CS558 Programming Languages

Fall 2015

Lecture 10

Object-oriented Programming

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

Typical key characteristics of OOP:

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

Important OO Languages: Simula 67, Smalltalk, C++, Java, C#, JavaScript, Python, Ruby, Scala, ...

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 **method invocation**, similar to function call in 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 = new ordered-list-set
s2 = new balanced-tree-set
if ... then s = s1 else s = 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 Abstract Data Type (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.

• However, OO programmers often violate encapsulation policies. (For example, object fields may be **public**, allowing them to be accessed from code outside of methods.)

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 dx, int dy);
}

class Line extends DisplayObject {
   int x0,y0,x1,y1;  // coordinates of endpoints
   Line (int x0,int y0,int x1,int y1) {
        this.x0 = x0; this.y0 = y0; this.x1 = x1; this.y1 = y1;
   }
   void translate (int dx, int dy) {
        x0 += dx; y0 += dy; x1 += dx; y1 += dy;
   }
   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, int y, String s) {
        this.x = x; this.y = y; this.s = s;
  }
  void translate (int dx,int dy) {
       x += dx;  y += dy;
  }
  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. To make this useful, we may need to refactor the design.

```
abstract class DisplayObject extends Object {
   int x0, y0;  // coordinates of object origin
   DisplayObject(int x0, int y0) {
        this.x0 = x0; this.y0 = y0;
   }
   abstract void draw();
   void translate(int dx,int dy) {
        x0 += dx; y0 += dy;
   }
}
```

```
class Line extends DisplayObject {
   int del_x, del_y;  // vector to other endpoint
   Line(int x0,int y0,int x1,int y1){
        super(x0,y0);
        del_x = x1 - x0; del_y = y1 - y0;
   }
   void draw () {
        moveto(x0,y0);
        drawto(x0+del_x,y0+del_y);
   }
}

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

Flexibility using Dynamic Dispatch

Our ability to inherit implementation code from a super-class in enhanced by dynamic dispatch.

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. We can define this at the level of the `DisplayObject` class, while still invoking the specific `translate` and `draw` methods from the relevant subclass.

Example

```
abstract class DisplayObject extends Object {
   int x0, y0;  // coordinates of object origin
   DisplayObject(int x0, int y0) {
        this.x0 = x0; this.y0 = y0;
   }
   abstract void draw();
   void translate(int dx,int dy) {
        x0 += dx; y0 += dy;
   }
   void translate_and_draw (int dx,int dy) {
        translate(dx,dy);
        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);
}
```

ADDING SUBCLASSES; OVERRIDING

As a program changes over time, it is easy to add new subclasses as long as they implement the methods expected of them.

Example, we could add `Circle` or `Square` as subclasses of `DisplayObject` without breaking any existing code.

For added flexibility, we can also have a new subclass selectively **override** the implementations of some superclass functions. 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 (for some strange reason).

```
class Bitmap extends DisplayObject {
  int sc;           // scale factor
  boolean[] b;      // bitmap
  Bitmap(int x0,int y0,int sc,boolean[] b) {
        super(x0 * sc,y0 * sc);
        this.sc = sc; this.b = b;
  }
  void translate (int dx,int dy) {
        x0 += dx * sc; y0 += dy * sc;
  }
  void draw () {
        moveto(x0,y0);
        blit(sc,b);
  }
}
```

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

```
void translate (int dx, int dy) {
   super.translate(dx * sc, dy * 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 dx, int dy);
  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 dx,int dy) { ... }
  public void draw () { ... }
}
```

```
class DisplayGroup extends Vector<Displayable>
                   implements Displayable {
  public void translate(int dx, int dy) {
     for (int i = 0; i < size(); i++)
       elementAt(i).translate(dx,dy);
  }
  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();
```

Representation of Objects

In a naive **interpreted** implementation, each object is represented by a heap-allocated record, containing

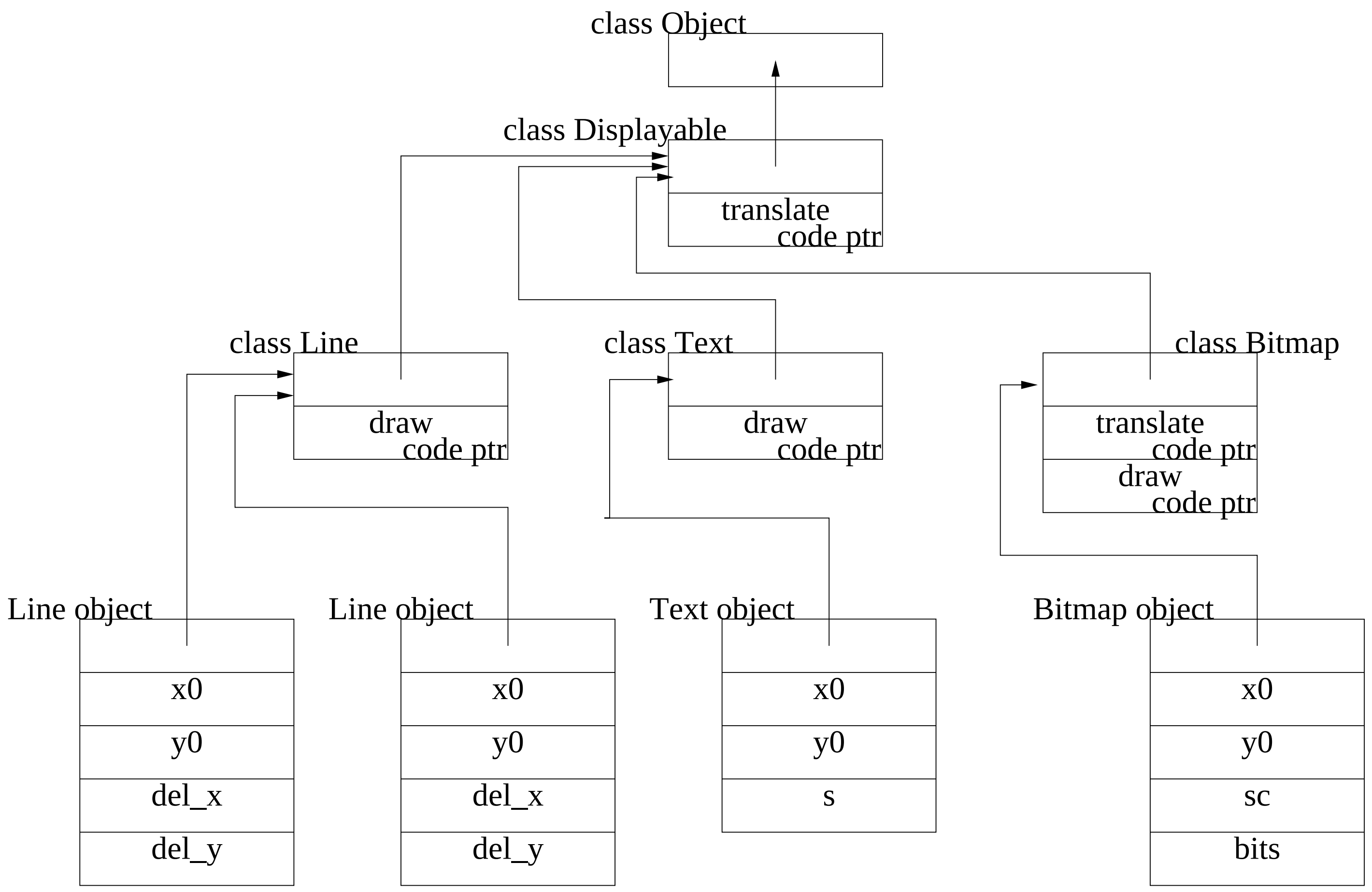
- Name and values of each instance field.
- Pointer to class description record.

Each class is represented by a (essentially static) record with:

- Name and code pointer for each class method.
- Pointer to super-class's record.

EXAMPLE

(based on the code from slides 11,12,14)



Interpreted Implementation of Operations

To perform a message **send** (function call) at runtime, the interpreter does a method lookup, starting from the receiver object, as follows:

• Use class pointer to find class description record.

• Search for method in class record. If found, invoke it; otherwise, continue search in superclass record.

• If no method found, issue “Message Not Understood” or similar error. (Can’t happen if language is statically typed.)

Instance fields are accessed in the object record; `self` always points to the receiver object record; and `super` always points to the superclass.

Can obviously improve on this naive scheme by **caching** results of searches; works well when the same methods are called repeatedly.

Efficient Implementation

How about “compiling” OO languages?

Dynamic binding makes compilation difficult:

• method code doesn’t know the precise class to which the object it is manipulating belongs,

• nor the precise method that will execute when it sends a message.

Instance fields are not so hard.

• Code that refers to instance fields of a given class will actually operate on objects of that class or of a subclass.

• Since a subclass always **extends** the set of instance variables defined in its superclass, compiler can consistently assign each instance variable a fixed (static) offset in the object record; this offset will be the same in every object record for that class and any of its subclasses.

• Compiled methods can then reference variables by offset rather than by name, just like ordinary record field offsets.

(But multiple inheritance systems require more work.)

Compilation (cont.)

Handling message sends is harder, because methods can be overridden by subclasses.

Simple approach: keep a per-class static **method table** (or **vtable**) and "compile" message sends into indirect jumps through fixed offsets in this table.

Example: Classes in our example code all have this vtable structure:

```
             ----------------------
(offset 0) | draw code ptr.      |
             ----------------------
(offset 1) | translate code ptr.|
             ----------------------
```

These tables can get large, and much of their contents will be duplicated between a class and its superclasses. Still, this approach is used by many compiled languages including C++, Java. (Again, multiple inheritance – and Java interfaces – cause complications.)