3. Programación Orientada a Objetos (POO) Avanzado

3.1. Herencia de clases

Programación Modular y Orientación a Objetos

Felipe Ibañez y Juan Miguel Lopez
felipe.anfurrutia@ehu.es
juanmiguel.lopez@ehu.es
Dpto. de Lenguajes y Sistemas Informáticos
UPV/EHU

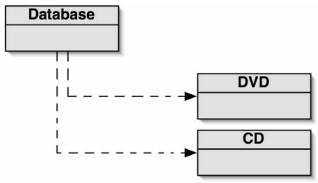
Basado en las transparencias de Juan Pavón Mestras
Dpto. de Ingeniería e Inteligencia Artificial
Universidad Complutense de Madrid

Contenido

- Motivación
- Herencia de clases
  - Definición
  - Superclase-subclase
  - Subtipos
  - Polimorfismo: variables polimórficas
  - Construcciones del lenguaje Java:
    - extends
    - super
    - enmascaramiento (*casting*)
    - la clase Object
    - clases envoltorio (wrappers) de tipos

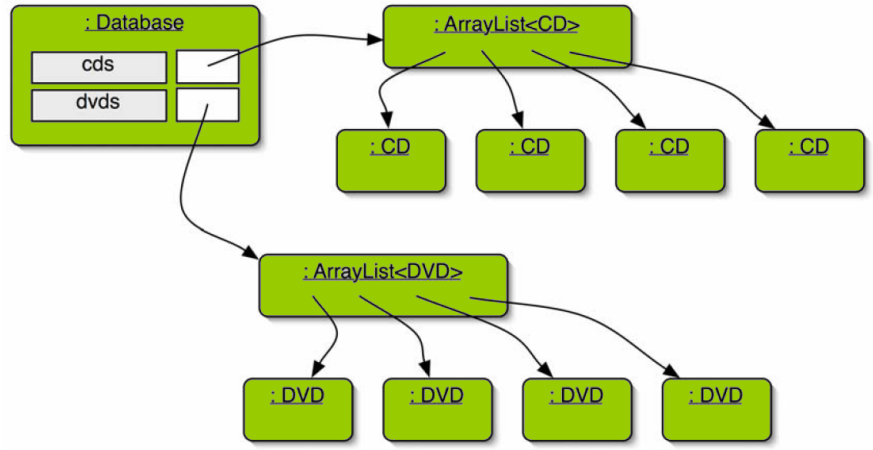
Motivación

- El ejemplo DoME: "Database of Multimedia Entertainment"
- Aplicación que permite gestionar información sobre discos de música (CD) y películas (DVD)
  - CD: title, artist, # tracks, playing time, got-it, comment
  - DVD: title, director, playing time, got-it, comment
  - Y permite buscar información y sacar listados
- Proyecto en chapter08/dome-v1 y dome-v2

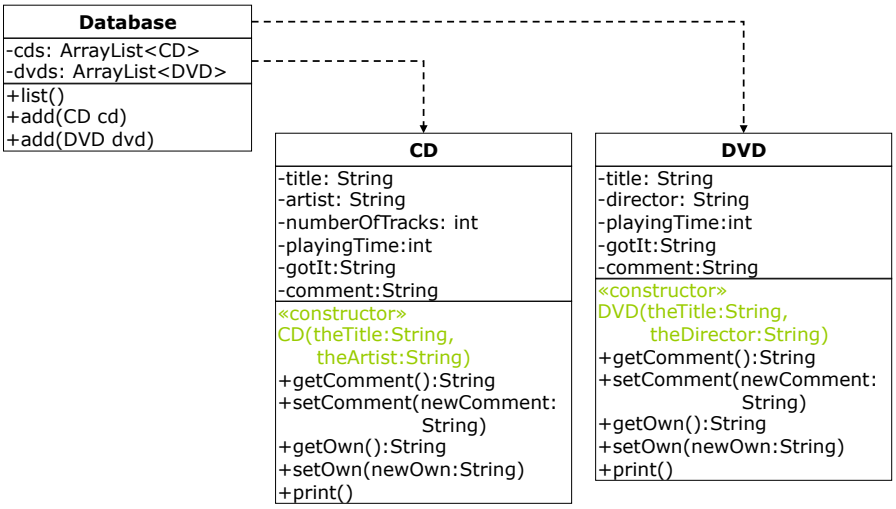


Modelo de objetos de DoME v1

- 2 listas: una de dvds y otra de cds



Modelo de clases de DoME



Código fuente de las clases CD y DVD

```
public class CD {
  private String title;
  private String artist;
  private String comment;
  public CD(String theTitle,
                String theArtist){
    title = theTitle;
    artist = theArtist;
    comment = " ";
  }

  public void setComment(String
                         newComment)
  { ... }

  public String getComment()
  { ... }

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

```
public class DVD {
  private String title;
  private String director;
  private String comment;
  public DVD(String theTitle,
                String theDirector){
    title = theTitle;
    director = theDirector;
    comment = " ";
  }

  public void setComment(String
                         newComment)
  { ... }

  public String getComment()
  { ... }

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

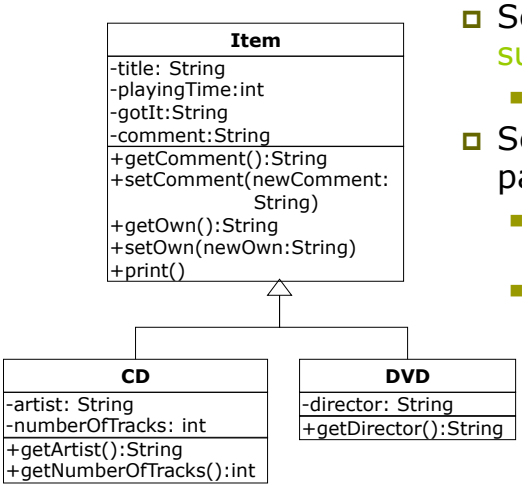
Código fuente de la clase de la base de datos

```
class Database {
   private ArrayList<CD> cds;
   private ArrayList<DVD> dvds;
   ...
  public void list() {
     for(CD cd : cds) {
        cd.print();
        System.out.println();  // empty line between items
     }

     for(DVD dvd : dvds) {
        dvd.print();
        System.out.println();  // empty line between items
     }
  }
}
```

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

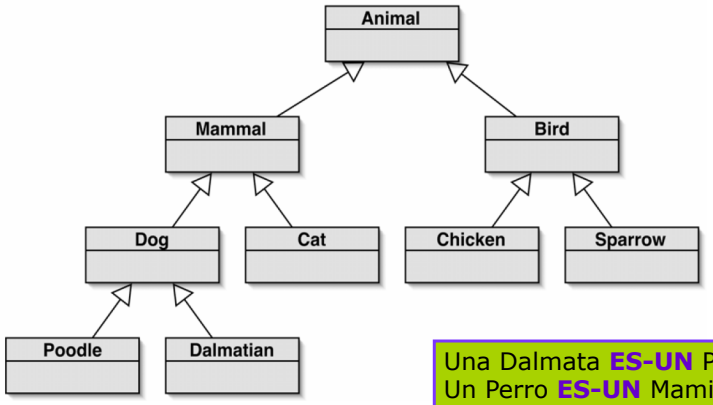
Uso de la herencia



- Se define una superclase: **Item**
  - define atributos comunes
- Se definen subclases para CD y DVD
  - heredan los atributos de la superclase
  - pueden tener sus propios atributos (características)

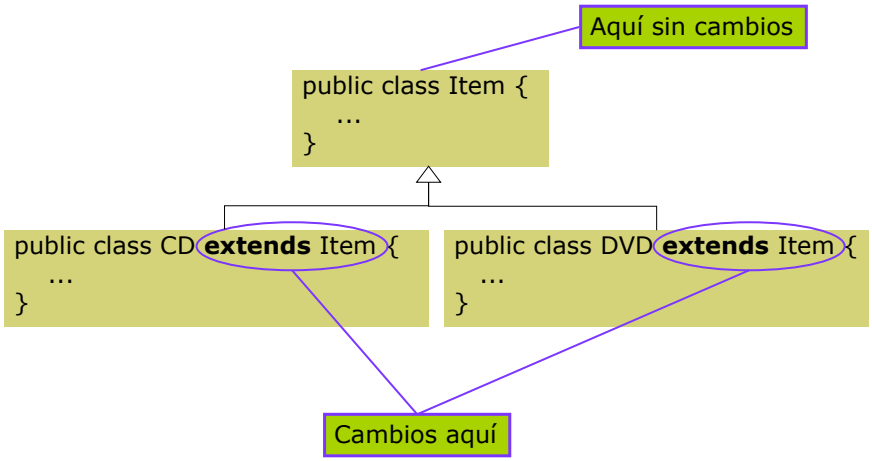
Jerarquías de herencia

- La relación "**ES-UN**"

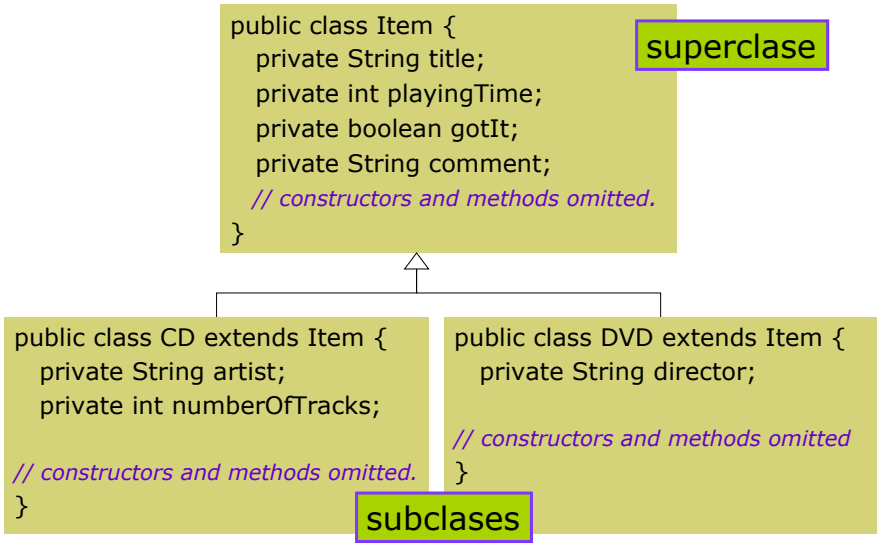


Una Dalmata **ES-UN** Perro
Un Perro **ES-UN** Mamifero
Un Mamifero **ES-UN** Animal

La herencia en Java: *extends*



La superclase y las subclases



Herencia y constructores

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

   /**
    * Initialise the fields of the item.
    */
   public Item(String theTitle, int time) {
      title = theTitle;
      playingTime = time;
      gotIt = false;
      comment = "";
   }

   // methods omitted
}
```

mismo nombre que la clase

Método constructor,
Inicializa los atributos

Herencia y constructores (2)

```
public class CD extends Item {
   private String artist;
   private int numberOfTracks;

   /**
    * Constructor for objects of class CD
    */
   public CD(String theTitle, String theArtist,
             int tracks, int time){
      super(theTitle, time);
      artist = theArtist;
      numberOfTracks = tracks;
   }

   // methods omitted
}
```

Llamada al constructor de la superclase, parecido al this

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

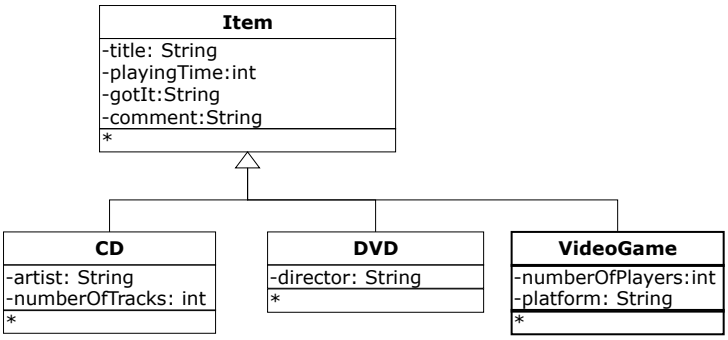
Llamadas a métodos de la superclase

```
// En class Item:
public String toString() {
    return title + "(" + comment + ")";
}
```

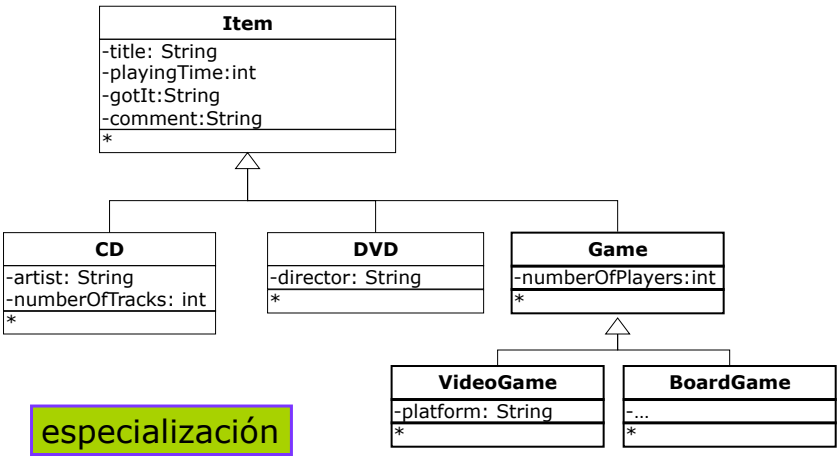
```
// En class CD:
public String toString(){
return artist + ": " + super.toString();
}
```

```
// o bien:
public String toString() {
return artist + ": " + getTitle();
}
```

Se pueden añadir nuevos tipos de item



Y definir jerarquías más profundas



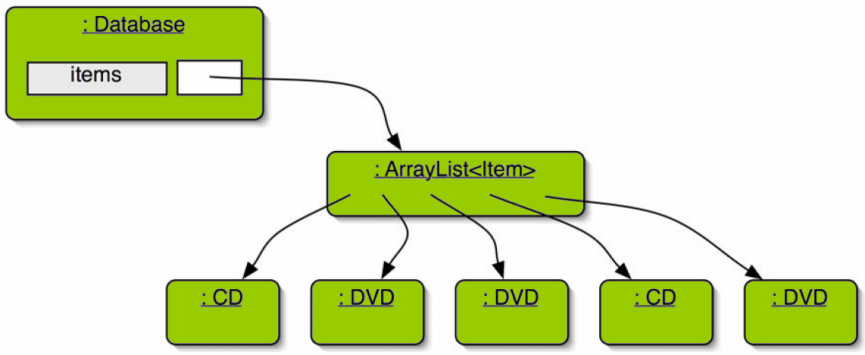
En resumen…

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

Nuevo código de la base de datos

```
public class Database {
    private ArrayList<Item> items;

    /**
     * Construct an empty Database.
     */
    public Database() {
        items = new ArrayList<Item>();
    }

    /**
     * Add an item to the database.
     * @param theItem The item to be added.
     */
    public void add(Item theItem) {
        items.add(theItem);
    }
   ...
}
```

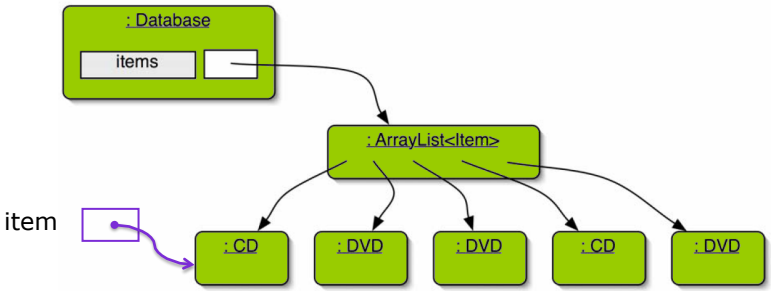
Diagrama de objetos (DoME v2)



Nuevo código de la base de datos

```
public class Database {
    ...
    /**
     * Print a list of all currently stored items to the
     * text terminal.
     */
    public void list()    {
        for(Item item : items) {
            item.print();
            System.out.println();   // empty line between items
        }
    }
    ...
}
```

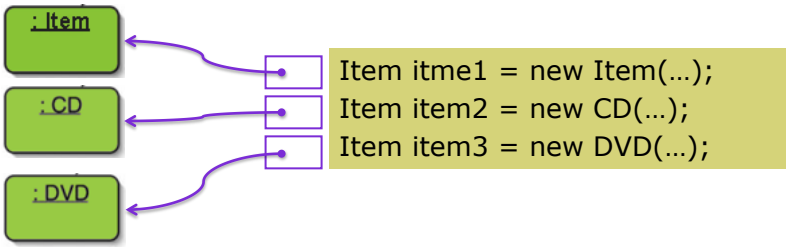
Nuevo código de la base de datos (2)



- Problema: Cómo es posible que una variable de tipo Item referencie un objeto de tipo CD o DVD?

Subclases y subtipos

- Las clases definen tipos
- Las subclases definen subtipos
- Los objetos de los subtipos se pueden usar como objetos de los supertipos
  - A esto se le llama **sustitución**
  - Ejemplo: los objetos de las subclases se pueden asignar a variables de la superclase



Subclases y subtipos (2)

- Al principio teníamos:
  public void add(CD theCD)
  public void add(DVD theDVD)

2 métodos con distintos tipos

- Luego hemos cambiado:
  public void add(Item theItem)

Un único método con un supertipo

- Que se puede utilizar de esta forma:
  DVD myDVD = new DVD(…);
  database.add(myDVD);

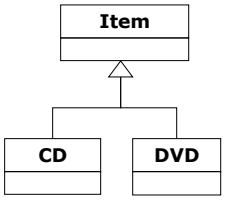
Instanciar un objeto de un subtipo y utilizar el método anterior

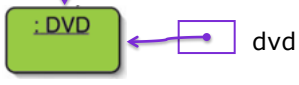
Subtipos y paso de parámetros

- De la misma manera que con la asignación, también se pueden usar subtipos para pasar como parámetros en métodos que tienen definidos parámetros de la superclase

```
public class Database {
    public void add(Item theItem) {
        …
    }
}
```

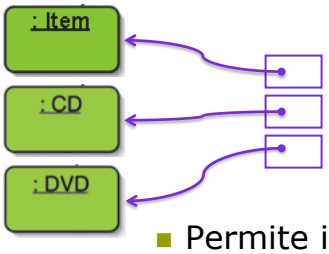
```
Database database = new Database();

DVD dvd = new DVD(…);
CD cd = new CD(…);

database.add(dvd);
database.add(cd);
```





Polimorfismo: variables polimórficas

- En Java, las variables de tipo referencia a objeto son polimórficas
  - Pueden referenciar objetos de distintos tipos
  - Los objetos deben ser del tipo declarado o sus subtipos




```
Item item1 = new Item(…);
Item item2 = new CD(…);
Item item3 = new DVD(…);
```

  - Permite invocar: `item2.print();`
  - Pero **NO** se puede invocar:

```
item2.getArtist(); //error de compilación
```

Enmascaramiento de tipos (*casting*)

- Se puede asignar un valor de subtipo a una variable de supertipo
  - ¡Pero no al revés!

```
CD cd = new CD(...);
item2 = cd;    //correcto;
cd = item2; ¡Error en tiempo de compilación!
```

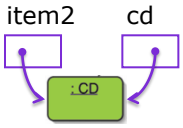
- Aunque si fuera necesario se puede hacer con la técnica de casting

```
cd = (CD) item2;
```

  - Pero sólo si *item2* es realmente un *CD*
  - *Ahora sí se puede invocar* *cd.getArtist();*

Enmascaramiento de tipos (*casting*)

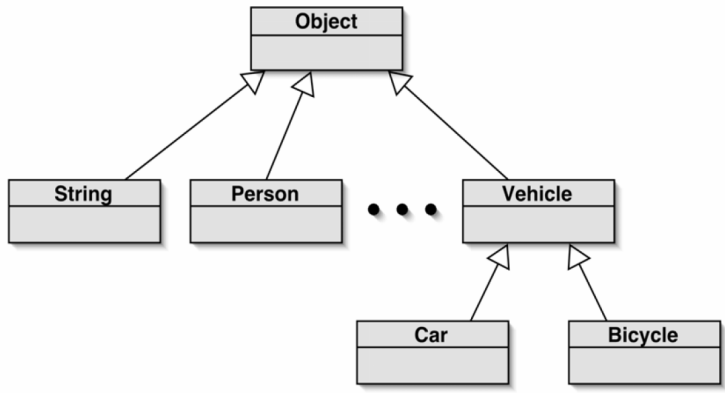
- Sintaxis:
  - Se indican el tipo del enmascaramiento entre paréntesis
    CD cd = (CD) item2;
  - El compilador comprueba si el objeto es del mismo tipo o subtipo
- En tiempo de ejecución:
  - El objeto no cambia en nada, es el mismo
  - Simplemente se permite usar la referencia adecuadamente
  - El interprete comprueba que el objeto es realmente de ese tipo
  - si no lo es, se recibe una excepción: **ClassCastException**
  - Para sortear el posible problema, se puede comprobar por código:
    if (var **instanceof** T) // si el objeto var pertenece a la
    // clase T o a una de sus subclases
  - Para el ejemplo anterior:
    if (item2 instanceof CD)
    CD cd = (CD) item2; //nunca va a fallar



- *Úsese con moderación*

La clase Object (la superclase)

- En Java, todas las clases heredan de la clase *java.lang.Object*



Las colecciones en Java son polimórficas

- Antes de tener tipos genéricos se definieron las colecciones como polimórficas
  - Operaciones del tipo:

    public void add(Object element)

    public Object get(int index)

  - permiten trabajar con cualquier tipo de objeto
    - Ya que todas las clases heredan de la clase Object

Clase Object

- public final Class<?> **getClass**()
  - Devuelve la clase del objeto
  - **getName()** sobre el objeto Class devuelve un String con el nombre de la clase
- public int **hashCode**()
  - Devuelve el valor hash code del objeto (identificador único)
- public String **toString**()
  - Devuelve la representación textual como String del objeto
  - Se recomienda que todas las clases redefinan este método
  - Por defecto, devuelve el siguiente texto:

    getClass().getName() + '@' + Integer.toHexString(hashCode())

Clase Object (2)

- public boolean **equals**(Object obj)
  - Comprueba si dos objetos son iguales
- protected Object **clone**() throws CloneNotSupportedException
  - Crea y devuelve una copia del objeto
  - La clase debe implementar la interfaz **Cloneable**
  - x.clone() != x
  - Shallow copy vs. Deep copy

Clase Object – Método finalize()

- **protected void finalize() throws Throwable**
  - Método invocado por el recogedor de basura (*garbage collector*) cuando no hay referencias al objeto y se va a eliminar
  - Sirve para hacer operaciones de limpieza y liberar recursos asociados al objeto
  - El método en la clase Object no realiza ninguna operación

Clases envoltorio (*wrapper*)

- Si se quieren utilizar los tipos primitivos (int, boolean, etc.) donde valga un Object, ¿cómo hacerlo?
- La respuesta es un conjunto de clases envoltorio (*wrappers*) que envuelven la variable

tipo simple	*clase wrapper*
int	Integer
float	Float
char	Character
boolean	Boolean
byte	Byte
…	…

Clases envoltorio (*wrapper*)

```
int i = 18;
Integer iwrap = new Integer(i);
…
int value = iwrap.intValue();
```

Envuelve el valor

Lo desenvuelve

Autoboxing y unboxing

- Aunque en ocasiones donde se espera un Object el compilador se encarga de hacer la conversión automática

```
private ArrayList<Integer> markList;
…
public void storeMark(int mark)
{
    markList.add(mark);
}
```

autoboxing

```
int firstMark = markList.remove(0);
```

unboxing

Autoboxing y unboxing (2)

- Aunque en ocasiones donde se espera un Object el compilador se encarga de hacer la conversión automática

- En el ejemplo anterior de Autoboxing:
  - La instrucción: markList.add(mark)
  - Lo convierte en: markList.add(new Integer(mark));

- En el ejemplo anterior de Unboxing:
  - La instrucción: int firstMark = markList.remove(0);
  - La convierte en:
    int firstMark = ((Integer) markList.remove(0)).intValue();