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 DVDs
  - CD: title, artist, # tracks, playing time, got-it, comment
  - DVD: title, director, playing time, got-it, comment
- allows (later) to search for information or print lists

DoME objects

<table>
<thead>
<tr>
<th>CD</th>
<th>DVD</th>
</tr>
</thead>
<tbody>
<tr>
<td>title</td>
<td>title</td>
</tr>
<tr>
<td>artist</td>
<td>director</td>
</tr>
<tr>
<td>#tracks</td>
<td></td>
</tr>
<tr>
<td>playing time</td>
<td></td>
</tr>
<tr>
<td>got it</td>
<td>got it</td>
</tr>
<tr>
<td>comment</td>
<td>comment</td>
</tr>
</tbody>
</table>
DoME classes

<table>
<thead>
<tr>
<th>CD</th>
<th>DVD</th>
</tr>
</thead>
<tbody>
<tr>
<td>title</td>
<td>title</td>
</tr>
<tr>
<td>artist</td>
<td>director</td>
</tr>
<tr>
<td>numberOfTracks</td>
<td>playingTime</td>
</tr>
<tr>
<td>playingTime</td>
<td>goft</td>
</tr>
<tr>
<td>comment</td>
<td>comment</td>
</tr>
<tr>
<td>setComment</td>
<td>getComment</td>
</tr>
<tr>
<td>getOwn</td>
<td>getOwn</td>
</tr>
<tr>
<td>print</td>
<td>print</td>
</tr>
</tbody>
</table>

top half shows fields

bottom half shows methods

DoME object model

Class diagram

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

CD source code

incomplete (comments!)
### Critique of DoME

- **code duplication**
  - CD and DVD 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

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

Inheritance in Java

```java
public class Item {
    // ... 
}
public class DVD extends Item {
    // ... 
}
public class CD extends Item {
    // ... 
}
```

Superclass

```java
public class Item {
    private String title;
    private int playingTime;
    private boolean gotIt;
    private String comment;
    // constructors and methods omitted.
}
```
Subclasses

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

    // constructors and methods omitted.
}

public class DVD extends Item
{
    private String director;

    // constructors and methods omitted.
}
```

**Inheritance and constructors**

```java
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
}
```

**Superclass constructor call**

- Subclass constructors must always contain a 'super' call.
- If none is written, the compiler inserts one (without parameters)
- Must be the first statement in the subclass constructor.
Review (so far)

Inheritance (so far) helps with:

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

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);
    }
    ...
}
```

avoids code duplication in client!
/**
 * Print a list of all currently stored CDs and DVDs to the text terminal.
 */
public void list()
{
    for(Item item : items)
    {
        item.print();
        // Print an empty line between items
        System.out.println();
    }
}

First, we had:
public void addCD(CD theCD)
public void addDVD(DVD theDVD)
Now, we have:
public void addItem(Item theItem)
We call this method with:
DVD myDVD = new DVD(...);
database.addItem(myDVD);

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.)

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

Subtyping and assignment

subclass objects may be assigned to superclass variables
Subtyping and parameter passing

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

DVD dvd = new DVD(...);
CD cd = new CD(...);
database.addItem(dvd);
database.addItem(cd);

subclass objects may be passed to superclass parameters

Objects First with Java - A Practical Introduction using BlueJ, © David J. Barnes, Michael Kölling

Object diagram

Class diagram

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.
Casting

• Can assign subtype to supertype.
• Cannot assign supertype to subtype!
  ```java
  Vehicle v;
  Car c = new Car();
  v = c; // correct;
  c = v; compile-time error!
  ```

• Casting fixes this:
  ```java
  c = (Car) v;
  ```
  (only ok if the vehicle really is a Car!)

Casting

• An object type in parentheses.
• Used to overcome 'type loss'.
• The object is not changed in any way.
• A runtime check is made to ensure the object really is of that type:
  ```java
  ClassCastException if it isn't!
  ```
• Use it sparingly.

The Object class

All classes inherit from Object.

Polymorphic collections

• All collections are polymorphic.
• The elements are of type Object.
  ```java
  public void add(Object element)
  public Object get(int index)
  ```
Collections and primitive types

- All objects can be entered into collections...
- ...because collections accept elements of type Object...
- ...and all classes are subtypes of Object.
- Great! But what about simple types?

Wrapper classes

- Primitive types (int, char, etc) are not objects. They must be wrapped into an object!
- Wrapper classes exist for all simple types:

<table>
<thead>
<tr>
<th>simple type</th>
<th>wrapper class</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Integer</td>
</tr>
<tr>
<td>float</td>
<td>Float</td>
</tr>
<tr>
<td>char</td>
<td>Character</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

int i = 18;
Integer iwrap = new Integer(i);  // wrap the value
int value = iwrap.intValue();    // unwrap it

In practice, autoboxing and unboxing mean we don’t often have to do this.

Autoboxing and unboxing

```java
private ArrayList<Integer> markList;

... public void storeMark(int mark) {
    markList.add(mark);       // autoboxing
}
```

```java
int firstMark = markList.remove(0);  // unboxing
```
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

- Variables can hold subtype objects.

- Subtypes can be used wherever supertype objects are expected (substitution).