Unidad 4: Mas técnicas de abstracción

Metodología de la Programación

Curso 2021-2022

© *Candi Luengo Díez , Francisco Ortín Soler y José Manuel Redondo López, Alberto M. Fernandez Álvarez*

Bibliografía

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

  Capítulo 12: Técnicas de abstracción adicionales

En este capítulo veremos…

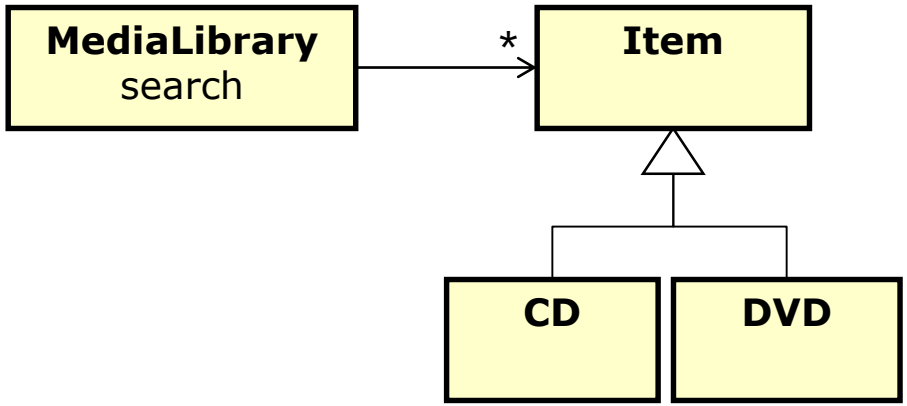
- **Actualizar nuestro Proyecto DoME** con nuevas funciones múltiples.
- Poner en práctica el **polimorfismo** con un problema real.
- Discutir la **igualdad de objetos.**
- Aprender los conceptos de **clase abstracta** y **método abstracto.**
- Conocer como **imprimir** el contenido de nuestros objetos.
- Aprender más sobre **herencia** e **interfaces**

Principales conceptos

- Métodos abstractos
- Clases abstractas
- La clase Object
- Herencia múltiple
- Interfaces

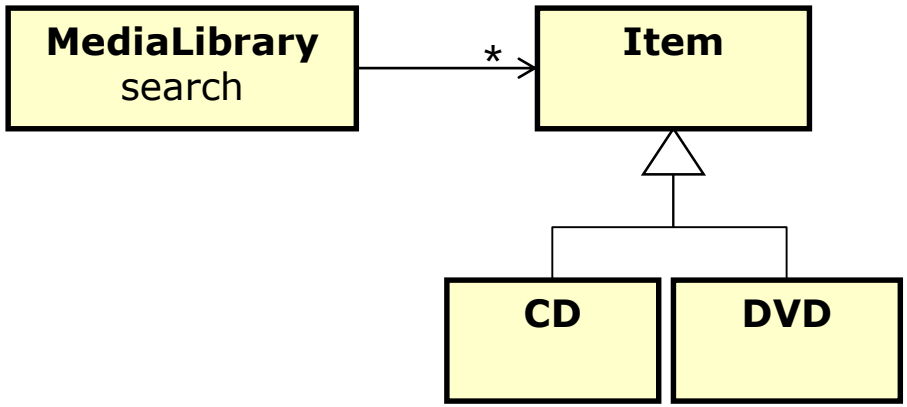
Volviendo al proyecto DoME

- En la aplicación DoME, queremos que la clase **`MediaLibrary`** ofrezca un método de búsqueda llamado **`search`**
- Este método `search` debe buscar los objetos CD, DVD,… que existen en el catálogo (**`MediaLibrary`**)
- Pregunta: ¿Como sería la declaración de este método?



Un nuevo método search

```
public class MediaLibrary {
  public ? search(?) {
        // ...
  }
  // ...
}
```



- ¿Que queremos buscar?
  - CDs, DVDs, Videogames…
- ¿Que información describe cada item?
  - `title` y `artist` para `CD`
  - `title` y `director` para `DVD`…
  - … y, ¿para futuros items?

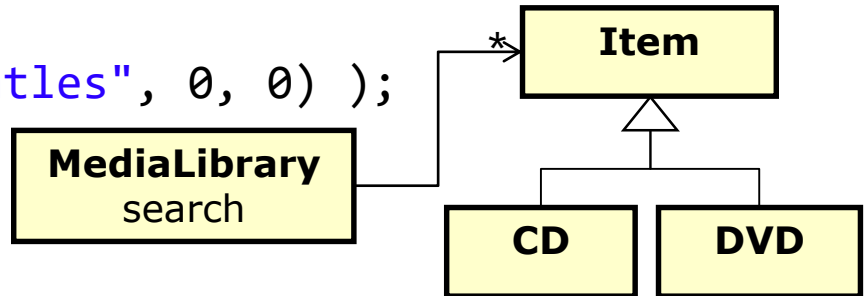
Un nuevo método search

- **Solución**: Pasar un `Item` con el estado necesario (atributos) para identificarlo
- Y devolver un objeto con toda la información que estábamos buscando (p.e., `playingTime`, `gotIt`, `comment`, y `numberOfTracks`)

```
public Item search(Item item)
```

- Funciona para cualquier tipo de `Item` (y futuros)

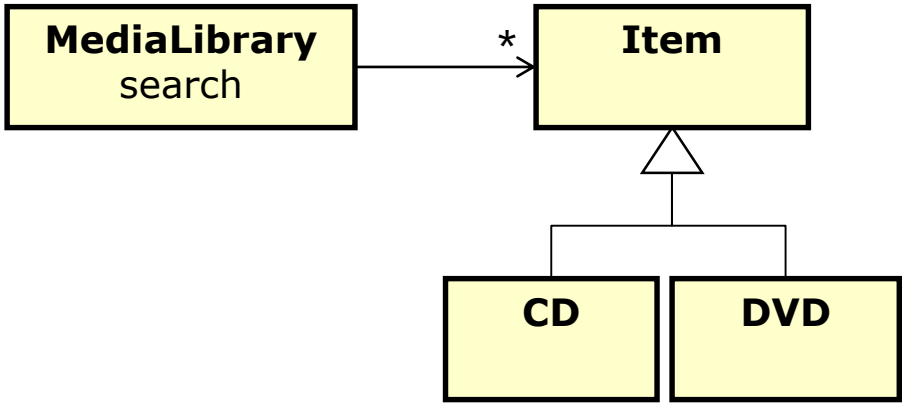
```
MediaLibrary ml = new MediaLibrary();
ml.add(new CD("Yesterday", “The Beatles", 0, 0) );)
// ...
CD yesterday = (CD)ml.search(
    new CD("Yesterday", “The Beatles", 0, 0) );
```



Implementación

- Como podemos implementar el método `search`?

```
public class MediaLibrary {
  public Item search(Item item) {
        // ...
  }
  // ...
}
```

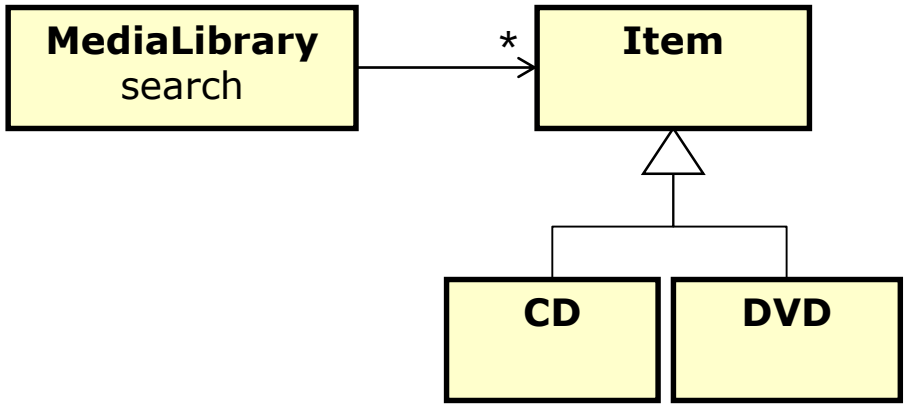


Implementación

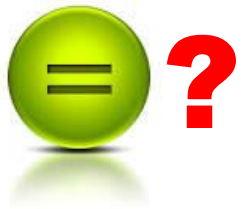
```
public class MediaLibrary {
 private ArrayList<Item> items;

  public Item search(Item item) {
     for(Item i: items)
        if ( ?  )
           return i;
    return null;
  }
  // ...
}
```

Necesitamos un método para saber si dos objetos son iguales o no

¿Dónde ponemos ese método?

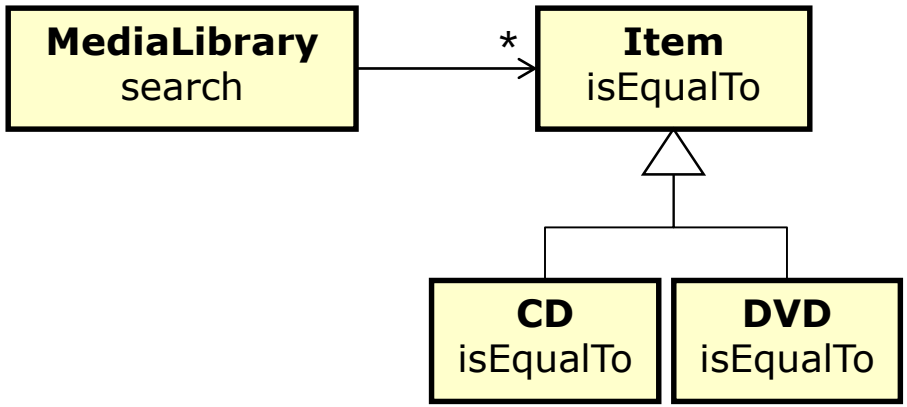


Método polimórfico

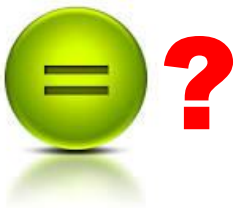


- Añadimos el método `isEqualTo` a `Item` para que sea polimórfico
  - Usando enlace dinámico, especificamos la forma en que cada elemento se compara : `CD` (`title` y `artist`), `DVD` (`title` y `director`)…
- Entonces, podemos escribir el siguiente código:

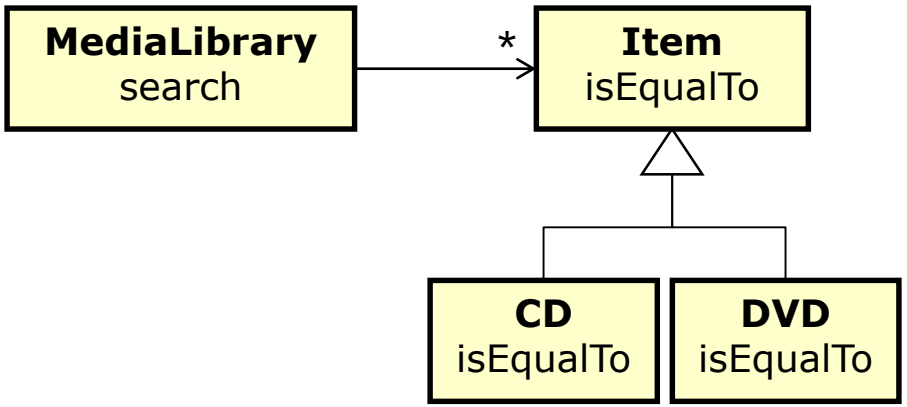
```
public class Database {
 private ArrayList<Item> items;
  public Item search(Item item) {
    for(Item i: items)
      if (i.isEqualTo(item))
        return i;
    return null;
   }
  // ...
}
```



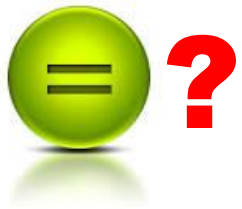
Método polimórfico



- ¿Como declaramos el método `isEqualTo` en `Item`?
- ¿Como lo implementamos en CD?
- ¿Como lo implementamos en DVD?

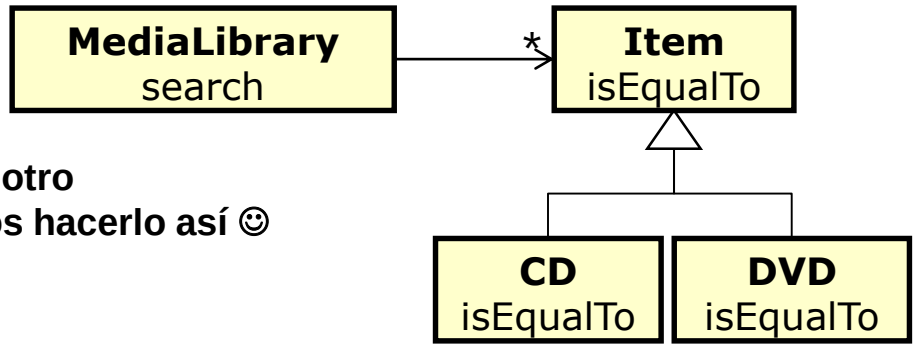


Método polimórfico

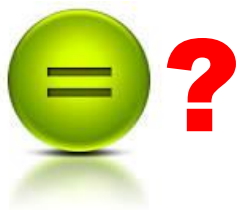


```
public class CD extends Item {
  @Override
  public boolean isEqualTo(Item item) {
   if (this == item) //igualdad de referencias
       return true;
    if (!(item instanceof CD))
      return false;
    CD cd = (CD)item; //acceder a los atributos del otro item
    return cd.getTitle().equals(this.getTitle()) &&
           cd.getArtist().equals(this.getArtist());
  }
  //...
}
```

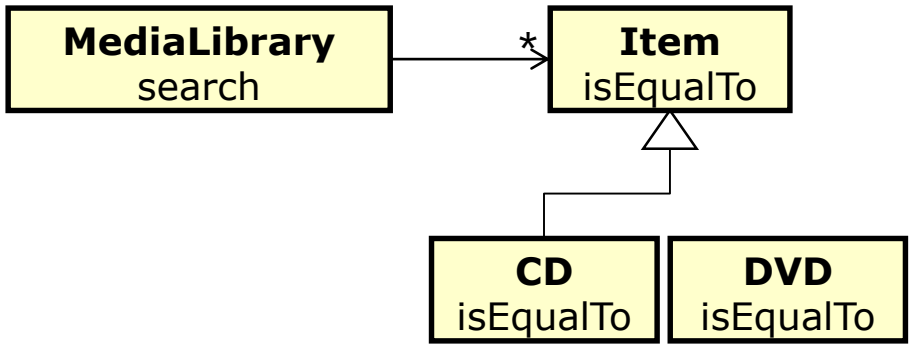
¿Por qué buscamos por title y artist y no por otro criterio?. Porque en este programa decidimos hacerlo así ☺



Método polimórfico

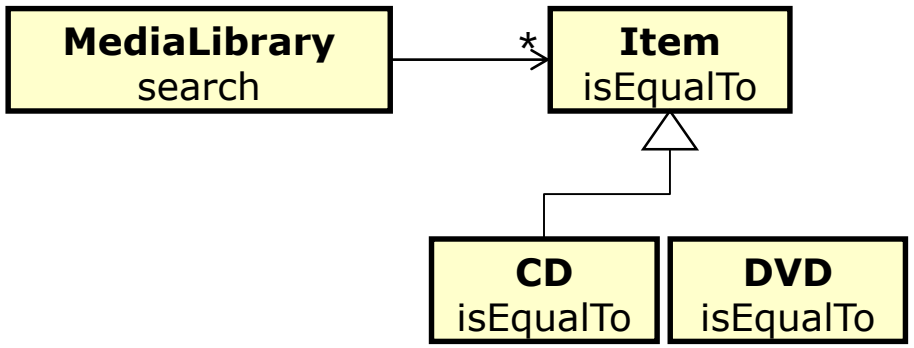


```
public class DVD extends Item {
  @Override
  public boolean isEqualTo(Item item) {
    if (this == item) //igualdad de referencias
       return true;
    if (!(item instanceof DVD))
      return false;
    DVD dvd = (DVD)item; //acceder a los atributos del otro item
    return dvd.getTitle().equals(this.getTitle()) &&
           dvd.getDirector().equals(this.getDirector());
  }
  //...
}
```

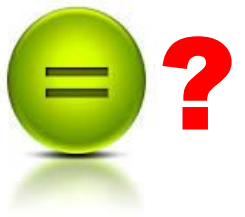


Polimorfismo + Enlace Dinámico

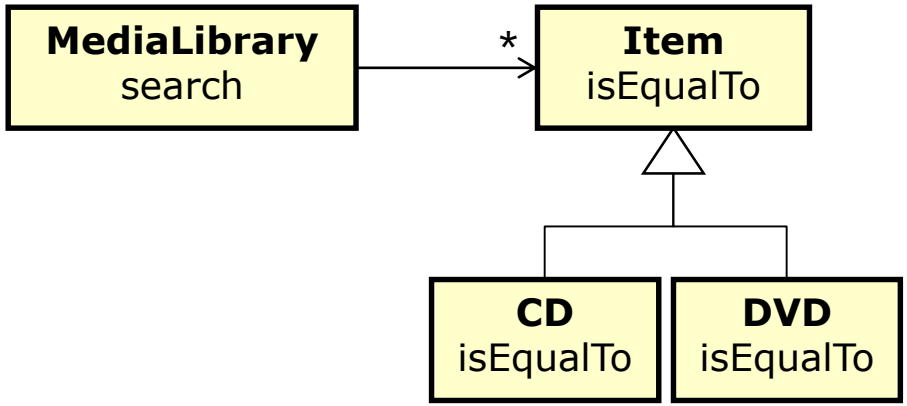
- Se han combinado los beneficios del Polimorfismo y el Enlace Dinámico
  - La generalización de `CD` y `DVD` con `Item` (polimorfismo) implica la implementación de un método de búsqueda único
  - La especialización de `isEqualTo` (enlace dinámico) implica un comportamiento heterogéneo
- Este código **no tiene que modificarse** cuando se añadan nuevos elementos



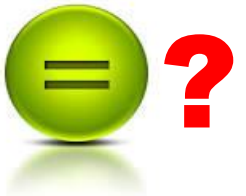
El método `isEqualTo` en `Item`



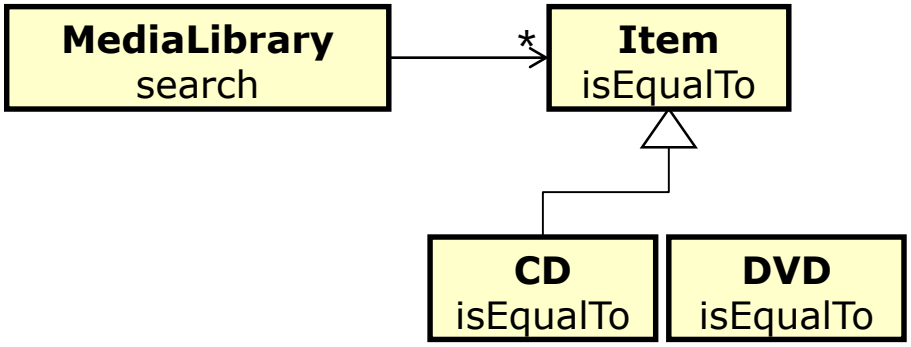
- ¿Como implementamos el método `isEqualTo` en `Item`?
  - ¿Cómo podemos comparar algo abstracto con otro algo también abstracto?
  - ¿Podemos comparar algo como esto en este programa?



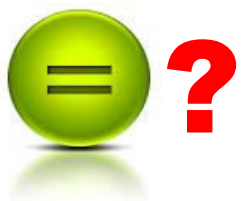
El método `isEqualTo` en `Item`



- No tiene sentido (en este programa)
- No existe una implementación correcta (en este programa)
- No hay una regla fija para determinar que ...
- ¡hay que pensar!

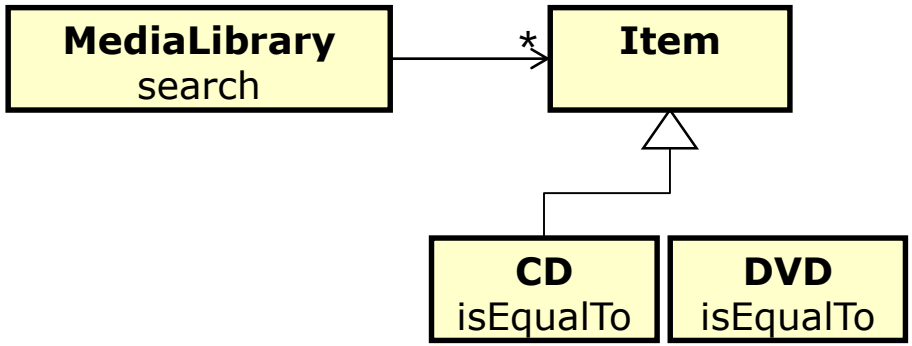


El método `isEqualTo` en `Item`



- Entonces, lo deberíamos quitar de `Item`
- Pero, … si lo hacemos, el método `search` será rechazado por el compilador…
  El método `isEqualTo(Item)` está indefinido para el tipo `Item`

```
public class MediaLibrary {
  private ArrayList<Item> items;
  public Item search(Item item) {
    for(Item i:items)
      if (i.isEqualTo(item))
        return i;
    return null;
   }
  // ...
}
```

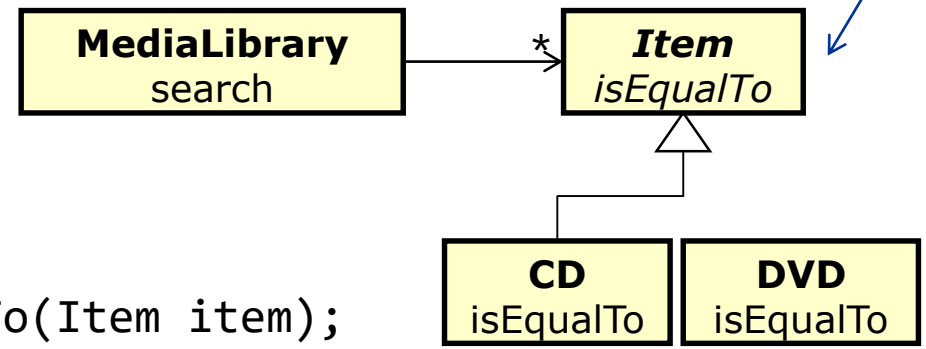


Métodos abstractos

- Cuando utilizamos polimorfismo, es muy común que:
  1. Haya que declarar un método
  2. No exista una implementación coherente
- Realmente necesitamos un mensaje
  - Los objetos se comunican enviando mensajes entre ellos
  - Los mensajes se implementan mediante métodos
  - Algunos mensajes no son implementados en los métodos (métodos abstractos)
  - Esos métodos (abstractos) están destinados a ser especializados

Itálica significa abstract en UML

- Java resuelve este problema declarando `isEqualTo` como **abstract** en `Item`
- Los métodos abstractos no tienen implementación




```
public class Item {
  public abstract boolean isEqualTo(Item item);
  // …
}
```

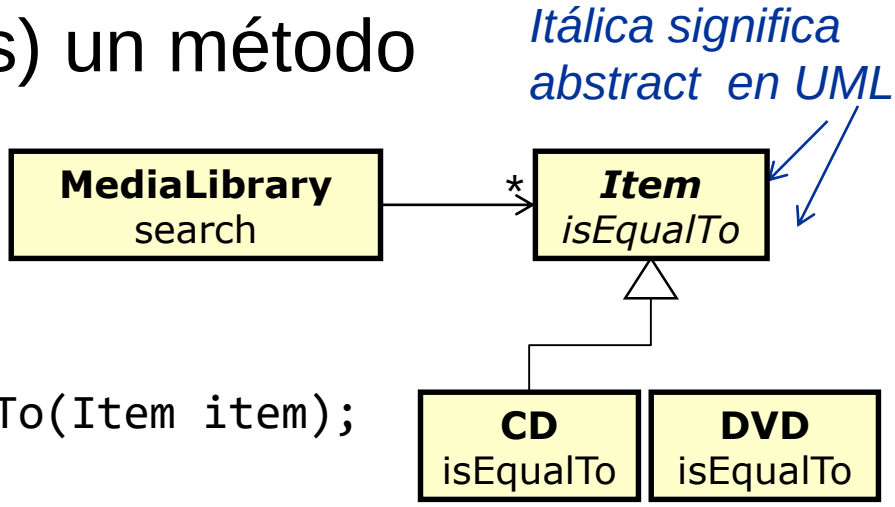
Clases abstractas

- Si se declara el método `isEqualTo` como abstracto en `Item`, el compilador avisa del siguiente error

  "*The type Item must be an abstract class to define abstract methods*"

- Es necesario declarar a `Item` también como **clase abstracta**

- Una clase con (al menos) un método abstracto, es una clase abstracta




```
public abstract class Item {
  public abstract boolean isEqualTo(Item item);
  // ...
}
```

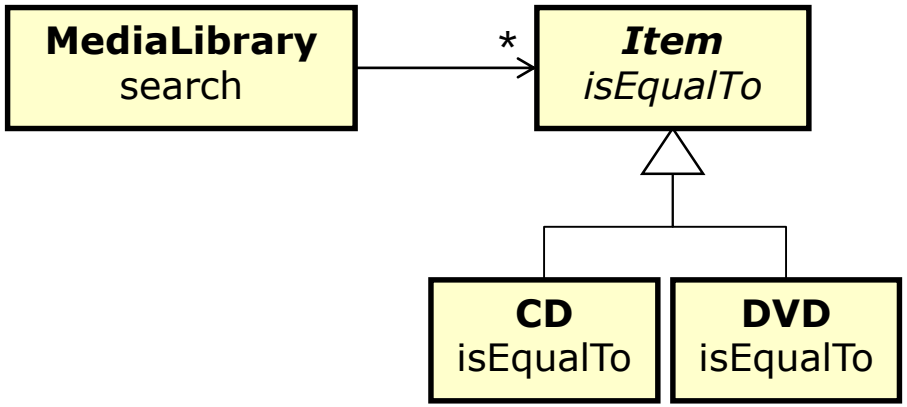
Clases abstractas

- Una clase abstracta es una clase cuyo propósito es servir de superclase para otras clases
  1. Ya sea para la generalización (polimorfismo)
  2. O para la reutilización de código, p.ej. extensión (herencia)
- Están implementadas parcialmente
  - Algunos métodos son abstractos
- Como efecto de esta implementación parcial, **no es posible crear instancias de clases abstractas**
  - Está controlado por el compilador estáticamente
  - Es una forma de evitar pasar un mensaje sin implementación a un objeto

Clases abstractas

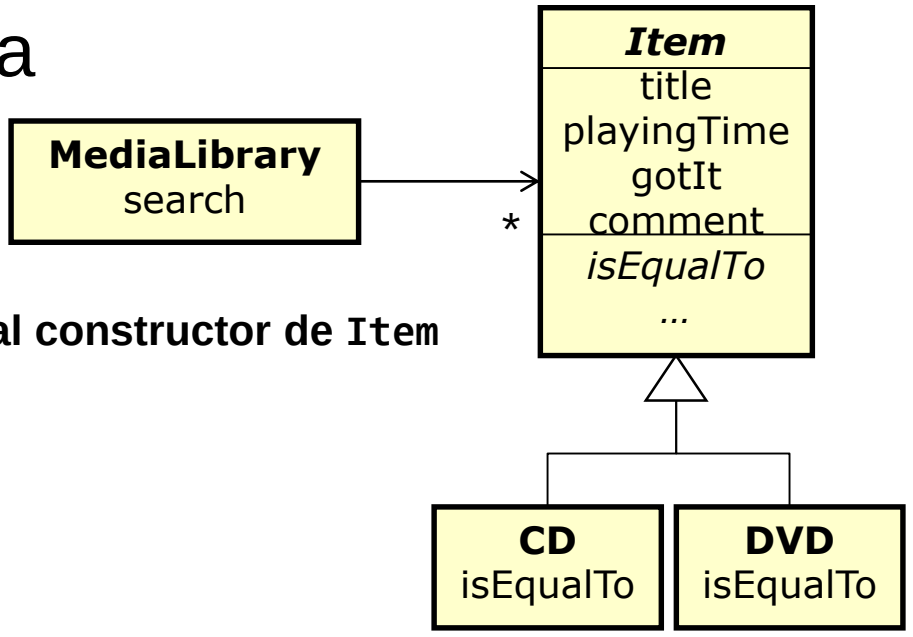
- Si decimos que NO es posible crear instancias de las clases abstractas

- Entonces, ¿tiene sentido implementar un constructor en una clase abstracta?



Clases abstractas

- Si `Item` tiene atributos (y los tiene), su constructor debería encargarse de inicializar sus valores
- Los constructores de las subclases llamarían a estos constructores de las clases abstractas
- Entonces, si tiene sentido tener un constructor en una clase abstracta



Clases abstractas

- Las clases abstractas obligan a las subclases a redefinir esos métodos declarados como abstractos
  - De lo contrario, las subclases serían también abstractas
- En Java, es posible declarar una clase abstracta sin métodos abstractos
  - Puesto que puede heredar los métodos abstractos de la superclase
  - Es probable que indique que se requiere la creación de subclases (una generalización)
- NO usarlas para evitar la creación de instancias
  - Usar constructores privados (p.e `java.lang.Math`)

Ejercicio

- Piense cómo podríamos declarar un método para **buscar cualquier objeto en un array**
- Debe ser aplicable a
  - CD
  - DVD
  - String
  - Integer
  - Double
  - …

Ejercicio

- Sólo hay una superclase común para todos los tipos : `Object`

```
public class Algorithms {
  public static int search(Object[] vector,
                           Object object) {
    // …
  }
}
```

Ejercicio

```
public class Algorithms {
  public static int search(Object[] vector,
                           Object object) {
    int i=0;
    for(Object element : vector) {
      if (element. ?? (object))
        return i;
      i++;
    }
  return -1; // no encontrado
  }
}
```

¿Tiene Object el método `isEqualTo`?

Ejercicio

- No, pero ofrece el método `equals`
  - Que es exactamente lo que estábamos buscando

```
public class Algorithms {
  public static int search(Object[] vector,
                           Object object) {
    int i=0;
    for(Object element : vector) {
      if (element.equals(object))
        return i;
      i++;
    }
  return -1; // no encontrado
  }
}
```

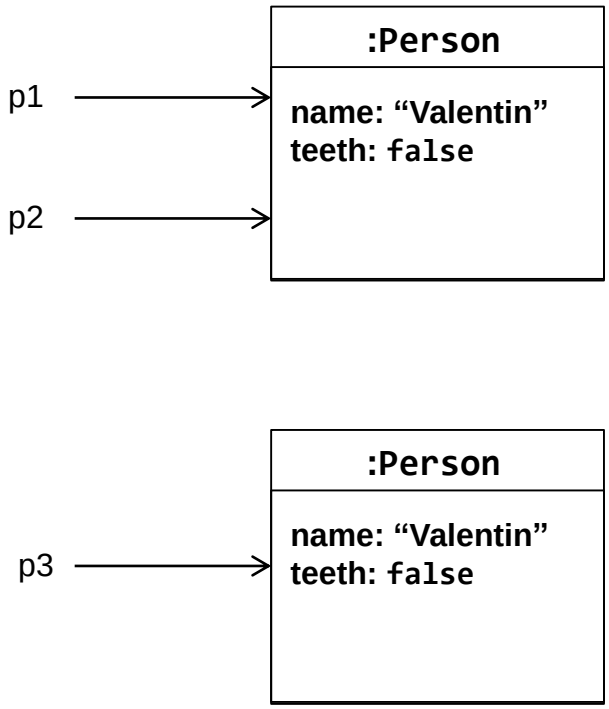
Ejercicio

- ¿Cuál es el único requisito para poder utilizar el método `search` de la clase `Algorithms`?
- Tan sólo hay que redefinir el método `equals` en las clases de nueva creación.
- Los objetos más utilizados en la API de Java tienen redefinido este método.

Igualdad en Java

- En Java, hay dos tipos de igualdad de objetos
  - **Igualdad** de **identidad** (o de referencias): dos referencias apuntan al mismo objeto
    - Se puede usar **el operador ==**
    - Funciona cuando los dos objetos son exactamente el mismo objeto.
    - Ejemplo: `Person p = new Person(); Person q;`
      `q = p;` **`p == q;`** // es true
  - **Igualdad** de **estado** (o de contenidos): dos objetos separados pasan a representar el mismo objeto
    - Entonces, se debe usar **método `equals`** de **`Object`**
    - Indica cuando dos objetos representan la misma entidad (`Person`, `DVD`, `CD`, `String`, `int`...)

Igualdad: Ejemplo



- `p1 == p2?` **`true`**
- `p2 == p3?` **`false`**
- `p1.equals(p2)?` **`true`**
- `p2.equals(p3)?` **`false`**
- Si `equals` esta redefinido en `Person`
  - `p1.equals(p2)?` **`true`** (obvimente)
  - `p2.equals(p3)?` **`true`** (siempre que en el método equals redefinido se compare los valores de name y teeth ☺)

equals() en Java

- El método equals implementa una relación de equivalencia en las **referencias a objetos no null**
  - Es *reflexiva*: `x.equals(x)` devolverá `true`
  - Es *simérica*: `x.equals(y)` devolverá `true` si y solo si `y.equals(x)` devuelve `true`
  - Es *transitiva*: si `x.equals(y)` devuelve `true` e `y.equals(z)` devuelve `true`, entonces `x.equals(z)` devolverá `true`
  - Es *consistente*: varias invocaciones de `x.equals(y)` constantemente devuelven `true` o constantemente devuelven `false`
  - `x.equals(null)` devolverá `false`

equals() en Java

- El método `equals` de la clase `Object` se comporta como sigue:
  - *Para cualquier referencia x e y (que no sean null), devuelve `true` si y solo si x e y se refieren al mismo objeto (`x == y` tiene el valor `true`)*
  - Así el método `equals()` de la clase `Object` funciona de hecho como una igualdad de identidad.

- Por lo general es necesario redefinir el método `hashCode()` cada vez que se redefine el método `equals()`.
  - **Los objetos iguales deben tener igual código hash**

Pregunta

- El siguiente código es válido para el compilador

```
System.out.println(
     new CD("Zapatillas", "El Canto del Loco",13, 170));

System.out.println(
     new DVD("Full metal jacket", "Stanley Kubrick", 180));
```

- ¿Como habrá sido declarado el método `println`? ¿Cómo será su signatura?

Método `toString` de `Object`

- Su signatura:
- `public void println(Object x)`
  - Con un parámetro que recibe un `Object`
- Y, qué método será invocado de la clase `Object`?

```
public String toString();
```

Devuelve una cadena representando al objeto. En general, el método toString devuelve una cadena que "representa textualmente" este objeto.

- ¿Hace lo mismo que el método `print` de la clase `Item`?

Método `toString` de `Object`

- ¿Hace lo mismo que el método `print` de la clase `Item`?
- No
  - **`print`** de **`Item`** escribe la cadena que representa al objeto en un `PrintStream` (la consola, salida de errores, etc.)
  - **`toString`** de **`Object`** devuelve un string (la cadena que representa al objeto)
- ¿Cuál crees que es más fácil de mantener?
- ¿Cual está más acoplado?

Método `toString` de `Object`

- El método **`toString`** de **`Object`** puede ser usado en más escenarios
  1. Para escribir la cadena que representa al objeto en la consola, ficheros, memoria, sockets…
  2. Para mostrar las excepciones que no fueron capturadas
  3. Para mostrar los objetos en interfaces gráficas como combo-boxes
  4. Cuando el operador + se aplica a objetos

  …
- **Es menos acoplado**
- **Es el comportamiento estándar en Java!**

Volviendo al proyecto DoMe

- Podemos mejorar la implementación de los métodos `list` en DoMe

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

Sustituye a:
`item.print(out);`

```
public abstract class Item {
  @Override
  public String toString() {
    return "Item [title=" + getTitle() +
          ", gotIt=" + getOwn() +
          ", comment=" + getComment() +
           ", playingTime=" + getPlayingTime() + "]";
  }
}
```

Sustituye a:
`public void print(PrintStream out)`

Volviendo al proyecto DoMe

```
public class CD extends Item {
  @Override
  public String toString() {
    return super.toString() +
                " Artista: " + getArtist() + "\n" +
              " Canciones: " + getNumberOfTracks() + "\n";
  }
}
```

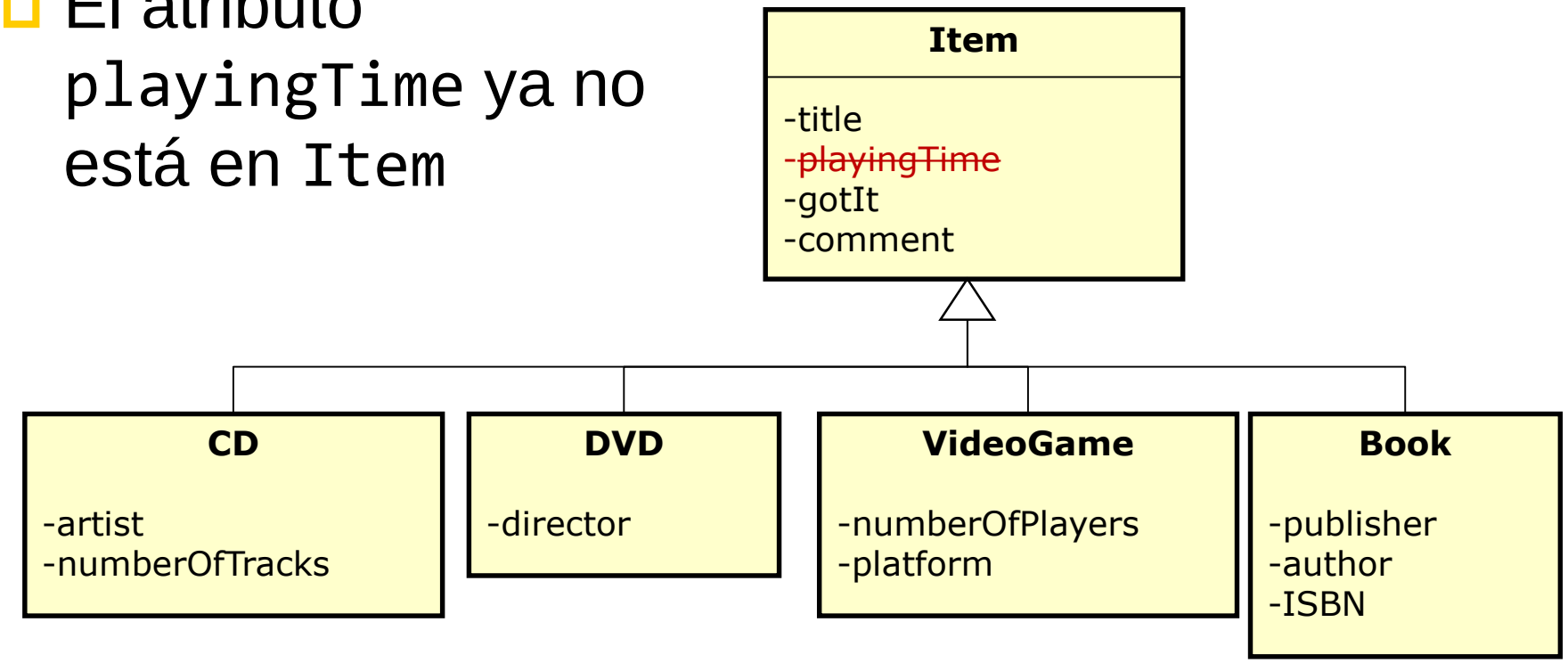
Code Style: Concatenación

```
public class DVD extends Item {
  @Override
  public String toString() {
    StringBuilder sb = new StringBuilder(super.toString());
    sb.append(" Director: ").append(getDirector()).append("\n");
    return sb.toString();
  }
}
```

Code Style: StringBuilder

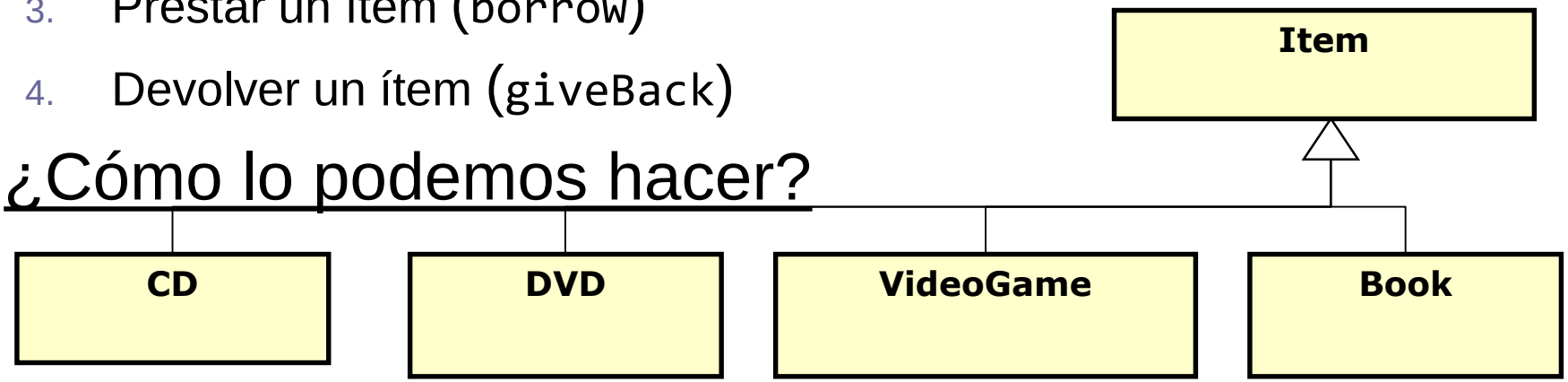
Extendiendo DoMe

- Vamos a añadir dos `Items más`
  - `VideoGame` (`numberOfPlayers`, y `platform`)
  - `Book` (`publisher`, `ISBN`, y `author`)
- El atributo `playingTime` ya no está en `Item`



Extendiendo DoMe

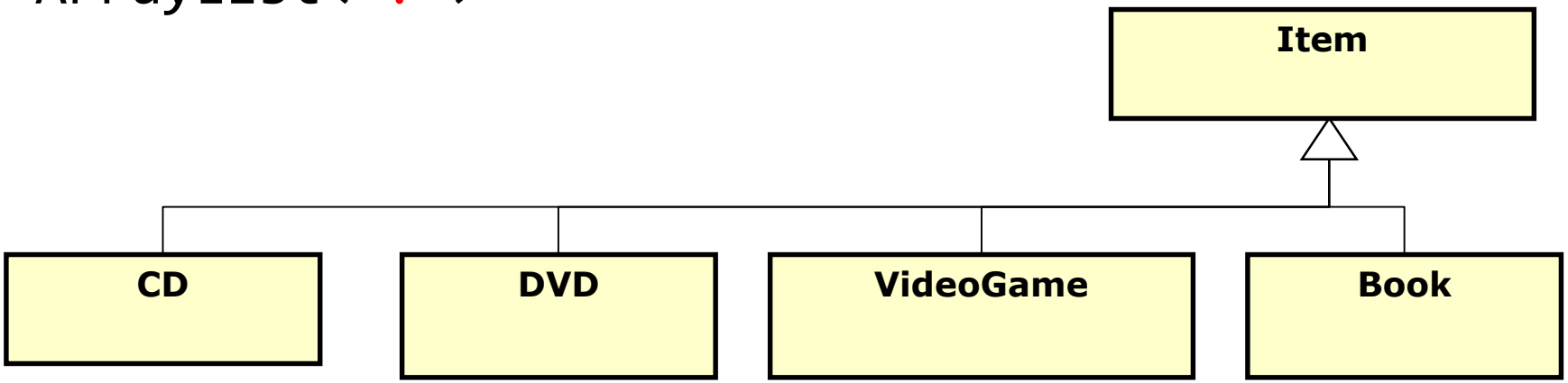
- También queremos añadir una nueva funcionalidad a MediaLibrary: los libros y CD pueden ser prestados
- La aplicación debe ofrecer:
  1. Listar los items que pueden ser prestados(`listBorrowableItems`)
  2. Listar los items que pueden ser prestados y están disponibles (no se han prestado aún) (`listAvailableItems`)
  3. Prestar un ítem (`borrow`)
  4. Devolver un ítem (`giveBack`)
- ¿Cómo lo podemos hacer?



Extendiendo DoMe

- Para listar los items que pueden ser prestados, añadimos otra colección `ArrayList` a la clase `MediaLibrary`
- Pero, ¿de qué tipo, debería ser la colección `ArrayList`?

```
ArrayList< ? >
```



Extendiendo DoMe

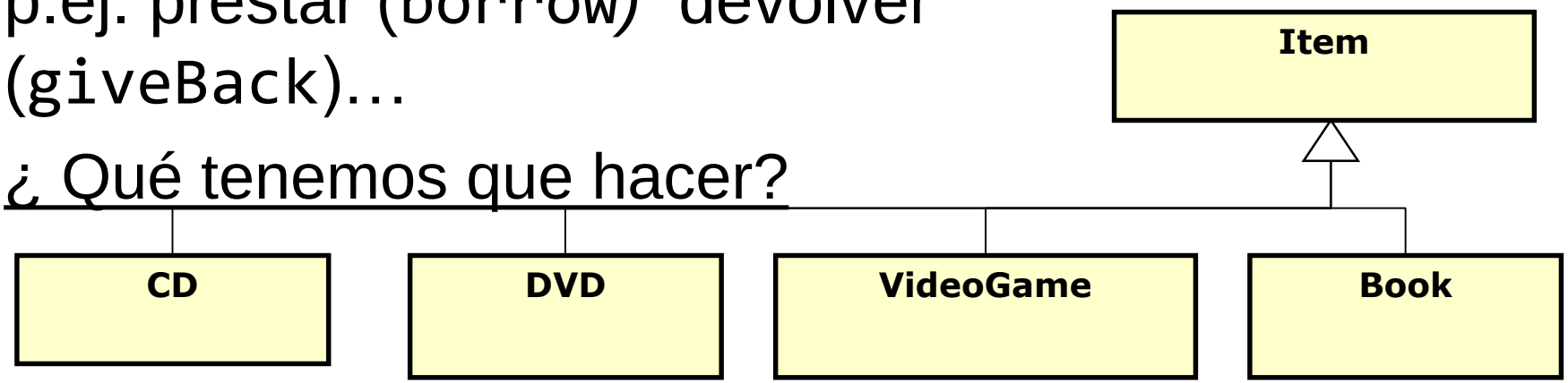
- ¿Object?
- Si añadimos el siguiente atributo a MediaLibrary

```
private ArrayList<Object> borrowableItems;
```

- Y para recorrer los objetos hacemos:

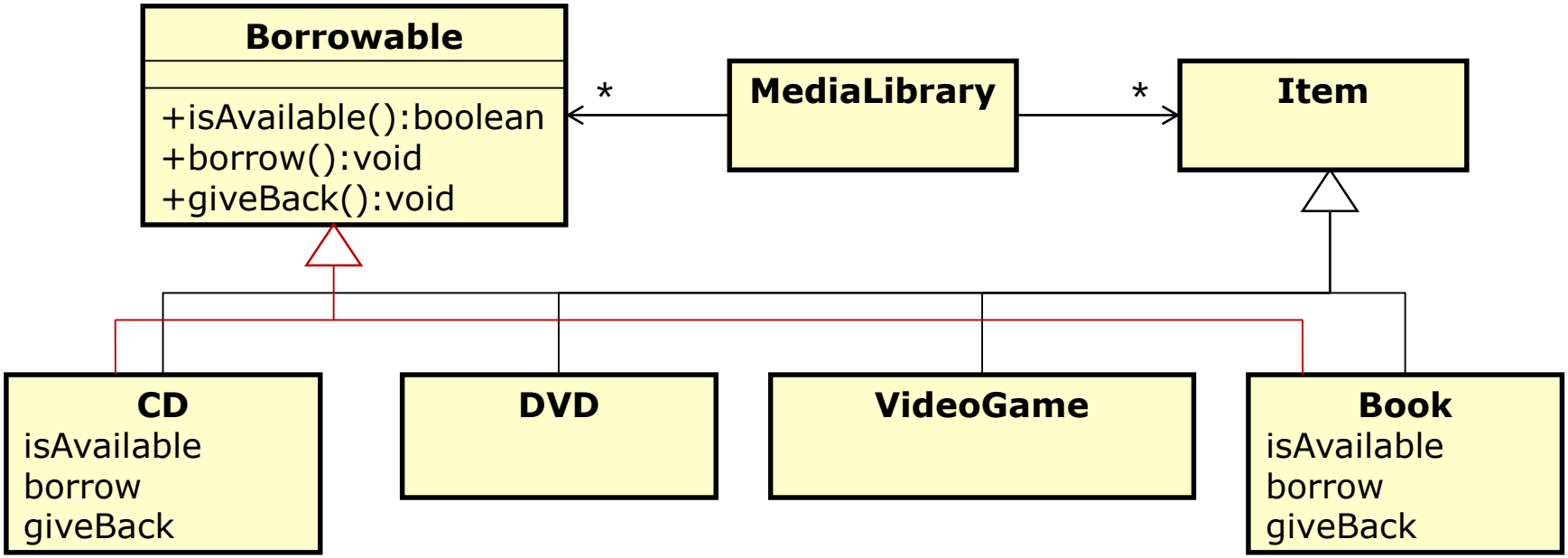
```
for(Object item : borrowableItems) {
        item. ?
```

- Pero Object no ofrece el mensaje que necesitamos p.ej. prestar (borrow) devolver (giveBack)…
- ¿ Qué tenemos que hacer?

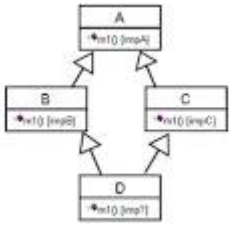


Generalización

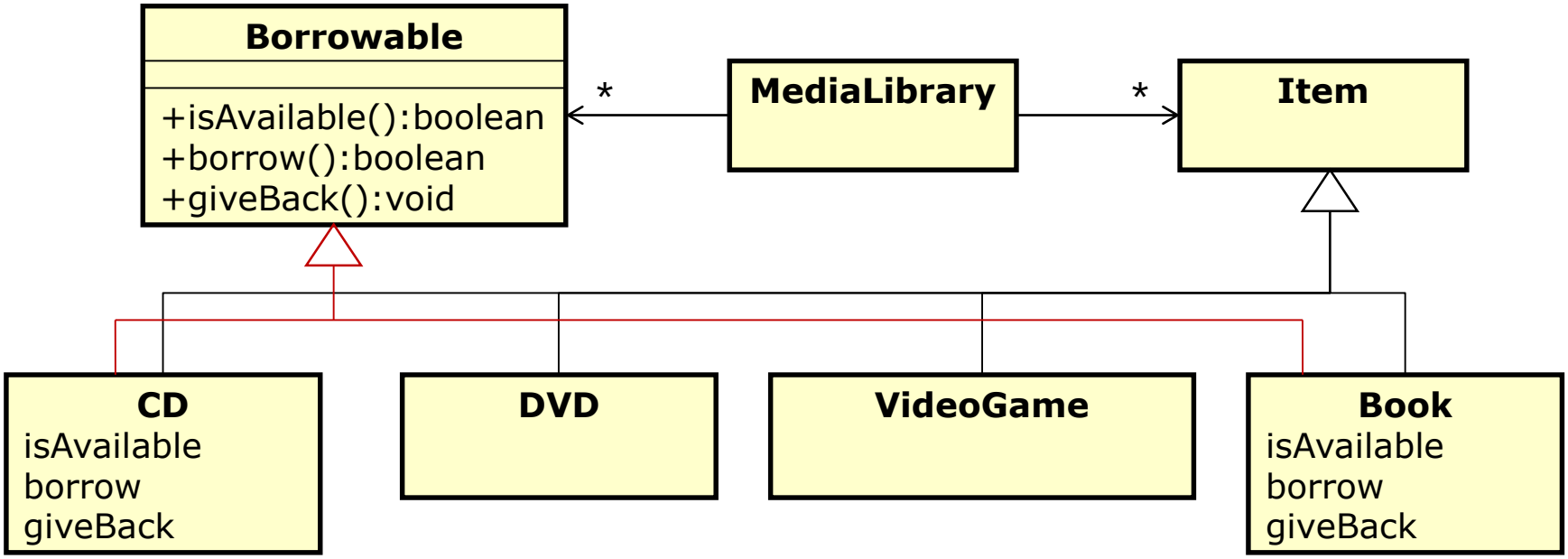
- Podemos generalizar, una vez más
- Una nueva generalización aparece “prestable” (`Borrowable`)
- Solo `CD` y `Book` son `Borrowable`
- `CD` y `Book` especializan los métodos de `Borrowable`



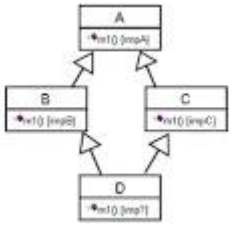
Herencia Múltiple



- En este diseño estamos reflejando la necesidad de **herencia múltiple**
- La herencia múltiple aparece cuando una clase hereda **directamente** de más de una superclase
  - Ambas clases CD y Book tienen dos superclases directas

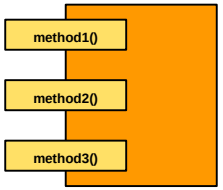


Herencia Múltiple



- El concepto de herencia múltiple es bastante fácil de entender
- Sin embargo, conlleva complicaciones importantes en la implementación de un lenguaje de programación
- Existen lenguajes que proporcionan herencia múltiple, y otros no. Ejemplos:
  - C++, Eiffel, Python SI soportan herencia múltiple
  - **Java**, C# y Ruby NO soportan herencia múltiple
- Para compensar la falta de herencia múltiple, Java soporta el concepto de **Interface**

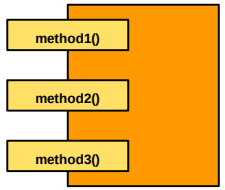
Interfaces



- Una interfaz se define como el conjunto de mensajes que una clase ofrece a sus clientes (public)
- En Java, este concepto es ofrecido como un tipo
- Por lo tanto, una interfaz Java es un conjunto de declaraciones de métodos, sin implementación

```
public interface Borrowable {
  boolean isAvailable();
  void borrow();
  void giveBack();
}
```

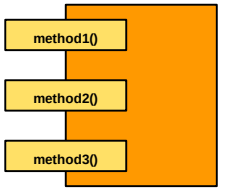
Interfaces



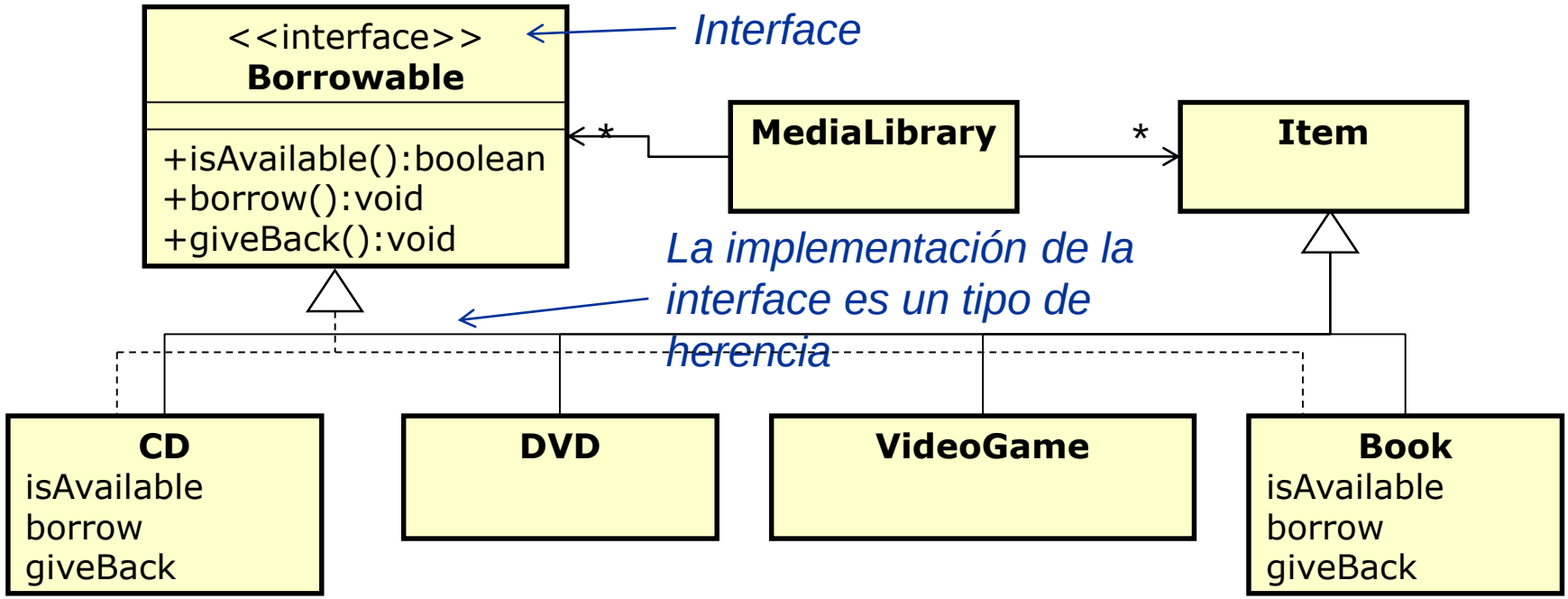
- En Java tienen las siguientes características:
  - Las Interfaces pueden ser declaradas como **public o no** (package)
  - Todos sus métodos son **abstract** (no es necesario poner la palabra abstract)
  - Todos sus métodos son **public** (no es necesario poner public)
  - Las interfaces no tienen constructores
  - Solo se permiten atributos con **`public static final`**
  - Una interface puede heredar de otras interfaces
  - En Java 1.8, soporta **Virtual Extension Methods** (implementaciones por defecto de los métodos de la interfaz)
    - http://www.javabeat.net/virtual-extension-methods-in-java-8/

```
public interface Borrowable {
  boolean isAvailable();
  void borrow();
  void giveBack();
}
```

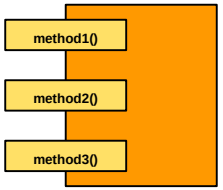
Interfaces



- Una clase puede heredar (implementar) **múltiples interfaces**
- Con esto se consigue dar a los tipos una nueva dimensión
  1. `Book` ≤ `Item` ≤ `Object` (inheritance tree)
  2. `Book` ≤ `Borrowable` (interface graph)

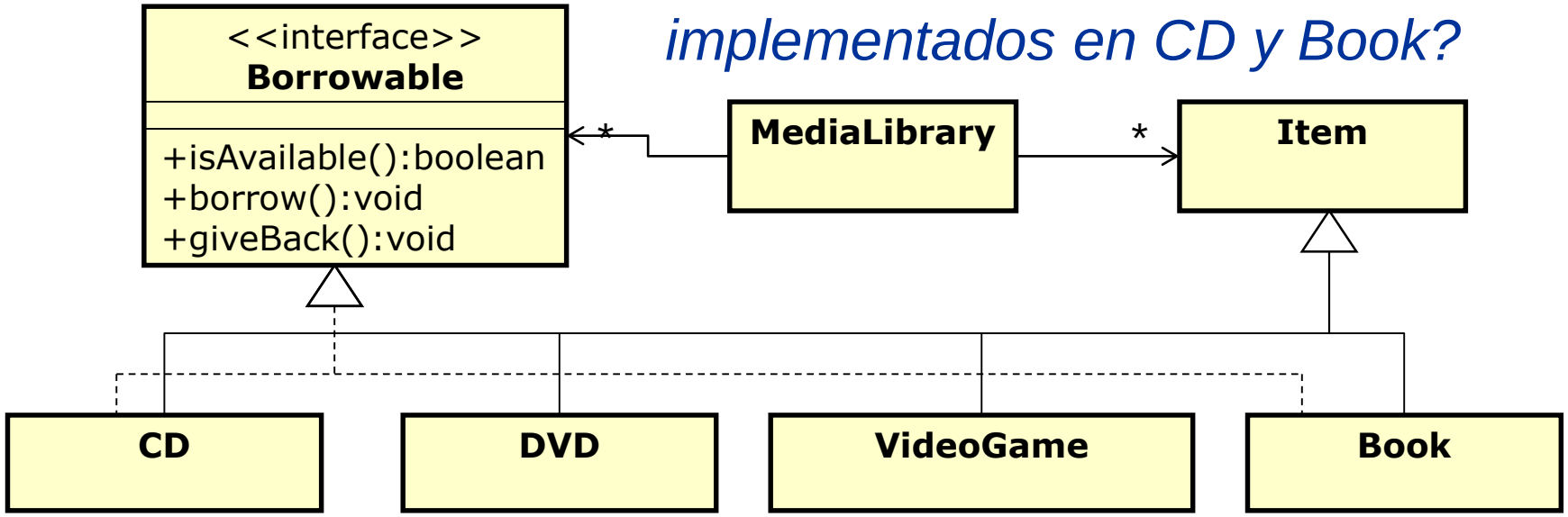


Interfaces

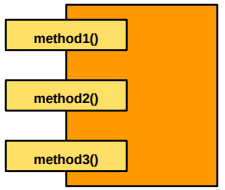



```
public class Book extends Item implements Borrowable {
    // ...
}
public class CD extends Item implements Borrowable {
    // ...
}
```

¿Que sucede si los tres métodos de Borrowable no son implementados en CD y Book?

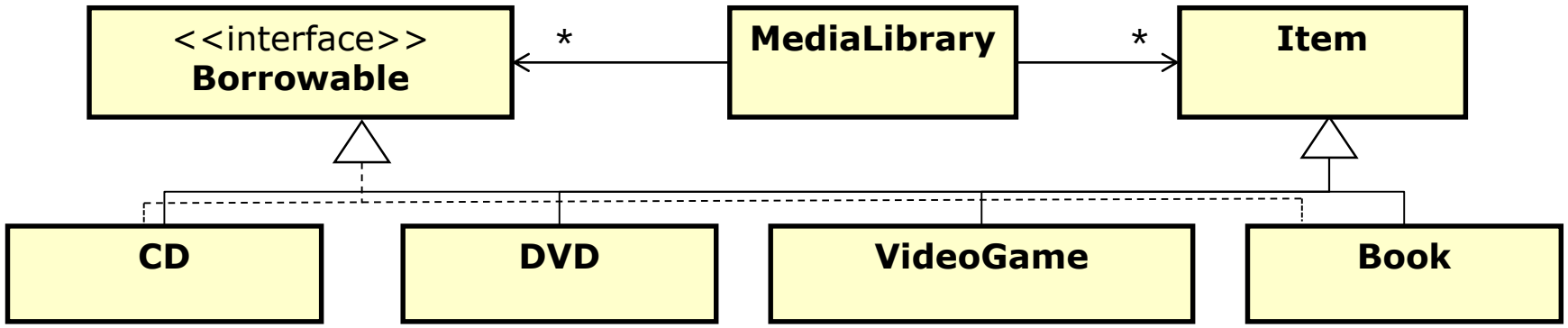


Implementación de la clase Book

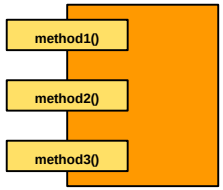


- Se deben implementar los tres métodos abstractos en CD y Book
  - En caso contrario, las dos clases serían abstractas

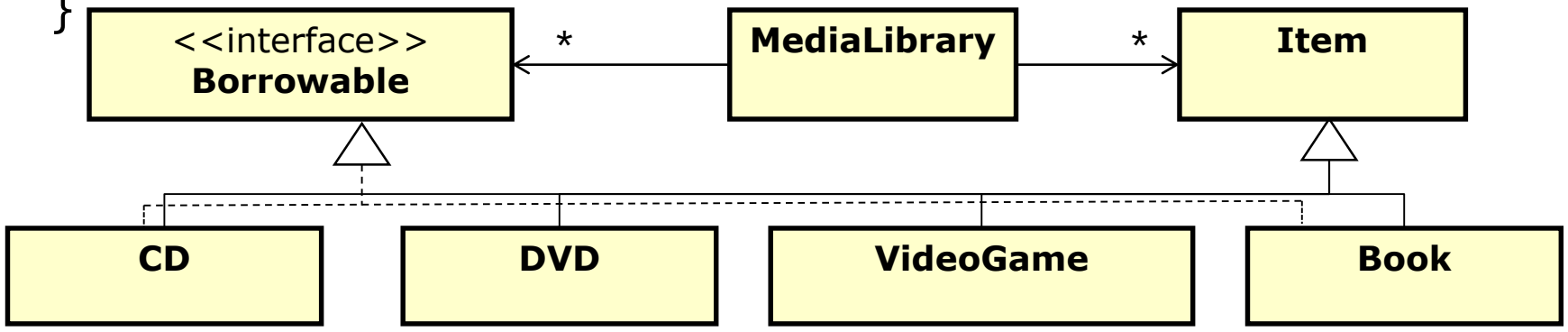
```
public class Book extends Item implements Borrowable {
   private String author;
   private String ISBN;
   private String publisher;
   private boolean isAvailable;
   // …
```



Implementación de la clase Book

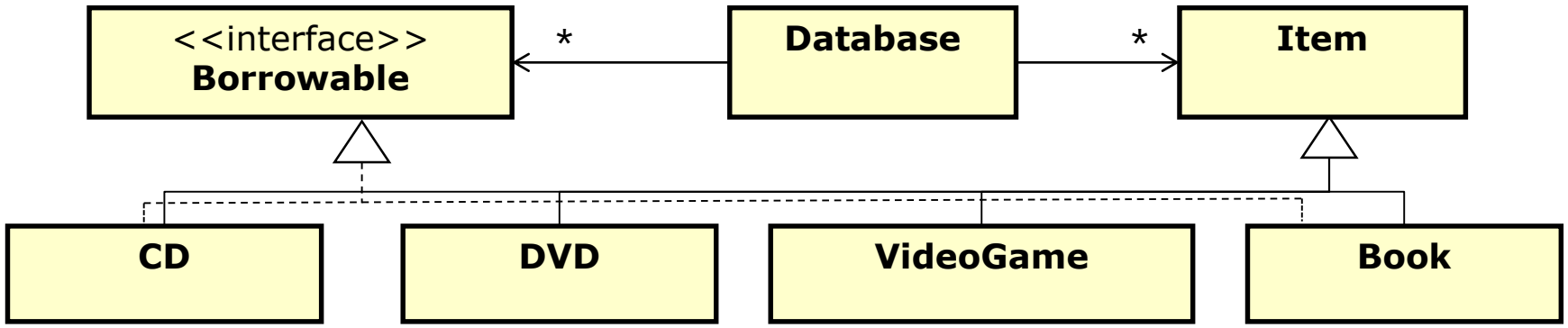



```
@Override
public boolean isAvailable() {
     return this.getOwn() && this.isAvailable;
}

@Override
public void borrow() {
    this.isAvailable = false;
}

@Override
public void giveBack() {
    this.isAvailable = true;
}
```



Recordando las nuevas características de DoMe

- Recordando la nueva funcionalidad, la aplicación debe ofrecer:
  1. Listar los items que pueden ser prestados
  2. Listar los items que se pueden prestar y están disponibles (no se han prestado aún)
  3. Prestar un item
  4. Devolver un item

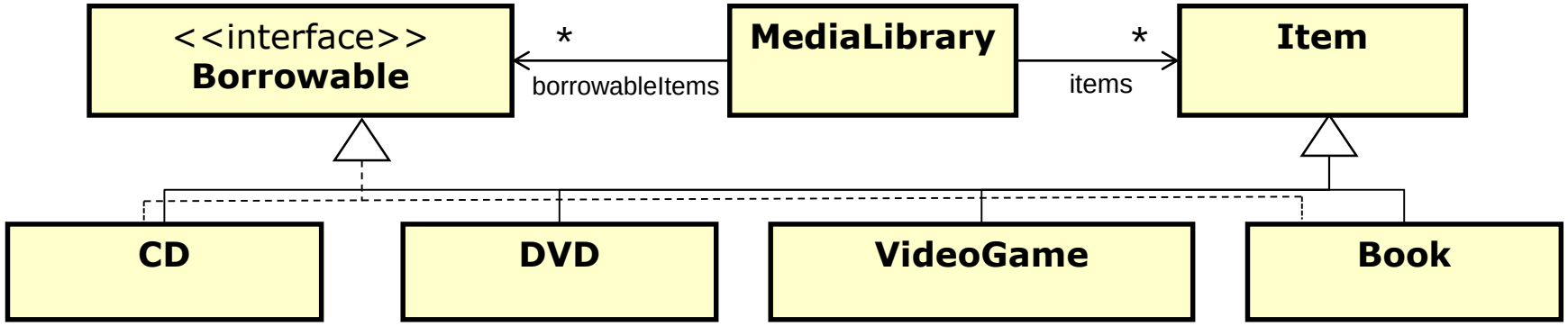


Primera característica

1. **Listar los items que pueden ser prestados**

```
public class MediaLibrary {

  private ArrayList<Item> items;
  private ArrayList<Borrowable> borrowableItems;

  public void add(Item theItem) {
    items.add(theItem);
    if (theItem instanceof Borrowable)
       borrowableItems.add((Borrowable)theItem);
  }
```

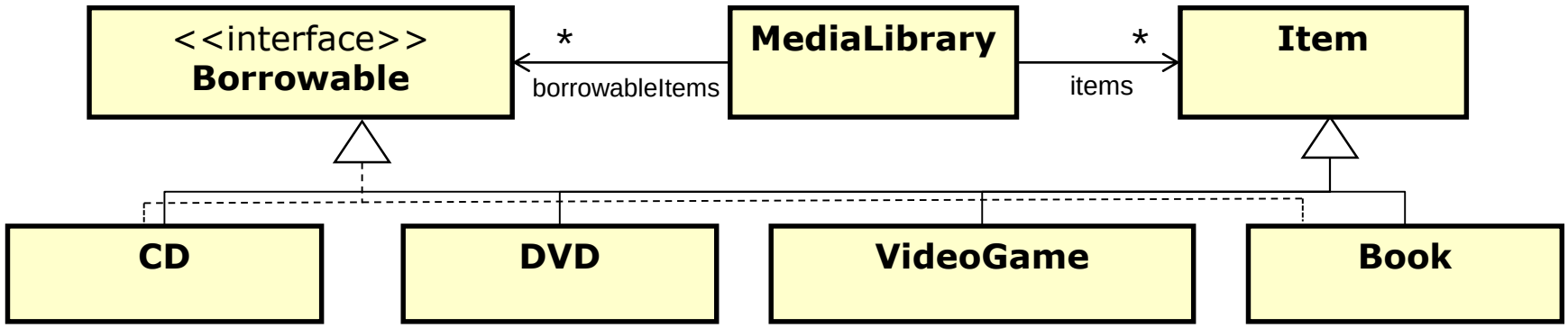
Significa: ¿Este item es Borrowable?



Primera característica

1. **Listar los items que pueden ser prestados**

```
public void listBorrowableItems(PrintStream out) {
  for (Borrowable item: borrowableItems)
    out.println(item);
}
```

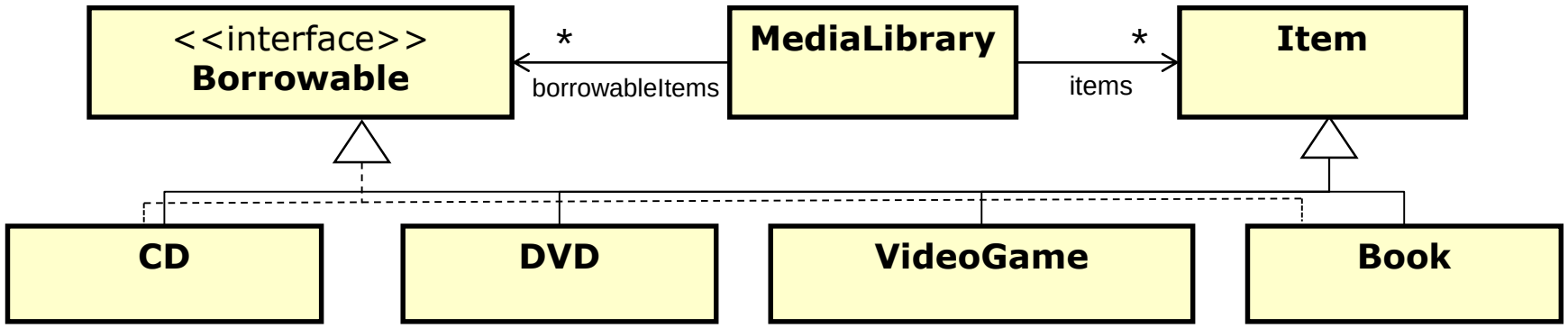


Segunda característica

2. **Listar los items que se pueden prestar y están disponibles** (no se han prestado aún)

```
public void listAvailableItems(PrintStream out) {
    for (Borrowable item: borrowableItems)
        if (item.isAvailable())
            out.println(item);
}
```

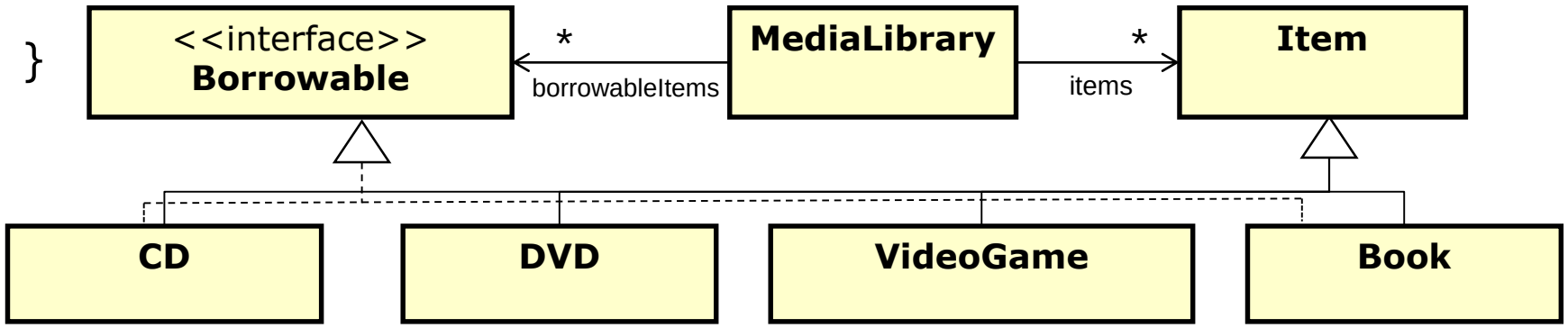
Polimorfismo



Tercera característica

3. **Prestar un item**

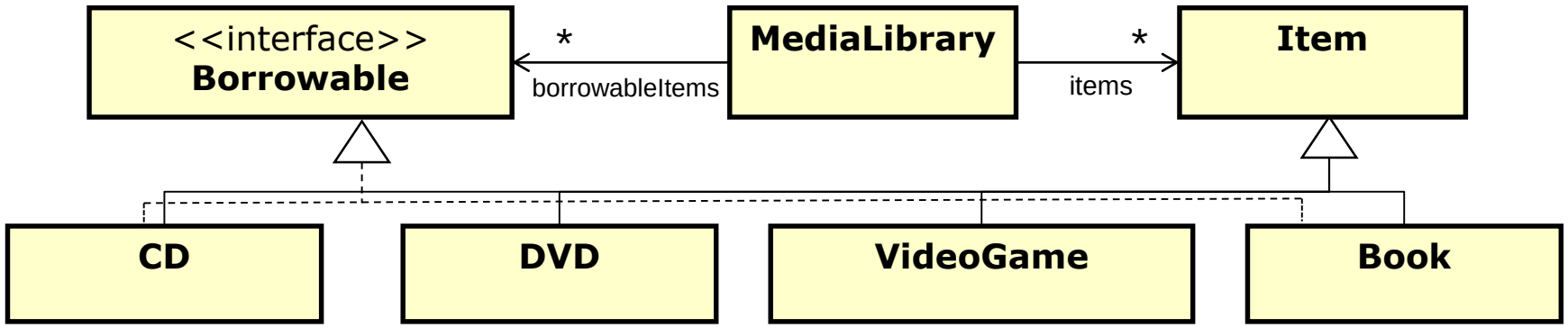
```
private Borrowable search(Borrowable item) {
  for(Borrowable i: borrowableItems)
    if (i.equals(item)) return i;
  return null;
}
public Borrowable borrow(Borrowable item) {
  Borrowable theItem = search(item);
  if (theItem == null) return null; // no encontrado
  if (! theItem.borrow()) return null; // no disponible
  theItem.borrow();
  return theItem; // se puede prestar
}
```



Cuarta característica

4. **Devolver un item**

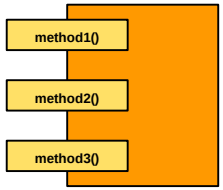
```
public boolean giveBack(Borrowable item) {
        Borrowable theItem = search(item);
        if (theItem == null)
                return false; // no encontrado
        if (theItem.isAvailable())
                return false; // ya disponible
        theItem.giveBack();
        return true; // correcto
}
```



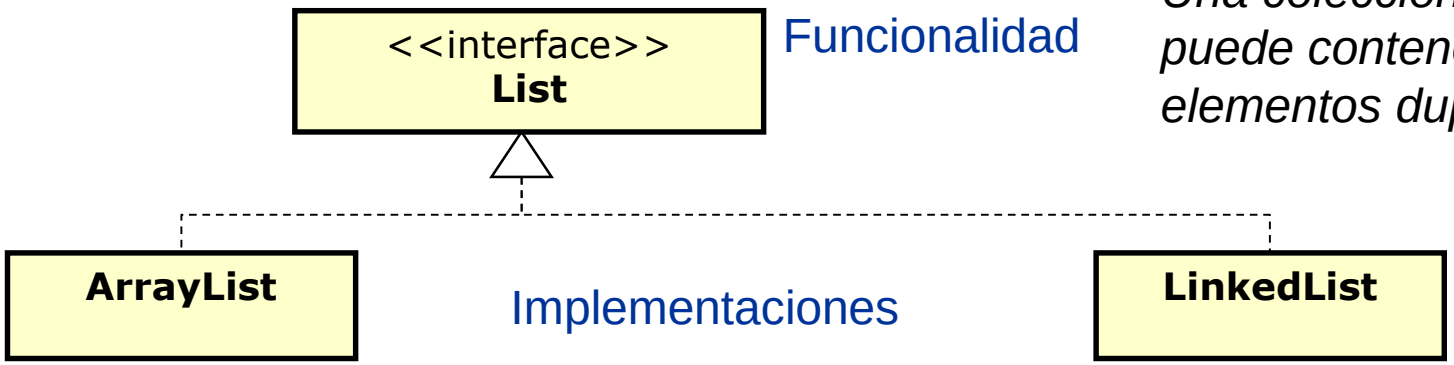
Resumen

- A menudo se necesita la generalización múltiple
- Dado que Java no proporciona herencia múltiple...
- … se utilizan las interfaces en su lugar
- Una clase Java puede implementar varias interfaces
- La implementación de una interface
  - No tiene nada que ver con la herencia (la falta de implementación se hereda debido a que cada método es abstracto)
  - Se utiliza para el polimorfismo múltiple (subtyping) ⇒ Se añade una nueva dimensión
  - Utiliza el enlace dinámico ya que todos los métodos de una interfaz deben ser redefinidos

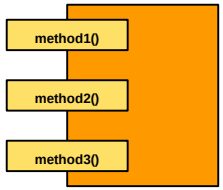
Interfaces como especificaciones



- Un uso común de las interfaces es para **separar una funcionalidad de su implementación**
  - Una interface define una funcionalidad
  - Cada clase que implementa esta interfaz define las posibles implementaciones
- Ejemplo en `java.util`:



Interfaces como especificaciones

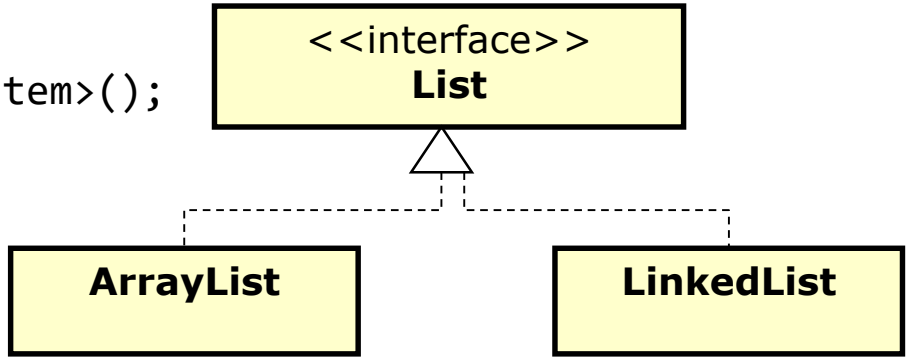


- Por lo tanto, cuando las interfaces se utilizan para separar la funcionalidad de su implementación, **es importante utilizar la interfaz en lugar de la clase**

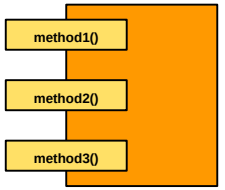
```
1. private ArrayList<Item> items = new ArrayList<Item>(); GOOD
2. private List<Item> items = new ArrayList<Item>(); BETTER
```

- La segunda opción es mejor, porque `items` no se acopla con una implementación específica

```
1.items = new LinkedList<Item>();
2.items = new MyOwnListImplementation<Item>();
MAGIC ☺
```



Interfaces vs. Clases Abstractas



- Una pregunta común en Java es cuándo utilizar clases abstractas o interfaces
- Dado que Java no proporciona herencia múltiple
  - **Sólo** usar clases abstractas cuando se desea heredar implementaciones (métodos y/o atributos)
- Si se puede elegir, las interfaces suelen ser preferibles
- Una forma común de programación en Java es
  1. Usar siempre interfaces para generalizar
  2. Usar solamente clases abstractas para la reutilización de la implementación (parcial)

Revisión

- Los métodos abstractos definen los mensajes que se pueden pasar a un objeto
- Las clases abstractas son superclases implementadas parcialmente
- La herencia múltiple no está soportada en Java
- Las interfaces permiten generalizaciones múltiples
- Uso común de interfaces para generalizar (polimorfismo y enlace dinámico)
- Uso común de clases abstractas para reutilizar el código a través de la herencia