Scarborough College
University of Toronto
Fall 1997

CSC A06F: Introduction to Computer Programming

Lecture notes 8:

Relationships between classes

Reading:

- Lewis and Loftus, chapter 8 and section 9.2.

Inheritance

Inheritance is a simple but powerful idea for designing object-oriented systems.

Idea: a class of objects can *inherit* the properties of another class.

E.g.,

- an elephant is a mammal: it has all the properties of a mammal, plus a few extra.
- a Corvette is a car; a car is a vehicle.
- a calculator is a computer.
- a Frame is a Window; a Window is a Container; a Container is a Component.

We try to model the real-world in our object-oriented systems, so we try to model the inheritance relationships as well.

Example: A fancy cash register

Design a new cash register that has the same behaviour as a regular cash register (`ringUpItem` and `summarize`), but also has a new behaviour:
`refundItem`

You could re-write the whole `CashRegister` class, but

- this takes time,
- you might make a mistake,
- would have different versions of the same code,
- this is less elegant.

Instead, you could **extend** the `CashRegister` class.

When you extend a class, you get all of its methods automatically, plus you can **add** new ones.

Extending a class

Also called **deriving** a new class.

The new class is a **subclass** or **child class** . The old class is a **parent class** or **superclass** .

```
class FancyCashRegister extends CashRegister {

   private long totalRefunds;

   // constructor
   public FancyCashRegister() {
      super();           // calls constructor of CashRegister
      totalRefunds = 0;
   }

   // add a new behaviour for refunds
   public void refundItem (String item, long price) {
      totalRefunds += price;
      total -= price;

      // print the item here

      ShowMoney.print(-price, priceLength);
   }
}
```

Constructors

If the subclass does not have a constructor, then it *automatically calls* the constructor of the parent class.

If the subclass ***does*** have a constructor, then you must call the constructor of the parent class yourself.

> Do this by calling the `super()` method, but it must be the first thing you do in the constructor.

Idea is that you initialize the parts of the object defined in parent class first, then initialize the parts for the subclass.

Adding new methods

Simple, just put them in the subclass.

An object can call all the `public` methods of both the subclass and the parent class.

Example:

```
FancyCashRegister fc = new FancyCashRegister();

...

fc.ringUpItem ("bananas", 500);
fc.refundItem ("elephant", 100000);
fc.summarize ();
```

Visibility: the `protected` modifier

In `CashRegister`, we have the instance variable:

```
private long total;
```

Since it's `private`, the subclass can't access it, even though it's defined. A new subclass *doesn't* inherit `private` variables or methods.

But we don't want to make it `public`, because then everyone can look at it.

Use the `protected` modifier instead in `CashRegister`.

```
protected long total;
```

A subclass inherits the `public` and `protected` variables and methods, so it can access them.

Overriding a method

What if the `FancyCashRegister` wants to also print the total refunds when it prints the summary?

We can redefine the `summarize` method (or any inherited method) in `FancyCashRegister`:

```
class FancyCashRegister extends CashRegister {

   ...

   public void summarize () {
      System.out.print ("Refunds    ");
      ShowMoney.print (totalRefunds, priceLength);
      super.summarize();
   }
}
```

Note:

- a call to `cr.summarize()` calls this method.
- `super.summarize()` calls the summarize method of the parent class.

Extending a class further

If we define another type of cash register:

```
class ReallyFancyCashRegister extends FancyCashRegister {

   // the only thing on sale today is elephants
   private String onsale = "Elephants";

   // no constructor, will just call constructor of parent class

   // check if the item is on sale today (returns true if it is)
   public boolean isItemOnSale (String item) {

      return (item.equals (onsale));
   }
}
```

then it inherits everything from `FancyCashRegister` and `CashRegister`.

So, all the the `public` methods can be called:

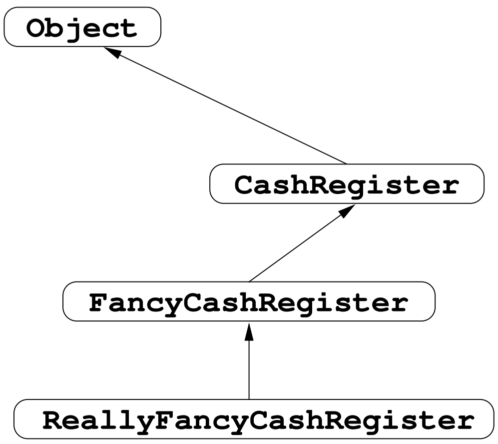
```
   ReallyFancyCashRegister rfcr = new ReallyFancyCashRegister ();

   boolean sale = rfcr.isItemOnSale (item);

   rfcr.refundItem (item, price);

   rfcr.ringUpItem (item, price);
```

Inheritance hierarchies



The `Object` class

Every class inherits from the `Object` class.

`Object` is the great-great-great-…-grandparent of all classes.

Every object is an `Object`

So, every class in Java, whether pre-defined or written by you, is in the same hierarchy!

`Object` provides some very generic methods:

```
boolean equals(Object arg)
String toString()
```

You can override these, or just call them. E.g.,

```
String s = rfcr.toString();
```


Objects of many forms: Polymorphism

A variable that is declared to refer to a particular class is *actually* allowed to refer to any other class that is related to it by inheritance.

A store with three different kinds (forms) of cash register:

```
public class Store {
   public static void main (String[] args) {

      CashRegister[] cr = new CashRegister[3];
      cr[0] = new CashRegister();
      cr[1] = new FancyCashRegister();
      cr[2] = new ReallyFancyCashRegister();

      // which summarize method is called?
      cr[0].ringUpItem("Bananas",100);
      cr[0].summarize ();

      cr[1].ringUpItem("Beans",3999);
      cr[1].summarize ();

      cr[2].ringUpItem("Elephants",100000);
      cr[2].summarize ();

      cr[0] = cr[2];
      cr[0].summarize ();
   }
}
```

Interfaces

An **interface** is a special kind of class definition. It defines methods, but does not give them any content. (So, it's not really a class at all.)

You must **implement** the interface with a class, and provide definitions for all the methods in the interface.

```
interface interface-name {
  abstract-method-declarations
}
```

```
class class-name implements interface-name {
  implementation-of-methods-named-in-interface
}
```

Interfaces provide a **standard** of interaction. If a class implements a certain interface, then you know exactly how to use that class.

You will have to implement pre-defined interfaces. E.g., an `ActionListener`.

Example

`ActionListener` is defined in package `java.awt`. It looks like this:

```
interface ActionListener {
   public void actionPerformed (ActionEvent event);
}
```

You can implement it like this:

```
class ButtonListener implements ActionListener {

   // when a button is pressed, it comes here, and just says so
   public void actionPerformed (ActionEvent e) {
      System.out.println ("Someone pressed the button: ");
      System.out.println (e);
   }
}
```