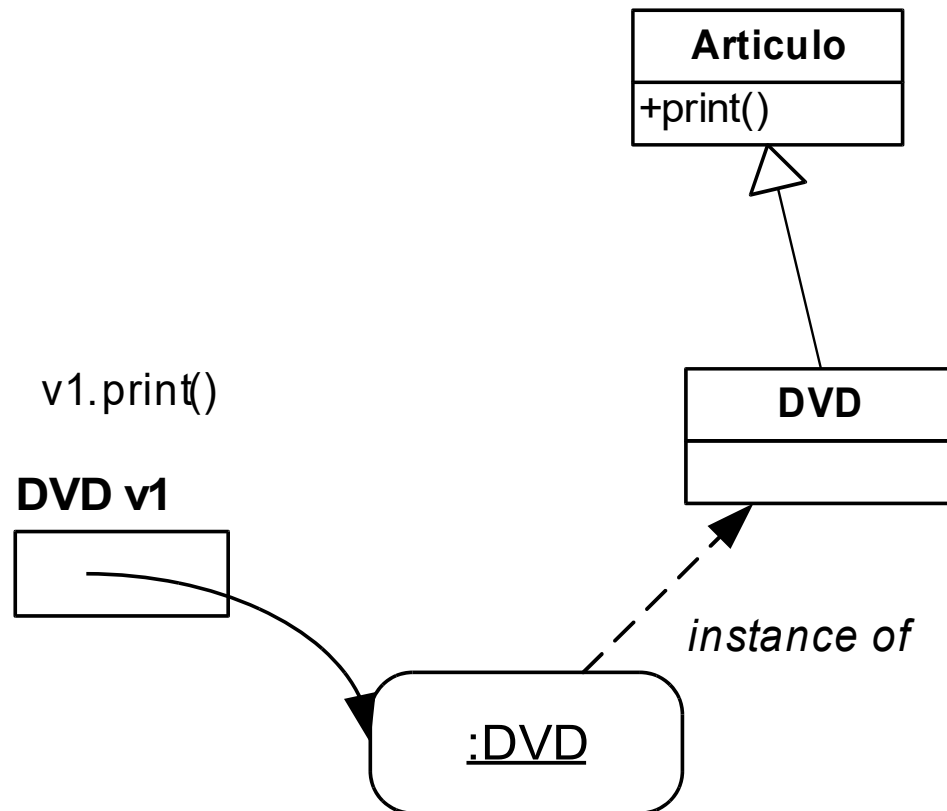
- Polimorfismo de métodos
 - **una llamada a un mismo método produce ejecuciones diferentes dependiendo del tipo dinámico de la variable utilizada en la llamada**
 - se da en una relación de herencia cuando un método de una clase padre se redefine en la clase hija
 - hay un tipo estático y otro dinámico
 - `print()` de `Articulo` es un método polimórfico
 - `mover()` de `Pieza` también

Búsqueda de métodos a través de la herencia



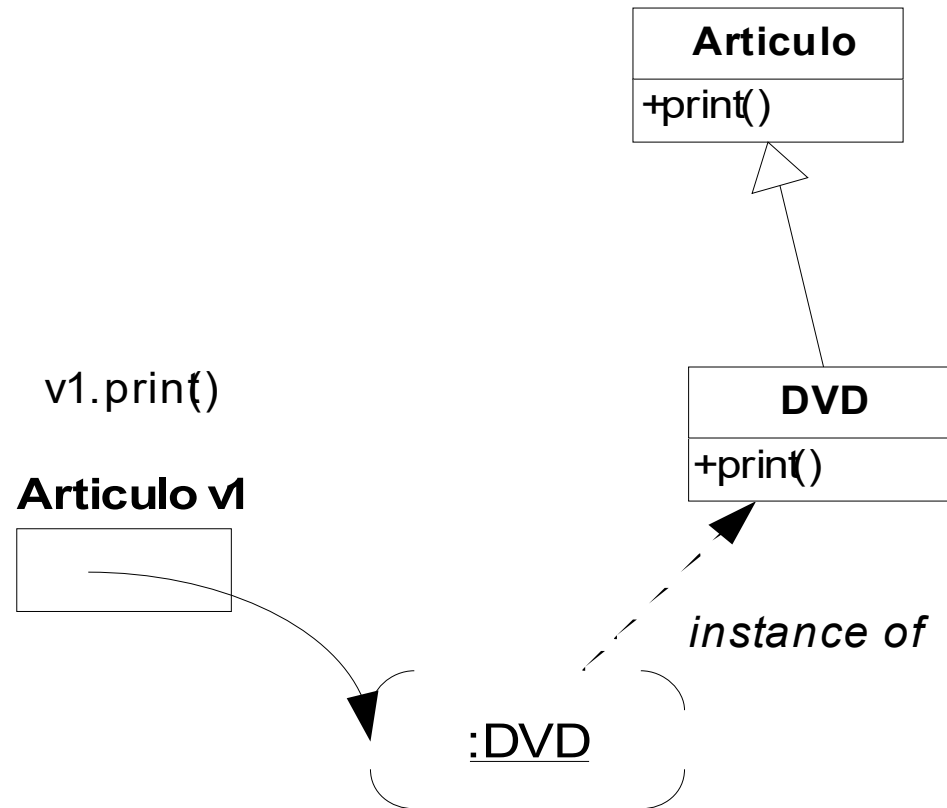
La búsqueda de métodos estáticos se hace en tiempo de compilación

Búsqueda de métodos a través de la herencia



Hay herencia pero no overriding (no se redefine `print()`). Se busca en la jerarquía de herencia hacia arriba hasta encontrar el método

Búsqueda de métodos a través de la herencia



Polimorfismo y redefinición
La primera versión
encontrada es la que se
ejecuta

Operador instanceof

■ instanceof

- verifica si un objeto dado es, directa o indirectamente, una instancia de una clase dada o de sus hijos
- *objeto instanceof clase*
- devuelve *true* si el tipo dinámico de objeto es clase (o alguna derivada)

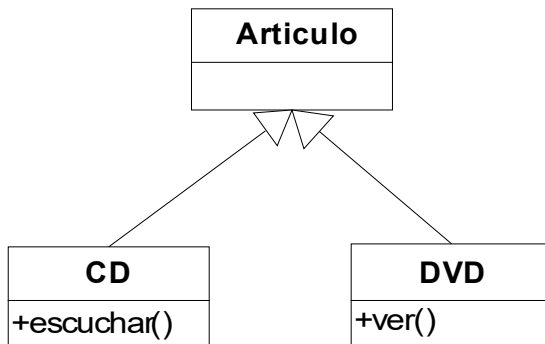
- `Vehiculo v = new Coche();`
`//asumimos que Coche deriva de //`
`Vehiculo`
- `Coche c = new Coche();`
- `Persona p = new Persona();`
- `Vehiculo v2 = new Vehiculo();`

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

Algunas consideraciones

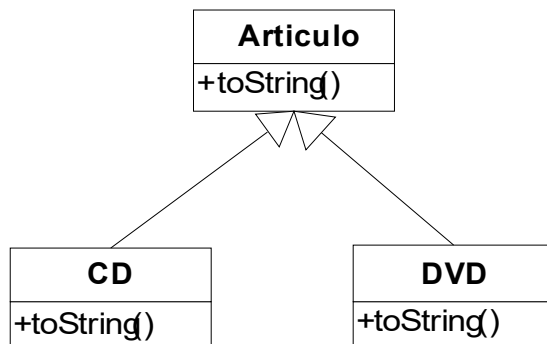
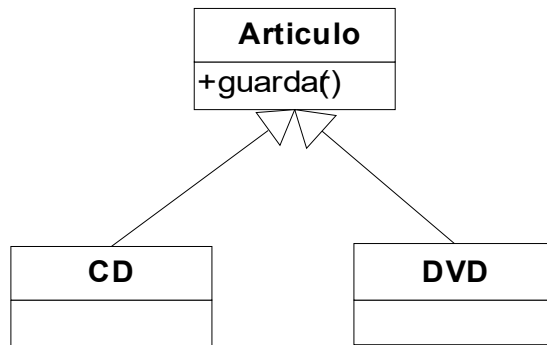
- Seleccionar el método adecuado en tiempo de ejecución se denomina **enlace dinámico**
- Si un método es privado, static, final o es un constructor el compilador sabe exactamente a qué método llamar (el enlace entre la llamada al método y el método es estático)
- En cualquier otro caso el método a invocar depende del tipo en tiempo de ejecución – **enlace dinámico** – (es lo que hace la JVM – el comportamiento por defecto en Java)

Ejer 7.13.



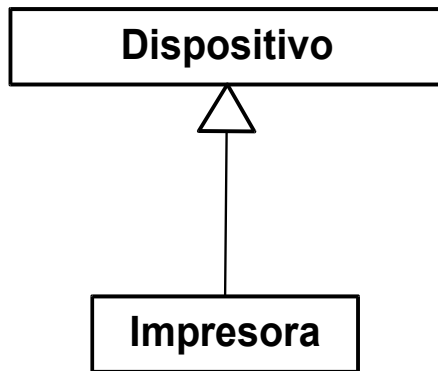
- `CD c1 = new CD();`
 - ¿Tipo estático de c1 ? **CD**
 - ¿Tipo dinámico de c1 ? **CD**
 - Si hago `c1.escuchar()` , ¿qué método se ejecuta? **escuchar()**
 - ¿Hay polimorfismo? **NO**
- `Articulo arti = new DVD();`
 - ¿Tipo estático de arti? **Articulo**
 - ¿Tipo dinámico de arti? **DVD**
 - Hacemos `arti.ver()` **ERROR en compilación**
 - `if (arti instanceof DVD)`
`((DVD) arti).ver()`

Ejer 7.13.



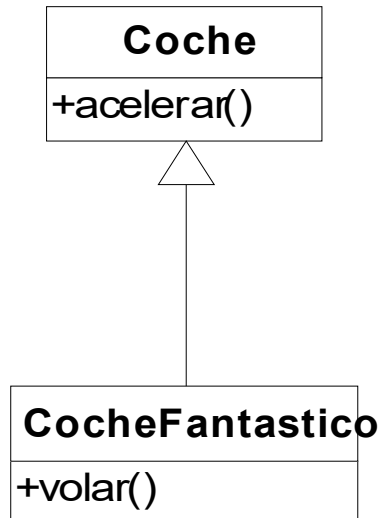
- `Articulo d1 = new DVD();`
`d1.guardar();`
 - ¿Tipo estático de d1 ? **Articulo**
 - ¿Tipo dinámico de d1 ? **DVD**
 - ¿algún problema? **ninguno**
se ejecutar guardar() heredado,
compilador no se queja
- `Articulo a1 = new CD();`
`a1.toString();`
 - ¿Tipo estático de a1? **Articulo**
 - ¿Tipo dinámico de a1? **CD**
 - Qué `toString()` se ejecuta? **el de CD,**
aplicamos polimorfismo

Ejer 7.14.



- `Dispositivo d1 = new Impresora();`
`String n = d1.getNombre();`
 - Para que el código compile `getNombre()` ha de estar en la clase padre (**Dispositivo**)
- se ejecuta `getNombre()` de **Impresora**
- `getNombre()` es método polimórfico

Ejer 7.14.



- `Coche miCoche = new CocheFantastico();`
`Coche tuCoche = new Coche();`
 - `miCoche.volar();`
 - **error compilación**
 - **se arregla**
`((CocheFantastico) miCoche).volar();`
 - `miCoche.acelerar();`
 - **bien**
 - `tuCoche.acelerar();`
 - **bien**
 - `tuCoche.volar();`
 - **error compilación, sin arreglo**

Modificadores `protected` y `final`

- Atributo o método **`protected` (#)**
 - acceso a él desde las subclases (estén o no en el mismo paquete) pero no desde otras clases fuera del paquete
 - evitar el uso de `protected` en atributos utilizando los accesores y mutadores para evitar debilitar la encapsulación
 - al redefinir un método se puede ampliar su visibilidad pero no disminuirla
- **`final`**
 - atributo – constante
 - método – no puede redefinirse

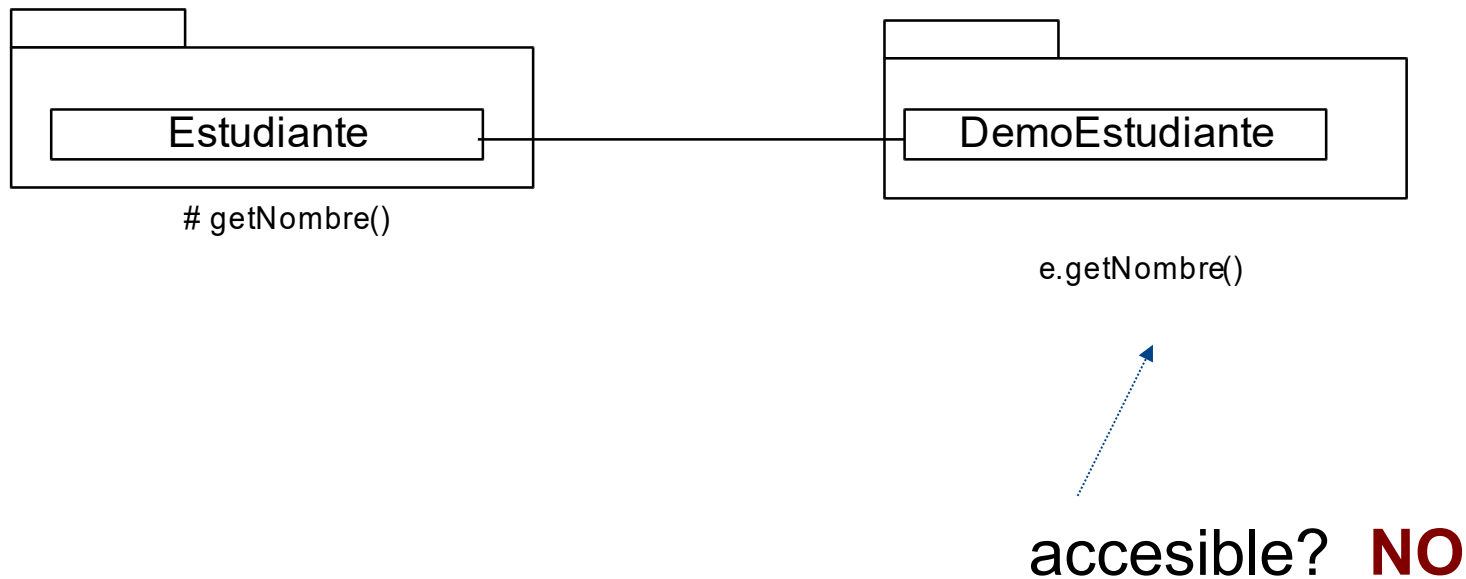
```
public final void escribir()  
{  
    .....  
}
```

Modificadores protected y final

- clase **final**
 - no se puede heredar de ella

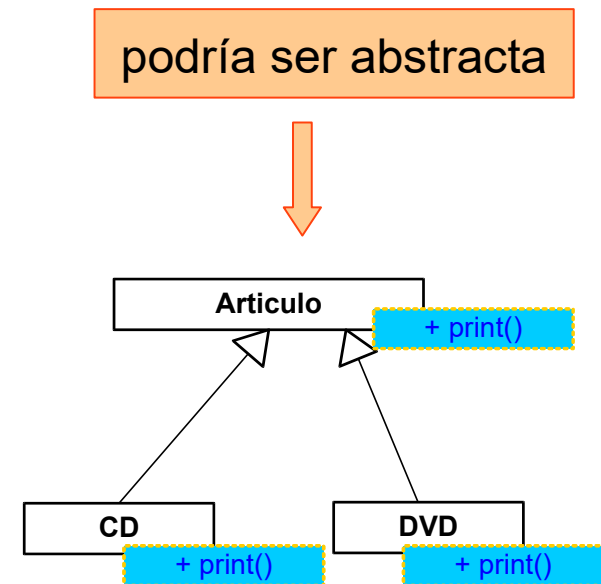
```
public final class EstudiantePrimerCiclo
{
    . . . . .
}
```

Modificadores protected y final

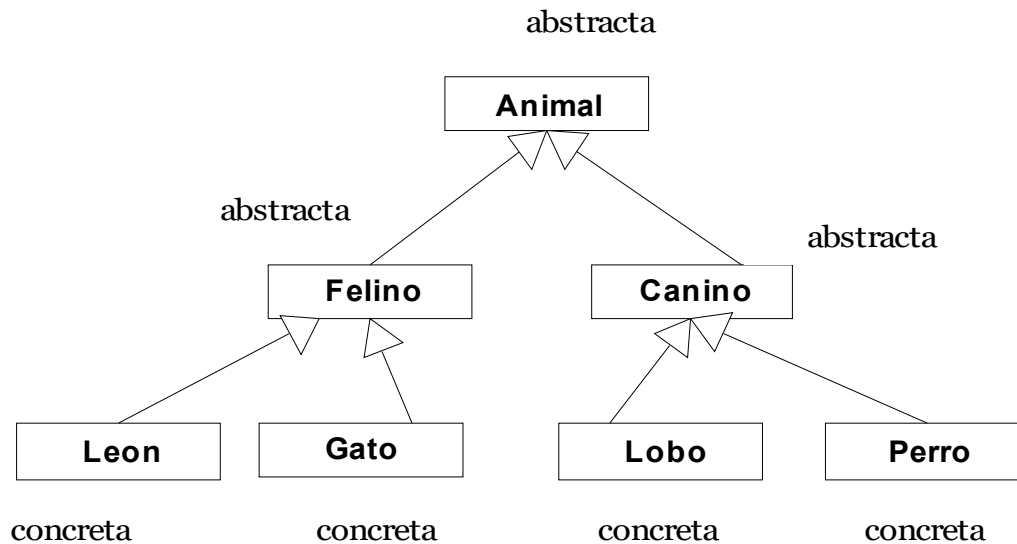


Clases abstractas

- no se pueden instanciar
- sirven como supertipos y plantillas de clase
- capturan las características y comportamientos comunes a otras clases
 - cuando deseamos definir una abstracción que agrupe objetos de distintos tipos y queremos hacer uso del polimorfismo
- se declaran con el modificador **abstract**



Clases abstractas



```
public abstract class Animal
{
}

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

Clases abstractas

- clases **concretas**
 - las que se instancian
- las clases abstractas pueden contener **métodos abstractos**, sin implementación para usar polimorfismo y
- métodos con implementación

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

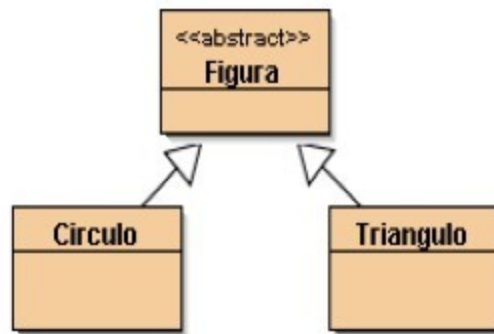
```
public abstract class Canino extends Animal
{
    public abstract void comer();
    public abstract void correr();
}
```

Clases abstractas

```
public class Perro extends Canino
{
    public void comer()
    {
    }
    public void correr()
    {
    }
}
```

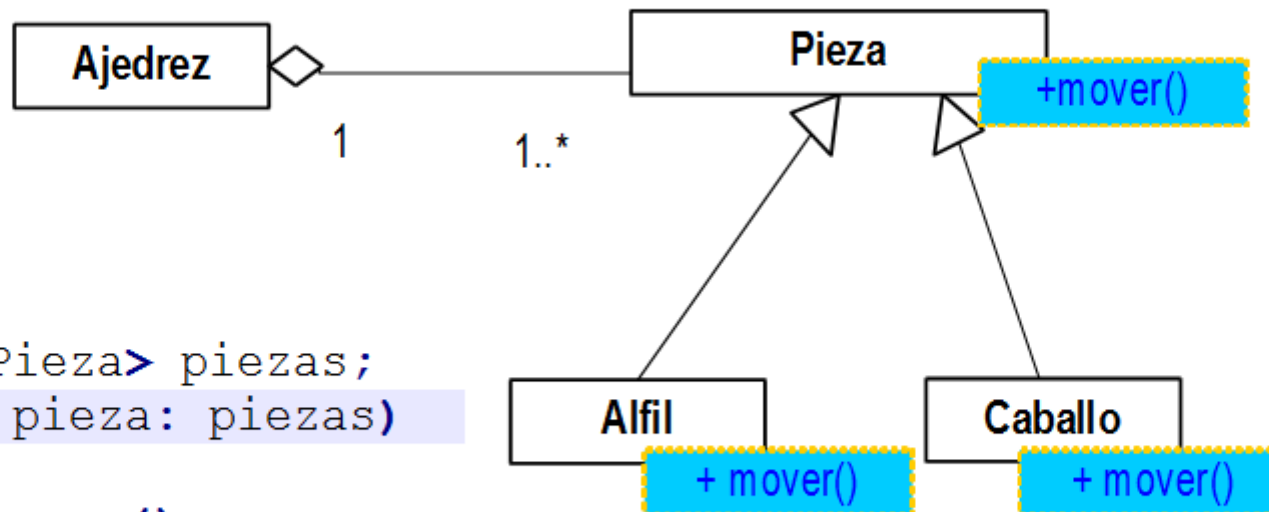
- toda clase abstracta deberá ser extendida
- todo método abstracto debe ser redefinido

Clases abstractas



En UML clase abstracta en cursiva o con el estereotipo <<abstract>>

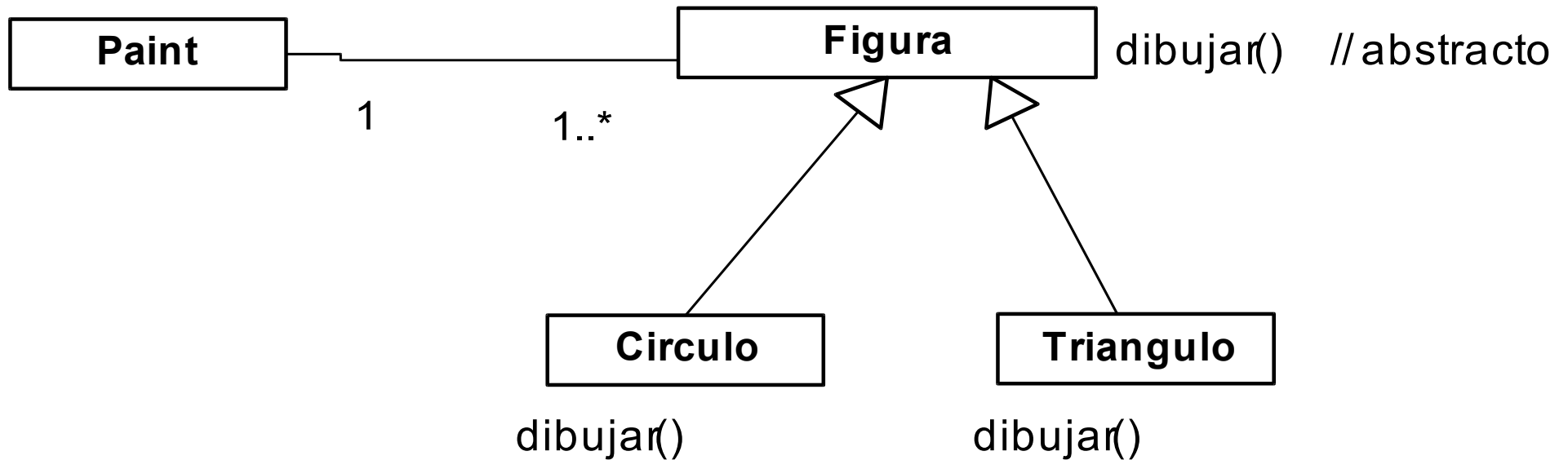
Más ejemplos



```
ArrayList<Pieza> piezas;
for (Pieza pieza: piezas)
{
    pieza.mover();
}
```

Clases abstractas

Más ejemplos



Resumen clases abstractas

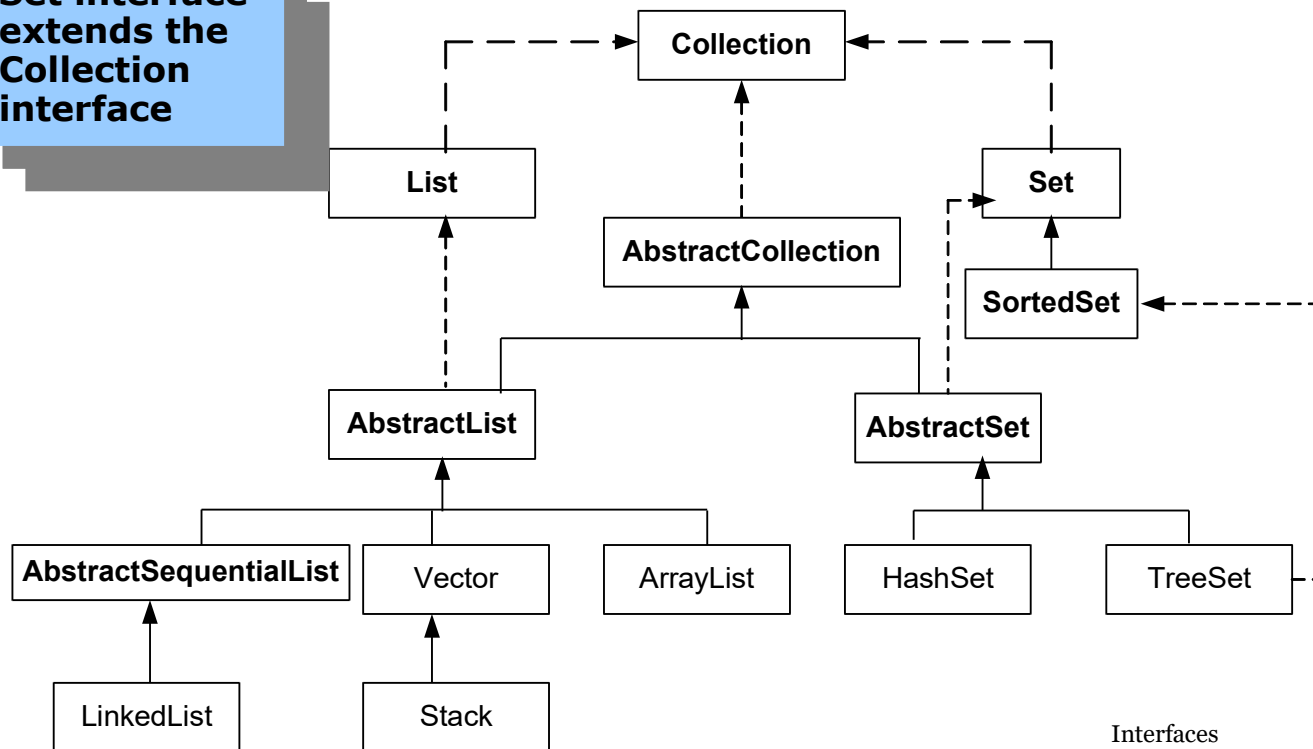
- no se pueden crear instancias
- sólo las clases abstractas pueden contener métodos abstractos
- las clases abstractas con métodos abstractos fuerzan a las subclases a redefinir e implementar los métodos declarados como abstractos
 - para que una subclase sea concreta debe implementar todos los métodos abstractos que herede
- una clase abstracta se puede utilizar como tipo aunque no se instancie

Resumen clases abstractas

- pueden contener atributos y métodos no abstractos
- un método abstracto no puede ser definido como final porque ha de ser redefinido
- su único uso es para ser extendidas (proporcionar un protocolo), sin embargo, hay una excepción, si tienen métodos estáticos pueden ser invocados
- proporcionan mayor flexibilidad, más abstracción, más extensibilidad, menor acoplamiento

Clases abstractas y la API de Java

Set interface extends the Collection interface



List interface extends the Collection interface

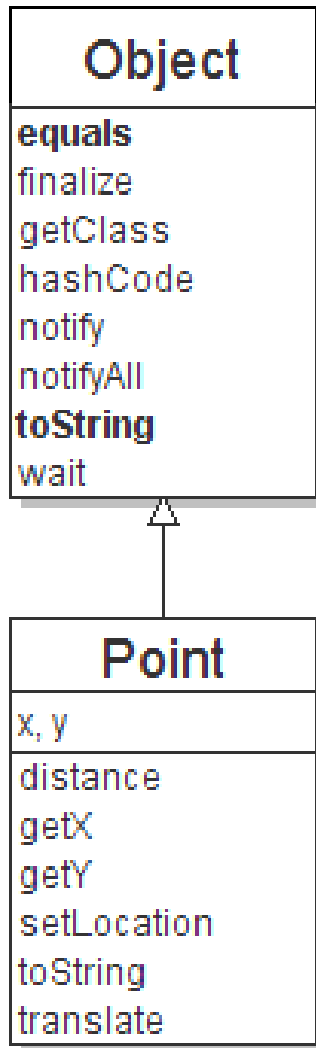
```
public interface Set extends Collection
```

Ejer Tienda Ordenador

- Definir Ordenador como clase abstracta
- Incluir `public void calcularDescuento()` en Ordenador como método abstracto

Ejer 7.15 y 7.16

Algunos métodos de la clase Object - toString()



- **toString()** - `public String toString()`
 - devuelve un `String` conteniendo una representación del objeto
 - por defecto, 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.
 - `clase@código_hash / Telefono@162b91`
 - se redefine en cada clase (es lo que hemos estado haciendo habitualmente)
 - `System.out.println(articulo);` lo mismo que `System.out.println(articulo.toString());`

Algunos métodos de la clase Object - equals()

- **equals()** - `public boolean equals(Object obj)`
 - por defecto equals() verifica la identidad
 - dos objetos son idénticos si y solo si son el mismo objeto (dos referencias que apuntan al mismo objeto). Las dos referencias son “alias”
 - dos objetos son iguales si son del mismo tipo y tienen el mismo valor, es decir, si los valores de sus atributos son iguales
 - `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

Algunos métodos de la clase Object - equals()

- **equals()** - `public boolean equals(Object obj)`
 - por defecto, equals() devuelve true si dos referencias constituyen un alias
 - implementación por defecto que proporciona Object para este método

```
public boolean equals(Object obj)
{
    return this == 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

Algunos métodos de la clase Object - equals()

■ equals() - **Cómo redefinirlo?**

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

Algunos métodos de la clase Object - equals()

- **equals()** - `public boolean equals(Object obj)`
 - muchas de las colecciones del framework de Java (ArrayList, Set, HashMap, ...) utilizan equals() para comparar objetos
 - hay que redefinir este método para efectuar correctamente las operaciones con estas colecciones
 - ej, para que la clave en un map sea un objeto Artículo hay que redefinir equals() para que el map funcione correctamente al localizar una clave
 - **si se redefine equals() obligatoriamente hay que redefinir hashCode()**

Algunos métodos de la clase Object - hashCode()

- **hashCode()** - `public int hashCode()`
 - 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
 - objetos que son iguales de acuerdo al método equals() deben devolver el mismo valor de hashCode()
 - hay que redefinir hashCode() en aquellas clases que redefinan equals()

Algunos métodos de la clase Object - hashCode()

- **hashCode()** - `public int hashCode()`

```
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, LocalDate, ...

Algunos métodos de la clase Object - clone()

- **clone()** - `protected Object clone()`
 - Realiza una copia (clonación) de un objeto
 - es una “*shallow copy*”, *copia ligera o no profunda*, 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`
 - `protected` en `Object` por lo que se suele sobrescribir haciéndolo público

Algunos métodos de la clase Object - clone()

- **clone()** - **protected Object clone()**

```
public class Alumno implements Cloneable
{
    private String nombre;
    private int nota;

    public Alumno clone() throws CloneNotSupportedException
    {
        return (Alumno) super.clone();
    }
}
```


Algunos métodos de la clase Object - getClass()

getClass() -

- API reflection de Java (reflexión) – capacidad de inspeccionar las clases en tiempo de ejecución sin conocer los tipos específicos con los que se está trabajando
- devuelve una instancia de `java.lang.Class` que contiene información sobre la clase del objeto
 - × representa los tipos en Java: clases, interfaces, arrays, primitivos
- nos proporciona información sobre la clase en tiempo de ejecución.
 - a esta instancia se le denomina **metaobjeto** (contiene información sobre una clase).

Algunos métodos de la clase Object - getClass()

■ getClass() -

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

DVD d = new DVD;
d.escribirClaseObjeto();
// visualiza "El objeto pertenece a la clase class DVD"
```

Ejer Tienda Ordenador

- Redefinir `equals()`, `hashCode()`
- Modificar `toString()` para incluir `getClass()`

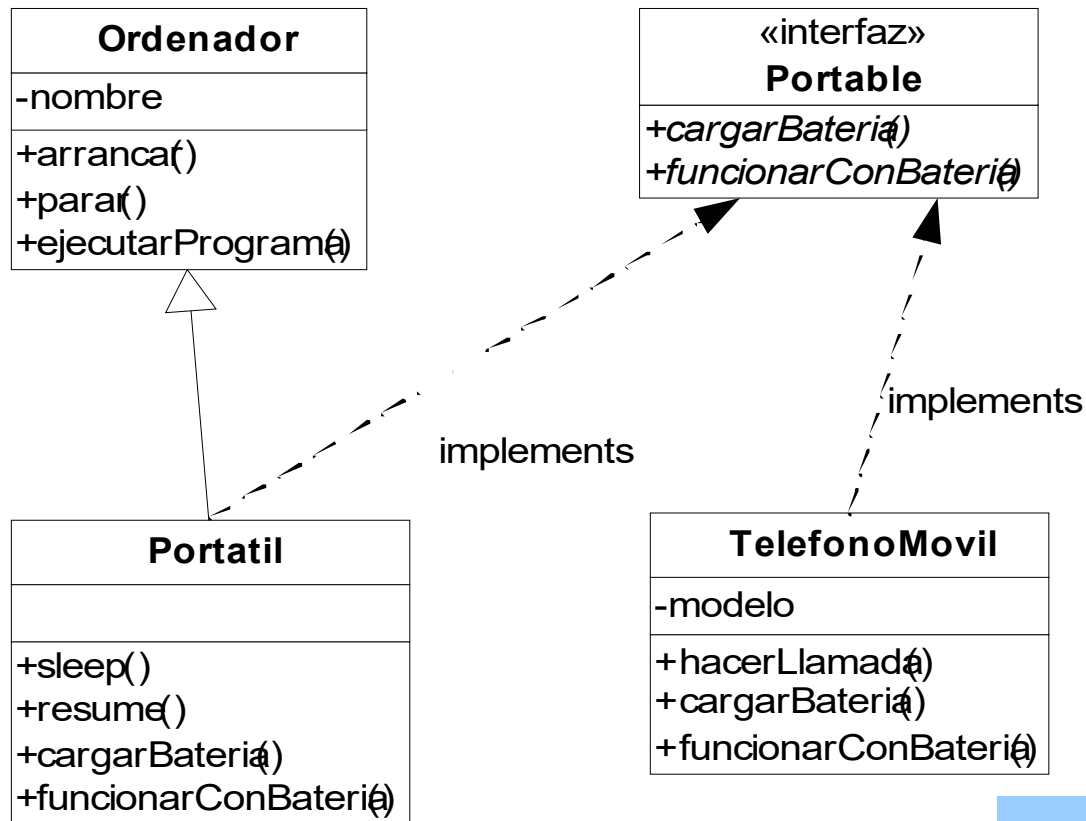
Interfaces

- clases abstractas puras
 - sin atributos
 - solo con métodos abstractos
 - especificaciones de un tipo
 - definen un protocolo de comportamiento
 - es la definición de un contrato que las clases que lo implementen se comprometen a cumplir
 - simulan la herencia múltiple
- herencia { de estructura (por extensión)
interfaces (de subtipos)

Interfaces

- las clases implementan un interfaz
- sin constructores
- permiten definir tipos y hacer que clases que no pertenecen a la misma jerarquía de herencia sean del mismo tipo por el hecho de definir el interfaz

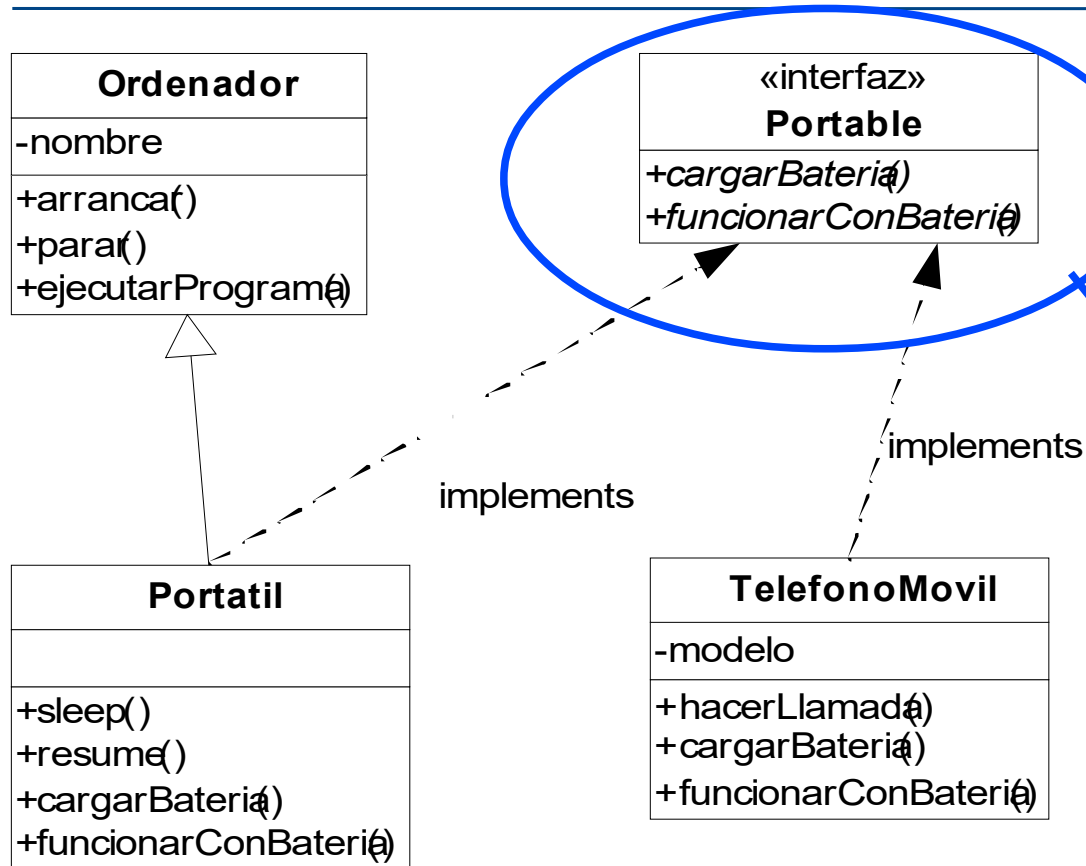
Interfaces



un interfaz no es parte de una jerarquía, clases que no tienen ninguna relación pueden implementar el mismo interfaz

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

Interfaces



interfaz Portable expresa el comportamiento que es común a los dispositivos portátiles, define lo que hacen los objetos pero no cómo

Definición de un interface

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



- todos los métodos en un interfaz son abstractos (sin implementación)
 - no hace falta indicarlo
 - en Java 8 sí podemos con métodos **default**
- no contienen métodos estáticos
 - en Java 8 sí se puede
- no contienen constructores
- todos los métodos tienen visibilidad public
 - no es necesario indicarlo

Definición de un interface

- no se permiten atributos
 - solo constantes `public static final`
 - no es preciso indicarlo
- pueden extender otros interfaces
 - en la API, el interface `Set` extiende el interface `Collection`
 - `public interface Collection<E> extends Iterable<E>`
 - `Set extends Collection`
 - `List extends Collection`
- se denotan habitualmente (no siempre) con adjetivos (`Comible`, `Movable`, `Editable`, `Observable`, `Comparable`)

Implementando un interface

- una clase implementa un interfaz (**implements**)
- si una clase implementa un interfaz
 - **debe proporcionar implementación de todos los métodos que define el interfaz**

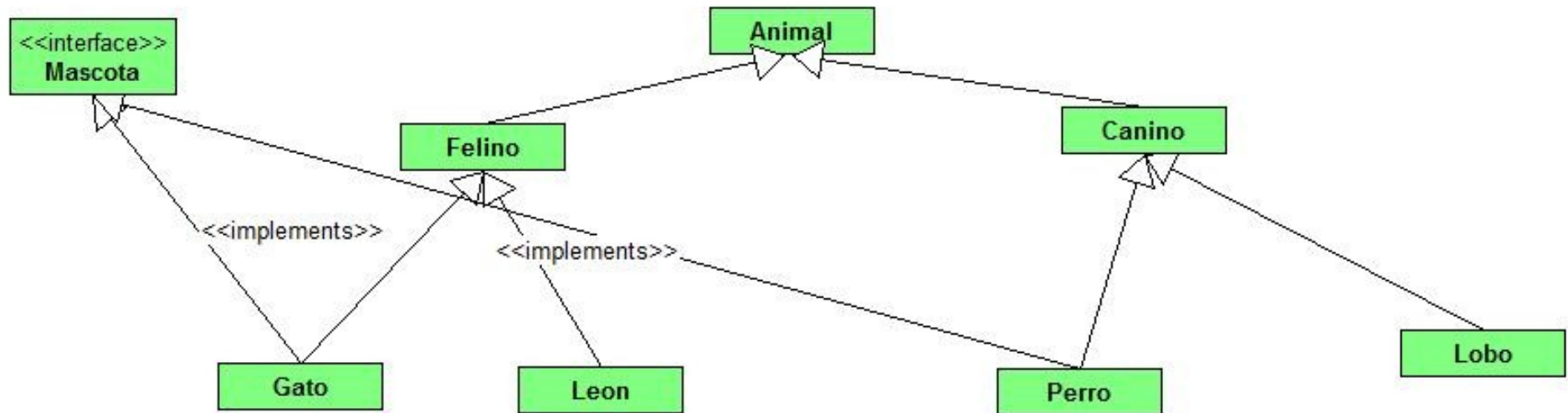
```
public class Portatil extends Ordenador implements Portable
{
    public void cargarBateria()
    {
        System.out.println("Cargando batería");
    }
    public void funcionarConBateria()
    {
        System.out.println("Funcionando con batería");
    }
}
```

extends va antes
que implements

Implementando más de un interface

```
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()
    {
        .....
    }
}
```

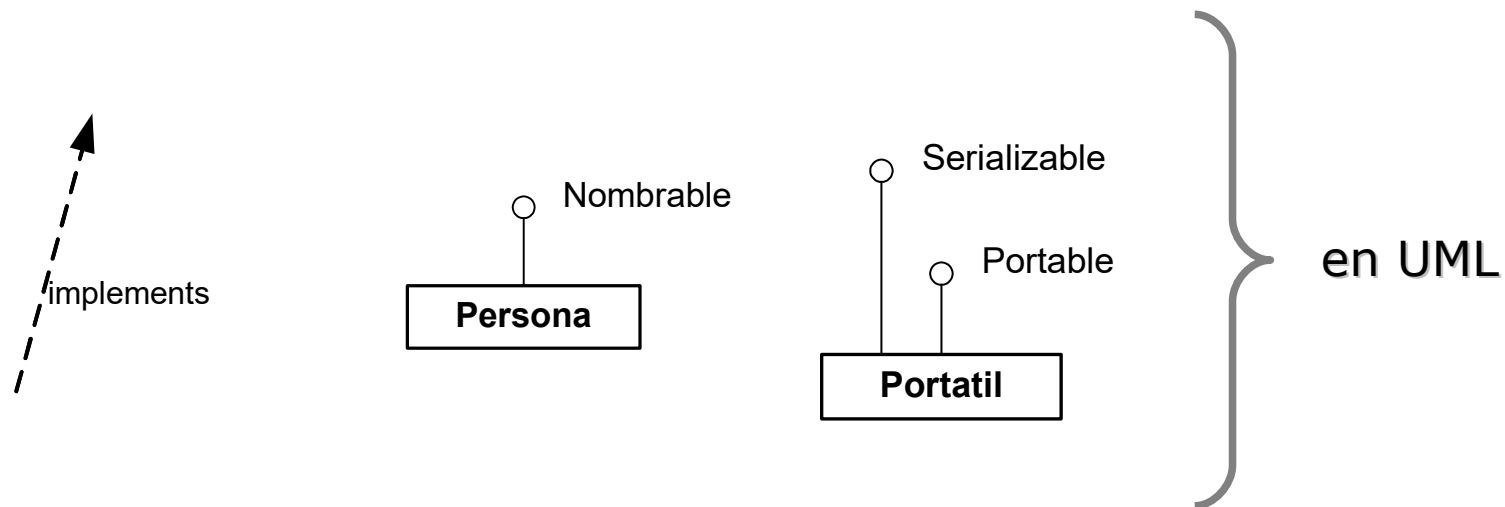
Otro ejemplo de interfaz



```
public interface Mascota
{
    public abstract void pasear();
    public abstract void jugar();
}
```

```
public class Perro extends Canino
    implements Mascota
{
    public void pasear() {...}
    public void jugar() {...}
    .....
}
```

Clase implementa interfaz - Notación en UML



Interfaces como tipos

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

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

- Herencia clases – **extends**
 - subclases heredan atributos y código - reutilización
 - subclases son subtipos de las superclases - polimorfismo

- Herencia tipos – **implements**
 - no se hereda estructura (código)
 - sí definen tipos – variable de un tipo del interfaz que apunta a objetos subtipos del interfaz - polimorfismo

Interfaces como tipos

- las variables pueden ser declaradas de un tipo interface
 - no hay 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

Extendiendo interfaces

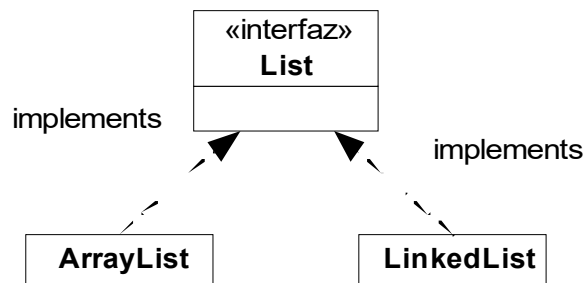
- un interfaz puede heredar o extender de otro interfaz proporcionando métodos adicionales
- la clase que implemente B deberá dar código a los tres métodos

```
interface A
{
    int hacerUno();
    int hacerDos();
}
interface B extends A
{
    int hacerTres();
}
```

Ventaja – evitan romper el código de las clases que implementan A si deseamos añadir nueva funcionalidad al interfaz A

Interfaces como especificaciones

- Además de permitir la herencia múltiple los interfaces
 - separan completamente funcionalidad de implementación
 - definen un contrato entre el proveedor del comportamiento (el propio interfaz) y un usuario del mismo (la clase que lo implementa)
 - definen un papel que las clases que lo implementan van a jugar

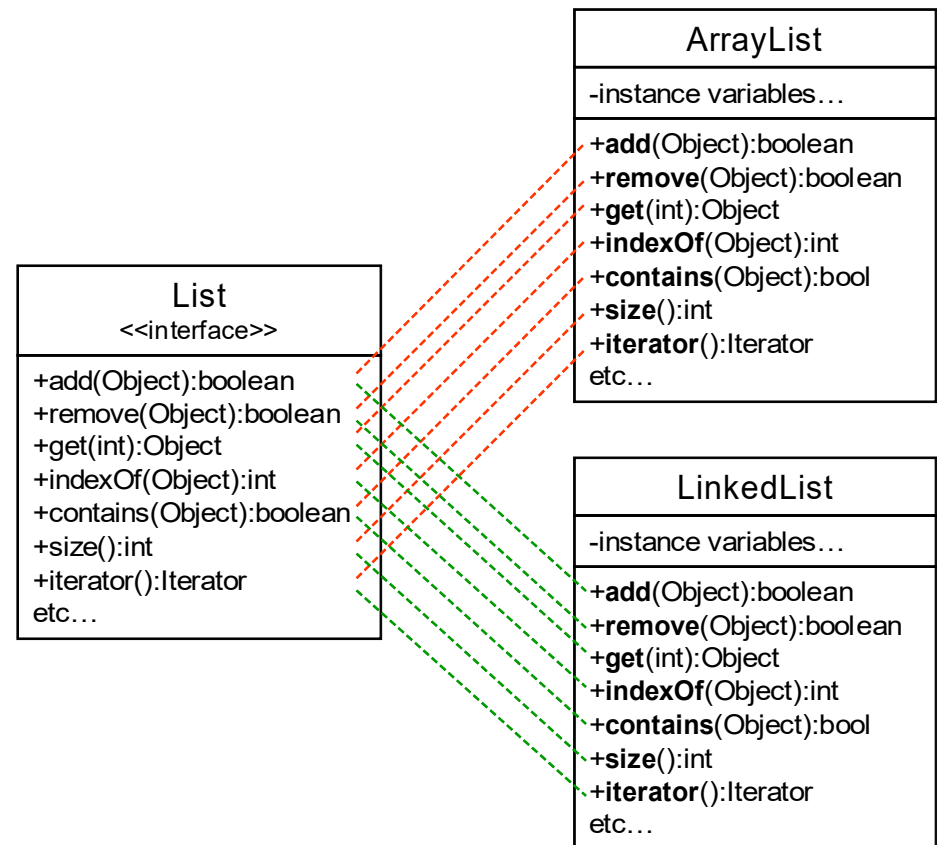


El interface `List` especifica la funcionalidad completa de una lista sin proporcionar implementación.

Las clases `ArrayList` y `LinkedList` proporcionan diferentes implementaciones del mismo interfaz

Interfaces como especificaciones

```
public interface List
{
    public boolean add(Object o);
    public boolean add(int i, Object);
    public Object remove(int i);
    public Object remove(Object o);
    public Object get(int i);
    public int indexOf(Object o);
    public boolean contains(Object o);
    public int size();
    public Iterator iterator();
}
```



Interfaces como especificaciones

```
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
```

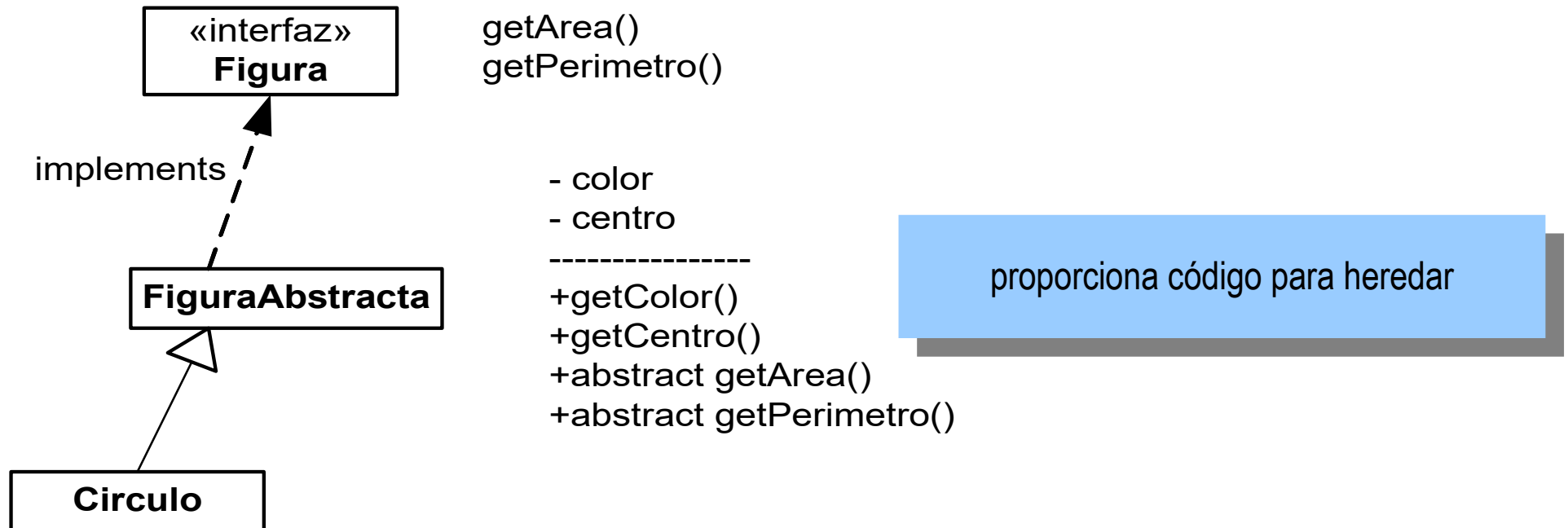
se declara la colección lista de un tipo interfaz (el que ofrece la funcionalidad)

- interface Map y las clases concretas HashMap y TreeMap, el interface Set y las clases HashSet y TreeSet
- interfaces también pueden formar jerarquías
 - Collection es uno de los interfaces raíz del framework de Java, List y Set heredan de ese interfaz.

Clases abstractas versus interfaces

- preferible el uso de interfaces a clases abstractas
 - permiten el uso del polimorfismo en su máxima expresión
 - menor acoplamiento entre clases, más flexibilidad,..
 - es muy fácil cambiar la implementación sin afectar a la aplicación
 - si se necesita heredar código se puede jugar con interfaces y clases abstractas
 - 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

Clases abstractas versus interfaces



```
List<Figura> listaFiguras = new ArrayList<Figura>();  
listaFiguras.add(new Circulo());
```

Clases abstractas versus interfaces

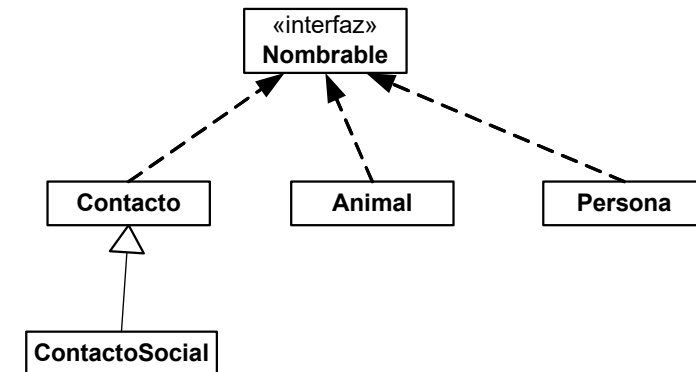
- Deberíamos utilizar clases abstractas cuando queremos heredar código de ellas
- Interfaces deberían ser utilizados cuando sus métodos son apropiados para nuestra aplicación y además fácilmente aplicados en otros programas
- Un interface no puede proporcionar ninguna implementación para sus métodos, una clase abstracta sí
- Un interface no puede contener variables de instancia, una clase abstracta sí
- Un interface y una clase abstracta pueden declarar constantes
- No se pueden crear instancias de interfaces ni de clases abstractas

Ejer 7.17

- Definiendo el interfaz Nombrable

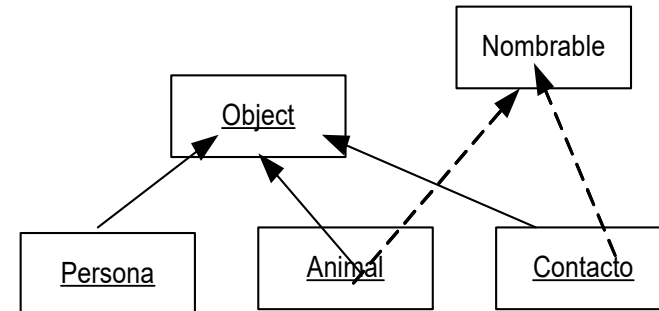
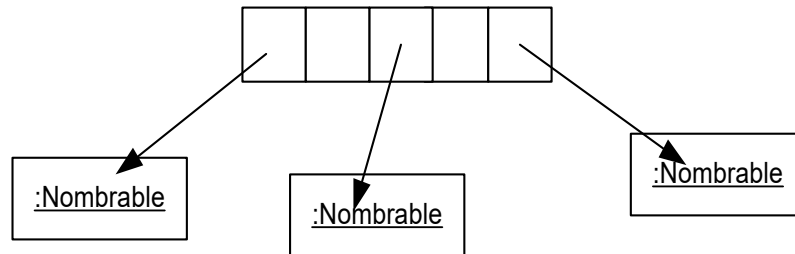
```
public interfaz Nombrable
{
    public String getNombre();
    public void print();
}
```

- `public class Contacto implements Nombrable`
- `public class Animal implements Nombrable`
- `public class Persona implements Nombrable`



- Las tres clases deben proporcionar implementación para sus métodos
- `Nombrable n = new Contacto()? SI`
- `Nombrable n = new ContactoSocial()? SI`

Ejer 7.17



Ejer 7.17

```
public void print Nombrables(String nombre)
{
    Iterator<Object> it = lista.iterator();
    while (it.hasNext())
    {
        Object obj = it.next();
        if (obj instanceof Nombrable)
        {
            Nombrable n = (Nombrable) obj;
            if (n.getNombre().equals(nombre))
            {
                n.print();
            }
        }
    }
}
```

Ejer 7.18

Interface Comparable

- en *java.lang*
- compara el objeto receptor con el que recibe como parámetro y devuelve un entero

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

<T> - tipo de los objetos a comparar

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

Ejemplo Clase Coche

```
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;
    }
}
```

Ejemplo Clase Coche

```
public class Coche implements Comparable<Coche>
{
    private int velocidad;
    .....
    public int compareTo(Coche otro)
    {
        return Integer.compare(this.velocidad,
                                otro.getVelocidad());
    }
}
```



Aún más fácil

Interface Comparable

- Los objetos de las clases que implementan el interface `Comparable` pueden ser comparados en términos de orden
 - las colecciones de la API basan las operaciones de ordenación y algunas búsquedas en el método `compareTo()` de este interface
 - podremos utilizar `Collections.sort(listaPersonas)` donde `Persona` implementa `Comparable`
 - también `Arrays.sort()`
 - La clase `String` y las clases envolventes `Integer`, `Double`, `Character`, implementan ya el interface `Comparable`

Ejercicio Clase Pais

```
public class Pais implements Comparable<Pais>
{
    private double  superficie;
    .....
    public int compareTo(Pais otro)
    {
        if (this.superficie == otro.getSuperficie()) {
            return 0;
        }
        if (this.superficie > otro.getSuperficie()) {
            return 1;
        }
        return -1;
    }
}
```

■ Ejer 7.19

Ejercicio Clase Pais

```
public class Pais implements Comparable<Pais>
{
    private double  superficie;
    .....
    public int compareTo(Pais otro)
    {
        return Double.compare(this.superficie,
                               otro.getSuperficie());
    }
}
```

■ Ejer 7.19

Interface Comparator

- en *java.util*
- un objeto `Comparator` encapsula un criterio de orden
 - esto permite 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`
 - ej. queremos ordenar una colección de países además de por superficie, por habitante

Interface Comparator

```
public interface Comparator<T>
{
    public int compare(T obj1, T obj2);
}
```

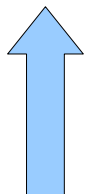
<T> - tipo de los objetos a comparar

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

Interface Comparator

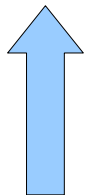
```
import java.util.Comparator;
public class ComparadorCoches implements Comparator<Coche>
{
    public int compare(Coche c1, Coche c2)
    {
        if (c1.getVelocidad() < c2.getVelocidad()) {
            return -1;
        }
        if (c1.getVelocidad() == c2.getVelocidad()) {
            return 0;
        }
        return 1;
    }
}
```



Clase que encapsula el criterio de ordenación

Interface Comparator

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

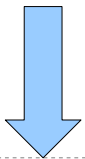


Clase que encapsula el criterio de ordenación

Interface Comparator

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

Lo mismo con clase anónima



```
Collections.sort(lista, new Comparator<Coche>() {  
    public int compare(Coche c1, Coche c2){  
        .....  
    }  
} );
```

Sintaxis clases anónimas

Ejercicio Clase Pais con Comparator

```
public class Pais implements Comparable<Pais>
{
    private double  superficie;
    private int  habitantes;
    .....
}
```

Un criterio de ordenación la superficie

```
public class ComparadorPorHabitantes
    implements Comparator<Pais>
{
    public int compare(Pais pais1, Pais pais2)
    {
        if (pais1.getHabitantes() == pais2.getHabitantes()) {
            return 0;
        }
        if (pais1.getHabitantes() > pais2.getHabitantes()) {
            return 1;
        }
        return -1;
    }
}
```

Otro criterio de ordenación los habitantes

Ejercicio Clase Pais con Comparator

```
public class Pais implements Comparable<Pais>
{
    private double  superficie;
    private int habitantes;
    .....
}
```

Un criterio de ordenación la superficie

```
public class ComparadorPorHabitantes
    implements Comparator<Pais>
{
    public int compare(Pais pais1, Pais pais2)
    {
        return Integer.compare(pais1.getHabitantes(),
                                pais2.getHabitantes());
    }
}
```

Otro criterio de ordenación los habitantes

¿Cómo utilizar el comparador?

```
ArrayList<Pais> paises = new ArrayList<Pais>();  
ArrayList<Pais> nueva = new ArrayList<Pais>();  
nueva.addAll(paises);
```

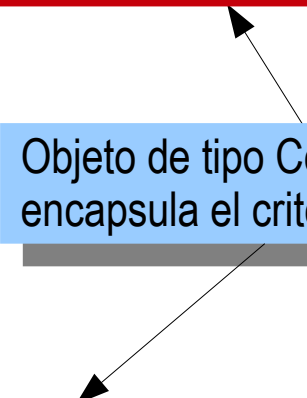
```
Collections.sort(nueva, new ComparadorPorHabitante());
```

ó

```
ComparadorPorHabitante comparador =  
    new ComparadorPorHabitante();
```

```
Collections.sort(nueva, comparador);
```

Objeto de tipo Comparator que
encapsula el criterio de comparación



¿Cómo utilizar el comparador?

- en orden descendente de habitantes

```
Comparator<Pais> comparadorHabitantesDesc =  
    Collections.reverseOrder(new ComparadorHabitantes());
```

```
Collections.sort(nueva, comparadorHabitantesDesc);
```

o bien

```
Collections.sort(nueva,  
    Collections.reverseOrder(new  
        ComparadorHabitantes()));
```

Comparadores en Java 8

- **default void sort(Comparator comp)**
 - nuevo método (método por defecto) introducido en el interface List en Java 8
 - ordena una lista según el criterio establecido por el comparador

Comparadores en Java 8

```
List<Pais> paises = new ArrayList<Pais>();
Comparator<Pais> comparadorPorNombre = new Comparator<Pais>() {
    public int compare(Pais p1, Pais p2)
    {
        return p1.getNombre().compareToIgnoreCase(p2.getNombre());
    }
};

paises.sort(comparadorPorNombre); // ordena países en orden creciente de nombre

paises.sort(comparadorPorNombre.reversed());
// ordena países en orden decreciente de nombre

.....

paises.sort(Comparator.naturalOrder());
// ordena países por su orden natural el que marca el interface Comparable
```

Ejemplo realizado sin expresiones Lambda

Algunos trucos para implementar compareTo() / compare() más fácilmente

```
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;
    }
}
```

```
public class Coche implements Comparable<Coche>
{
    private int  velocidad;
    .....
    public int compareTo(Coche otro)
    {
        return (this.velocidad - otro.getVelocidad());
    }
}
```

Algunos trucos para implementar compareTo() / compare() más fácilmente

```
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;
    }
}
```

```
public class Coche implements Comparable<Coche>
{
    private int  velocidad;
    .....
    public int compareTo(Coche otro)
    {
        return Integer.compare(this.velocidad,
                                otro.getVelocidad());
    }
}
```

Algunos trucos para implementar compareTo() / compare() más fácilmente

```
public class Pais implements Comparable<Pais>
{
    private double  superficie;
    .....
    public int compareTo(Pais otro)
    {
        if (this.superficie == otro.getSuperficie()) {
            return 0;
        }
        if (this.superficie < otro.getSuperficie()) {
            return -1;
        }
        return 1;
    }
}
```

para valores double
usar **Math.signum()**

```
public class Pais implements Comparable<Pais>
{
    private double  superficie;
    .....
    public int compareTo(Pais otro)
    {
        return (int) (Math.signum(this.superficie -
                                   otro.getSuperficie()));
    }
}
```

Algunos trucos para implementar compareTo() / compare() más fácilmente

```
public class ComparadorSuperficie implements Comparator<Pais>
{
    public int compare(Pais p1, Pais p2)
    {
        if (p1.getSuperficie() == p2.getSuperficie()) {
            return 0;
        }
        if (p1.getSuperficie() < p2.getSuperficie()) {
            return -1;
        }
        return 1;
    }
}
```

```
public class ComparadorSuperficie implements Comparator<Pais>
{
    public int compare(Pais p1, Pais p2)
    {
        return (int) (Math.signum(p1.getSuperficie() -
                                   p2.getSuperficie()));
    }
}
```

Algunos trucos para implementar compareTo() / compare() más fácilmente

```
public class ComparadorSuperficie implements Comparator<Pais>
{
    public int compare(Pais p1, Pais p2)
    {
        if (p1.getSuperficie() == p2.getSuperficie()) {
            return 0;
        }
        if (p1.getSuperficie() < p2.getSuperficie()) {
            return -1;
        }
        return 1;
    }
}
```


```
public class ComparadorSuperficie implements Comparator<Pais>
{
    public int compare(Pais p1, Pais p2)
    {
        return Double.compare(p1.getSuperficie(),
                               p2.getSuperficie());
    }
}
```


Comparadores con expresiones Lambda en Java 8

```
Collections.sort(países, new Comparator<Pais>()  
{  
    public int compare(Pais p1, Pais p2)  
    {  
        return p1.getHabitantes() - p2.getHabitantes();  
    }  
});
```

```
Collections.sort(países,  
    (p1, p2) -> p1.getHabitantes() - p2.getHabitantes() );
```

expresión Lambda

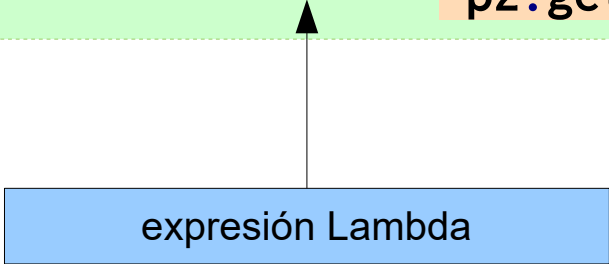


Comparadores con expresiones Lambda en Java 8

```
Collections.sort(países, new Comparator<Pais>() {  
    public int compare(Pais p1, Pais p2) {  
        return Integer.compare(p1.getHabitantes(),  
                                p2.getHabitantes());  
    }  
});
```

```
Collections.sort(países,  
    (p1, p2) -> Integer.compare(p1.getHabitantes(),  
                                p2.getHabitantes() );
```

expresión Lambda



Ordenar sin Collections.sort() y con expresiones Lambda en Java 8

```
países.sort(new Comparator<Pais>() {  
    public int compare(Pais p1, Pais p2) {  
        return p1.getHabitantes() - p2.getHabitantes();  
    }  
});
```

```
países.sort((p1, p2) -> p1.getHabitantes() - p2.getHabitantes() );
```

expresión Lambda

```
graph BT; A[expresión Lambda] --> B["países.sort((p1, p2) -> p1.getHabitantes() - p2.getHabitantes() );"];
```

Ordenar sin Collections.sort() y con expresiones Lambda en Java 8

```
países.sort(new Comparator<Pais>() {  
    public int compare(Pais p1, Pais p2) {  
        return Integer.compare(p1.getHabitantes(),  
                                p2.getHabitantes());  
    }  
});
```

```
países.sort((p1, p2) -> Integer.compare(p1.getHabitantes(),  
                                          p2.getHabitantes() ));
```

expresión Lambda

```
graph BT; A[expresión Lambda] --> B[países.sort((p1, p2) -> Integer.compare(p1.getHabitantes(), p2.getHabitantes() ));];
```

Ejer 7.20

Interface Cloneable

- en *java.lang*
- ¿Cómo hacer una copia de un objeto?
 - hay que clonarlo
- Para clonar objetos de una clase la clase debe implementar
 - el interface **Cloneable**
- Cloneable es
 - un interfaz vacío (sin métodos) – *tagging* interfaz
 - **public class Estudiante implements Cloneable**
 - indica que la clase Estudiante se puede clonar utilizando el método `clone()` de `Object`

Interface Cloneable

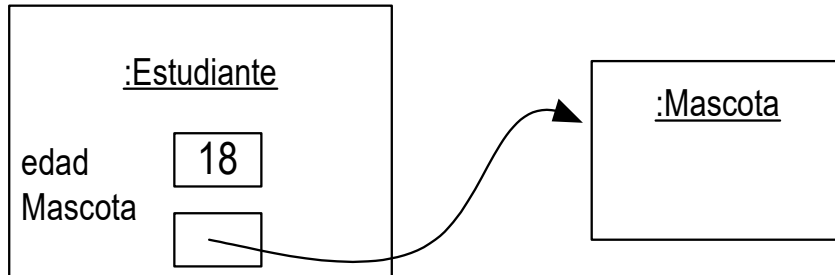
- `protected Object clone()` throws `CloneNotSupportedException`
 - en `Object`
 - se redefine en la clase que queremos clonar haciéndolo público
 - esa clase implementará el interface `Cloneable`
 - se produce una excepción si el objeto a clonar no implementa el interface `Cloneable` o no es una instancia de la clase que lo implementa
- `clone()` es `protected` en `Object` para evitar que otras clases lo utilicen si no se redefine aunque se herede

Interface Cloneable

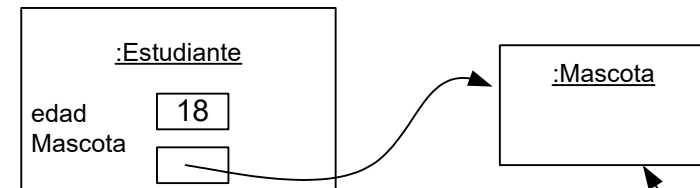
```
public class Coche implements Cloneable
{
    private int velocidad;
    .....
    @override
    public Coche clone()
    {
        .....
    }
    .....
}
```


¿Cómo redefinir clone() ?

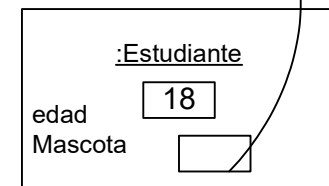
- por defecto
 - clone() hace una *shallow copy* (copia no profunda)
 - copia tipos primitivos y tipos referencia pero no clona los atributos que sean objetos



es lo que hace clone() de
Object – copia no profunda
(`:Mascota` no lo clona)



es lo que hace clone() de
Object – copia no profunda
(`:Mascota` no lo clona)



el clonado por
defecto

copia no profunda después de clonar

¿Cómo redefinir clone()?

```
public class Estudiante implements Cloneable
{
    public Object clone()    // también public Estudiante clone()
    {
        try
        {
            Estudiante clonado = (Estudiante) super.clone();
            return clonado;
        }
        catch (CloneNotSupportedException e)
        {
            return null;
        }
    }
}
```

Copia no profunda
(shallow copy)

¿Cómo redefinir clone() ?

```
public class Estudiante implements Cloneable
{
    public Object clone()    // también public Estudiante clone()
    {
        try
        {
            Estudiante clonado = (Estudiante) super.clone();
            clonado.setMascota((Mascota) mascota.clone());
            return clonado;
        }
        catch (CloneNotSupportedException e)
        {
            return null;
        }
    }
}
```

Copia profunda
(deep copy)

¿Cómo redefinir clone()?

```
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();
```

Copia profunda – aquí solo hay tipos primitivos
(deep copy)

Interfaces y Java 8 – mejoras introducidas

- **Default method** – *métodos por defecto*
 - métodos públicos con implementación en el interfaz
 - la clase que implementa el interfaz puede redefinir el método o heredarlo tal cual

```
public interface Visualizable
{
    default void print()
    {
        System.out.println(toString());
    }
}
```

Ventaja – evitan romper el código de las clases que implementan un interfaz si éste evoluciona añadiendo nueva funcionalidad

Interfaces y Java 8 – mejoras introducidas

■ métodos static –

```
public interface Visualizable
{
    static void print(Visualizable p)
    {
        p.print();
    }
}
```

.....

.....

```
Visualizable producto = new Producto(.....);
Visualizable.print(producto);
```

Interfaces y Java 8 – mejoras introducidas

■ métodos static –

```
public interface SecuenciaEnteros
{
    .....
    // actúa como método factory (fábrica de objetos)
    public static SecuenciaEnteros digitosDe(int n)
    {
        return new SecuenciaDigitos(n);
    }
}

.....
SecuenciaEnteros digitos = SecuenciaEnteros.digitosDe(1729);
```

Interfaces y Java 8 – mejoras introducidas

■ interfaces funcionales

- cualquier interface que contenga un único método abstracto
 - Comparator, Comparable, ActionListener, Runnable
- Java 8 añade muchos interfaces funcionales
 - usados extensivamente junto con expresiones lambda

Sintaxis clases anónimas

- Clase anónima
 - clase local sin nombre
 - puede ser una subclase de una clase padre existente o
 - una clase que implementa un interfaz existente

- Sintaxis clase anónima

```
new nombre-clase-padre(argumentos)
{
    cuerpo de la clase
}
```

```
new interfaz(argumentos)
{
    cuerpo de la clase
}
```

Sintaxis clases anónimas

```
public abstract class Persona
{
    abstract void comer();
}
```

Clase anónima usando
clase padre

```
public class DemoClaseAnonima
{
    public static void main(String args[])
    {
        Persona p = new Persona() {
            public void comer()
            {
                System.out.println("Comiendo ...");
            }
        };
        p.comer();
    }
}
```

Clase sin nombre que
hereda de la clase
Persona

Sintaxis clases anónimas

```
public interfaz Comible
{
    void comer();
}
```

Clase anónima usando
interfaz

```
public class DemoClaseAnonima
{
    public static void main(String args[])
    {
        Comible c = new Comible() {
            public void comer()
            {
                System.out.println("Comiendo ...");
            }
        };
        c.comer();
    }
}
```

Clase sin nombre que
implementa el interfaz
Comible

A Comparator