Lecture 8
Chapter 8: Improving structure with inheritance

Main concepts to be covered

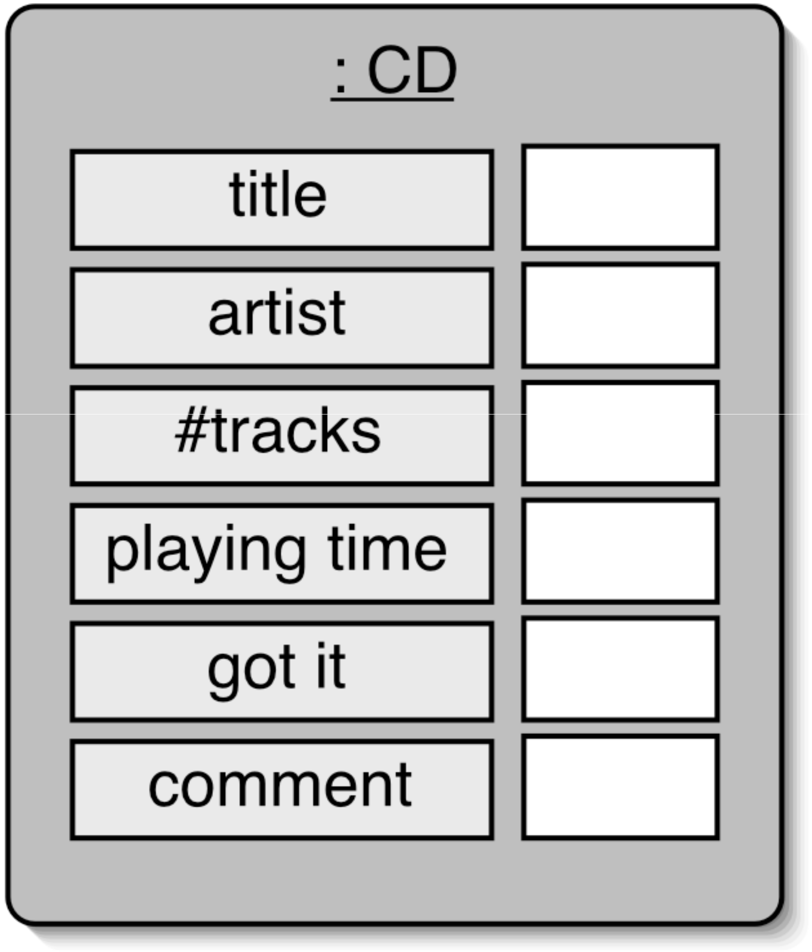
- Inheritance
- Subtyping
- Substitution
- Polymorphic variables

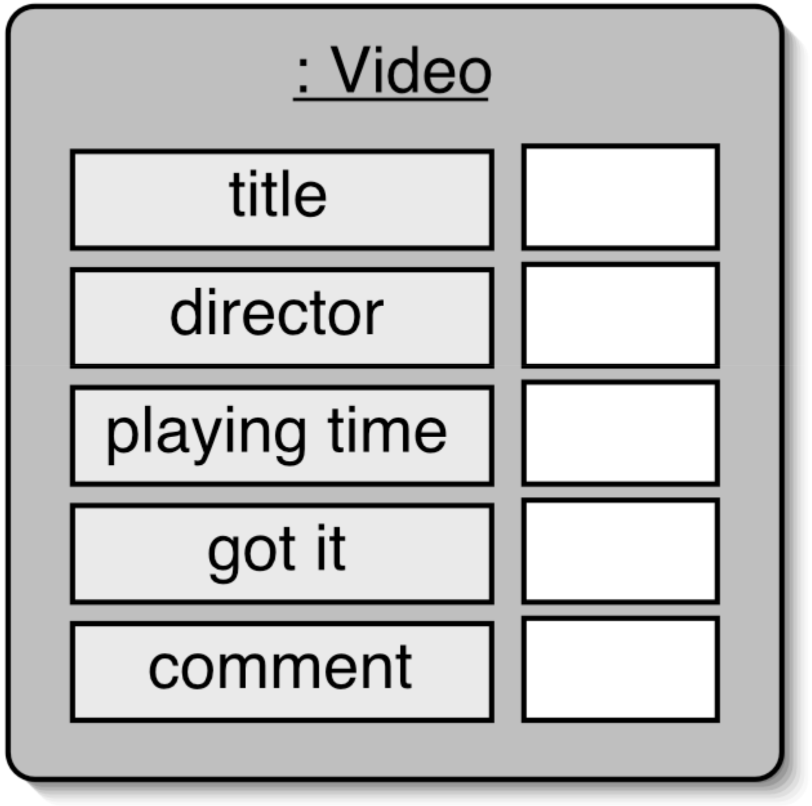
The DoME example

"Database of Multimedia Entertainment"

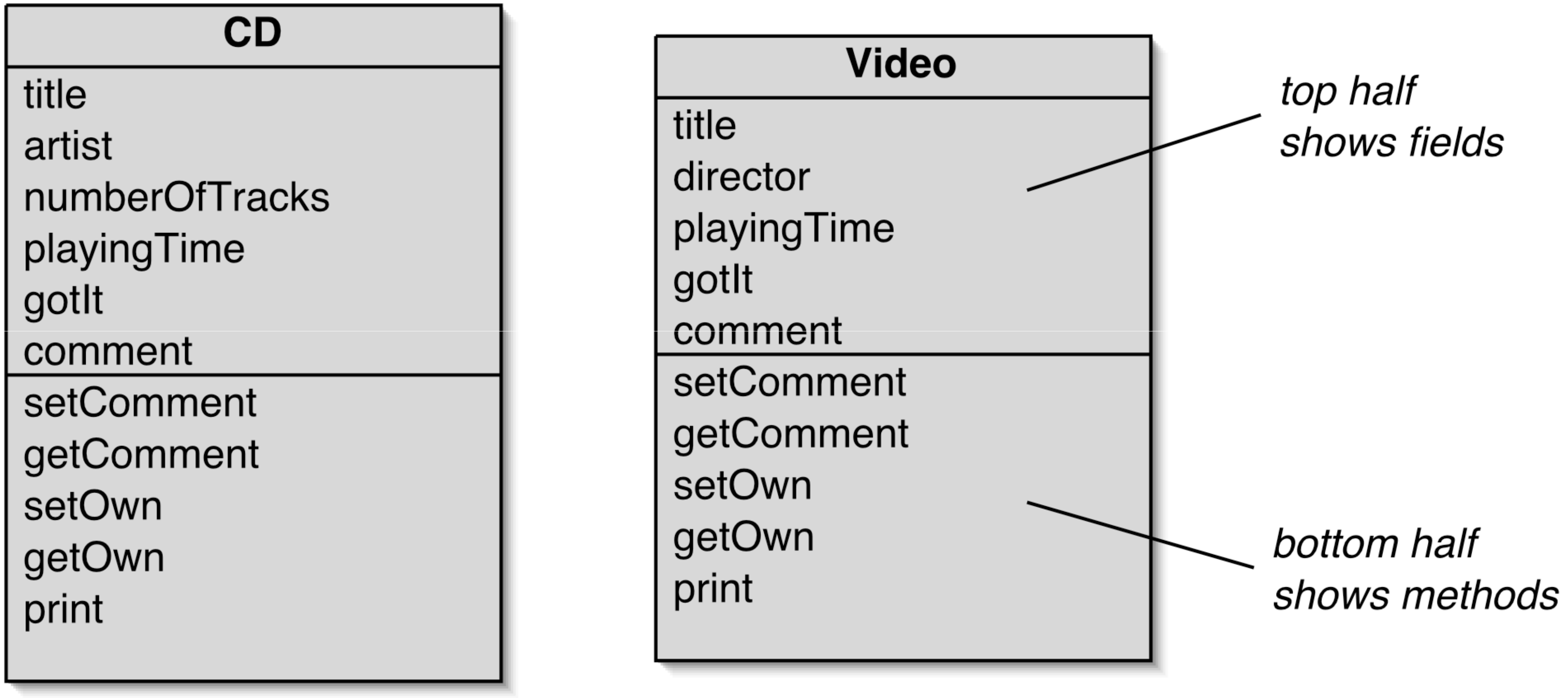
- stores details about CDs and videos
  - CD: title, artist, # tracks, playing time, got-it, comment
  - Video: title, director, playing time, got-it, comment
- allows (later) to search for information or print lists

DoME objects

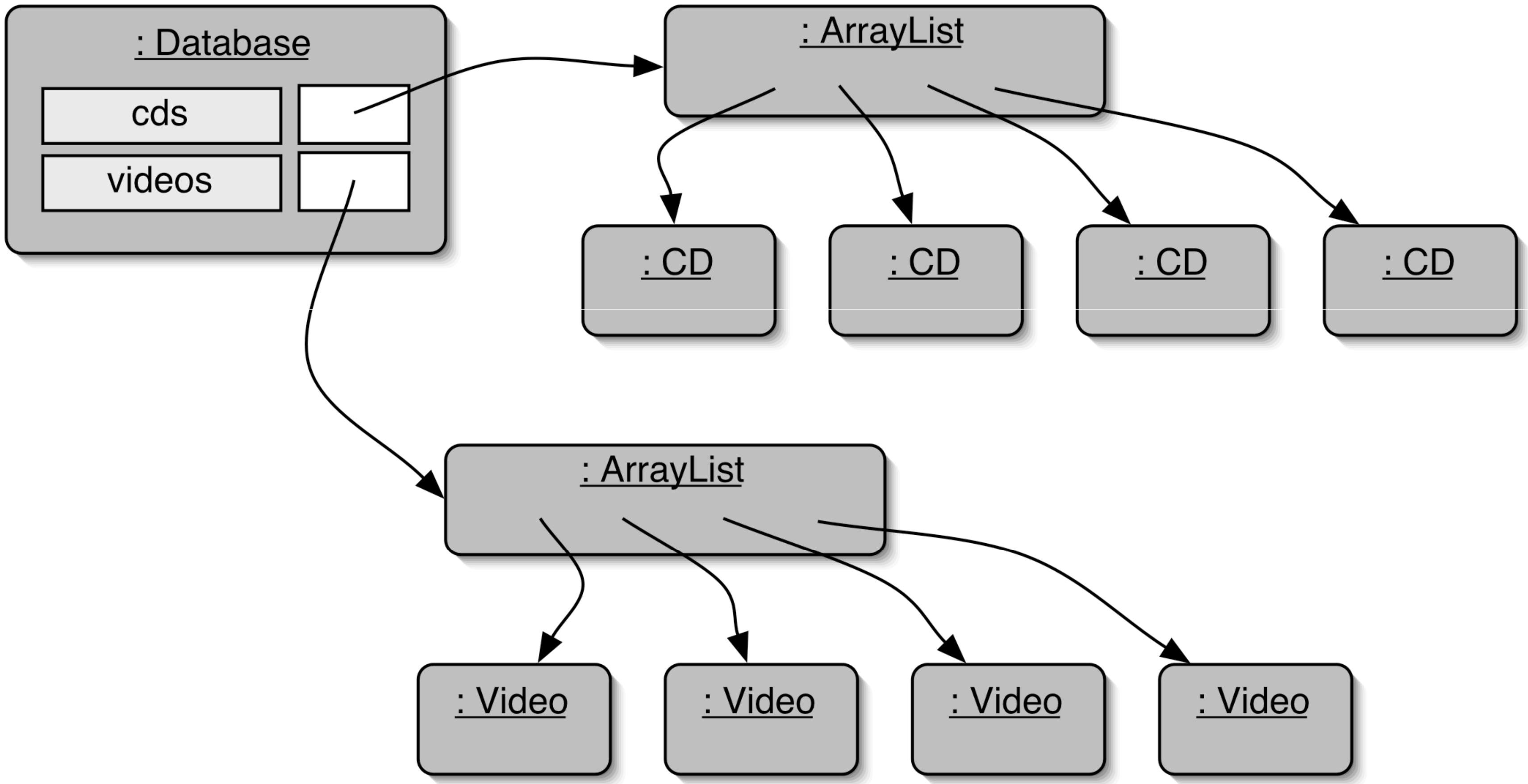




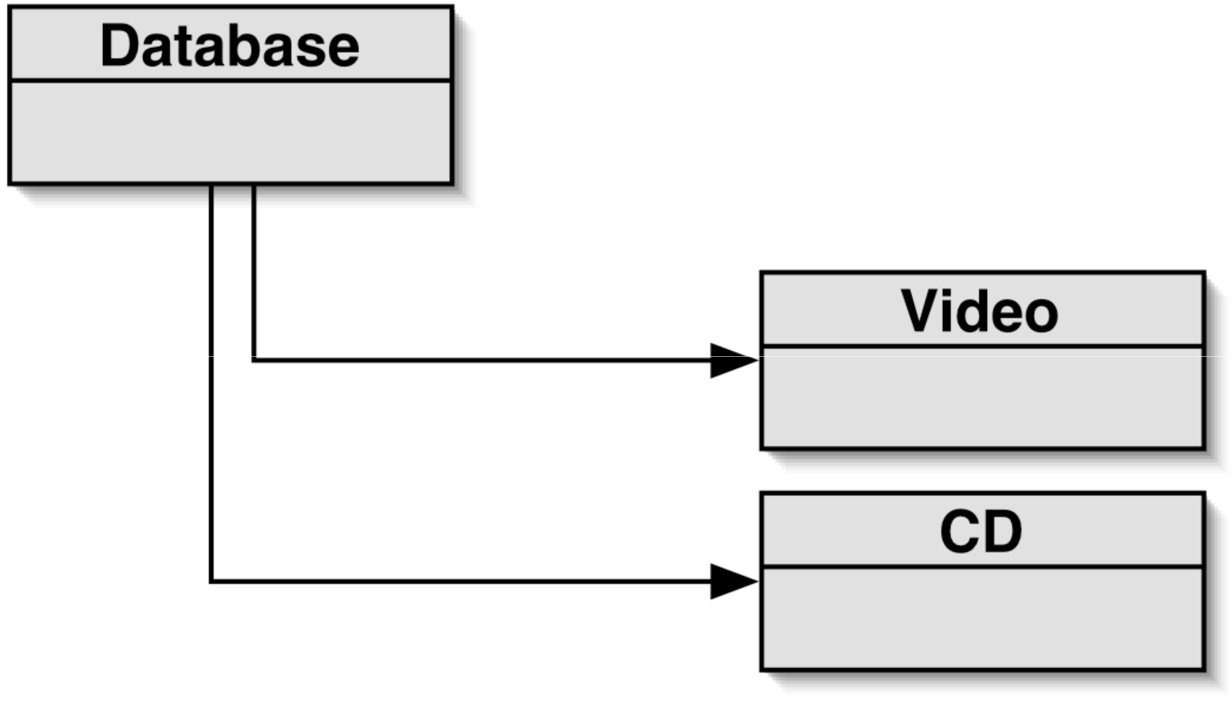
DoME classes (with details)



A possible implementation



BlueJ Class diagram



Note lack of detail: standard library classes not shown

CD source code

[incomplete (comments!)]

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

Video source code

[incomplete (comments!)]

```
public class Video {
   private String title;
   private String director;
   private String comment;

   Video(String theTitle, String theDirect)
   {
      title = theTitle;
      director = theDirect;
      comment = " ";
   }

   void setComment(String newComment)
   { ... }

   String getComment()
   { ... }

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

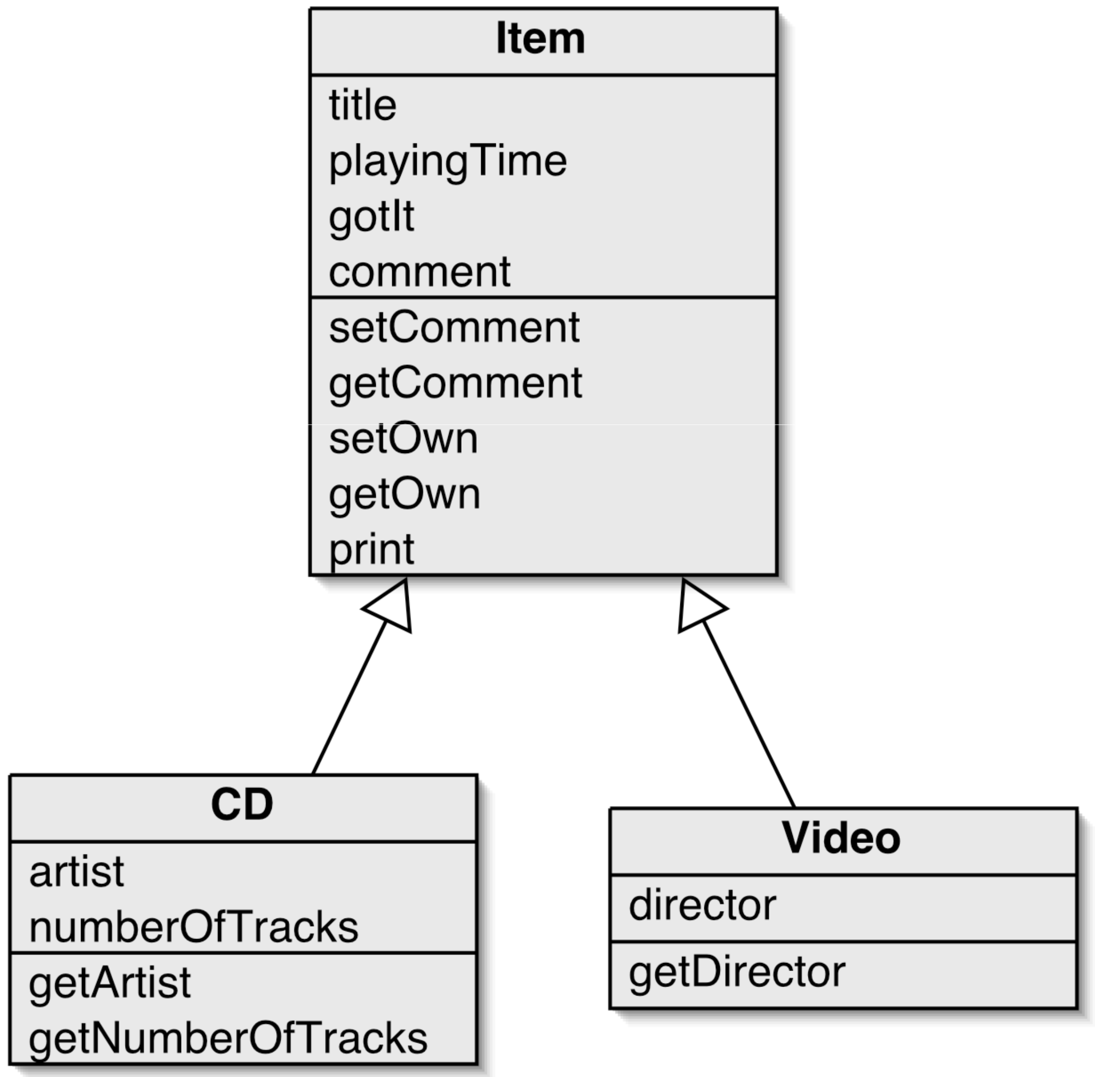
Database source code

```
public class Database {

   private ArrayList<CD> cds;
   private ArrayList<Video> videos;
   ...

   public void list()
   {
      for(Iterator<CD> iter = cds.iterator(); iter.hasNext(); ) {
          CD cd = iter.next();
          cd.print();
          System.out.println();   // empty line between items
       }

      for(Iterator iter <Video> = videos.iterator();
             iter.hasNext(); ) {
          Video video = iter.next();
          video.print();
          System.out.println();   // empty line between items
       }
   }
}
```

Critique of DoME

- code duplication
  - CD and Video classes very similar (large part are identical)
  - makes maintenance difficult/more work
  - introduces danger of bugs through incorrect maintenance
- code duplication also in Database class

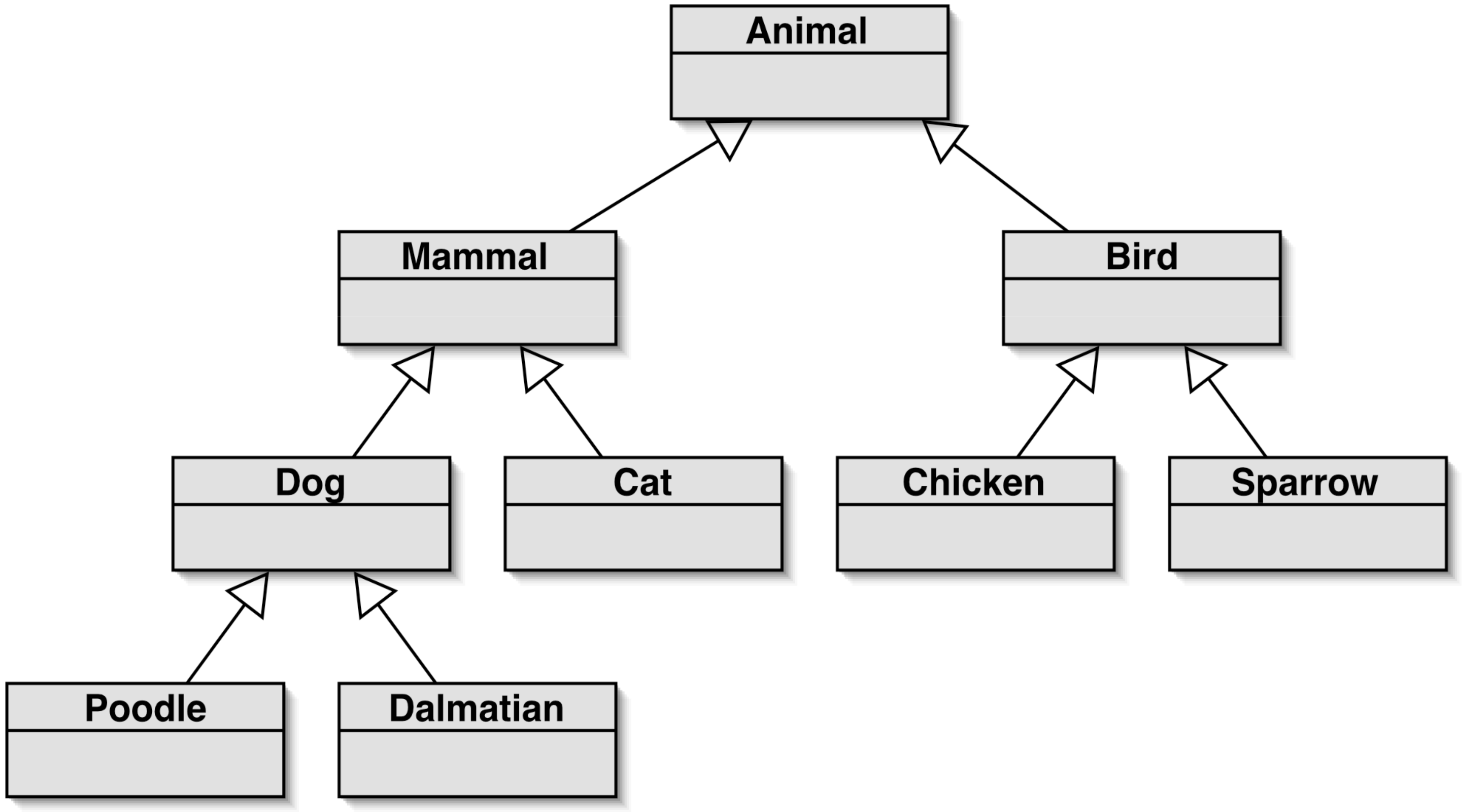
Using inheritance



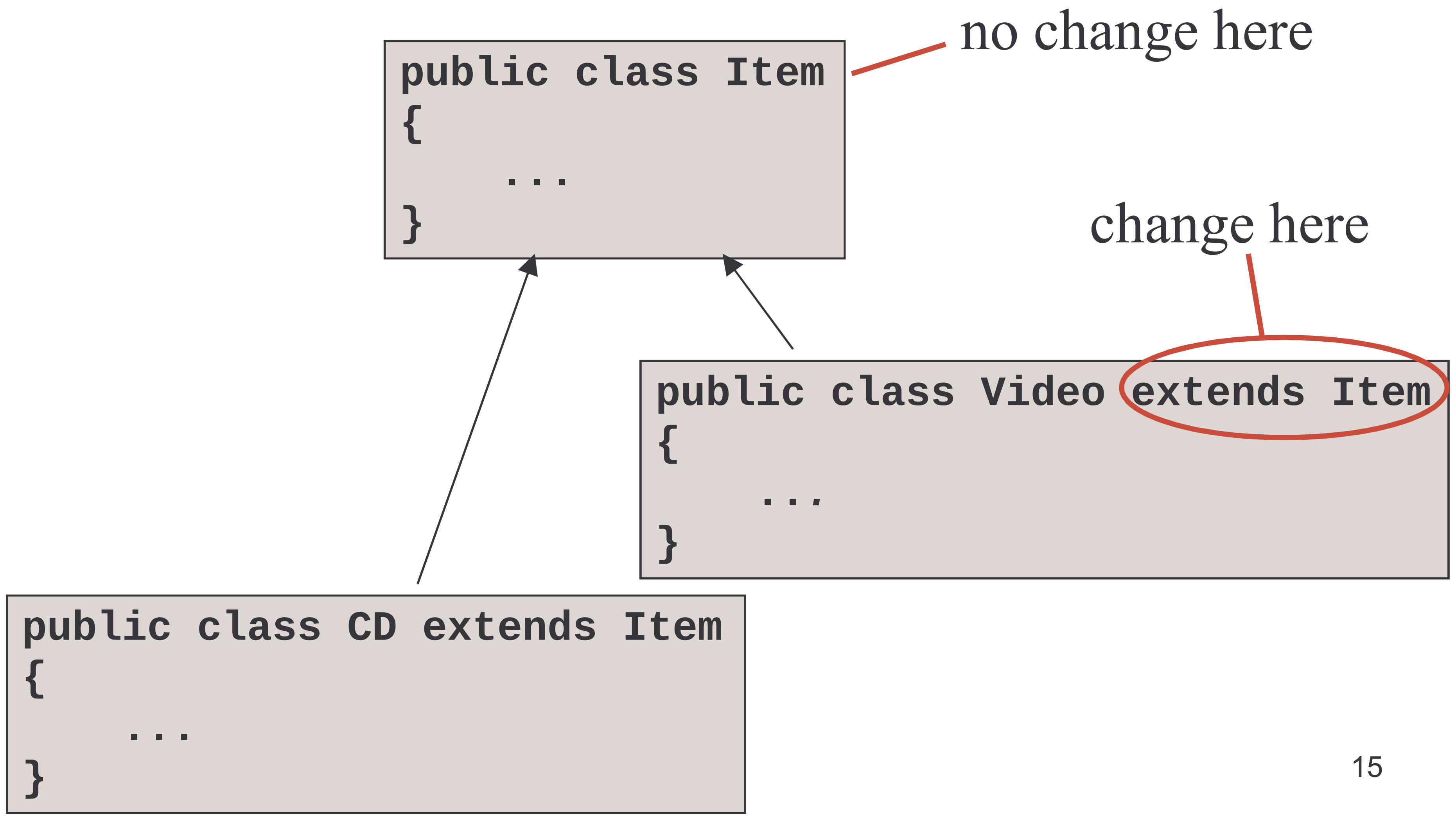
Using inheritance

- define one **superclass** : Item
- define **subclasses** for Video and CD
- the superclass defines common attributes
- the subclasses **inherit** the superclass attributes
- the subclasses add own attributes

Inheritance hierarchies



Inheritance in Java



```
public class Item
{
    ...
}
```

```
public class Video extends Item
{
    ...
}
```

```
public class CD extends Item
{
    ...
}
```

Superclass

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

    // constructors and methods omitted.
}
```

Subclasses

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

    // constructors and methods omitted.
}
```

```
public class Video extends Item
{
    private String director;

    // constructors and methods omitted.
}
```

Inheritance and constructors

```
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
}
```

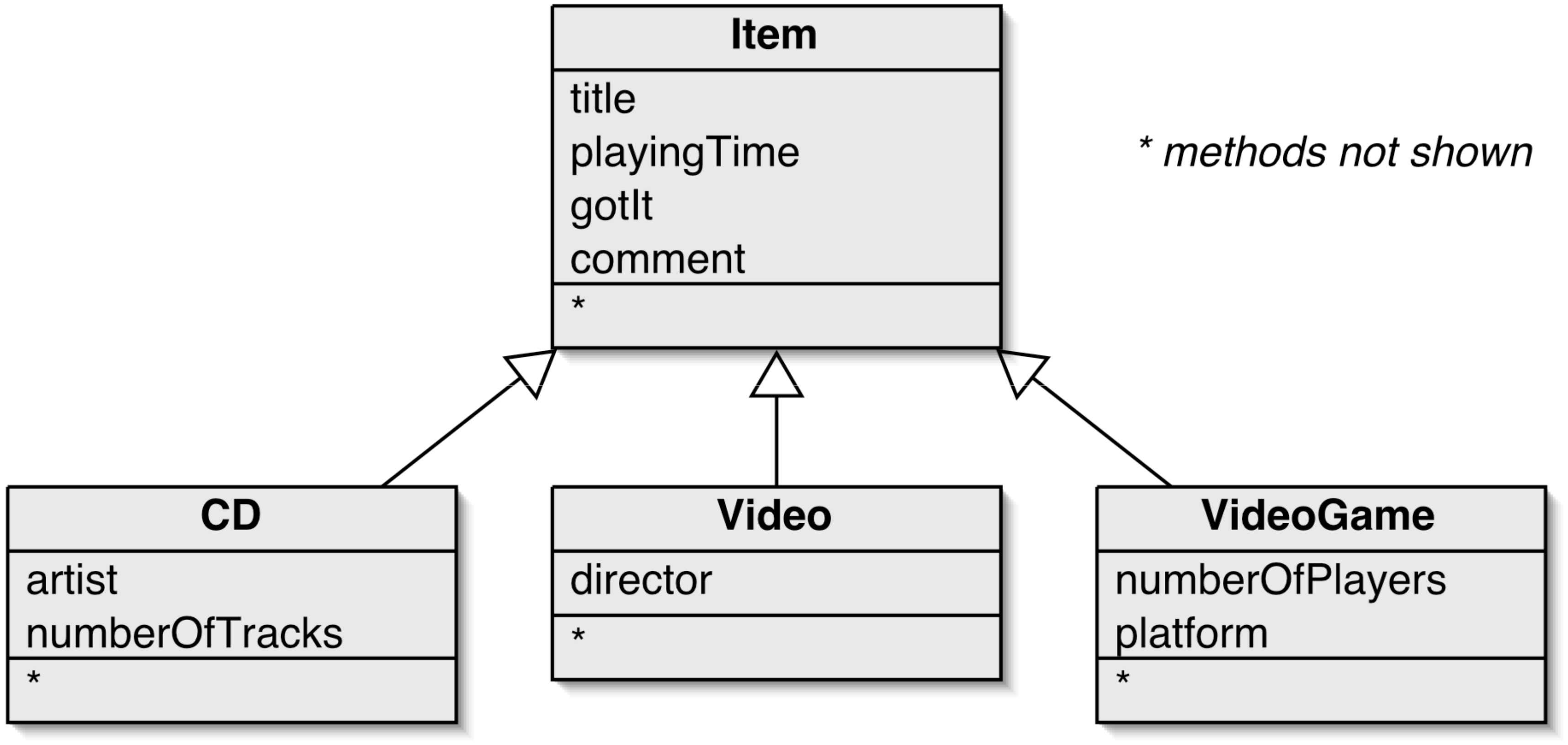
Inheritance and constructors

```
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
}
```

Superclass constructor call

- Subclass constructors must always contain a 'super' call.
- If none is written, the compiler inserts one (without parameters).
  - If the superclass has no constructor without parameters, error occurs.
- Must be the first statement in the subclass constructor.
- Good style to always write super explicitly.

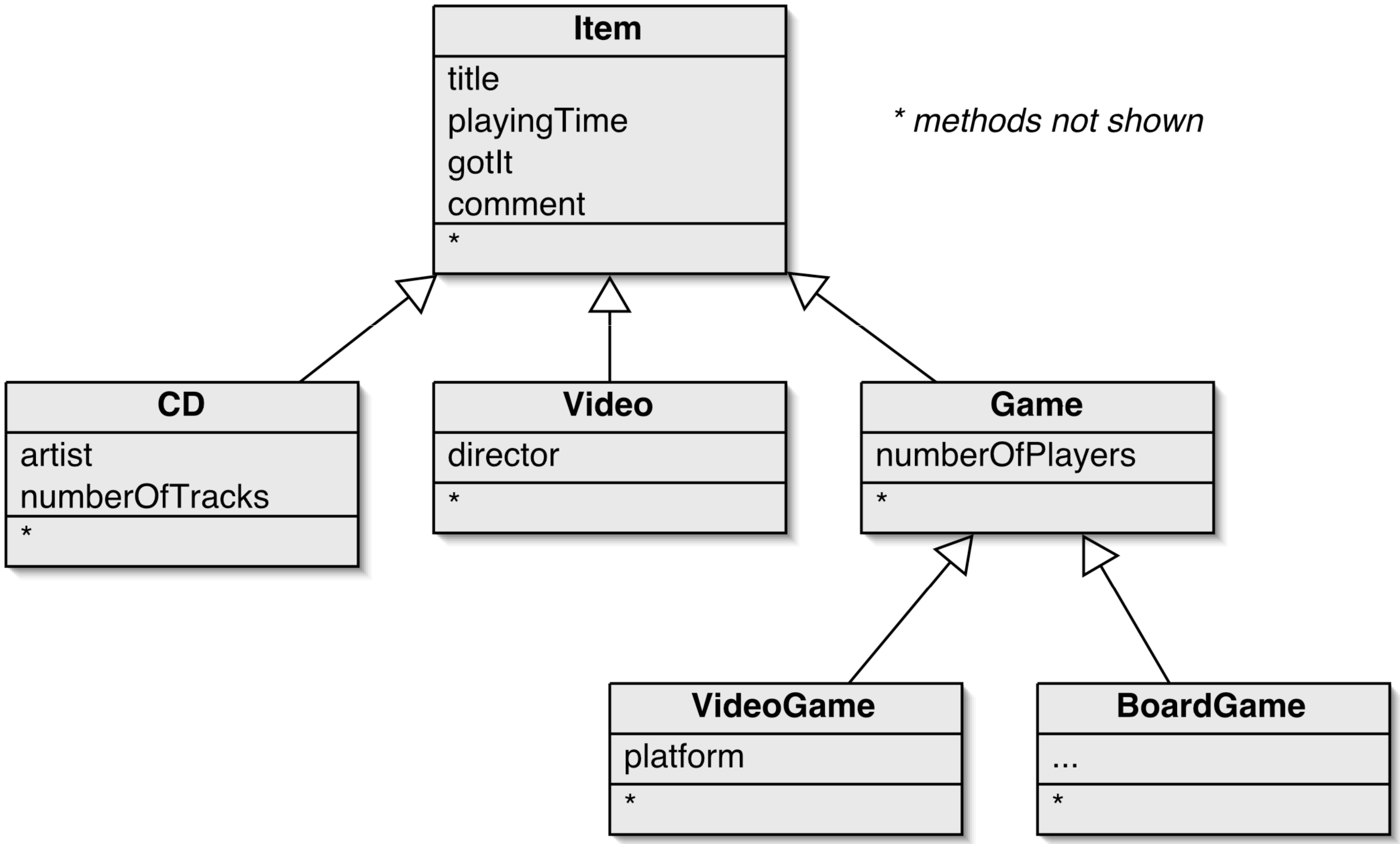
Adding more item types



** methods not shown*

Easy: inheritance lets us reuse existing code

Deeper hierarchies



** methods not shown*

Review (so far)

Inheritance (so far) helps with:

- Avoiding code duplication
- Code reuse
- Easier maintenance
- Extensibility

New Database source code

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

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

    /**
     * Add an item to the database.
     */
    public void addItem(Item theItem)
    {
        items.add(theItem);
    }
    ...
}
```

Compare to earlier slide.

Inheritance avoids code duplication in client!

New Database source code

```
/**
 * Print a list of all currently stored CDs and
 * videos to the text terminal.
 */
public void list()
{
   for(Iterator<Item> iter = items.iterator(); iter.hasNext(); )
   {
       Item item = iter.next();
       item.print();
       System.out.println();   // empty line between items
   }
}
```

Subtyping

First, we had:

```
  public void addCD(CD theCD)
   public void addVideo(Video theVideo)
```

Now, we have:

```
 public void addItem(Item theItem)
```

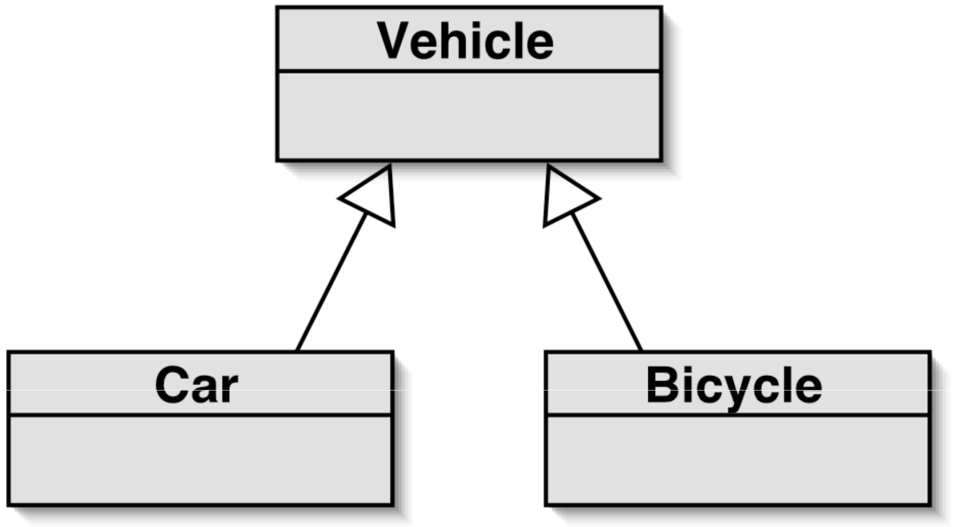
We call this method with:

```
    Video myVideo = new Video(...);
     database.addItem(myVideo);
```

Subclasses and subtyping

- Classes define types.
- Subclasses define subtypes.
- Objects of subclasses can be used where objects of supertypes are required. (This is called **substitution** .)

Subtyping and assignment



subclass objects may be assigned to superclass variables

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

Ok, since cars and bicycles *are* vehicles

Only substitute subtypes

```
Car c1 = new Vehicle();  // error
Car c2 = new Bicycle();  // error
```

Not ok:

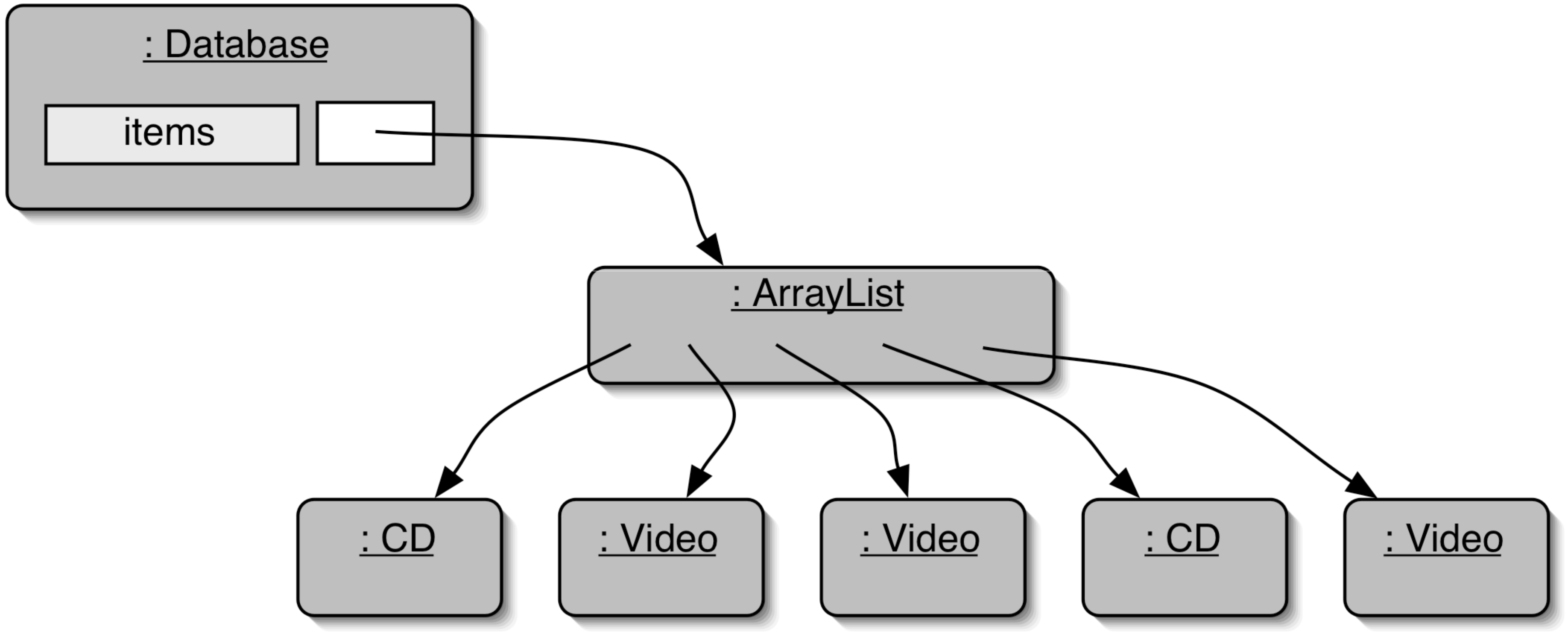
- A vehicle is not a kind of car
- A bicycle is not a kind of car

Subtyping and parameters

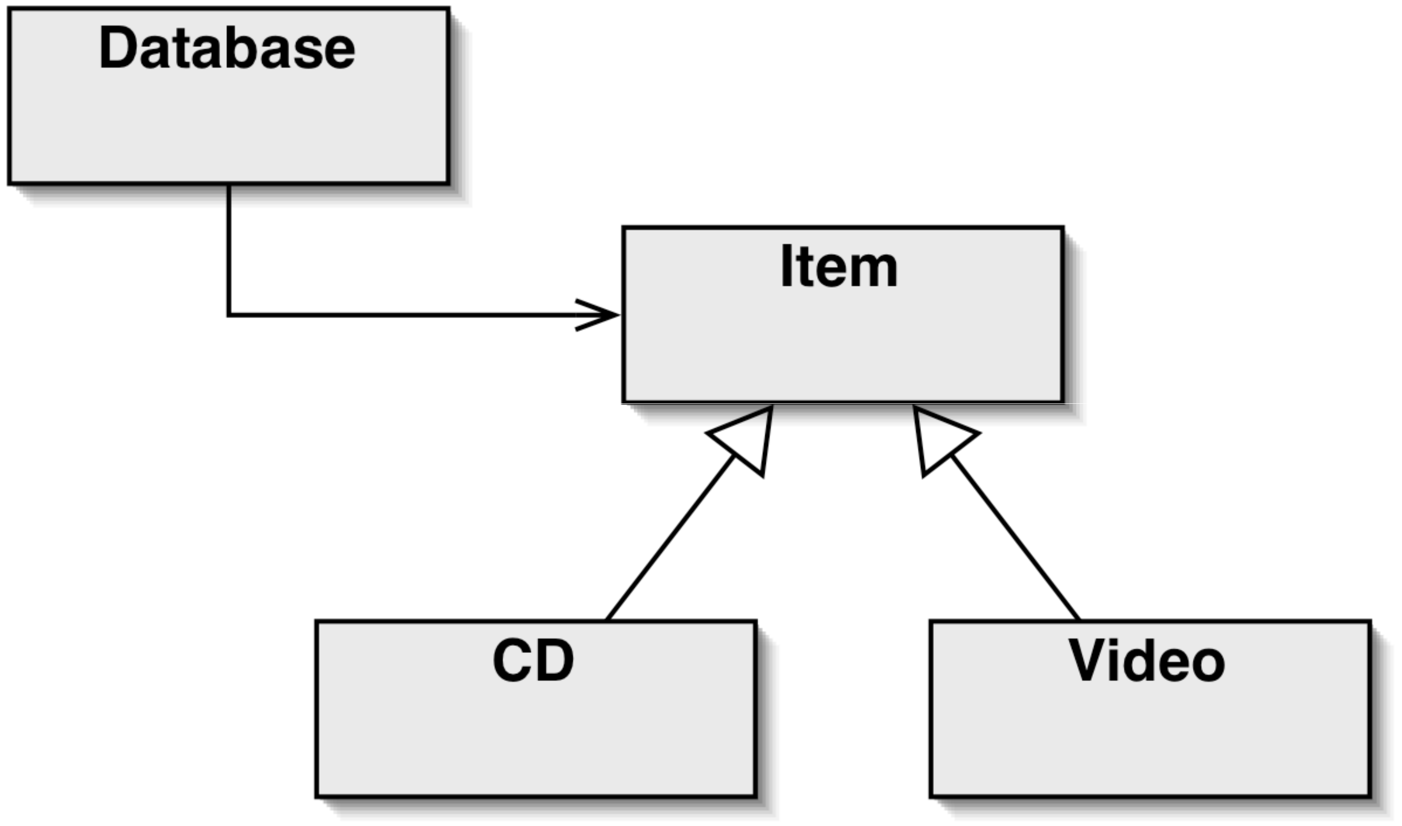
```
public class Database
{
    public void addItem(Item theItem)
    {
        ...
    }
}

Video video = new Video(...);
CD cd = new CD(...);

database.addItem(video);
database.addItem(cd);
```

subclass objects may be passed to superclass parameters

Object diagram



Class diagram



Big arrowheads indicate inheritance

Polymorphic variables

- Object variables in Java are **polymorphic**
  - They can hold objects of more than one type
- They can hold objects of the declared type, or of subtypes of the declared type

- Reminder: polymorphism allows us to write code referring to a superclass, but get run-time behaviour belonging to different subclasses as appropriate

Limitations of inheritance

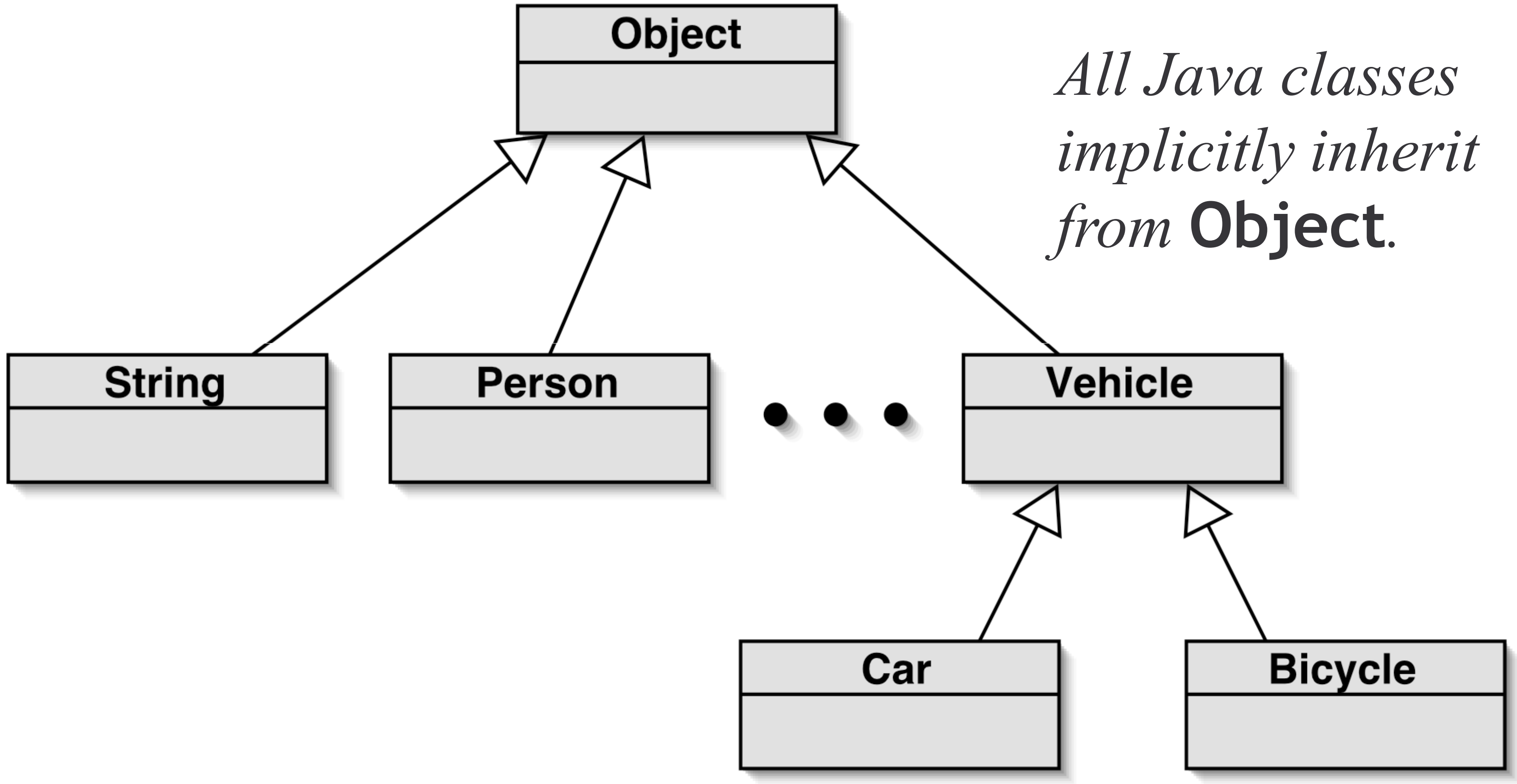
Inheritance is important but it has limitations, including:

- A subclass inherits everything
  - There is no good way to hide unwanted inherited fields or methods
- Inheritance is a static relationship fixed at compile time
  - There is no way to change the relationships between objects at run-time
- A class can only extend one other class

Alternatives to Inheritance

- In many situations there are better alternatives: composition / delegation
  - Many books overemphasise inheritance, many designers overuse it
  - More on this in a lecture on design

The Object class



All Java classes implicitly inherit from **Object**.

Polymorphic collections

- Untyped collections are polymorphic.
- The elements are of type Object.

```
public void add(Object element)
public Object get(int index)
```

- Typed collections (Java 1.5) are polymorphic if their type has subtypes

Casting revisited

- Can assign subtype to supertype.
- Cannot assign supertype to subtype!

```
 Object Iterator.get(int);
String s1 = myList.get(1);
```

error! get() returns Object

- Casting fixes this:

```
  String s1 = (String) myList.get(1);
```

(but only if the element really is a String)

- Remember:
  - Better to use typed collections
  - Casting to different types when getting from collections may indicate bad design

Wrapper classes

- Untyped collections accept any object because all objects are subtypes of Object
- Typed collections can be made for any object type
- Great! But what about primitive types?
  - They are not objects
  - They must be wrapped inside an object to be added to a collection

primitive type	*wrapper class*
int	Integer
float	Float
char	Character
…	…

Wrapper classes

- Let’s add an int to a collection called myCollection

```
...
int i = 18;
Integer iwrap = new Integer(i);
myCollecton.add(iwrap);
...
Integer element = (Integer) myCollection.get(0);
int value = element.intValue();
```

wrap the int value

add the wrapper

retrieve the wrapper

unwrap

Autoboxing

- New in Java 5
- Java automatically casts primitives into their wrapper types and back (unboxing) as needed
- Previous example becomes

```
int i = 18;
myCollecton.add(i);
...
Integer element = (Integer) myCollection.get(0);
```

i automatically wrapped

...

and unwrapped

- Or, if myCollection is typed as <Integer>

```
int i = 18;
myCollecton.add(i);
...
Integer element = myCollection.get(0);
```

Review

- Inheritance allows the definition of classes as extensions of other classes
- Inheritance
  - avoids code duplication
  - allows code reuse
  - simplifies the code
  - simplifies maintenance and extending

Review

- Variables can hold subtype objects (polymorphism)
  - Polymorphism is a key OOP idea
  - Subtypes can be used wherever supertype objects are expected (substitution)
  - Can only substitute subtypes for their supertypes
- Although inheritance is important it is often overused
- Primitive types need to be wrapped before they can be used in collections
  - Prior to Java 5 (un)wrapping was manual
  - In Java 5 (un)wrapping is automatic: called autoboxing.