CS301 W'99 Lecture Notes
Lecture 16

Object-oriented Programming

• A set of programming **techniques**.

• An **architectural** style.

• A **modeling** approach.

What is it?

• Program is structured as collection of **objects** interacting via explicit **interfaces**.

• Objects encapsulate **state**.

• Objects are (usually) grouped into **classes** that share common interface.

• Classes are related by **inheritance**.

• Operations in interface use **dynamic binding**.

Why?

• "Real world" can often by modeled by collection of interacting objects. Classic examples: simulation, user interfaces. But OOP can be used for any kind of programming task.

• Building programs around objects allows model, specification, program to **share** common framework.

• A data object can be added or changed without affecting other existing objects, leading to easier maintenance. (Compare top-down procedural design.)

• Data abstraction/encapsulation enables easier **re-use** of code.

How do objects/classes differ from ADT's?

Classes are a lot like ADT definitions, and objects are a lot like values of the ADT. What's the difference?

• In most OO languages, there is a superficial syntactic difference: each function defined for an object takes the object itself as an implicit argument.

```
s.add(x)     ; OO style
Set.add(s,x) ; ADT style
```

• There is a corresponding change in **metaphor**: instead of applying functions to values, we talk of "sending messages to objects."

• OO languages have some form of **inheritance** and **dynamic binding**.

Important OO Languages: Simula 67, Smalltalk, C++, Java

Differences among languages: Are there types? Is everything an object?

(Note: Some OO languages, e.g., Self, JavaScript, have objects but no classes.)

Inheritance

Often one object class differs only slightly from another one, perhaps previously defined.

To avoid having to define it twice, we might like to **inherit** most of the definition of one class from the other, possibly making just a few alterations. If class A inherits from class B, we say A is a **subclass** of B, and B is a **superclass** of A. This generalizes to an inheritance **hierarchy** among different classes.

At least two kinds of inheritance notions are useful, distinguished by what we mean by "definition." These are not always well-separated in discussions.

Specification Inheritance

Inheritance of specification is relevant where one class has similar external **behavior** (available operations) to the another. This kind of inheritance is usually related to a conceptual "is-a" relationship between the concepts represented by the classes.

For example, in a GUI, we might manipulate "lines," "text," and "bitmaps," all of which are conceptually a specialized kind of "display object." Thus all should respond appropriately to messages like "display yourself" or "translate your screen origin."

Goal is to allow us to manipulate arbitrary collections of display objects **uniformly**, without caring which particular kind of object we have.

Key idea: if B is a subclass of A, should be able to use a B instance wherever an A instance is wanted. (Not vice-versa, since Bs may be able to do things that As cannot.) This is sometimes called "simulation."

Note that the implementations of these operators may differ widely from one subclass to another, and might not be implemented in the common superclass at all.

Java Example

```
abstract class DisplayObject extends Object {
   abstract void draw();
   abstract void translate(int delta_x, int delta_y);
}

class Line extends DisplayObject {
   int x0,y0,x1,y1;  // coordinates of endpoints
   Line (int x0_arg,int y0_arg,int x1_arg,int y1_arg) {
        x0 = x0_arg; y0 = y0_arg;
        x1 = x1_arg; y1 = y1_arg;
   }
   void translate (int delta_x, int delta_y) {
        x0 += delta_x;
        y0 += delta_y;
        x1 += delta_x;
        y1 += delta_y;
   }
   void draw () {
        moveto(x0,y0);
        drawto(x1,y1);
   }
}
```



```
class Text extends DisplayObject {
  int x,y;    // coordinates of origin
  string s;   // text contents
  Text(int x_arg, int y_arg, String s_arg) {
        x = x_arg; y = y_arg;
        s = s_arg;
  }
  void translate (int delta_x,int delta_y) {
       x += delta_x;
       y += delta_y;
  }
  void draw () {
       moveto(x,y);
       write(s);
  }
}

Vector v = new Vector();
v.addElement (new Line(0,0,10,10));
v.addElement (new Text(5,5,"hello"));
for (int i = 0; i < v.size(); i++) {
   DisplayObject d = (DisplayObject) v.elementAt(i);
   d.draw();
}
```

Inheritance of Implementation

Alternatively, we may have two classes whose **implementations** are very similar.

Then we'd like one class to inherit its implementation from the other, to avoid writing the same code twice.

Example: Could handle common code for translation in the superclass.

Note: In general, A can inherit implementation from B even when the **conceptual** object represented by A is **not** a specialization of that represented by B.

Example

```
abstract class DisplayObject extends Object {
   int x0, y0;  // coordinates of object origin
   DisplayObject(int x0_arg, int y0_arg) {
        x0 = x0_arg; y0 = y0_arg;
   }
   abstract void draw();
   void translate(int delta_x,int delta_y) {
        x0 += delta_x;
        y0 += delta_y;
   }
}
```



```
class Line extends DisplayObject {
   int del_x, del_y;  // vector to other endpoint
   Line(int x0_arg, int y0_arg, int x1_arg, int y1_arg) {
        super(x0_arg,y0_arg);
        del_x = x1_arg - x0_arg;
        del_y = y1_arg - y0_arg;
   }
   void draw () {
        moveto(x0,y0);
        drawto (x0+del_x,y0+del_y);
   }
}

class Text extends DisplayObject {
   String s;
   Text(int x0_arg, int y0_arg, String s_arg) {
       super(x0_arg,y0_arg);
       s = s_arg;
   }
   void draw () {
        moveto(x0,y0);
        write(s);
   }
}
```


Extension without code change

In particular, we often want to **extend** an existing system with new features, changing existing code as little as possible. Try to do this by adding a new object class that inherits most of its functionality from an existing class, but implements its own distinctive features.

The key idea here is that calls are always dispatched to the original **receiving** object, so that superclass code can access functionality defined in the **subclasses**.

Example: Consider adding a `translate_and_draw` function for all display objects.

Example

```
abstract class DisplayObject extends Object {
   int x0, y0;  // coordinates of object origin
   DisplayObject(int x0_arg, int y0_arg) {
        x0 = x0_arg; y0 = y0_arg;
   }
   abstract void draw();
   void translate(int delta_x,int delta_y) {
        x0 += delta_x;
        y0 += delta_y;
   }
   void translate_and_draw (int delta_x, int delta_y) {
        translate(delta_x,delta_y);
        draw();
}

...

Vector v = new Vector();
v.addElement (new Line(0,0,10,10));
v.addElement (new Text(5,5,"hello"));
for (int i = 0; i < v.size(); i++) {
   DisplayObject d = (DisplayObject) v.elementAt(i);
   d.translate_and_draw(1,1);
}
```

Overriding in subclasses

Sometimes we want a new subclass to **override** the implementation of a superclass function. Again, the rule that all internal messages go to the original receiver is essential here, to make sure most-specific version of code gets invoked.

Example: Add new `bitmap` object, with its own version of `translate`, which scales the argument.

```
class Bitmap extends DisplayObject {
  int sc;          // scale factor
  boolean[] b;     // bitmap
  Bitmap(int x0_arg,int y0_arg,int sc_arg,boolean[] b_arg) {
        super(x0_arg * sc_arg,y0_arg * sc_arg);
        sc = sc_arg; b = b_arg;
  }
  void translate (int delta_x,int delta_y) {
        x0 += x0 * (delta_x * sc);
        y0 += y0 * (delta_y * sc);
  }
  void draw () {
        moveto(x0,y0);
        blit(b);
  }
}
```

An alternative way to implement `translate` is using the `super` pseudo-variable:

```
void translate (int delta_x, int delta_y) {
        super.translate(delta_x * sc, delta_y * sc);
}
```


Specification vs. Implementation

Often we'd like to use both specification-based and implementation-based inheritance, but the superclasses we want for these purposes may be different.

For example, suppose we want to define a class `DisplayGroup` whose objects are **collections** of display objects that can be translated or drawn as a unit. We want to be able to insert and retrieve the elements of a group just as for objects of the Java library class `Vector`, using `addElement`, `removeElementAt`, `size`, etc.

For specification purposes, our group class should clearly be a subclass of `DisplayObject`, but for implementation purposes, it would be very convenient to make it a subclass of `Vector`.

Some languages permit **multiple inheritance** to handle this problem. Java has only single inheritance, but it also has a notion of **interfaces**; these are like abstract class descriptions with no variables or method implementations at all, and are just the thing for describing **specification** inheritance.

So in Java, we could define an interface `Displayable` rather than the abstract class `DisplayObject`, and make `DisplayGroup` a subclass of `Vector` that **implements** `Displayable`.


```
import java.util.*;

interface Displayable {
  void translate(int delta_x, int delta_y);
  void draw();
}

class Line implements Displayable {
  int x0,y0,x1,y1;  // coordinates of endpoints
  Line (int x0_arg,int y0_arg,int x1_arg,int y1_arg) { ... }
  public void translate (int delta_x, int delta_y) { ... }
  public void draw () { ... }
}

class DisplayGroup extends Vector implements Displayable {
  public void translate(int delta_x, int delta_y) {
     for (int i = 0; i < size(); i++) {
       Displayable d = (Displayable) (elementAt(i));
       d.translate(delta_x,delta_y);  }
  }
  public void draw () {
     for (int i = 0; i < size(); i++) {
       Displayable d = (Displayable) (elementAt(i));
       d.draw(); }
  }
}
...
DisplayGroup d = new DisplayGroup();
d.addElement(new Line(1,2,3,4));
d.addElement(new Line(4,5,6,7));
d.translate(100,200);
d.draw();
```


Alternative Approach

Another approach would be to define `DisplayGroup` as a subclass of `DisplayObject` using an `Vector` **field** to hold the group contents. But then we have to redefine all the (useful) `Vector` methods explicitly (and boringly) for `DisplayGroup`, and pay the cost of extra method calls.

```
class DisplayGroup extends DisplayObject {
  Vector contents;
  DisplayGroup() { contents = new Vector(); }

  void addElement(DisplayObject d) {
    contents.addElement(d);
  }
  DisplayObject elementAt(int index) {
    return (DisplayObject) (contents.elementAt(index));
  }
     ...
}
```

An advantage of this approach is that we can localize the casting of vector contents to the bodies of the `DisplayObject` methods.