

# CS558 Programming Languages

Winter 2008

Lecture 15

# OBJECT-ORIENTED PROGRAMMING

**Object-oriented** programs are structured in terms of **objects**: collections of variables (usually called **fields**) and functions (usually called **methods**).

OOP is particularly appropriate for programs that model discrete real-world processes for simulation or interaction. Idea: one program object corresponds to each real-world object. But OOP can be used for any programming task.

Key characteristics of OOP:

- **Dynamic Dispatch**
- **Encapsulation**
- **Subtyping**
- **Inheritance**

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

Differences among languages: Are there static types? Are there **classes**? Are all values objects?

## PROCEDURAL VS. OO PROGRAMMING

The fundamental control structure in OOP is function call, similar to ordinary procedural programming, but:

- 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) ; procedural style
```

- Corresponding change in **metaphor**: instead of applying functions to values, we talk of “sending messages to objects.”

## DYNAMIC METHOD DISPATCH

A more important difference is that in OOP, the receiving object itself controls how each message is processed. E.g., the effect of `s.add` can change depending on exactly which object is bound to `s`. This is a form of **dynamic overloading**.

Example:

```
s1 = empty ordered-list-set
s2 = empty balanced-tree-set
s = if (...) then s1 else s2
s.add(42);
```

The implementation of the `add` method is completely different in `s1` and `s2`; choice of which runs is determined at runtime.

# CLASSES

In OOP, we typically want to create multiple objects having the same structure (field and method names).

In most OO languages this is done by defining a **class**, which is a kind of template from which new objects can be created.

- Different **instances** of the class will typically have different field values, but all will share the same method implementations.
- Classes are not essential; there are some successful OO languages (e.g. Javascript) in which new objects are created by **cloning** existing **prototype** objects.

## CLASSES VS. ADT's

Class definitions are much like ADT definitions.

- In particular, objects often (though not always) designed so that their data fields can only be accessed by the object's own methods. This kind of **encapsulation** is just what ADT's offer.
- Using encapsulation makes it possible to change the representation or implementation of an object without affecting **client** code that interacts with the object only via method calls. This helps support modular development of large programs.
- Unfortunately, OO programmers often violate encapsulation policies.

## DEFINING SIMILAR CLASSES: SUBTYPING

Often one object class differs only slightly from another one, perhaps previously defined. There are (at least) two useful kinds of similarities: **subtyping** and **inheritance**. (These are often confused.)

**Subtyping** is relevant where one class has similar external **behavior** (available member functions and public fields) to the another, in particular when the objects of one class conceptually form a **subset** of the objects of the other.

- For example, in a GUI, we might manipulate “lines,” “text,” and “bitmaps,” all of which are conceptually a specialized kind of “display object.” (We might say that “a line **is** a display object.”) Thus all should respond appropriately to messages like “display yourself” or “translate your screen origin.”
- Key idea is **principle of safe substitution**: if B is a subtype of A, we should be able to use a B instance wherever an A instance is wanted. (Not vice-versa, since B’s may be able to do things that A’s cannot.) This is sometimes called “simulation.”

## SUBTYPING AND SUBCLASSES

In many OO languages (including Smalltalk, C++, and Java) we declare subtypes by defining **subclasses**. (Java also offers another mechanism; more later).

- Subtyping should obviously be a **transitive** relationship, and so is subclassing. This generalizes to a **hierarchy** among different classes.

## SUBTYPING EXAMPLE

Uniform manipulation of heterogeneous **collections**.

```
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<DisplayObject> v = new Vector<DisplayObject>();
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 = v.elementAt(i);
    d.translate(3,4);
    d.draw();
}

```

## DEFINING SIMILAR CLASSES: INHERITANCE

Classes might also be related because their **implementations** are similar. To avoid having to write the code twice, we might like to **inherit** most of the implementation of one class from the other, possibly making just a few alterations.

In Smalltalk, C++, Java, this is again expressed by making the class that inherits the implementation a **subclass** of the class providing the implementation.

- This works nicely when the inheriting class is also a subtype of the providing class.
- But note: Sometimes we'd like B to inherit implementation from A even when the **conceptual** object represented by B is **not** a specialization of that represented by A; i.e. B is not really a subtype of A. More later.

## EXAMPLE REVISITED

Handle common code for translation in the superclass.

```
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 the course of a lengthy development project, we often want to **extend** an existing program with new features, changing existing code as little as possible. We can 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**.

(In C++, this is only true for methods declared as **virtual**; in Java it is true for all methods by default.)

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<DisplayObject> v = new Vector<DisplayObject>();
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 = v.elementAt(i);
    d.translate_and_draw(3,4);
}
```

## 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 += delta_x * sc; y0 += delta_y * sc;
    }
    void draw () {
        moveto(x0,y0);
        blit(b);
    }
}

```

Another way to implement translate is to use the super pseudo-variable:

```

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

```

## SUBTYPING VS. INHERITANCE

Often we'd like to use both subtyping and inheritance, but the subclassing structure 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 subtyping purposes, our group class should clearly be a subclass of `DisplayObject`, but for inheritance 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 **subtypes**.

## JAVA INTERFACE EXAMPLE

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`.

```
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,int y0,int x1,int y1) { ... }
    public void translate (int delta_x,int delta_y) { ... }
    public void draw () { ... }
}
```

```

class DisplayGroup extends Vector<Displayable>
    implements Displayable {
    public void translate(int delta_x, int delta_y) {
        for (int i = 0; i < size(); i++)
            elementAt(i).translate(delta_x,delta_y);
    }
    public void draw () {
        for (int i = 0; i < size(); i++)
            elementAt(i).draw();
    }
}
...
DisplayGroup d = new DisplayGroup();
d.addElement(new Line(0,0,10,10));
d.addElement(new Line(20,20,40,40));
d.translate(3,4);
d.draw();

```