CS558 Programming Languages

Winter 2013

Lecture 8

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.

Key characteristics of OOP:

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

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

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
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.

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

Two Kinds of Inheritance

Often classes share strong similarities, either in their external **specifications** or in their internal **implementations**, or both.

OO languages typically allow us to declare one class to be a **subclass** of another. The subclass is said to **inherit** from its **superclass**. Subclassing is a transitive relationship, which leads to an inheritance **hierarchy**.

Subclassing can be used to describe both specification and implementation inheritance, which often causes confusion.

Specification inheritance is relevant where one class provides the same methods and fields as another, in particular when the objects of one class conceptually form a **subset** of the objects of the other.

Implementation inheritance is relevant where the method implementations of two classes 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.

Specification Inheritance

Specification inheritance occurs naturally when we model concepts that already form a hierarchy.

• 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 the specification of B inherits from the specification 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."

• In a statically typed language where classes are types, this form of inheritance is called **subtyping**.

Specification Inheritance Example

Uniform manipulation of heterogeneous **collections**.

```
class Displayable():  # this is just documentation
    pass

class Line(Displayable):
    def __init__(self,x0,y0,x1,y1):
       self.x0 = x0  # first endpoint
       self.y0 = y0
       self.x1 = x1  # second endpoint
       self.y1 = y1

    def translate(self,delta_x,delta_y):
       self.x0 = self.x0 + delta_x
       self.y0 = self.y0 + delta_y
       self.x1 = self.x1 + delta_x
       self.y1 = self.y1 + delta_y

    def draw(self):
       moveto(self.x0,self.y0)
       drawto(self.x1,self.y1)
```



```
class Text(Displayable):
  def __init__(self,x,y,s):
     self.x = x  # origin of text
     self.y = y
     self.s = s  # contents of text

  def translate(self,delta_x,delta_y):
     self.x = self.x + delta_x
     self.y = self.y + delta_y

  def draw(self):
     moveto(self.x,self.y)
     write(self.s)

# a tuple of various Displayables
v = (Line(0,0,10,10), Text(5,5,"Hello"))

for d in v:
   d.translate(3,4)
   d.draw()
```


Inheriting Implementation

Inheritance of implementations is about code re-use: we can write code just once (in a super-class) and use it in all the subclasses.

• This works nicely when the inheriting class also inherits the specification 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.

• In any case, we may need to do some refactoring in order to maximize code re-use.

Example revisited, handling translation task in the superclass code...


```
class Displayable():
    def __init__(self,x0,y0):
        self.x0 = x0  # coordinates of object origin
        self.y0 = y0

    def translate(self,delta_x,delta_y):
        self.x0 = self.x0 + delta_x
        self.y0 = self.y0 + delta_y

class Line(Displayable):
    def __init__(self,x0,y0,x1,y1):
       super().__init__(x0,y0) # first endpoint
       self.delta_x = x1 - x0  # vector to second endpoint
       self.delta_y = y1 - y0

    def draw(self):
       moveto(self.x0,self.y0)
       drawto(self.x0 + self.delta_x,self.y0 + self.delta_y)

class Text(Displayable):
    def __init__(self,x,y,s):
       super().__init__(x,y)
       self.s = s  # contents of text

    def draw(self):
       moveto(self.x0,self.y0)
       write(self.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 **dynamically dispatched** to the original **receiving** object, so that superclass code can access functionality defined in the **subclasses**.

(In C++, dynamic dispatch is only used for methods declared as **virtual**; in most OO languages it is true for all methods by default.)

Example: Consider adding a `translate_and_draw` function for all display objects. Although this function is defined in the superclass, the `draw` code it invokes lives in the subclasses.

EXAMPLE

```
class Displayable():
    def __init__(self,x0,y0):
        self.x0 = x0  # coordinates of object origin
        self.y0 = y0

    def translate(self,delta_x,delta_y):
        self.x0 = self.x0 + delta_x
        self.y0 = self.y0 + delta_y

    def translate_and_draw(self,delta_x,delta_y):
        self.translate(delta_x,delta_y)
        self.draw()
...
v = (Line(0,0,10,10), Text(5,5,"Hello"))
for d in v:
   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(Displayable):
    def __init__(self,x0,y0,sc,bits):
        super().__init__(x0 * sc,y0 * sc)
        self.sc = sc
        self.bits = bits

    def translate(self,delta_x,delta_y):
        self.x0 = self.x0 + self.sc * delta_x
        self.y0 = self.y0 + self.sc * delta_y

    def draw(self):
        moveto(self.x0,self.y0)
        blit(self.bits)
```

Another way to implement `translate` is to invoke the super-class method explicitly:

```
    def translate(self,delta_x,delta_y):
        super().translate(self.sc * delta_x, self.sc * delta_y)
```

Subclassing: Specification vs. Implementation

Often we'd like to use inheritance for both specification and impelementation — 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 `Displayable`s that can be translated or drawn as a unit. We want to be able to insert and manipulate the elements of a group just as for objects of the Python library class `list`, using `append`, `del`, etc.

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

Multiple Inheritance Example

Some languages, like Python, permit **multiple inheritance** to handle this situation:

```
class DisplayGroup(Displayable,list):
    def translate(self, delta_x, delta_y):
        for d in self:
            d.translate(delta_x,delta_y)

    def draw(self):
        for d in self:
            d.draw()

d = DisplayGroup(50,50)
d.append(Line(0,0,10,10))
d.insert(0,Line(20,20,40,40))
d.reverse()
d.translate_and_draw(3,4)
```


Alternative Approaches

Multiple inheritance introduces semantic complications and poses some implementation challenges, so many OO languages support it in only limited ways or not at all.

For example, Java has only single inheritance, but it also has a notion of **interfaces**; these are like class descriptions with no fields or method implementations at all, and are just the thing for describing specifications.

- If we implemented our example in Java, we might treat `Displayable` as an interface and make `DisplayGroup` a subclass of (only) `Vector` (the Java library equivalent of `list`).

Representation of Objects

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

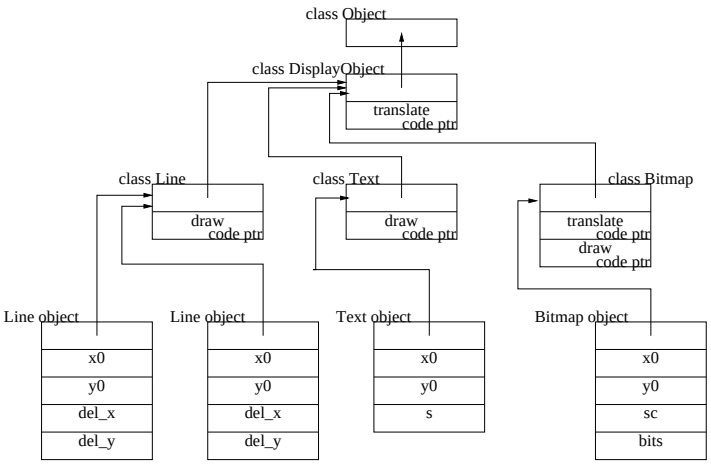
- Name and values of each instance field.
- Pointer to class description record.

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 12 and 16)



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.)