Object-oriented Programming

Object-oriented programs are structured in terms of **objects**; collections of variables and operations. Typically, we want many objects (created dynamically) having the same set of variables and operators but with different variable values. Objects are like ADT instances; the ADT itself is usually called a **class**.

Object-oriented programming 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.

How do objects/classes differ from ADT's?

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

  \[
  \text{s.add(x)} ; \text{ OO style} \\
  \text{Set.add(s,x); ADT style}
  \]

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

- OO languages have some form of **inheritance**.

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

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

(Some OO languages, e.g., 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 B inherits from class A, we say B is a **subclass** of A, and A is a **superclass** of B. 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.

Inheritance of specification is relevant where one class has similar external **behavior** (available operations) to the other. 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. This is essentially a form of **dynamic overloading**, applied consistently to **every** operator in the language.

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.
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);
    }
}

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

Example

abstract class DisplayObject extends Object {
    int x0, y0; // coordinates of 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.

(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.

```java
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(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.

```java
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:

```java
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.
Interface Example

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) {
        for (i = 0; i < size(); i++) {
            Displayable d = (Displayable) (elementAt(i));
            d.translate(delta_x,delta_y);
        }
    }
    public void draw () {
        for (i = 0; i < size(); i++) {
            Displayable d = (Displayable) (elementAt(i));
            d.draw();
        }
    }
}
class DisplayGroup extends Vector implements Displayable {
    public void translate(int delta_x, int delta_y) {
        for (i = 0; i < size(); i++) {
            Displayable d = (Displayable) (elementAt(i));
            d.translate(delta_x,delta_y);
        }
    }
    public void draw () {
        for (i = 0; i < size(); i++) {
            Displayable d = (Displayable) (elementAt(i));
            d.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();

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.

Implementation of Objects

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

- Name and values of each instance variable.
- 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.

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” error.
  (Can’t happen in strongly-typed languages; more later.)

Instance variables are accessed in the object record. Pseudo-variable this always points to the receiver object record; super always points to the super-class.
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 variables are not so hard.

- Method code can only access variables in objects of its own class or 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.

(Multiple inheritance schemes – and Java interfaces – cause problems.)

Compilation (cont.)

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

Simple approach: keep a per-class static method table and “compile” message sends into indirect jumps through fixed offsets in this table.

But these tables can get very large, and much of their contents will be duplicated between a class and its superclasses.

Alternative: Use naive lookup scheme, but keep a dynamic method cache recording the

pairs that have been discovered by recent lookups. Since the same methods tend to be called repeatedly on the same object classes, this can speed things up a lot.

These “compilation” schemes are further hindered because OO environments are often very dynamic – new versions of classes can be reloaded at any time – so frequent recompilation and cache flushing may be needed.

Strong Typing vs. Polymorphism

Smalltalk is untyped and every value is an object; this makes it very easy to use “container” objects. But lack of compile-time typing allows “Message Not Understood” errors at run-time, and treating base types like integers as objects can be inefficient.

C++ and Java are strongly typed, and distinguish base types from objects: this makes them more secure, but less flexible.

```java
// class Integer extends Object

class List {
    Object item;
    List next;
    List(Object i, List n) {
        item = i; next = n;
    }
}

static List reverse(List l) { ... }

List l = new List(new Integer(1),
                  new List(new Integer(2),
                             new List(new Integer(3),
                                        null)));

List l1 = reverse(l);
int i = ((Integer)l1.item).intValue();
```

Typing and Inheritance

Inheriting functions that take or return objects causes typing problems. To maintain soundness, (original) C++ and Java require that an overriding function definition has the same argument and return types as the base class function it overrides.

What we’d like to write:

```java
class Point {
    public int x,y;
    Point(int x, int y) {
        this.x = x; this.y = y;
    }
    Point copy() {return new Point(x,y);}
    boolean equal(Point p1) { // ILLEGAL Java
        return (p1.x == x && p1.y == y);
    }
}

class ColorPoint extends Point {
    public int color;
    ColorPoint(int x, int y, int c) {
        super(x,y);
        color = c;
    }
    ColorPoint copy() { return new ColorPoint(x,y,color); // ILLEGAL Java }
    boolean equal(ColorPoint p1) { // ILLEGAL C++
        return super.equal(p1) && color == p1.color;
    }
}
```
Problems with Copy

Point p = new Point(10,20);
ColorPoint cp = new ColorPoint(30,40,Red);
Point q = new ColorPoint(50,60,Blue);

Point p1 = p.copy(); // want Point.copy
ColorPoint cp1 = cp.copy(); // want ColorPoint.copy
Point q1 = q.copy(); // want ColorPoint.copy

In Java and original C++, the overriding definition of copy in ColorPoint isn’t legal. But there’s no fundamental reason why it shouldn’t work, and current C++ standard allows it (though some compilers still might not).

Runtime Type Testing

Java does provide an alternative using runtime type testing:

```java
class ColorPoint extends Point {
    boolean equal(Point p1) {
        if (p1 instanceof ColorPoint) {
            return super.equal(p1) &&
                color == ((ColorPoint) p1).color;
        } else {
            return super.equal(p1);
        }
    }
}
```

C++ Value Model and Inheritance

Unlike Java, C++ uses the “value model” for representing objects. So in declarations like this

```cpp
Point p(10,20);
ColorPoint cp(30,40,Red);
```

variables p and cp represent actual instance records, not pointers to records. In order to allocate storage space for such instances, the compiler uses the declared type of the variable, so p has space for only x and y, not col.

Now, an assignment like

```cpp
... p = cp; ...
```

works by truncating the ColorPoint to fit in the space of a Point; the col field is lost.

Not likely to be what you wanted!

So, in practice, C++ programmers routinely use explicit pointers to objects instead:

```cpp
Point *p = new Point(10,20);
ColorPoint *cp = new ColorPoint(30,40,Red);
```

Now assignment is a pointer assignment rather than a record copy.

Problems with Equal

Point p = new Point(10,20);
ColorPoint cp = new ColorPoint(30,40,Red);
ColorPoint cp1 = new ColorPoint(30,40,Blue);
Point p1 = cp1;
cpl.equal(cp); // (a) want ColorPoint.equal
cpl.equal(p1); // (b) ColorPoint.equal or Point.equal?
cpl.equal(p); // (c) only Point.equal can work

Problem with equal is more fundamental. In example, first call makes sense, but what about others?

In C++, overriding definition of equal is illegal. If ColorPoint.equal were allowed to override Point.equal it would be executed in all three cases. But it expects its argument to be a ColorPoint with a col field. This is fine for (a) and (b), but we’d get a runtime error (core dump) for (c).

Java treats the two versions of equal as being statically overloaded, based on the declared type of their arguments. No overriding occurs. This gives ColorPoint.equal for (a), but Point.equal for (b) and (c).