What is OOP? (part I)

The object idea:
- bundling of data (data members) and operations (methods) on that data
- restricting access to the data

An object contains:
- data members : arranged as a set of named fields
- methods : routines which take the object they are associated with as an argument
  (known as member functions in C++)
- constructors : routines which create a new object

A class is a construct which defines the data, methods and constructors associated with all of its instances (objects).
The *inheritance* and *dynamic binding* ideas:

- classes can be extended (*inheritance*):
  - by adding new fields
  - by adding new methods
  - by *overriding* existing methods (changing behavior)

If class B extends class A, we say that B is a *subclass or derived* class of A, and A is a *superclass or base* class of B.

- dynamic binding: wherever an instance of a class is required, we can also use an instance of any of its subclasses; when we call one of its methods, the overridden versions are used.

- There should be an *is-a* relationship between a derived class and its base class.
in class-based OOLs, each object is an instance of a class (Java, C++, C#, Ada95, Smalltalk, OCaml, etc.)

in prototype-based OOLS, each object is a clone of another object, possibly with modifications and/or additions (Self, NewtonScript, Javascript)

- Clones (in this context) are **not** copies.
- Clones inherit fields from the prototype object.
- Changes to prototype object (e.g., assignments) propagate to the clone.
- Clones can modify, add, remove, or hide fields (language dependent).
- Usually only the *changes* are stored in the clone object.
Other common OOP features

- multiple inheritance (inheriting from more than one parent class/object)
  - C++
  - Java (of interfaces only)
  - problem: how to handle diamond shaped inheritance hierarchy

- classes often provide package-like capabilities:
  - visibility control
  - ability to define types and classes in addition to data fields and methods
Java Features

- an imperative language (like C++, Ada, C, Pascal)
- is interpreted (like Scheme, APL)
- can be compiled
- is garbage-collected (like Scheme, ML, Smalltalk, Eiffel, Modula-3)
- is object-oriented (like Eiffel, more so than C++, Ada)
- a successful hybrid for a specific-application domain
- a reasonable general-purpose language for non-real-time applications

Work in progress: language continues to evolve
- C# is latest, incompatible variant
Original design goals (white paper 1993)

- simple
- object-oriented (inheritance, polymorphism)
- distributed
- interpreted
- multi-threaded
- robust
- secure
- architecture-neutral

Obviously, “simple” was dropped.
Critical concern: write once – run everywhere

Consequences:

- portable interpreter
- definition through virtual machine: the JVM
- run-time representation has high-level semantics
- supports dynamic loading
- high-level representation can be queried at run-time to provide reflection
- dynamic features make it hard to fully compile, safety requires numerous run-time checks
Conventional imperative languages are fully compiled:

- run-time structure is machine language
- minimal run-time type information
- language provides low-level tools for accessing storage
- safety requires fewer run-time checks because compiler (least for Ada and somewhat for C++) can verify correctness statically
- languages require static binding, run-time image cannot be easily modified
- different compilers may create portability problems
Notable omissions

- no operator overloading (syntactic annoyance)
- no separation of specification and body
- no enumerations until latest language release
- no generic facilities until latest language release
Most statements are like their C counterparts:

- `switch` (including C’s falling through behavior)
- `for`
- `if`
- `while`
- `do ... while`
- `break` and `continue`
  
  • Java also has *labeled* versions of `break` and `continue`, like Ada.

- `return`

Java has no `goto`!
class HelloWorld {
    public static void main (String[] args) {
        System.out.println("Hello, world");
    }
}
Encapsulation of type and related operations

class Point {
    private double x, y;  // private data members

    public Point (double x, double y) {  // constructor
        this.x = x;    this.y = y;
    }

    public void move (double dx, double dy) {
        x += dx;    y += dy;
    }

    public double distance (Point p) {
        double xdist = x - p.x, ydist = y - p.y;
        return Math.sqrt(xdist * xdist + ydist * ydist);
    }

    public void display () { ... }
}
class ColoredPoint extends Point {
    private Color color;

    public ColoredPoint (double x, double y,
            Color c) {
        super(x, y);
        color = c;
    }

    public ColoredPoint (Color c) {
        super(0.0, 0.0);
        color = c;
    }

    public Color getColor () { return color; } 

    public void display () { ... }  // now in color! 
}
Point p1 = new Point(2.0, 3.0);
ColoredPoint cp1 = new ColoredPoint(2.0, 3.0, Blue);

Point p2 = p1; // OK
Point p3 = cp1; // OK

ColoredPoint cp2 = cp1; // OK
ColoredPoint cp3 = p1; // Error

cp1.move(1.0, 1.0); // cp1 and p3 affected

p1.display(); // Point’s display
cp1.display(); // ColoredPoint’s display
p3.display(); // ColoredPoint’s display
The same classes, translated into C++:

class Point {
   double m_x, m_y;  // private data members

public:

   Point (double x, double y)  // constructor
      : m_x(x), m_y(y) { }

   virtual ~Point () { }

   virtual void move (double dx, double dy) {
      m_x += dx;  m_y += dy;
   }

   virtual double distance (const Point& p) {
      double xdist = m_x - p.m_x, ydist = m_y - p.m_y;
      return sqrt(xdist * xdist + ydist * ydist);
   }

   virtual void display () { ... }
};
class ColoredPoint : public Point {
    Color color;

public:

    ColoredPoint (double x, double y, Color c) : Point(x, y), color(c) {
        color = c;
    }

    ColoredPoint (Color c) : Point(0.0, 0.0), color(c) { }

virtual Color getColor () { return color; }

virtual void display () { ... }  // now in color!
};
Point *p1 = new Point(2.0, 3.0);
ColoredPoint *cp1 = new ColoredPoint(2.0, 3.0, Blue);

Point *p2 = p1; // OK
Point *p3 = cp1; // OK

ColoredPoint *cp2 = cp1; // OK
ColoredPoint *cp3 = p1; // Error

cp1->move(1.0, 1.0); // cp1 and p3 affected

p1->display(); // Point’s display
cp1->display(); // ColoredPoint’s display
p3->display(); // ColoredPoint’s display
A typical implementation of a class in C++; using **Point** as an example:

<table>
<thead>
<tr>
<th>Point instance</th>
<th>Point vtable</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>d’tor</td>
</tr>
<tr>
<td>x</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td></td>
</tr>
<tr>
<td></td>
<td>move</td>
</tr>
<tr>
<td></td>
<td>distance</td>
</tr>
<tr>
<td></td>
<td>display</td>
</tr>
</tbody>
</table>

- **Point version**
- **Point version**
- **Point version**
- **Point version**
For `ColoredPoint`, we have:

<table>
<thead>
<tr>
<th>ColoredPoint instance</th>
<th>ColoredPoint vtable</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>d’tor</td>
</tr>
<tr>
<td></td>
<td>ColoredPoint version</td>
</tr>
<tr>
<td>x</td>
<td>move</td>
</tr>
<tr>
<td></td>
<td>Point version</td>
</tr>
<tr>
<td>y</td>
<td>distance</td>
</tr>
<tr>
<td></td>
<td>Point version</td>
</tr>
<tr>
<td>color</td>
<td>display</td>
</tr>
<tr>
<td></td>
<td>ColoredPoint version</td>
</tr>
<tr>
<td></td>
<td>getColor</td>
</tr>
<tr>
<td></td>
<td>ColoredPoint version</td>
</tr>
</tbody>
</table>

Non-virtual member functions are never put in the vtable.
Method modifiers

- **access modifiers:**
  - **public** - method is visible to external classes and all packages
  - **protected** - method is visible to subclasses and containing package
  - **private** - method is only visible within the class, no package access
  - **package** - a namespace to which classes belong

- **abstract** - method must be implemented in a subclass
- **static** - method cannot rely on class data members
- **final** - method cannot be overridden
- **synchronized** - method’s scope is a critical section
- **native** - the method contains native (e.g., C) code
- **strictfp** - method must use strict IEEE floating point math.
A new construct: interfaces

A Java interface allows otherwise unrelated classes to satisfy a given requirement.

This is orthogonal to inheritance.

- **inheritance**: an A *is-a* B (has the attributes of a B, and possibly others)
- **interface**: an A *can-do* X (and possibly other unrelated actions)
- Interfaces are a better model for multiple inheritance

See also, Scott, section 9.4.3.
public interface Comparable {
    public int compareTo (Object x) throws ClassCastException;
    // returns -1 if this < x,
    // 0 if this = x,
    // +1 if this > x
}

// Implementation needs to cast x to the proper class.

// Any class that may appear in a container should implement Comparable, so the container can support sorting.
### Comparison with C++

<table>
<thead>
<tr>
<th>Java</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>methods</td>
<td>virtual member functions</td>
</tr>
<tr>
<td>public/protected/private members</td>
<td>similar</td>
</tr>
<tr>
<td>static members</td>
<td>same</td>
</tr>
<tr>
<td>abstract methods</td>
<td>pure virtual member functions</td>
</tr>
<tr>
<td><strong>final</strong> methods</td>
<td>no analogous feature</td>
</tr>
<tr>
<td><strong>interface</strong></td>
<td>pure virtual class with no data</td>
</tr>
<tr>
<td>implementation of an interface</td>
<td>virtual inheritance</td>
</tr>
</tbody>
</table>
Simulating a first-class function with an object

A simple first-class function:

```haskell
fun mkAdder nonlocal = (fn arg => arg + nonlocal)
```

The corresponding C++ class:

```cpp
class Adder {
    int nonlocal;

public:
    Adder (int i) : nonlocal(i) { }
    int operator() (int arg) { return arg + nonlocal; }
};
```

`mkAdder 10` is roughly equivalent to `Adder(10)`. 
A simple unsuspecting object (in Java, for variety):

class Account {
    private float theBalance;
    private float theRate;

    Account (float b, float r) {
        theBalance = b;
        theRate = r;
    }

    public void deposit (float x) {
        theBalance = theBalance + x;
    }
    public void compound () {
        theBalance = theBalance * (1.0 + rate);
    }
    public float balance () { return theBalance; }
}
The corresponding first-class function:

```
(define (Account b r)
  (let ((theBalance b) (theRate r))
    (lambda (method)
      (case method
        ((deposit)
          (lambda (x) (set! theBalance (+ theBalance x))))
        ((compound)
          (set! theBalance (* theBalance (+ 1.0 theRate))))
        ((balance)
          theBalance))))

new Account(100.0, 0.05) is roughly equivalent to
(Account 100.0 0.05).
```
ML datatypes and OO inheritance organize data and routines in orthogonal ways:

<table>
<thead>
<tr>
<th></th>
<th>data variants</th>
<th>data operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>datatypes</td>
<td>all together/closed</td>
<td>scattered/open</td>
</tr>
<tr>
<td>classes</td>
<td>scattered/open</td>
<td>all together/closed</td>
</tr>
</tbody>
</table>

**datatypes**
- easy to add new operations
- harder to add new variants

**classes**
- easy to add new variants
- harder to add new operations
A couple of facts:

- In mathematics, an ellipse (from the Greek for absence) is a curve where the sum of the distances from any point on the curve to two fixed points is constant. The two fixed points are called foci (plural of focus).

  *from http://en.wikipedia.org/wiki/Ellipse*

- A circle is a special kind of ellipse, where the two foci are the same point.

If we need to model circles and ellipses using OOP, what happens if we have class `Circle` inherit from class `Ellipse`?
class Ellipse {
    ...

    public move (double dx, double dy) { ... } 

    public resize (double x, double y) { ... } 
}

class Circle extends Ellipse {
    ...

    public resize (double x, double y) { ??? } 
}

We can’t implement a resize for Circle that lets us make it asymmetric!
In Java, if class B is a subclass of class A, then Java considers “array of B” to be a subclass of “array of A”:

```java
class A { ... }
class B extends A { ... }

B[] b = new B[5];
A[] a = b;        // allowed (a and b are now aliases)

a[1] = new A();   // Bzzzt! (Type error)
```

The problem is that arrays are *mutable*; they allow us to replace an element with a different element.