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 Overloading**
- **Encapsulation**
- **Subtyping**
- **Inheritance**

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

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.”

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; ADT’s vs. OOP

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.

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.”

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.

---

**Example**

Subtyping allows us to manipulate arbitrary collections of display objects uniformly, without caring which particular kind of object we have.

Note that the implementations of the methods may be completely different in different subtypes.

```java
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;
    }
    void draw () {
        moveto(x0,y0);
        drawto(x1,y1);
    }
}
```

---

**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: can handle common code for translation in the superclass.

```java
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;
    }
}
```
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. 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);
}
```
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, 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.

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.

Alternative Approach

Another approach would be to define DisplayGroup as a subclass of DisplayObject using a 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.

```java
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.

Representation 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.
- Pointer to super-class’s record.

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

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

Here is an example based on the code from slides 6, 7, and 12.
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” 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 superclass.

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.

- Code that refers to instance variables 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.

(Central inheritance schemes 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 (or vtable) and “compile” message sends into indirect jumps through fixed offsets in this table.

Example: Classes in slides 6,7,12 all have this vtable structure:

```
| Offset 0 | Draw code ptr. |
| Offset 1 | Translate code ptr. |
```

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

Alternative: Use naive lookup scheme, but keep a dynamic method cache recording the (receiver class, code pointer) 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.

C++ Value Model and Inheritance

Unlike Java, C++ uses the “direct model” for representing object values.

So in declarations like this:

```
DisplayObject x(10,20);
Line y(30,40,2,4);
```

variables x and y 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 x has space for only x0 and y0, not del.x and del.y.

Now, an assignment like

```
... x = y; ...
```

works by truncating the Line to fit in the space of a DisplayObject; the del.x and del.y fields are lost.

Not likely to be what you wanted!

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

```
DisplayObject *x = new DisplayObject(10,20);
Line *y = new Line(30,40,2,4);
```

Now assignment is a pointer assignment rather than a record copy.
Types vs. Classes

As noted, many languages use a single class hierarchy to describe both subtyping and inheritance.

- Historically, it took a long time for language designers to figure out that subtyping and inheritance are fundamentally distinct ideas.
- From a software engineering point of view, inheritance is a mixed blessing, because it tends to break encapsulation.
- Availability of interfaces in Java and C# is a major improvement.

We can define sensible subtyping rules without reference to classes at all, just based on object structure.

In fact, subtyping makes sense even without static types, if we think of it as a discipline that will prevent our getting runtime errors like “method not understood” or “field not defined.”

Subtyping Rules

We extend our language’s type system with a subsumption rule:

\[
\frac{T \vdash e : t^1 \quad t^1 < : t}{T \vdash e : t}
\] (Sub)

As usual, we want this rule to be sound, i.e., using it to prove a program type-correct won’t allow well-typed programs to go wrong at runtime. In effect, this codifies the principle of safe substitution.

Now we must define the \(<:\) relation so that subsumption remains sound. We assume

- \(T \vdash e : t\) (Reflexive)
- \(t^1 < : t^2 \quad t^2 < : t^3\) (Transitive)

Two easy rules for records:

\[
\begin{align*}
\{l_1 : t_1, \ldots, l_n : t_n\} < : \{l_1 : t_1, \ldots, l_n : t_n\} & \tag{W} \\
\{l_1 : t_1, \ldots, l_k : t_k, \ldots, l_n : t_n\} < : \{l_1 : t_1, \ldots, l_k : t_k, \ldots, l_n : t_n\} & \tag{E}
\end{align*}
\]

The (Width) rule says that we can always make a subtype by adding fields to a record. The (Depth) rule says that we can always make a subtype by replacing the type of any field with a subtype.

(Also, we assume that order in records doesn’t matter.)

Principles of Subtyping

Let’s model object interfaces by record types with named fields and methods.

Example (corresponding to slides 8 and 9):

```latex
type DisplayObject =
  { x0 : int ref,
    y0 : int ref,
    draw : unit \to \text{void},
    translate : int \times int \to \text{void} }

type Line =
  { x0 : int ref,
    y0 : int ref,
    del_x : int ref,
    del_y : int ref,
    draw : unit \to \text{void},
    translate : int \times int \to \text{void} }
```

This notation is similar to ML.

We write \(t^1 < : t\) to mean \(t^1\) is a subtype of \(t\). For example, we would expect that Line \(< : \text{DisplayObject}\).
Functions (cont.)

The rule we really want is:
\[
t_1 < \cdot t'_1 < t_2 \wedge t'_1 < t_1 \\
(Fun)
\]

For example, given

\[
f : \text{DisplayObj} \rightarrow \text{DisplayObj} \\
h : \text{Object} \rightarrow \text{Line}
\]

(where Object is the universal supertype), we could validly replace \(f\) by \(h\).

We say that the \(<\cdot\) relation is **covariant** on the result type of functions, and **contravariant** on the argument type.

This rule suffices for type safety. But as a practical matter, most statically-typed object-oriented languages put even stronger restrictions on function subtypes.

- Both Java and C++ require that the argument types of a function overridden in a subclass be **equal** to those in the superclass definition. (Functions of the same name but with different argument types are treated as distinct, **statically** overloaded, functions.)
- Java also requires this for the result type; C++ allows the result in the overriding method to be a subclass of the superclass result.

Strong Typing vs. Polymorphism

Smalltalk has no static types 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 ll = reverse(l);
int i = ((Integer)ll.item).intValue();
```

Mutable cells

What about subtyping for **mutable** variables (ref cells) and array elements?

When can we safely replace a variable \(x : t_x\) by a variable \(y : t_y\)? A mutable variable can appear in two contexts:

- It can be stored into, e.g. \(x := e\). To safely replace \(x\) by \(y\), \(y\) must able to hold any value that \(x\) can, i.e., we must have \(t_x < \cdot t_y\).
- It can be fetched from, e.g. \(z := x\). To safely replace \(x\) by \(y\), the value in \(y\) must be containable by \(z\), which expects a value from \(x\), i.e., we must have \(t_y < \cdot t_x\).

Combining these requirements, we see that \(t_x\) and \(t_y\) must be equal. So **no** non-trivial subtyping should be permitted on references and array elements.

Java appears to break this rule. If \(B\) is a subclass of \(A\), then \(B[]\) is treated as a subtype of \(A[]\). This is fine when fetching, but can be unsound when storing, because it allows an A value to be stored as an element of an B[], e.g.

```java
B[] bx = new B[100];
A[] ax = bx; // permitted because B[] <: A[]
ax[0] = new A(); // oops!;
```

To guard against this, Java inserts an (expensive) **runtime** check on every array store operation to make sure that the stored value actually belongs to the same class as the array.