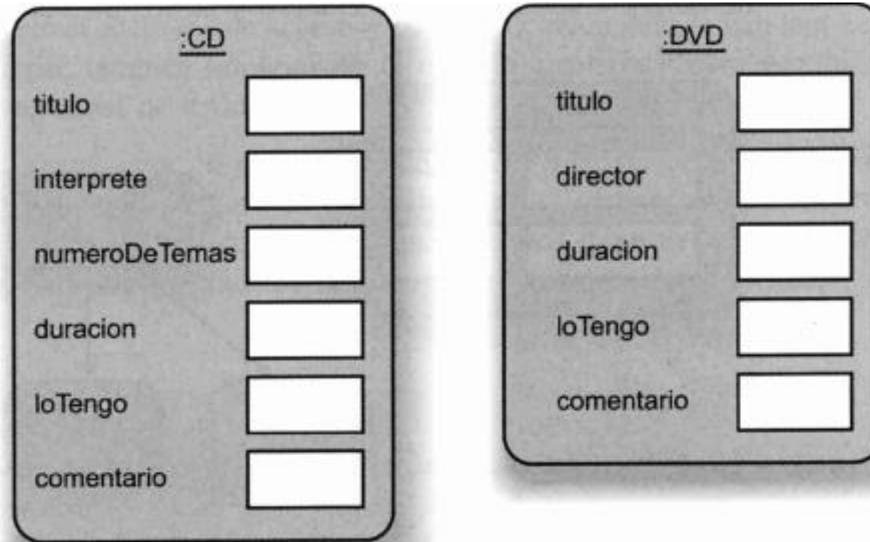
Programación orientada a objetos

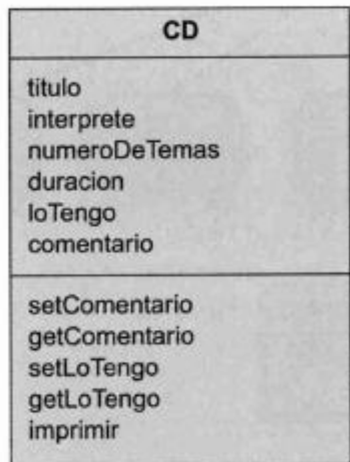
Capítulo 8

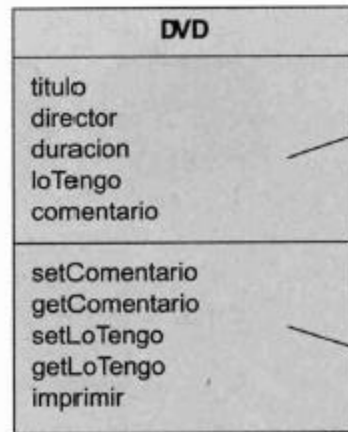
Mejora de las estructuras mediante herencia

Tema 8: Extensión de clases: Herencia. Semana 8	
1- El uso de la herencia. 2- Jerarquías de herencia. 3- Herencia en Java. 4- Herencia y derechos de acceso. 5- Herencia e inicialización. 6- Reutilización de código por medio de la herencia. 7- Subtipos a. Subclases y subtipos. b. Subtipos y asignación. c. Subtipo y paso de parámetros. d. Variables polimórficas. e. Enmascaramiento de tipos. 8- La clase Object. 9- Tipos estáticos y dinámicos. 10- Sobreescritura de métodos. 11- Llamada a métodos con la palabra reservada super. 12- Métodos polimórficos 13- Acceso protegido	1- Estudiar el capítulo 8 del libro base para la “Unidad Didáctica II”. 2- Realizar los ejercicios correspondientes del libro base. 3- Realizar los ejercicios resueltos en examines de años anteriores en los que se utilice la herencia.

Clases y objetos de DoME



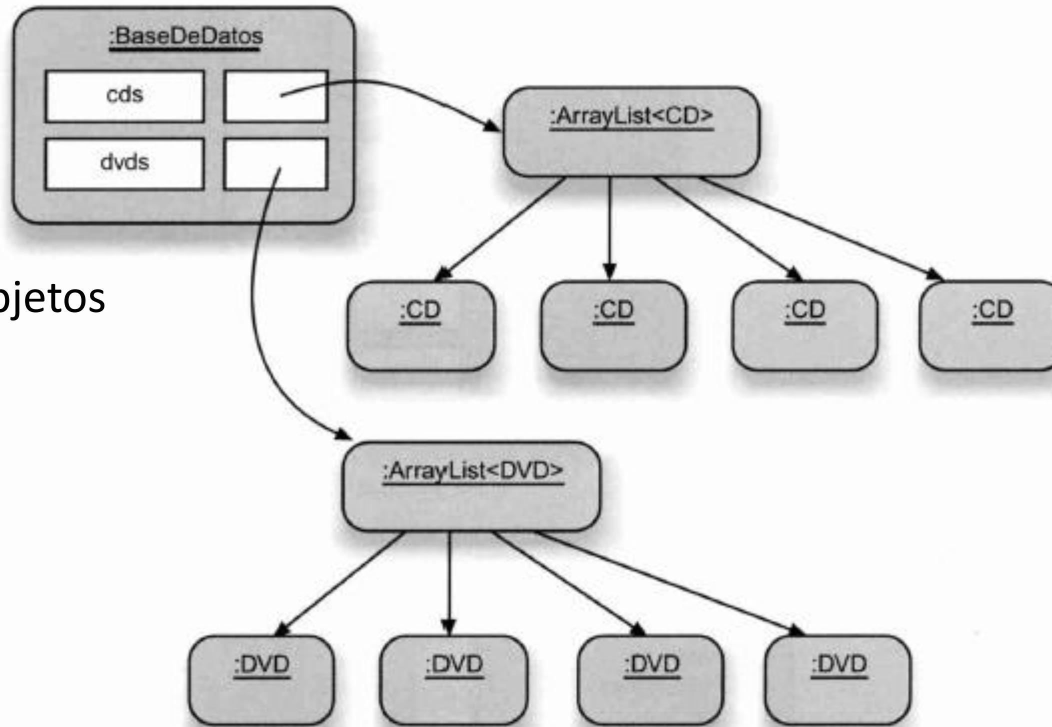




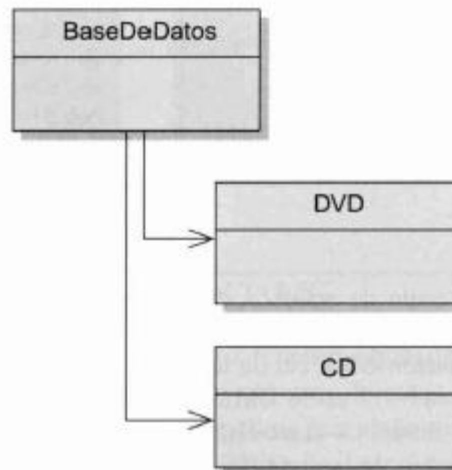
en la parte central se muestran los campos

en la parte inferior de muestran los métodos

Modelo de objetos



Modelo de clases



```java
public class CD
{
   private String title;
   private String artist;
   private String comment;

   CD(String theTitle, String theArtist)
   {
       title = theTitle;
       artist = theArtist;
       comment = " ";
   }

   void setComment(String newComment)
   { ... }

   String getComment()
   { ... }

   void print()
   { ... }
   ...
}
```

```java
public class DVD
{
   private String title;
   private String director;
   private String comment;

   DVD(String theTitle, String theDirector)
   {
       title = theTitle;
       director = theDirector;
       comment = " ";
   }

   void setComment(String newComment)
   { ... }

   String getComment()
   { ... }

   void print()
   { ... }
   ...
}
```

Crítica de la v1 de DoME

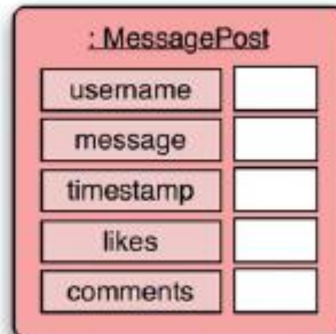
- Duplicación de código
- Las clases CD y DVD son muy parecidas (casi idénticas)

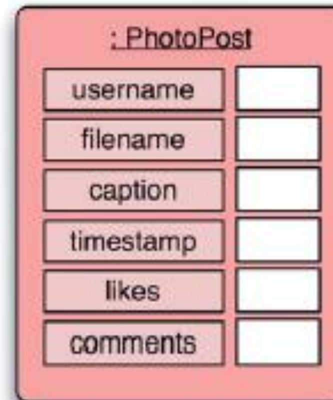
Esto hace el mantenimiento difícil y más trabajoso

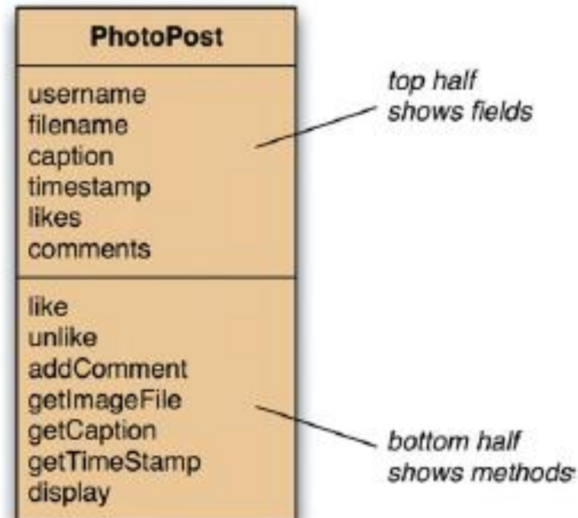
-Riesgo de errores con un mantenimiento inadecuado

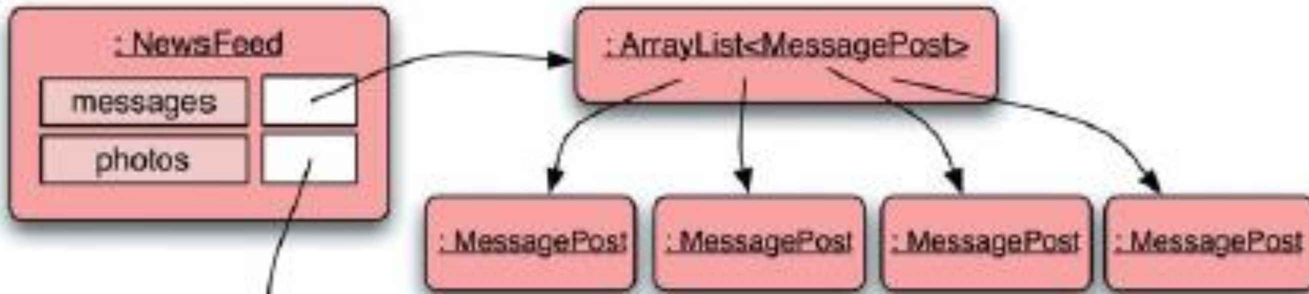
-También hay duplicación de código en la clase Database

network

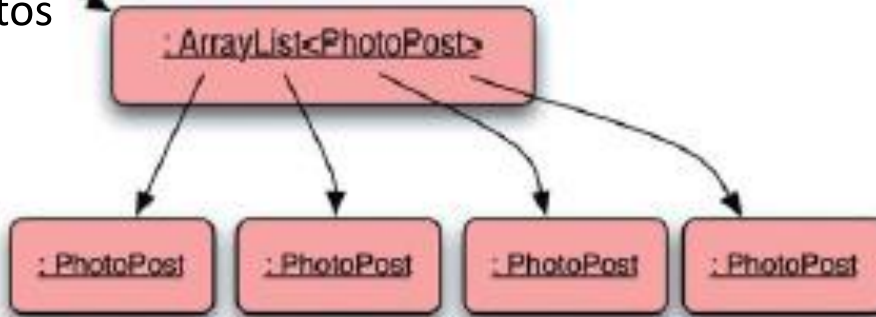




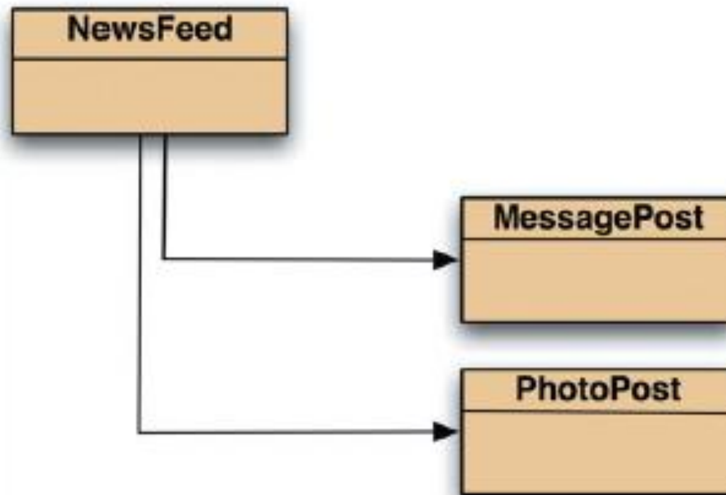




Modelo de objetos



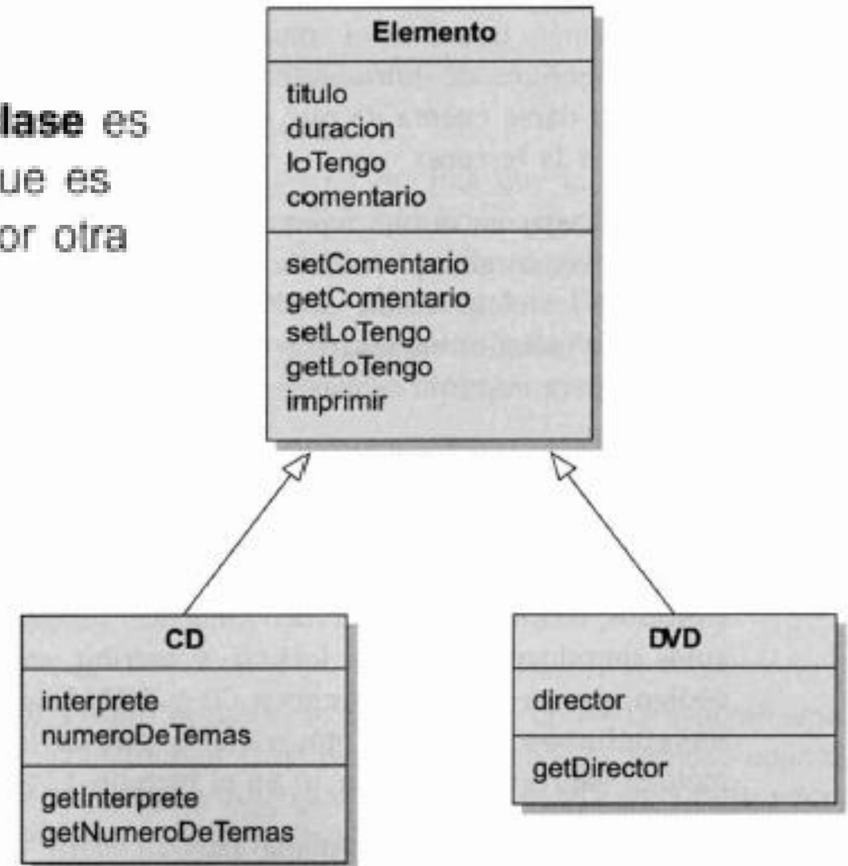
Modelo de clases



8.2 Usar herencia

La **herencia** nos permite definir una clase como una extensión de otra.

Una **superclase** es una clase que es extendida por otra clase.



Una **subclase** es una clase que extiende a otra clase. Hereda todos los campos y los métodos de la superclase.

Definimos una clase que contiene todas las cosas en común de ambas clases

Terminología de la “herencia”

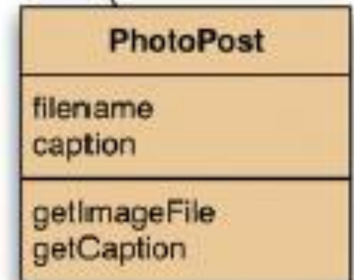
- La clase CD ***deriva*** de la clase elemento
- La clase CD ***extiende*** a la clase elemento
- La herencia se denomina relación ***“ES UN”***
- La clase a partir de la que se derivan o heredan las otras se denominan:
  - clase padre, clase base o superclase
- Las clases heredadas se denominan:
  - Clases derivadas, clases hijo o subclases
- Las instancias de las subclases contienen todos sus campos mas los de la superclase
- Las instancias de las subclases tienen todos los métodos definidos en ambas, la superclase y la subclase

MessagePost
username
message
timestamp
likes
comments
like
unlike
addComment
getText
getTimeStamp
display

PhotoPost
username
filename
caption
timestamp
likes
comments
like
unlike
addComment
getImageFile
getCaption
getTimeStamp
display
top half
shows fields
bottom half
shows methods

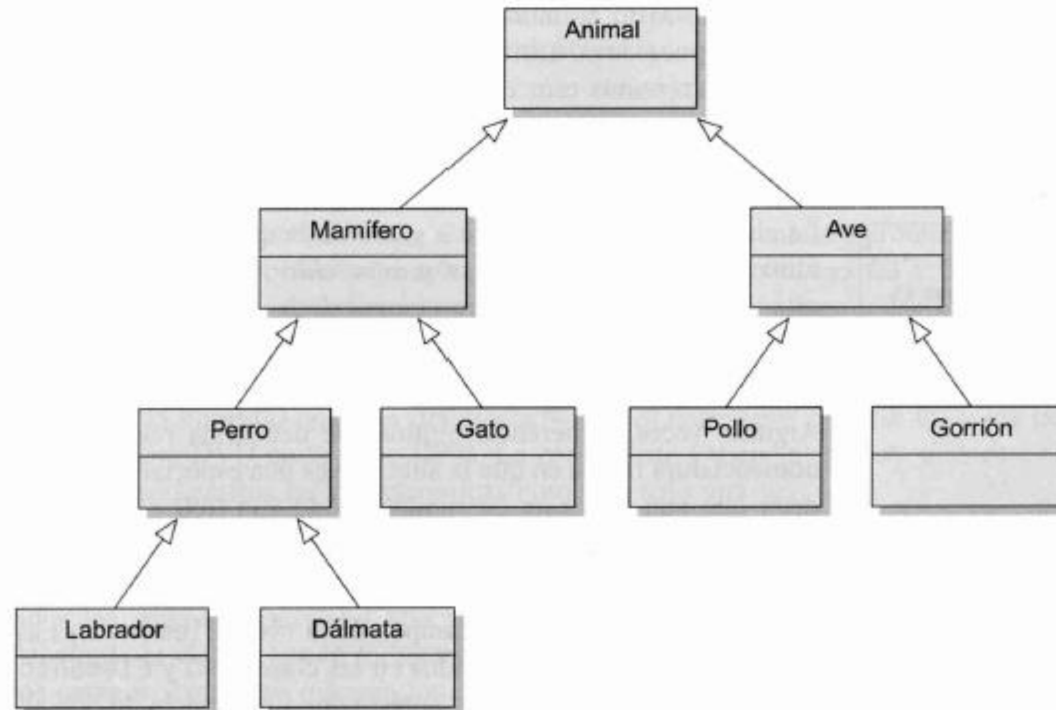
Post
username
timestamp
likes
comments
like
unlike
addComment
getTimeStamp
display

MessagePost
message
getText


PhotoPost
filename
caption
getImageFile
getCaption

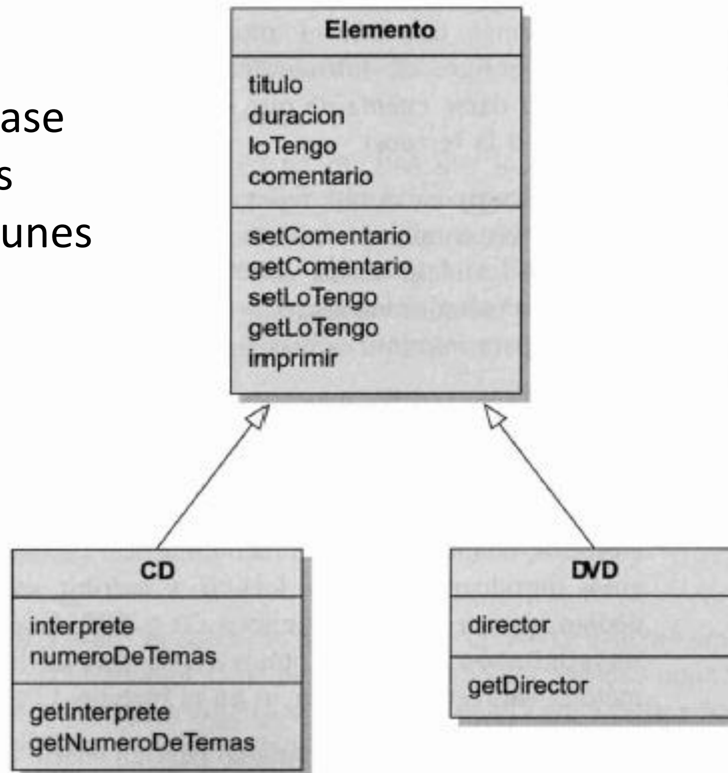
8.3 Jerarquía de herencia

Las clases que están vinculadas mediante una relación de herencia forman una **jerarquía de herencia**.



8.4 Herencia en Java

En la superclase definimos los campos comunes




```
public class Elemento
{
   private String titulo;
   private int duracion;
   private bolean loTengo;
   private String comentario;
   // se omitieron constructores y métodos
}
```

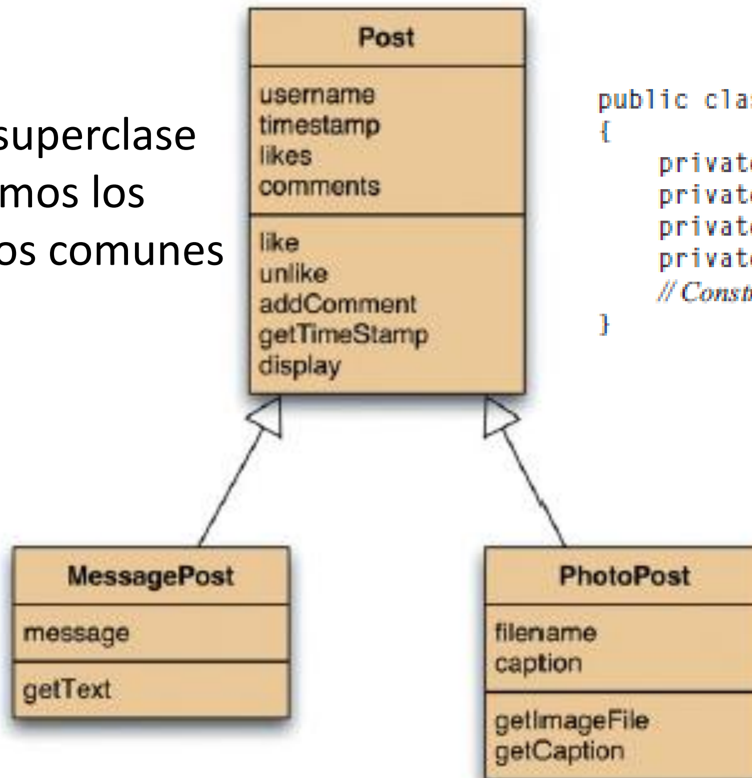
```
public class CD extends Elemento
{
   private String interprete;
   private int numeroDeTemas;
   // se omitieron constructores y métodos
}
```

```
public class DVD extends Elemento
{
   private String director;
   // se omitieron constructores y métodos
}
```

Uso de la herencia

- Se define una **superclase**
- Se definen **subclases**
- La superclase define atributos comunes
- Las subclases **heredan** los atributos de la superclase
- Las subclases pueden tener sus propios atributos

En la superclase definimos los campos comunes




```
public class Post
{
    private String username;  // username of the post's author
    private long timestamp;
    private int likes;
    private ArrayList<String> comments;
    // Constructors and methods omitted.
}
```

```
public class MessagePost extends Post
{
    private String message;
    // Constructors and methods omitted.
}
```

```
public class PhotoPost extends Post
{
    private String filename;
    private String caption;
    // Constructors and methods omitted.
}
```

8.4.1 Herencia y derecho de acceso

- Una subclase no puede acceder a los miembros privados de la superclase
  - Si un método de una subclase necesita acceder a un campo privado de su superclase, la superclase necesitará ofrecer los métodos apropiados.
- Una subclase puede invocar a cualquier método público de su superclase como si fuera propio, no necesita ninguna variable

8.4.2 Herencia e iniciación

```
public class Elemento
{
    private String titulo;
    private int duracion;
    private boolean loTengo;
    private String comentario;
    /**
     * Inicializa los campos del elemento.
     * @param elTitulo  el título de este elemento.
     * @param tiempo La duración de este elemento.
     */
    public Elemento(String elTitulo, int tiempo)
    {
        titulo = elTitulo;
        duracion = tiempo;
        loTengo = false;
        comentario = "";
    }
    // Se omitieron métodos
}
public class CD extends Elemento
{
    private String interprete;
    private int numeroDeTemas;
    /**
     * Constructor de objetos de la clase CD
     * @param elTitulo El título del CD.
     * @param elInterprete  El intérprete del CD.
     * @param temas  El número de temas del CD.
     * @param tiempo La duración del CD.
     */
    public CD(String elTitulo, String elInterprete, int
temas, int tiempo)
    {
        super(elTitulo, tiempo);
        interprete = elInterprete;
        numeroDeTemas = temas;
    }
    // Se omitieron métodos
}
```

La palabra clave ***super*** es una llamada al constructor de la superclase.

La llamada al ***super*** debe tener los mismos parámetros que tenga el constructor de la superclase

El constructor de la superclase inicializa los campos correspondientes y le pasa el control al constructor de la subclase

Constructor de superclase. El constructor de una subclase debe tener siempre como primera sentencia una invocación al constructor de su superclase. Si el código no incluye esta llamada, Java intentará insertarla automáticamente.

Llamada al constructor de la superclase

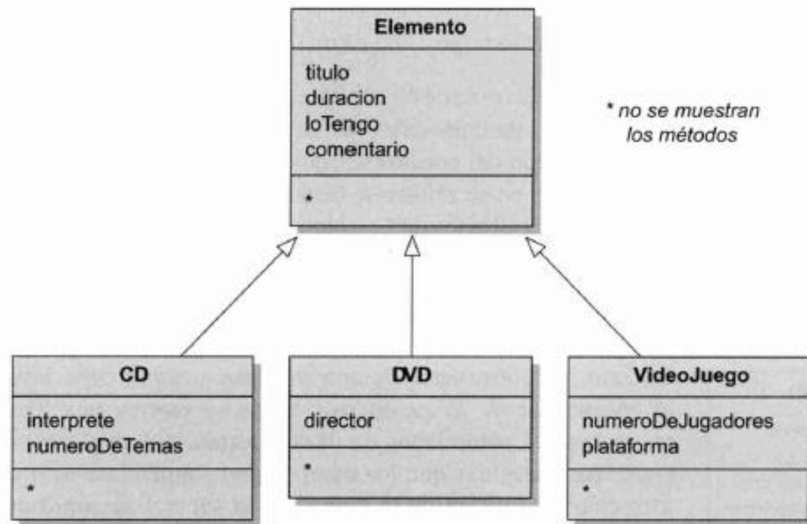
- Los constructores de una subclase siempre deben contener una llamada a un constructor de la superclase
  - Utilizando **super(parámetros);**
- Siempre tiene que ser ***la primera instrucción*** del código de un constructor
- Si no se pone nada, el compilador asume que hay una llamada sin parámetros:
  - **super();**
- Esto implica que la superclase tendría que tener definido un constructor sin parámetros
  - Si sólo tuviera constructores con parámetros, entonces el compilador señalaría el error

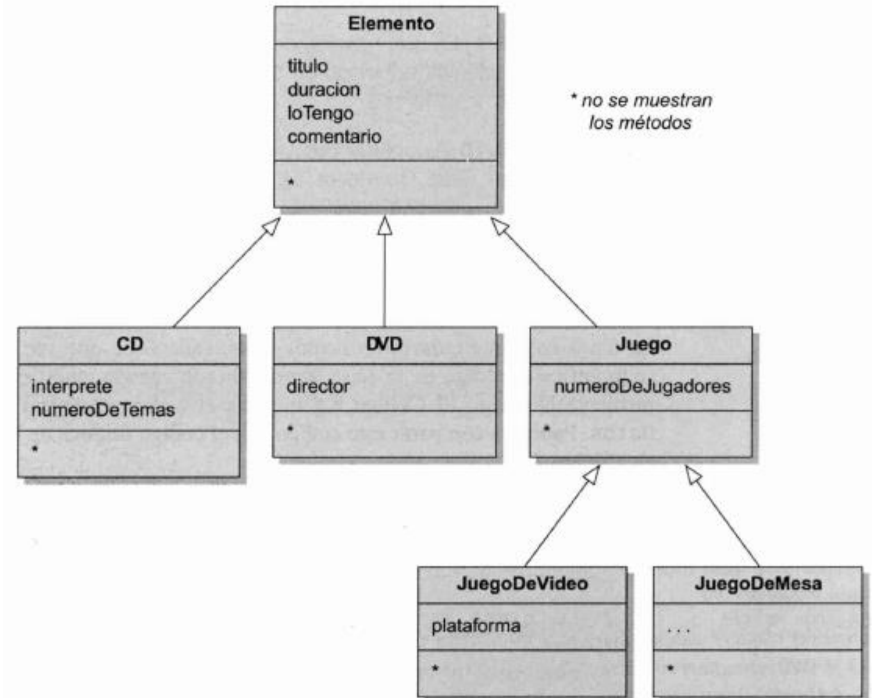
```java
public class Post
{
    private String username;  // username of the post's author
    private long timestamp;
    private int likes;
    private ArrayList<String> comments;

    /**
     * Constructor for objects of class Post.
     *
     * @param author    The username of the author of this post.
     */
    public Post(String author)
    {
        username = author;
        timestamp = System.currentTimeMillis();
        likes = 0;
        comments = new ArrayList<String>();
    }

    // Methods omitted.
}
```

```java
public class MessagePost extends Post
{
    private String message;  // an arbitrarily long, multi-line message

    /**
     * Constructor for objects of class MessagePost.
     *
     * @param author    The username of the author of this post.
     * @param text      The text of this post.
     */
    public MessagePost(String author, String text)
    {
        super(author);
        message = text;
    }

    // Methods omitted.
}
```

8.5 Agregar otras subclases

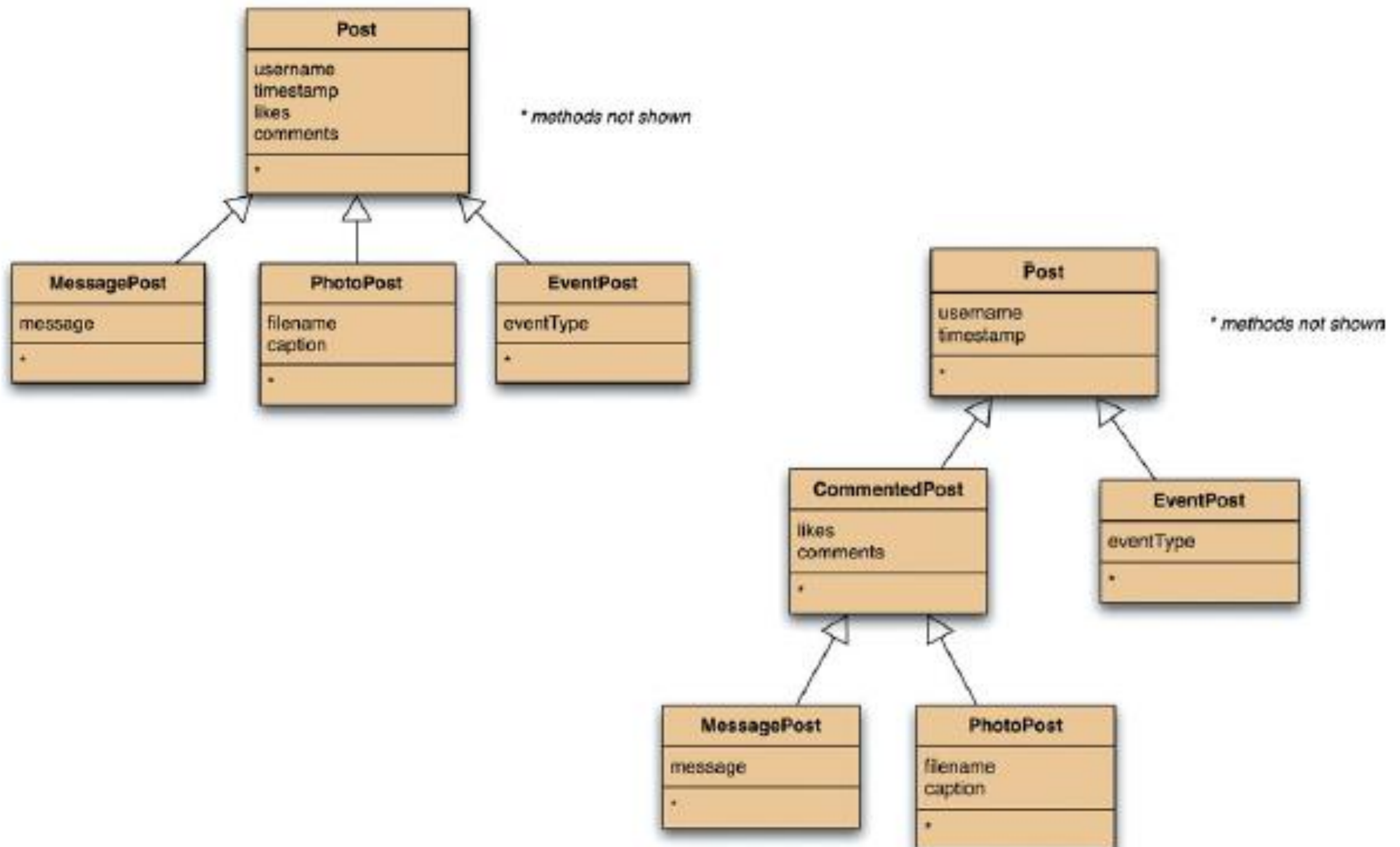
La herencia nos permite **reutilizar** en un nuevo contexto clases que fueron escritas previamente.





Las clases que no se piensan usar para crear instancias, cuyo propósito es exclusivamente servir como superclase de otra, se denomina "***clases abstractas***"

Refactorización de las clases



8.6 Ventajas de la Herencia

- La herencia contribuye a:
  - Evitar duplicación de código
  - Reutilizar código
  - Mejorar el mantenimiento
  - Extensibilidad

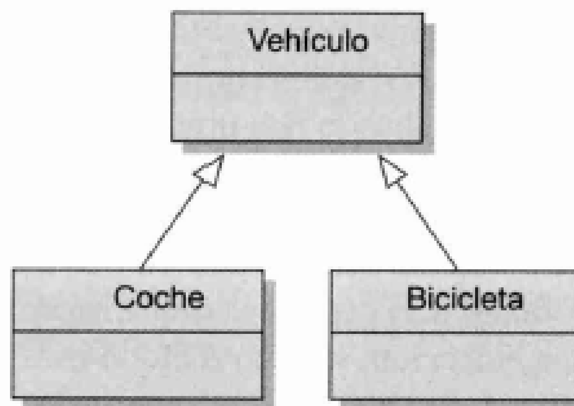
```
 * @author Michael Kölling and David J. Barnes
 * @version 2006.03.30
 */
public class BaseDeDatos
{
    private ArrayList<CD> cds;
    private ArrayList<DVD> videos;
    /**
     * Construye una BaseDeDatos vacía.
     */
    public BaseDeDatos()
    {
        cds = new ArrayList<CD>();
        dvds = new ArrayList<DVD>();
    }
    /**
     * Agrega un CD a la base.
     * @param elCD El CD que se agregará a la base de
datos.
     */
    public void agregarCD(CD elCD)
    {
        cds.add(elCD);
    }
```

Comparación del código de basededatos, al incorporar la "herencia" en el diseño

```
 * @author Michael Kölling and David J. Barnes
 * @version 2006.03.30
 */
public class BaseDeDatos
{
    private ArrayList<Elemento> elementos;
    /**
     * Construye una BaseDeDatos vacía.
     */
    public BaseDeDatos()
    {
        elementos = new ArrayList<Elemento>();
    }
    /**
     * Agrega un elemento en la base.
     */
    public void agregarElemento(Elemento elElemento)
    {
        elementos.add(elElemento);
    }
    /**
     * Imprime una lista en la terminal de texto de todos
     * los elementos almacenados actualmente.
     */
    public void listar()
    {
        for(Elemento elemento : elementos )
        {
            elemento.imprimir();
         System.out.println(); //una línea vacía entre
elementos
        }
    }
}
```

8.7 Subclases y subtipos

Subtipo. Por analogía con la jerarquía de clases, los tipos forman una jerarquía de tipos. El tipo que se define mediante la definición de una subclase es un subtipo del tipo de su superclase.



Variables y subtipos. Las variables pueden contener objetos del tipo declarado o de cualquier subtipo del tipo declarado.

Imagine que tenemos una clase `Vehiculo` con dos subclases, `coche` y `Bicicleta` (Figura 8.9). En este caso la regla de tipos admite que las siguientes sentencias son todas legales:

```
Vehiculo v1 = new Vehiculo();
Vehiculo v2 = new Coche();
Vehiculo v3 = new Bicicleta();
```

Sustitución. Se pueden usar objetos de subtipos en cualquier lugar en el que se espera un objeto de un supertipo. Esto se conoce como sustitución.

Sin embargo, no está permitido hacer esto de otra manera:

```
Coche a1 = new Vehiculo();      // ¡Es un error!
Coche a2 = new Bicicleta();     // ¡Es un error!
```

8.7.3 Subtipos y paso de parámetros

- El paso de parámetros se comporta exactamente de la misma manera que la asignación ordinaria a una variable

```
public void agregarElemento(Elemento elElemento)
{
   ...
}
```

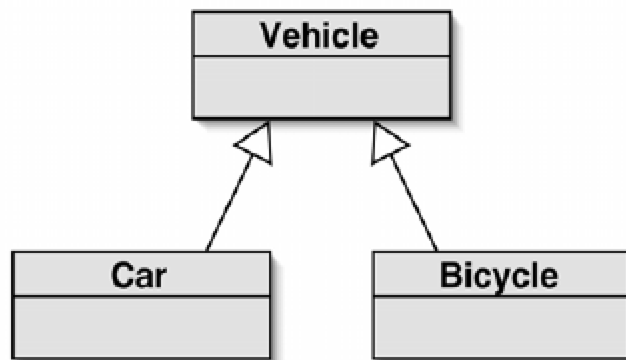
Ahora podemos usar este método para agregar objetos DVD y CD en la base de datos:

```
BaseDeDatos bd = new BaseDeDatos();
DVD dvd = new DVD(...);
CD cd = new CD(...);

bd.agregarElemento(dvd);
bd.agregarElemento(cd);
```

Debido a las reglas de los subtipos, sólo necesitamos un método (con un único tipo de parámetro) para agregar tanto objetos DVD como CD.

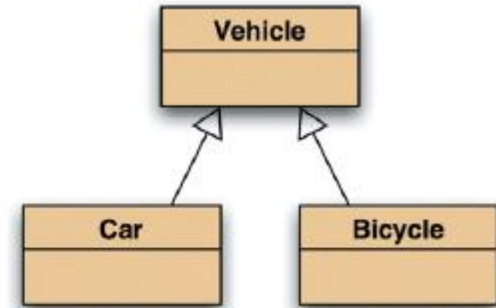
Polimorfismo

- Variables Polimórficas
  - Las variables que contienen objetos son variables polimórficas
  - El término polimórfico se refiere al hecho de que una misma variable puede contener objetos de diferentes tipos




```
Vehicle v1 = new Vehicle();
Vehicle v2 = new Car();
Vehicle v3 = new Bicycle();
```

8.7.5 Casting (Enmascaramiento de tipos)




```
Vehiculo v;
  Coche a = new Coche();
  v = a;      // es correcta
  a = v;      // es un error
   .
  a = (Coche) v;   // correcto
```

El operador de enmascaramiento consiste en el nombre de un tipo (en este caso, `Coche`) escrito entre paréntesis, que precede a una variable o a una expresión. Al usar esta operación, el compilador creerá que el objeto es un `Coche` y no informará ningún error.

Ahora considere este fragmento de código en el que `Bicicleta` también es una subclase de `Vehiculo`:

```
Vehiculo v;
  Coche a;
Bicicleta b;
a = new Coche();
  v = a;      // correcta
  b = (Bicicleta) a;     // ¡error en tiempo de compilación!
  b = (Bicicleta) v;     // ¡error en tiempo de ejecución!
```

Las últimas dos asignaciones fallarán. El intento de asignar la variable `a` en la variable `b` (aun estando enmascarada) dará por resultado un error en tiempo de compilación. El compilador se dará cuenta de que `Coche` y `Bicicleta` no constituyen una relación subtipo/supertipo, y por este motivo la variable `a` nunca puede contener un objeto `Bicicleta`; esta asignación no funcionará nunca.

casting

- Se especifica indicando el tipo de objeto entre paréntesis
- El objeto no cambia en nada
  - Simplemente se permite usar la referencia adecuadamente
- En tiempo de ejecución se comprueba que el objeto es realmente de ese tipo
  - **ClassCastException** si no lo es
- En Java esto se puede comprobar

  if (v **instanceof** A) // si el objeto v pertenece a la clase A o una de sus subclases
- Para el ejemplo anterior:

  if (v instanceof Car)

  c = (Car)v;
- *debe usarse con moderación*

8.8 La clase "Object"

Concepto

Todas aquellas clases que no tienen una superclase explícita tienen como su superclase a la clase **Object**.

- Todas las clases tienen una superclase de nombre *java.lang.Object*
- Que todos los objetos tengan una super clase en común tiene dos objetos:
  - Poder declarar variables polimórficas de tipo "Object", que puedan contener cualquier objeto
  - Definir algunos métodos que están automáticamente disponibles para cada objeto disponible

8.9 Autoboxing y clase envoltorio

Clases «envoltorio»

En Java, cada tipo primitivo tiene su correspondiente clase «envoltorio» que representa el mismo tipo pero que en realidad, es un tipo objeto. Estas clases hacen posible que se usen valores de tipos primitivos en los lugares en que se requieren tipos objeto mediante un proceso conocido como *autoboxing*. La siguiente tabla enumera los tipos primitivos y sus correspondientes clases envoltorio del paquete `java.lang`. Excepto `Integer` y `Character`, los nombres de las clases envoltorio coinciden con los nombres de los tipos primitivos, pero con su primera letra en mayúscula.

Tipo primitivo	Tipo envoltorio
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`char`	`Character`
`boolean`	`Boolean`

Términos introducidos en este capítulo

herencia, superclase(padre), subclase(hijo), es-un, jerarquía de herencia, clase abstracta, subtipo, sustitución, variable polimórfica, pérdida de tipo, enmascaramiento, autoboxing, clases envoltorio

Resumen de conceptos

- **herencia** La herencia nos permite definir una clase como extensión de otra.
- **superclase** Una superclase es una clase que es extendida por otra clase.
- **subclase** Una subclase es una clase que extiende (deriva de) otra clase. Hereda todos los campos y métodos de su superclase.
- **jerarquía de herencia** Las clases que están vinculadas mediante una relación de herencia forman una jerarquía de herencia.
- **constructores de superclase** El constructor de una subclase siempre debe invocar al constructor de la superclase en su primera sentencia. Si el código fuente no incluye esta llamada, Java intentará insertarla automáticamente.
- **reutilización** La herencia nos permite reutilizar en un nuevo contexto clases escritas previamente.
- **subtipos** Por analogía con la jerarquía de clase, los tipos forman una jerarquía de tipos. El tipo definido mediante una definición de subclase es un subtipo del tipo de su superclase.
- **variables y subtipos** Las variables pueden contener objetos de su tipo declarado o de cualquier subtipo de su tipo declarado.
- **sustitución** Los objetos subtipo pueden usarse cada vez que se espera un super-tipo. Esto se conoce como sustitución.
- **Object** Todas las clases que no tienen una superclase explícita tienen a `Object` como su superclase.
- **Autoboxing** El proceso de autoboxing se lleva a cabo automáticamente cuando se usa un valor de un tipo primitivo en un contexto que requiere un tipo objeto.