Unidad 2: Mejora de la estructura mediante la herencia

Metodología de la Programación

Curso 2017-2018

© *Candi Luengo Díez , Francisco Ortín Soler y José Manuel Redondo López*

Bibliografía

- **Programación orientada a objetos con Java. 6th Edición**
  David Barnes, Michael Kölling.
  Pearson Education. 2017

  Capítulo 10: Mejora de la estructura mediante la herencia

Principales conceptos

- **Arquitectura** de un programa
- **Problemas comunes** cuando se implementa software
- **Herencia**
- **Generalización / polimorfismo**
- Uso de **cast** con diferentes tipos de objetos
- Técnica **autoboxing**

El ejemplo DoME

"Database of Multimedia Entertainment"

- Aplicación para almacenar información de CDs y DVDs
- El propósito es crear un catalogo de CDs y DVDs
- **Funcionalidades**
  - Añadir CDs and DVDs
  - Almacenar su información
  - Buscar CDs y DVDs
  - Mostrar informes (listados) en pantalla
  - Borrar CDs y DVDs

Clases en el DoME

Atributos y Métodos de las **clases** CD y DVD

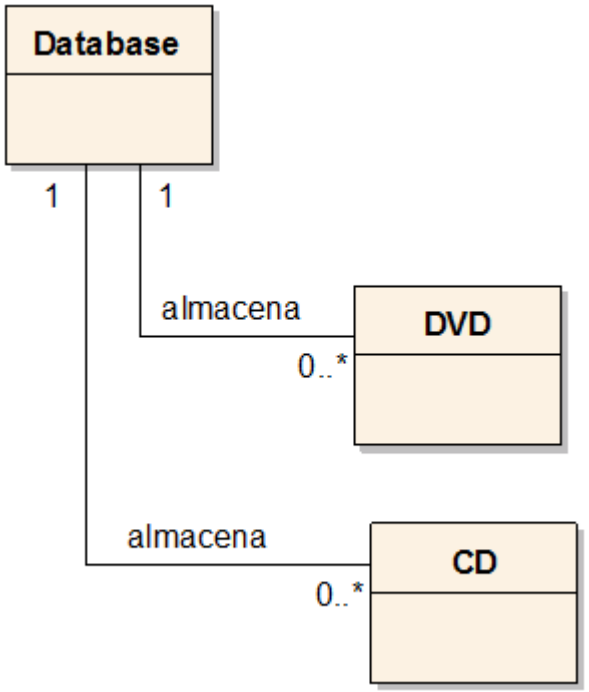
CD
- title:String - artist:String - numberOfTracks:int - playingTime:int - gotIt:boolean - comment:String
+getComment():String +setComment(comment:String) +getOwn():boolean +setOwn(ownIt:boolean) +print(PrintStream)

DVD
- title:String - director:String - playingTime:int - gotIt:boolean - comment:String
+getComment():String +setComment(comment:String) +getOwn():boolean +setOwn(ownIt:boolean) +print(PrintStream)

Indica si lo tengo en propiedad

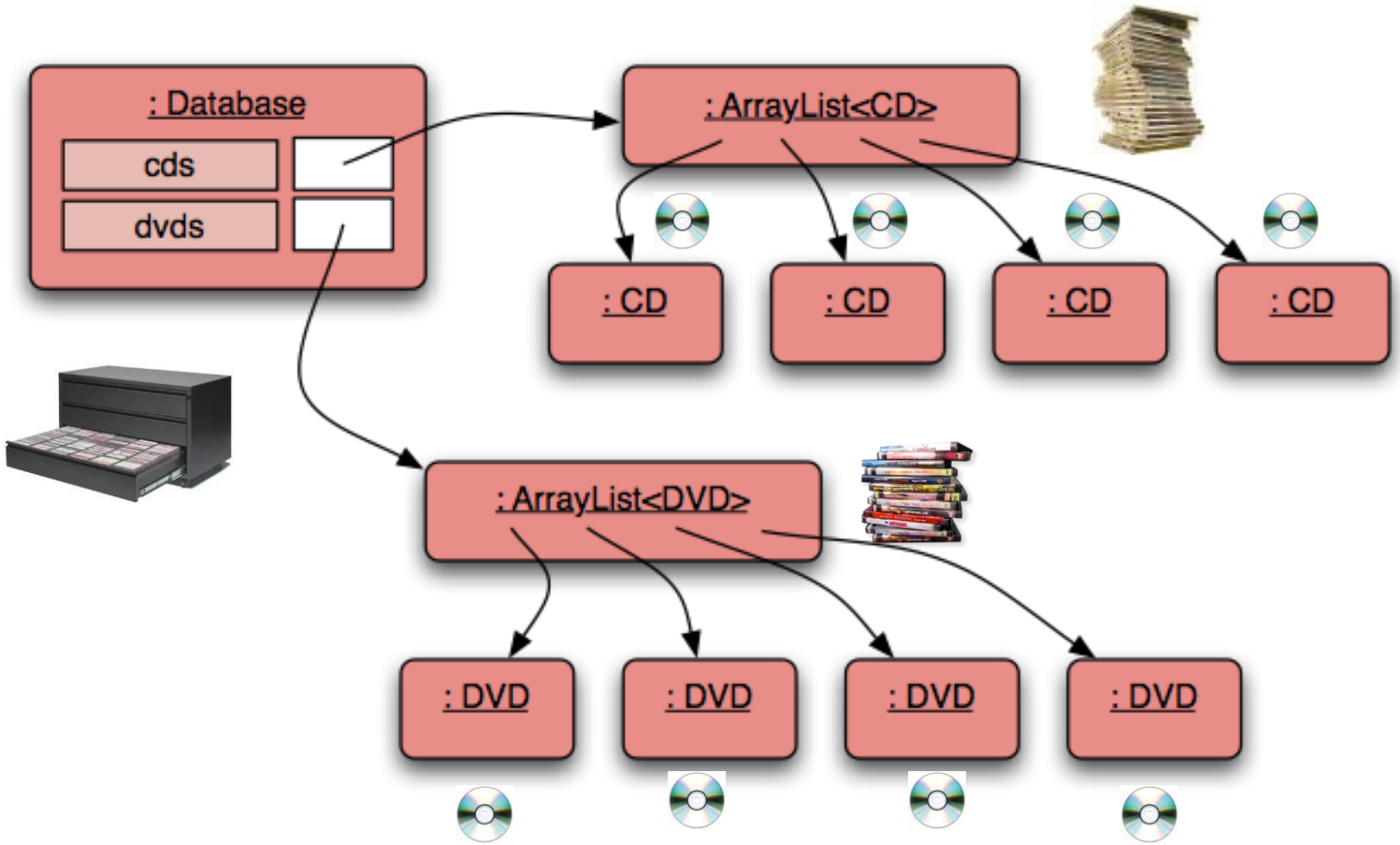
El ejemplo DoME

Diagrama de clases no detallado



El ejemplo DoME

Diagrama de Objetos en la aplicación DoME



Código fuente

Clase **CD**

```
public class CD {
  private String title;
  private String artist;
  private int numberOfTracks;
  private int playingTime;
  private boolean gotIt;
  private String comment;

  public CD(String theTitle, String theArtist,
            int tracks, int time){
    setTitle(theTitle);
    setArtist(theArtist);
    setNumberOfTracks(tracks);
    setPlayingTime(time);
    setOwn(false);
    setComment("");
  }
```

Código fuente de la clase **CD**

```
public void setComment(String comment) {
        this.comment = comment;
}
public String getComment() { return comment; }

public void setOwn(boolean ownIt) {  gotIt = ownIt; }

public boolean getOwn() { return gotIt; }

...

 public void print(PrintStream out) {
        out.print("CD: " + title + " (" + playingTime + " mins)");
        if (gotIt) out.println("*");
        else out.println();
        out.println(" " + artist);
        out.println(" tracks: " + numberOfTracks);
        out.println(" " + comment);
}   }
```

```
public final class System
{
   static PrintStream err;
   static PrintStream out;
   …
```

Código fuente

Clase **DVD**

```
public class DVD {
        private String title;
        private String director;
        private int playingTime;
        private boolean gotIt;
        private String comment;

        public DVD(String theTitle, String theDirector,
                        int time) {
                setTitle(theTitle);
                setDirector(theDirector);
                setPlayingTime(time);
                setOwn(false);
                setComment("");
        }
```

Código fuente clase **DVD**


```
public void setComment(String comment) {
        this.comment = comment;
}
public String getComment() { return comment; }
public void setOwn(boolean ownIt) {
        gotIt = ownIt;
} ...
public boolean getOwn() { return gotIt; }
public void print(PrintStream out) {
        out.println("DVD: " + title + " (" + playingTime + " mins)");
        if (gotIt) out.println("*");
        else out.println();
        out.println(" " + director);
        out.println(" " + comment);
} }
```

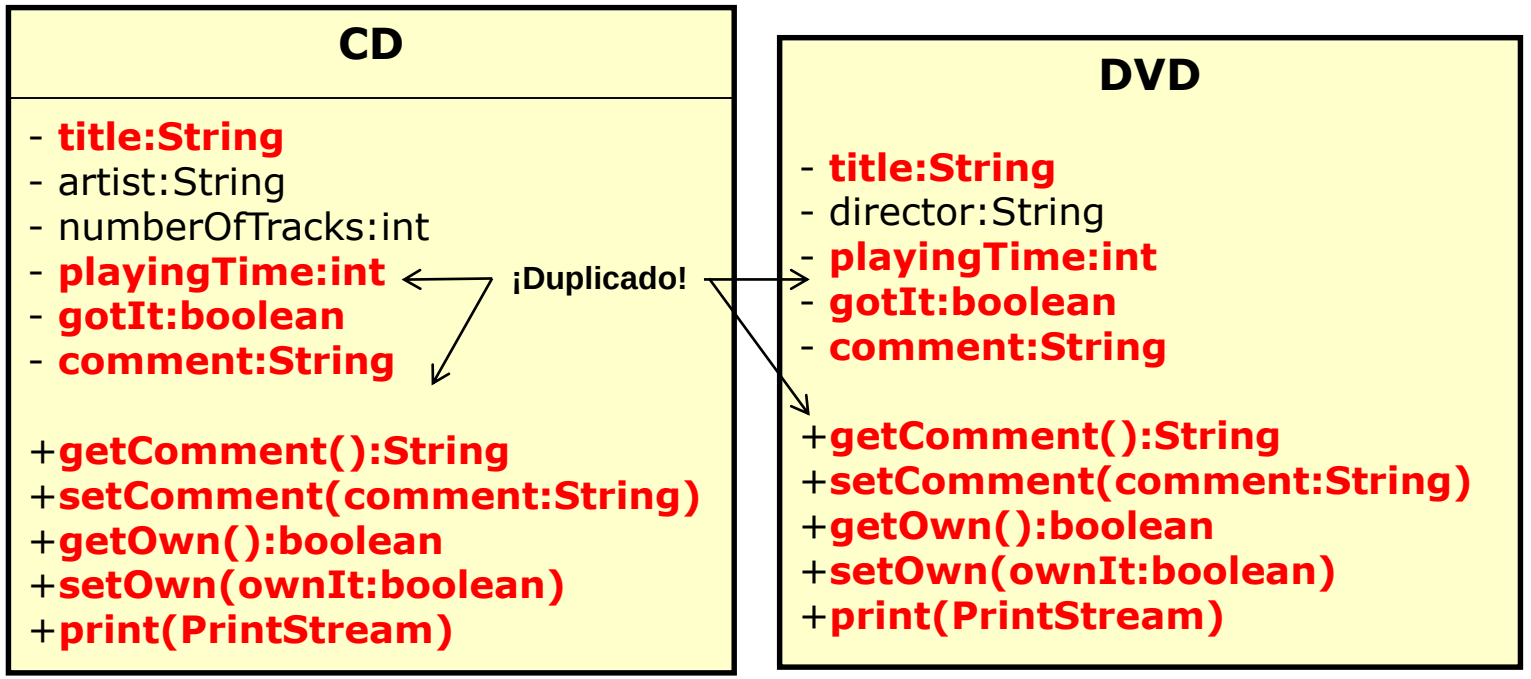
Database código fuente

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

  public Database() {
    cds = new ArrayList<CD>();
    dvds = new ArrayList<DVD>();
  }
  public void add(CD theCD) {
    cds.add(theCD);
  }
  public void add(DVD theDVD) {
    dvds.add(theDVD);
  }
```

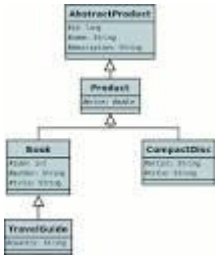
```
  public void list() {
    for (CD cd : cds) {
      cd.print(System.out);
      System.out.println();
      }
    for (DVD dvd : dvds) {
      dvd.print(System.out);
      System.out.println();
      }
    }

}
```

Críticas al ejemplo DoME

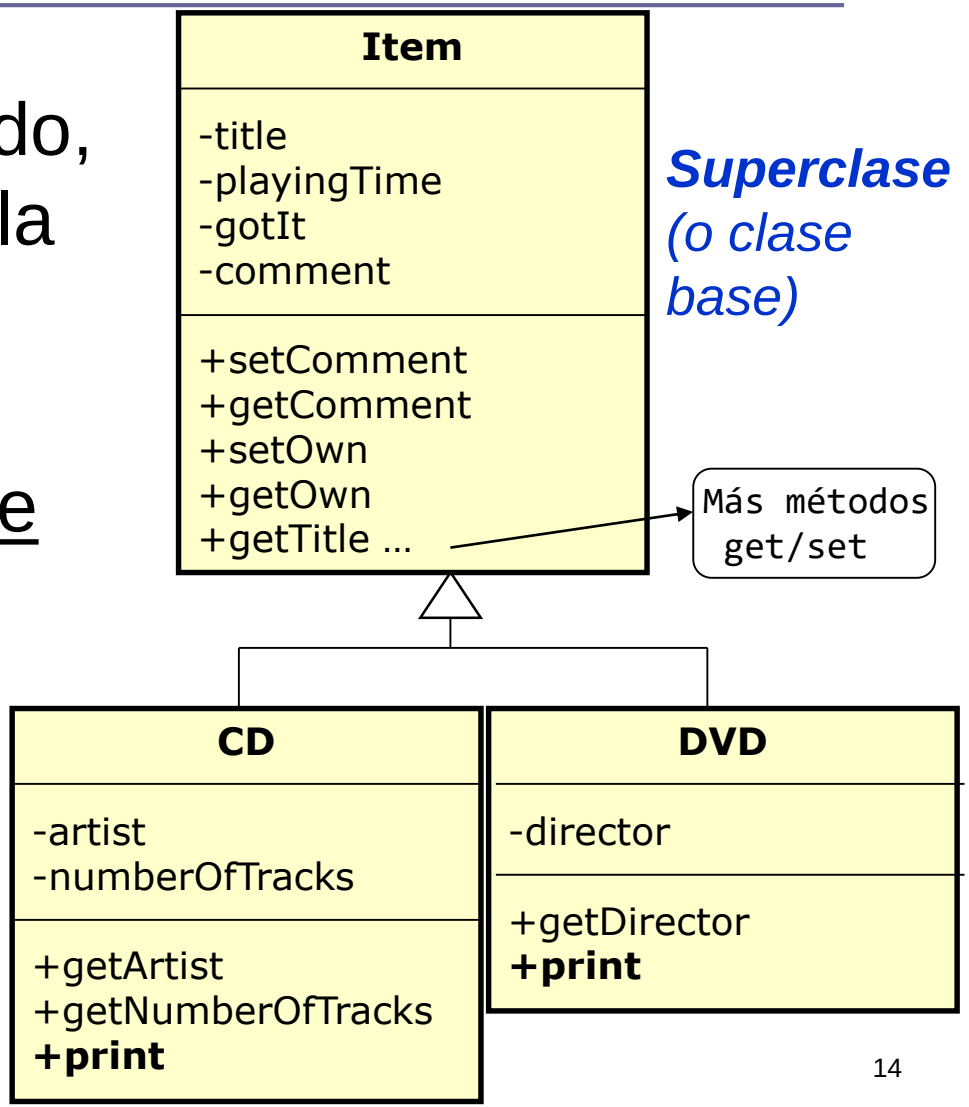
- La **duplicación del código** implica menor mantenibilidad.
  - Cambios, extensiones y depuración del código debe realizarse en varios lugares
- Las clases CD y DVD son muy similares



Usando herencia



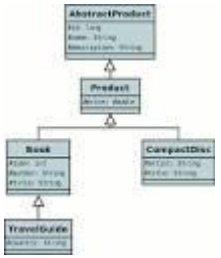
- La extracción de *superclases* refactorizando, resuelve el problema de la duplicación de código
- La **herencia**, es una técnica de reutilización de código
- Las subclases heredan todos los atributos y métodos de la superclase



Superclase *(o clase base)*

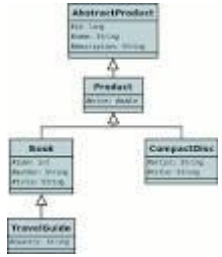
Subclases *(o clases derivadas)*

Usando herencia



- Se crea una **superclase** : **`Item`**
- Se crean las **subclases** : **`DVD`** y **`CD`**
- La superclase define atributos y métodos comunes a las subclases.
- Las subclases **heredan** los atributos y métodos de su superclase.
- Las subclases añaden sus propios atributos y/o métodos

Usando herencia



Item

-title
-playingTime
-gotIt
-comment

+setComment
+getComment
+setOwn
+getOwn
+getTitle …

CD

-artist
-numberOfTracks

+getArtist
+getNumberOfTracks
+print

DVD

-director

+getDirector
+print

:CD

title = "Yellow Submarine"
artist = "The Beatles"
playingTime = 90
numberOfTracks = 12
gotIt = true
comment = ""

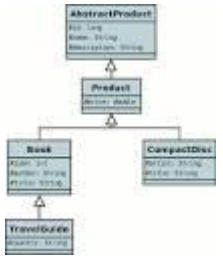
Métodos:
`setComment`
`getComment`
`setOwn`
`getOwn`
`getTitle`
`getArtist`
`getNumberOfTracks`
`print`

:DVD

title = "Inteligencia Artificial"
director = "Steven Spielberg"
playingTime = 160
gotIt = true
comment = "Buena"

Métodos:
`setComment`
`getComment`
`setOwn`
`getOwn`
`getTitle`
`getDirector`
`print`

Herencia en Java - Sintaxis



```
public class Item {
   private String title;
   private int playingTime;
   private boolean gotIt;
   private String comment;
   // methods...
}
```

```
public class DVD extends Item {
   private String director;
   // methods..
}
```

```
public class CD extends Item {
   private String artist;
   private int numberOfTracks;
   // methods...
}
```

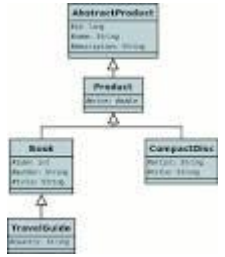
Herencia y ocultación de información

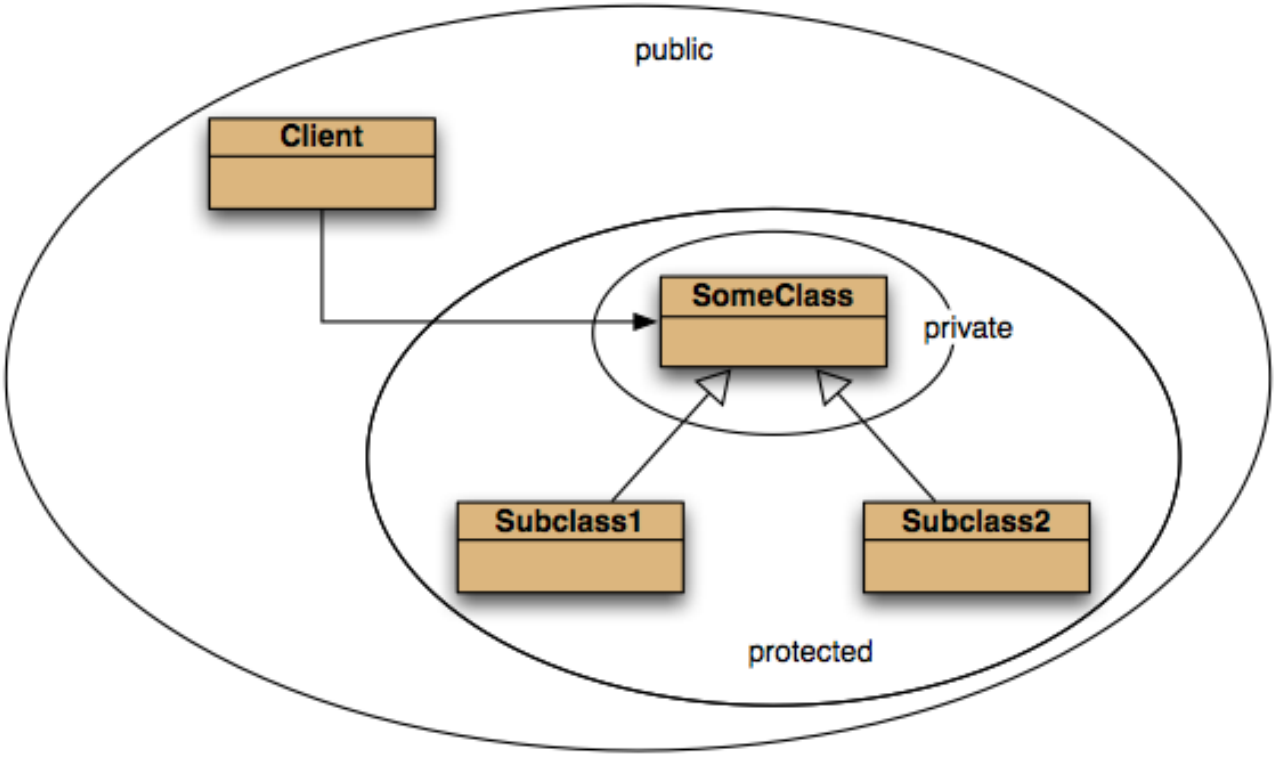
Access Levels

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

- **Recordando**
  - `Los miembros` **`private`** solo pueden ser accedidos en su propia clase.
  - `Los miembros` **`public`** son accesibles en todas las clases (del mismo package y de otros).
  - `Los miembros sin modificador de acceso public o private` (ocultación por defecto) solo pueden ser accedidos en la propia clase y propio paquete (que esté contenida la clase).
- **Con herencia**
  - `Los miembros` **`protected`** solo pueden ser accedidos en la propia clase, propio paquete y además, en las subclases de su clase que se encuentren en otro paquete.

Niveles de acceso





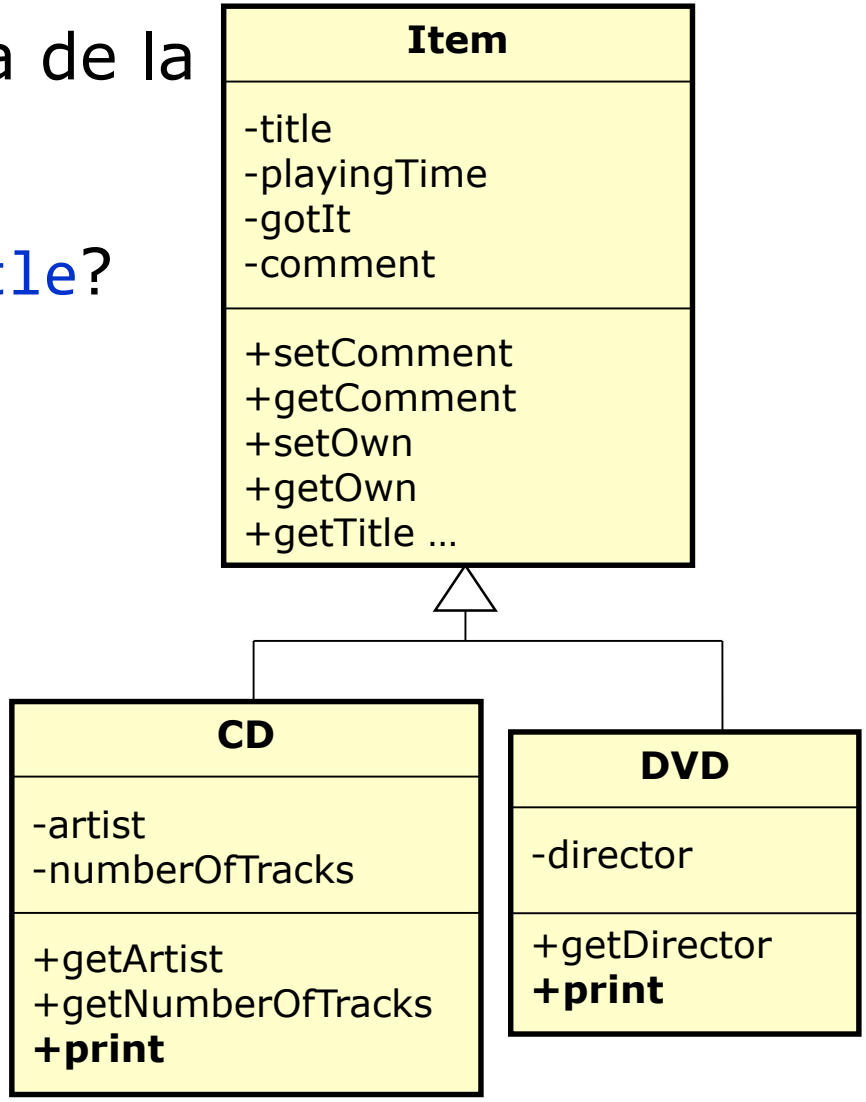
La herencia representa una forma más cerrada de acoplamiento.

La herencia vincula las clases de manera muy cercana y la modificación de la superclase puede romper fácilmente la subclase.

Pregunta

- Si creamos una instancia de la clase `CD`,

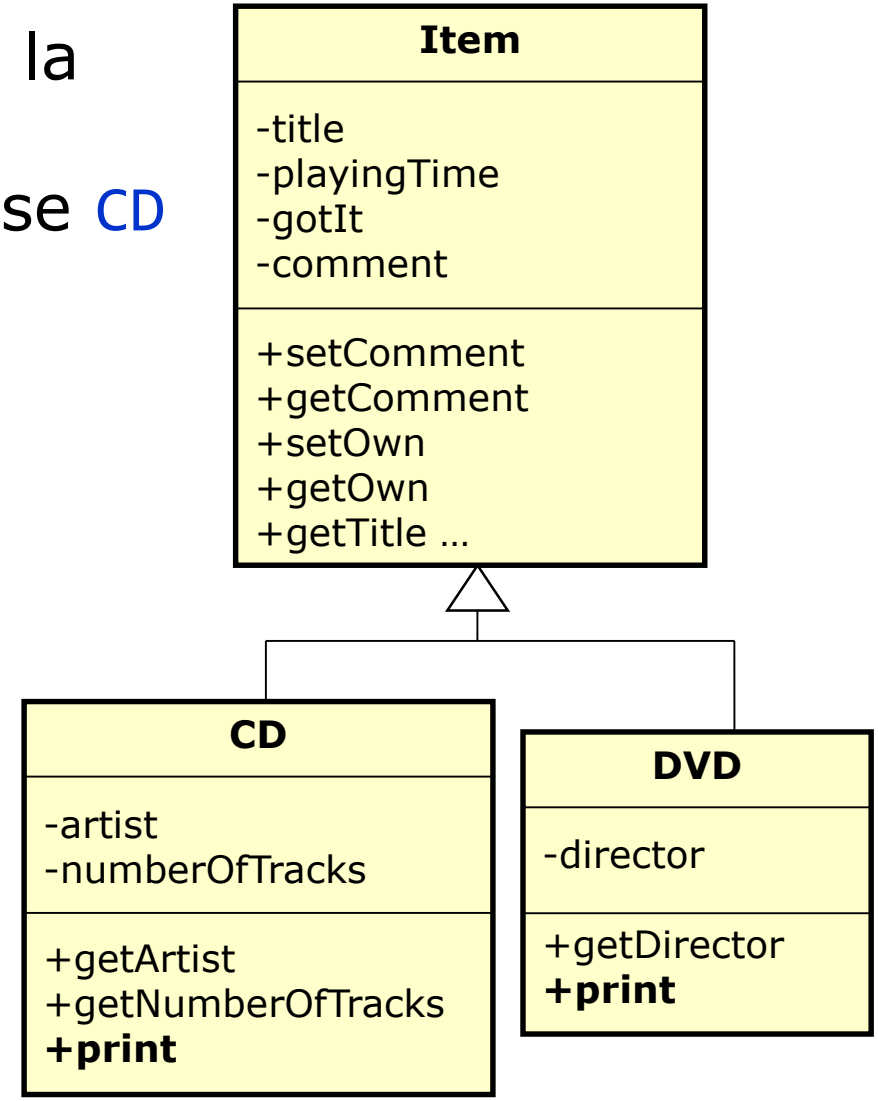
¿Contiene la propiedad `title`?



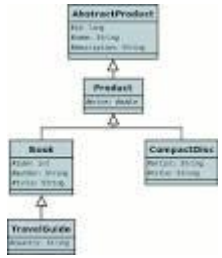
Pregunta

- Si queremos acceder a la propiedad `title` en el método `print` de la clase `CD`

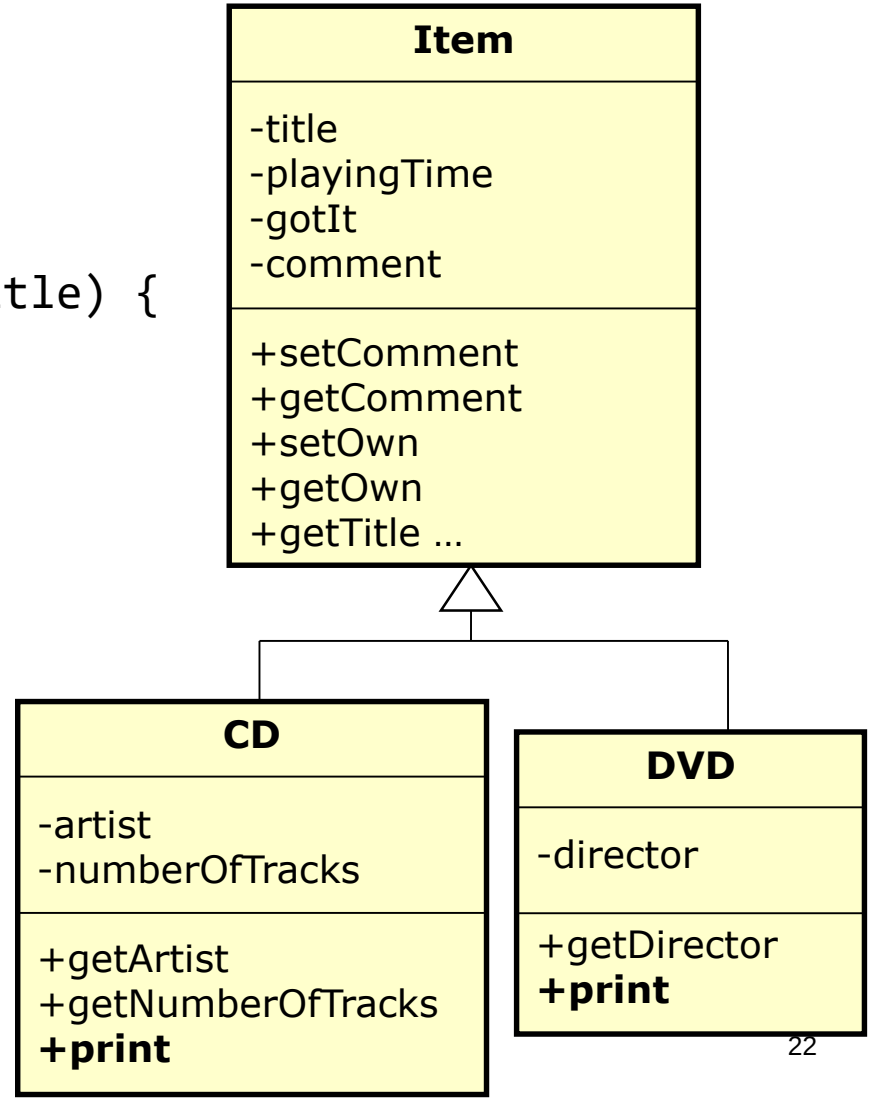
  ¿Como lo hacemos?



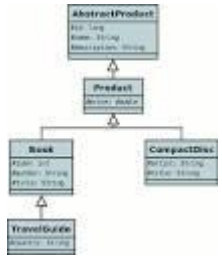
Herencia y ocultación de la información



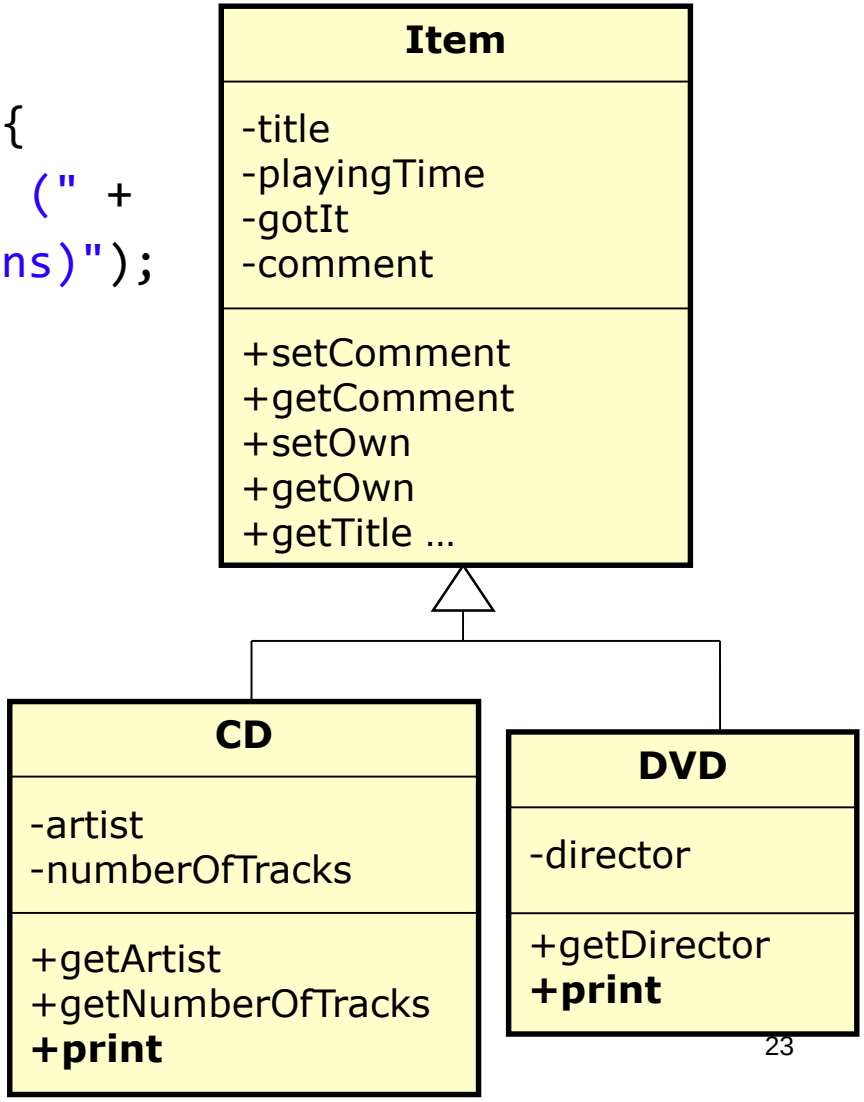
```
public class Item {
private String title;

   protected void setTitle(String title) {
         this.title = title;
    }
   protected String getTitle() {
         return title;
    }
    protected int getPlayingTime() {
         return playingTime;
    }
    // Resto del código...
}
```



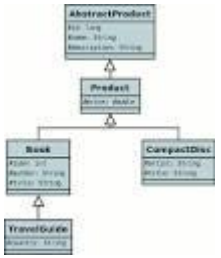
Herencia y ocultación de la información



```
public class CD extends Item {
  public void print(PrintStream out) {
    out.print("CD: " + getTitle() + " (" +
              getPlayingTime() + " mins)");
    if (getOwn())
      out.println("*");
    else
      out.println();
    out.println(" " + artist);
    out.println(" tracks: " +
                numberOfTracks);
    out.println(" " + getComment());
  }
  // Resto del código
}
```



Herencia y constructores



- En la mayoría de los lenguajes orientados a objeto se aplican las siguientes reglas
  1. Los constructores NO se heredan

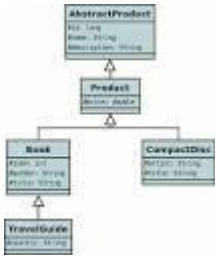
```
public class A {
  private int a;
  public A(int a) {
    this.a = a;
  }
}
public class B extends A {
  private int b;
}
```

La clase B no hereda el constructor de A

No se puede poner

```
B b = new B(3);
```

Herencia y constructores

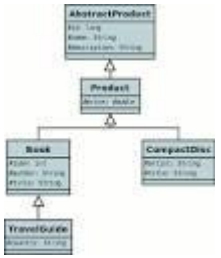


2. Los constructores de las subclases deben invocar a un constructor de la superclase.

```
public class A
{  private int a;
   public A(int a) {
      this.a = a;
    }
  }
public class B extends A
{  private int b;
   public B(int a, int b) {
     super(a);
     this.b = b;
   }
}
```

1. Primero, se invoca al constructor de la superclase con **super**
2. Segundo, se ejecuta el resto de instrucciones del cuerpo del constructor

Herencia y constructores

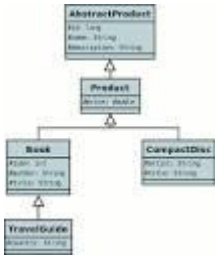


3. La llamada al constructor por defecto de la superclase es `super()` sin parámetros.

```
public class A
{  private int a;
   public A() { this.a = 1; }
    public A(int a) {
       this.a = a;
    }
 }
public class B extends A
{  private int b;
   public B(int a, int b) {
     super();
     this.b = b;
 }
```

Se realiza la llamada al constructor A() **incluso si no se escribe explícitamente**

Herencia y constructores

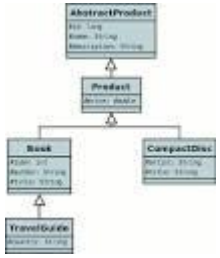


4. Si no se incluye un constructor en la clase el compilador inserta automaticaticamente un constructor por defecto sin parámetros.

```
public class A {
  private int a;
  public A() { }
}

public class B extends A {
  private int b;
  public B(int b) {
    super();
    this.b = b; }
}
```

1. Se invoca al constructor A() aunque no se escriba explícitamente.

Beneficios de la herencia



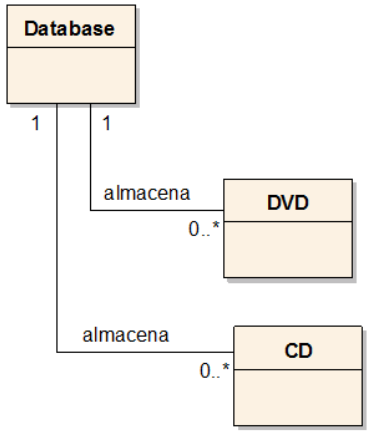
- Evita la duplicación de código
  - El código de la superclase se puede usar en todas las subclases.
- Se reutiliza código
  - Si un algoritmo funciona con un cierto tipo de clase, entonces también funcionará con sus subclases sin realizar cambios.
- Facilita el mantenimiento
  - Más fácil de aislar, corregir los errores y añadir nuevas funciones mediante la creación de nuevas subclases.
- Facilita la extensibilidad (Capacidad de ampliación)
  - Se puede partir del código existente y modificarlo para satisfacer nuevos requisitos
- **PERO** ….la herencia también tiene desventajas…
  - *¿Cuales?*
  - Se verán en otros cursos

Volviendo a la clase `DataBase`

- El método `list` de la clase `Database`

```
public void list() {
  for (CD cd : cds) {
   cd.print(System.out); System.out.println(); }
  for (DVD dvd : dvds) {
   dvd.print(System.out); System.out.println(); }
}
```

- **Está duplicado** porque CD and DVD son de diferentes tipos
- ¿Que sucede si queremos añadir más tipos de elementos?
- ¿Cómo podemos evitar la duplicación de código y añadir nuevos elementos?



Generalización / Polimorfismo

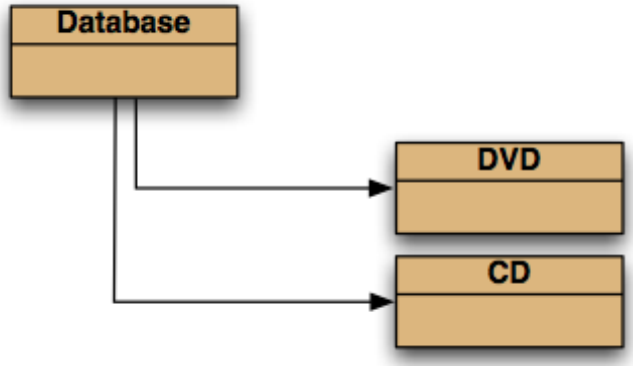
- Lo que necesitamos es la Generalización
- `Item` es una generalización de `DVD` y `CD`
- Esto significa que **`Item`** puede ser:
  - Un objeto de tipo `Item`
  - Un objeto de tipo `CD`
  - Un objeto de tipo `DVD`
  - Un objeto de otras futuras subclases de Item
- Esta característica se denomina **Polimorfismo**
  - Y la variable objeto `Item` es *polimórfica*
- En Java, **herencia** implica **polimorfismo** aunque **son dos conceptos diferentes**

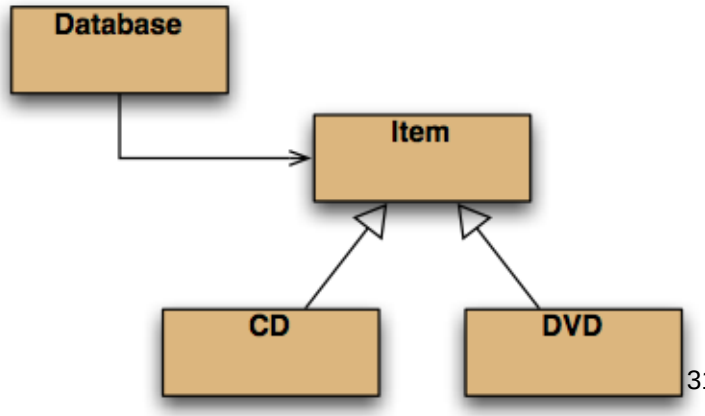
Usando polimorfismo

- Como `Item` es una generalización de `DVD` y `CD`

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

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

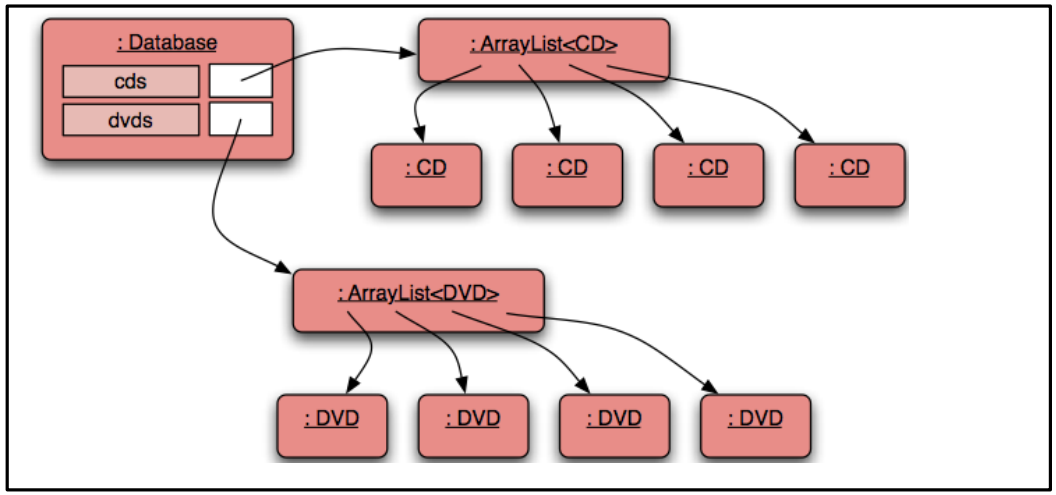
```
public class Database {

  private ArrayList<Item> items;

  public Database() {
    items = new ArrayList<Item>();
  }
```





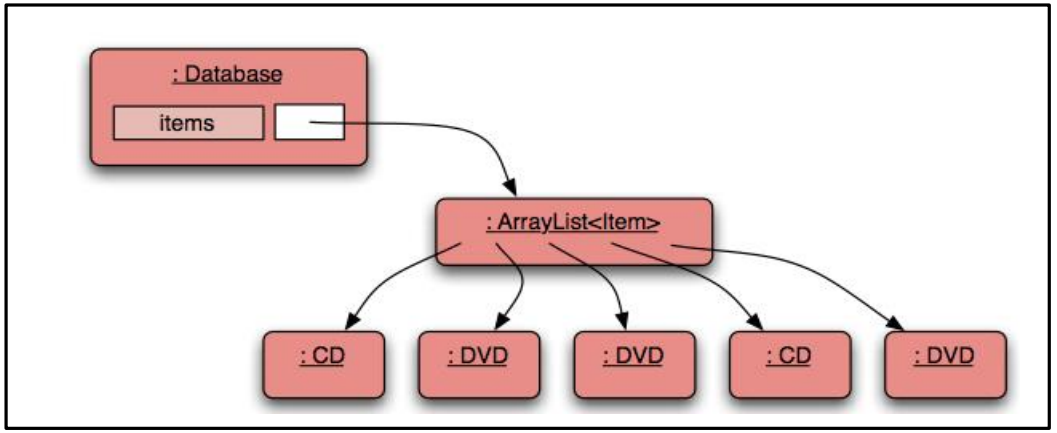
Usando polimorfismo

- Si `Item` es una generalización de `DVD` y `CD`



Se transforma en

¡Es mucho más simple!



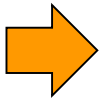
Usando polimorfismo

- Como `Item` es una generalización de `DVD` y `CD`:

```
public class Database {
  public void add(CD theCD) {
    cds.add(theCD);
  }
  public void add(DVD theDVD) {
    dvds.add(theDVD);
  }
```

```
public void list() {
    for (CD cd : cds) {
      cd.print(System.out);
      System.out.println();
    }
    for (DVD dvd : dvds) {
      dvd.print(System.out);
      System.out.println();
    }
  }
```

```
public class Database {
  public void add(Item theItem)
{
    items.add(theItem);
  }
```



Puede ser `System.out`, `System.err`

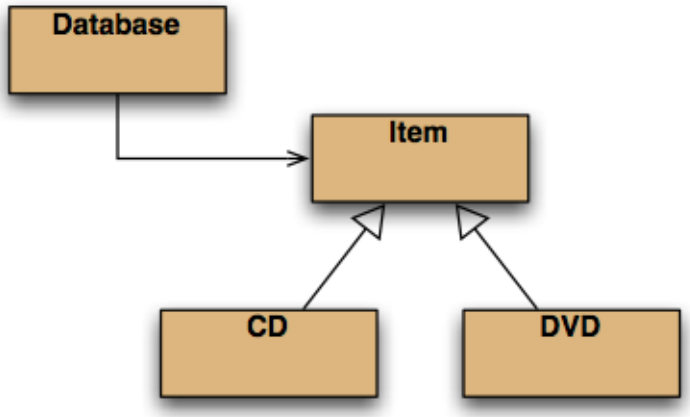
```
public void list(PrintStream out)
{
    for (Item item: items) {
      item.print(out);
      out.println();
    }
  }
```

Pregunta

- Con el nuevo diseño, ¿Qué cambios hay que realizar en el siguiente código si añadimos más items (p.ej., `VideoGame`)?

```
public class Database {

  private ArrayList<Item> items;

  public Database() {
    items = new ArrayList<Item>();
  }
```

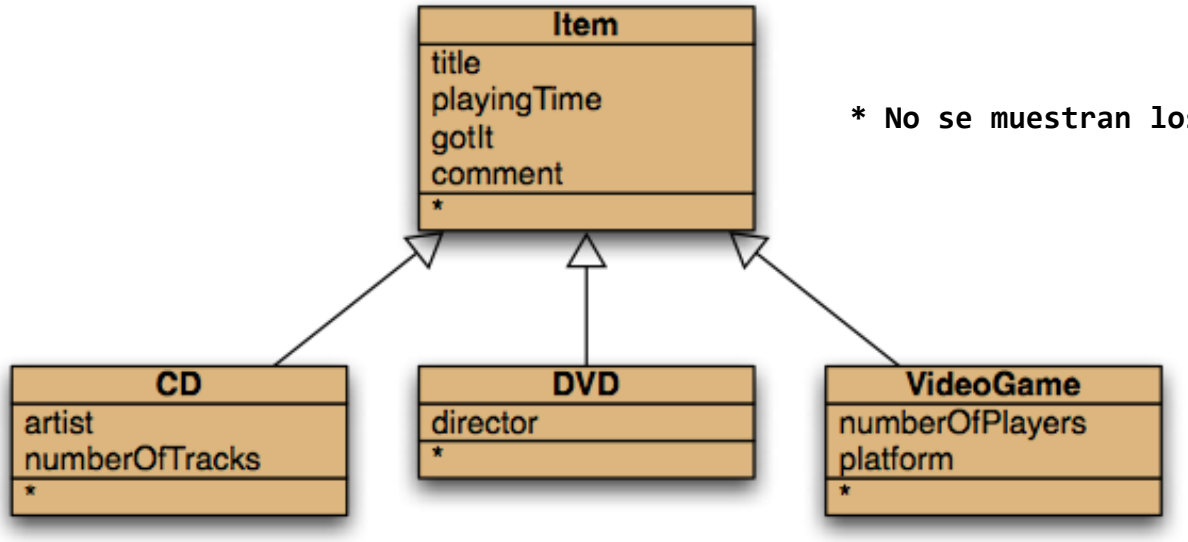
```
public class Database {
  public void add(Item theItem) {
    items.add(theItem);
  }
```




```
public void list(PrintStream out) {
    for (Item item: this.items) {
      item.print(out);
      out.println();
    }
  }
```

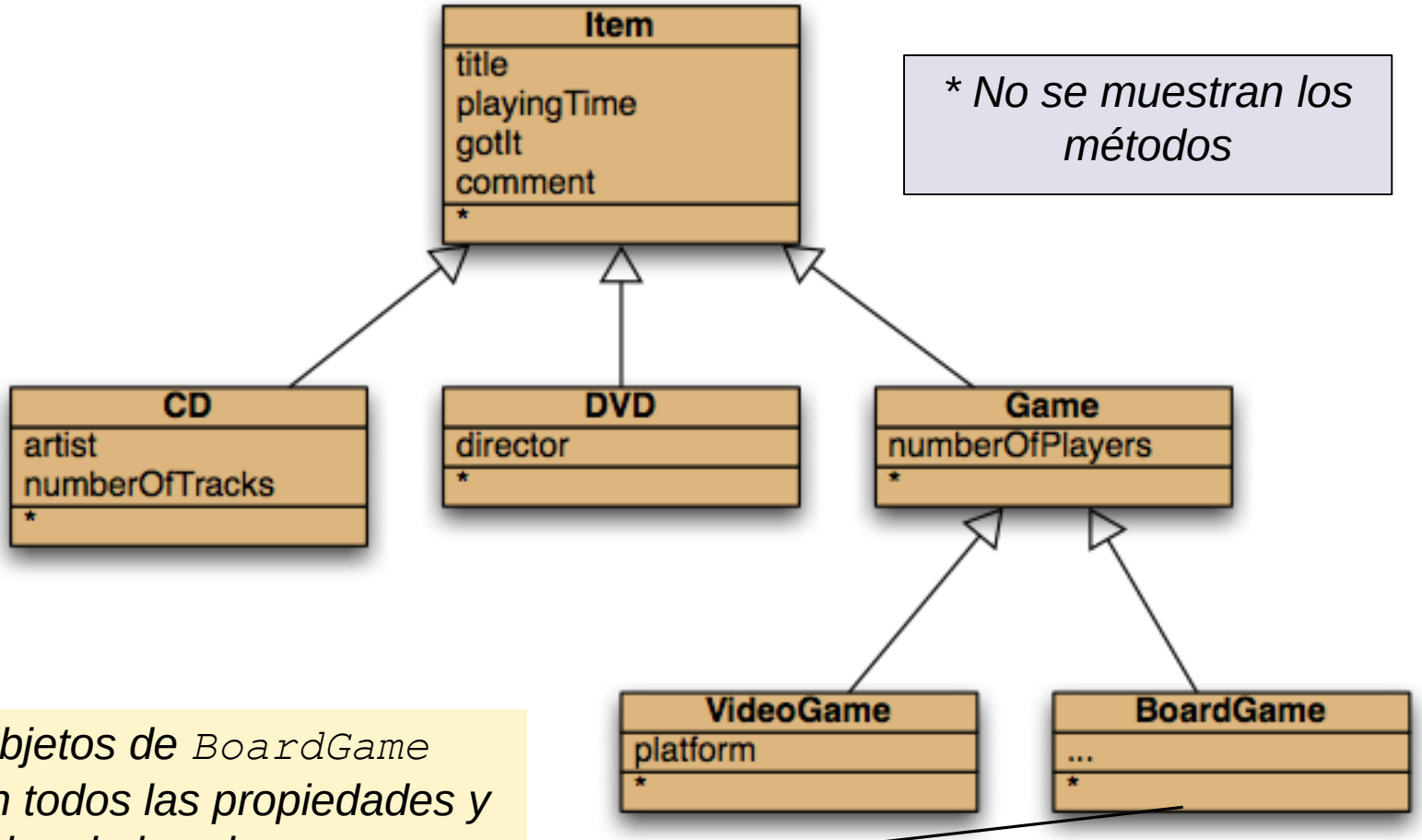
Polimorfismo = Matenibilidad

- Ningún cambio
- No es necesario cambiar **una sola línea de código**
- 100% código mantenible
- **El polimorfismo** es una herramienta esencial para crear software mantenible



* No se muestran los métodos

Añadiendo más elementos



** No se muestran los métodos*

Los objetos de `BoardGame` *tienen todos las propiedades y métodos de las clases* `Item`, `Game` *y* `BoardGame`

Subtipos

- La mayoría de los lenguajes de programación ofrecen el concepto de subtipo
- En Java, por ejemplo
  - `byte` ≤ `short` ≤ `int` ≤ `long`
  - `char` ≤ `int`
  - `float` ≤ `double`
- Se dice que
  - *byte es un subtipo de short*
  - *un byte promociona a un short*
- Con el **polimorfismo**, *una subclase es un subtipo de una superclase* (subclase ≤ superclase, `CD` ≤ `Item`, `DVD` ≤ `Item`)

Subtipos y sustitución

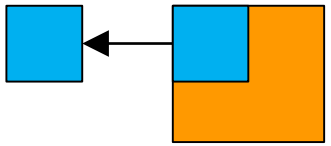
- Subtipos implican sustitución
- El *principio de sustitución de Liskov* establece que:

  Si T_1 es un subtipo de T_2 ($T_1 \leq T_2$) entonces, las variables de tipo T_2 pueden ser sustituidas con las variables de tipo T_1
- En programación, la sustitución puede ocurrir en asignaciones y paso de parámetros
  - Esto ayuda a crear código mantenible

```
Item item = new Item(…);   RIGHT!
item = new CD(…);          RIGHT!
item = new DVD(…);         RIGHT!
CD cd = new Item(…);       WRONG!
DVD dvd = new CD(…);       WRONG!
```

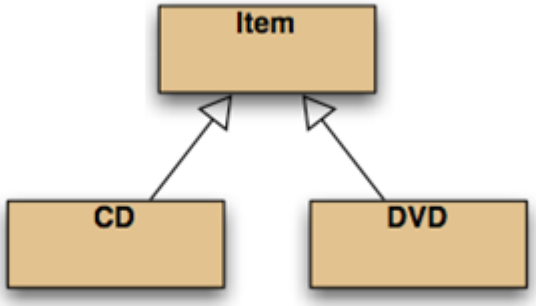
```
Database db = new Database();
db.add(new Item(…));       RIGHT!
db.add(new CD(…));         RIGHT!
db.add(new DVD(…));        RIGHT!
db.add(new Troll());       WRONG!
```

(Up – Down) Cast



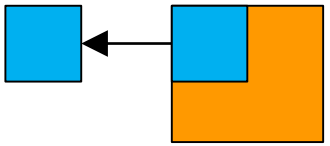
- En el árbol de herencia, la promoción se realiza **hacia arriba**

```
Item item = new CD(…);
item = new DVD(…);
```



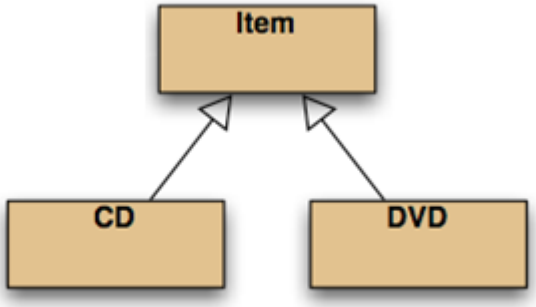
- Y esto es así porque el polimorfismo representa generalización
  - Como todos los objetos `CD` y `DVD` son también de tipo `Item`, **es seguro** realizar la sustitución
  - Por lo tanto, *up-casting* es correcto

(Up – Down) Cast



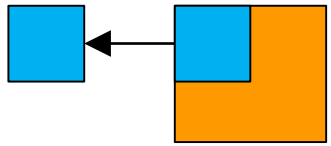
- El siguiente código da un error de compilación

```
void m(Item item) {
  DVD dvd = item;
}
```



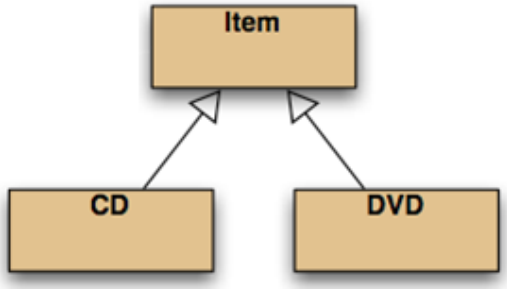
- Esto es debido a que la variable `item` puede ser un `DVD`, un `CD`, o una `VideoGame`…
  - Y por ejemplo, no tendría sentido llamar al método `getDirector` con un objeto `cd` así (`cd.getDirector()`)
  - Por lo tanto, down-casting no está permitido implícitamente

(Up – Down) Cast



- ¿Es posible realizar down-cast con un casting?

```
void m(Item item) {
  DVD dvd = (DVD)item;
}
```



- ¿Que ocurre si la variable `item` es un `CD`, o un `VideoGame`…?
  - Se produce un error (excepción) en tiempo de ejecución
    - ¿Se puede hacer down-cast? **SI**
    - ¿Se puede romper la ejecución, si el tipo no es apropiado? **SI**
    - ¿Sucede lo mismo que con up-casting? **NO**

- Si se necesita la conversión, es mejor usar el operador **`instanceof`** para hacer el código más seguro

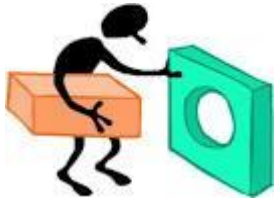
```
void m(Item item) {
  if (item instanceof DVD) {
    DVD dvd = (DVD)item;
  }
}
```

Coleccionando objetos

- En la clase `Database`, la colección `ArrayList<Item>` almacena cualquier `Item`, `DVD`, `CD`, `VideoGame`…
- ¿Es posible coleccionar cualquier objeto? **NO**
- Object es una clase de la librería estándar de Java que sirve como superclase para todos los objetos.
  - Algunos métodos disponibles automáticamente para todos los objetos existente son: **toString**, **equals**, **hashCode**,…
- De esta forma, un `ArrayList<Object>` (o simplemente `ArrayList`) puede coleccionar cualquier objeto.

```
public void add(Object object)
public Object get(int index)
public void set(int index, Object object)
…
```

Colecciones y tipos primitivos



- Todos los objetos se pueden almacenar en las colecciones …
  - ... porque las colecciones aceptan elementos de tipo `Object` ...
  - ... y todas las clases son subtipos del tipo `Object`.

- ¡Muy bien! ¿Pero qué pasa con los tipos simples (o tipos primitivos)? `boolean`, `char`, `byte`, `short`, `int`, `long`, `float` and `double`

Clases envoltorio

- Los tipos primitivos no son objetos. Se deben transformar en objetos.
- En Java 1.5 se añadió **Autoboxing**
  - Esta técnica convierte automáticamente un tipo primitivo en un objeto, y al revés.
- Las **clases envoltorio** (**wrapper)** están en el paquete `java.lang` y existen para todos los tipos simples

Tipo primitivo	Wrapper Class
boolean	Boolean
char	Character
byte	Byte
short	Short

Tipo primitivo	Wrapper Class
int	Integer
long	Long
float	Float
double	Double

Autoboxing = Boxing + Unboxing

- **Autoboxing** es una técnica que convierte automáticamente un tipo primitivo en un objeto y al revés.
  - **Boxing**: Un tipo primitivo es convertido automáticamente en un objeto (wrapper object)
  - **Unboxing**: Un objeto (wrapper object) es convertido automáticamente en un tipo primitivo.

Autoboxing, boxing and unboxing

Ejemplo

```
public static Integer m(Integer i) {
  return i;
}
public static void main(String... args)
{
  Character ch = 'a'; // boxing
  char myChar = ch;   // unboxing

  int n =                // unboxing
          m(3);          // boxing
}
```

Revisión

- La herencia permite la definición de nuevas clases como extensión de otras clases ya existentes.
- La herencia
  - *Evita la* ***duplicación de código***
  - *Permite* ***reutilizar el código***
  - *Mejora el* ***mantenimiento***
  - *Mejora la* ***extensibilidad***
- El **polimorfismo** ofrece **generalización**, lo que permite la sustitución de una superclase por una subclase.
- El **polimorfismo** es una herramienta fundamental para el **mantenimiento** del software.