Lecture 27

We show another example of a class hierarchy, a polymorphic data structure and the use of dynamic method binding.

We see that Java's strong- and (mostly) static-typing gives us a problem in getting these kinds of examples to work. Abstract classes and abstract methods come to the rescue.

Inventory for a camera store

- All inventory objects contain a description of the object, an identification number, the quantity on hand and the price
- Lenses have a focal length and they may or may not zoom
- Films have a recommended storage temperature, a speed and the number of exposures
- Cameras may or may not come with a lens, and have a maximum shutter speed and a body colour

Outline class definitions

Pretend that these definitions include constructors and other methods

```
public class InventoryItem
{
   protected String description;
   protected int inventoryNumber;
   protected int quantityOnHand;
   protected int price;
}


public class Lens extends InventoryItem
{
   public void display()
   {  // displays inventoryNumber,
      // quantityOnHand & focalLength
   }

   protected double focalLength;
   protected boolean isZoom;
}
```



```
public class Film extends InventoryItem
{
   public void display()
   {  // displays inventoryNumber,
      // quantityOnHand & numberOfExposures
   }

   protected int recommendedTemp;
   protected int numberOfExposures;
}


public class Camera extends InventoryItem
{
   public void display()
   {  // displays inventoryNumber,
      // quantityOnHand & bodyColour
   }

   protected boolean hasLens;
   protected int maxShutterSpeed;
   protected String bodyColour;
}
```


Displaying the inventory

```
// Assume the inventory is kept in an array
// (a polymorphic data structure)
InventoryItem[] inventory =
                  new InventoryItem[5];

// The array would be filled from a file,
// but we'll do it explicitly
inventory[0] = new Lens(...);
inventory[1] = new Film(...);
inventory[2] = new Lens(...);
inventory[3] = new Camera(...);
inventory[4] = new Camera(...);

// Now exploit dynamic method binding
for (int i = 0; i < inventory.length; i++)
{  inventory[i].display();
}
```

This won't compile!

Why won't it compile?

- Consider this (which also won't compile):

  ```
  InventoryItem invItem = new Lens(...);
  invItem.display();
  ```

- The *compiler* sees that the variable `invItem` is of type `InventoryItem` and complains that this class has no `display` method
  ...even though at *run-time* it is `Lens' display` method that will be invoked
- The compiler is being cautious: it wants to be sure that whatever `invItem` ends up containing will definitely have a `display` method
- It knows that if `InventoryItem` had a `display` method then whatever `invItem` ends up containing will have one (if only by inheritance)

Giving `InventoryItem` a 'dummy' `display` method

- So we must add a `display` method to `InventoryItem`
- We could write a 'dummy' version of `display` and include it in `InventoryItem` (as we did for `talk` in `Animal`), e.g.:

```
public void display()
{  System.out.println(
      "I will never be printed");
}
```

- It's a dummy because it exists only to be overridden
- It is the responsibility of subclasses to provide real definitions of `display`

Abstract version of `InventoryItem`

- A neater solution is to make `InventoryItem` abstract (it can never be instantiated) and make `display` an abstract method (subclasses must override it)

```
public abstract class InventoryItem
{
   public abstract void display();

   protected String description;
   protected int inventoryNumber;
   protected int quantityOnHand;
   protected int price;
}
```

- The abstract class acts as a 'specification': new subclasses are obliged to contain definitions of a `display` method, so the inventory printing program will still work.