Programming Languages

OOP

CSCI-GA.2110-001
Spring 2023

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).

What is OOP? (part II)

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.

Styles of OOLs

- 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.
  - Changes to the clone do not propagate back to the prototype.

Prototype OOP example

```
var original = { a:'A', b:'B' };
var clone = owl.util.clone(original);
// clone.a == 'A'
// clone.b == 'B'
clone.a = 'Apple';
// clone.a == 'Apple'
// original.a == 'A'  // unchanged
original.b = 'Banana'
// clone.b == 'Banana'  // change shows through
clone.c = 'Car'
// original.c is undefined
original.a = 'Blah'
// clone.a == 'Apple' // clone's new val hides original
delete clone.a
// clone.a = 'Blah'  // original value visible again
// repeating "delete clone.a" won't delete orig. value
```

Courtesy: http://oranlooney.com/functional-javascript

Other common OOP features

- multiple inheritance (inheriting from more than one parent class/object)
  - C++
  - Java (of interfaces only)
  - problem: how to handle possible name mangling due to 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

Constructors

All OOLs languages have the concept of a *constructor* which creates instances of an object.

Observation: An object cannot be used until fully constructed.

Plock's constructor best practices:

1. Should be limited to initializing data members only.
2. Do not pass data members to other objects (including constructors).
3. Do not perform business logic.
4. Never call a method that uses runtime method binding.
5. Never call a method that throws exceptions.
6. Even better: do not call methods at all.
7. Do not use `this`/`self` unless expressly permitted.
8. Do not spawn threads.
9. Do not use constructor chaining—a fad that needs to go away.

Java Features

- an imperative language (like C++, Ada, C, Pascal)
- is interpreted (like Scheme, APL)
- portions can be compiled using Just-in-Time compilation.
- 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

From a 1993 white paper:

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

Obviously, “simple” was dropped.

Portability

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

Contrast w/conventional languages

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 Java omissions

- no operator overloading (syntactic annoyance)
- no separation of specification and body
- no enumerations until version 5 (2004)
- no generic facilities until version 5 (2004)
- no lambdas until version 8 (2014)
  - Closures capture free variables by-value
  - Non-locals must be final or “effectively final.”
- destructors : supported (`finalize`) but virtually never used
  - Never know when `finalize` will run
  - `finalize` may never run
  - Not needed for deallocating memory (due to garbage collection)
  - Convention is to define and manually invoke a method called `close` to clean up resources (sockets, file handles) since these are usually time sensitive.
  - Objects with finalizers much slower to garbage collect

Classes in Java

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 () { ... }
}
```

Extending a class

```
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!
}
```

Dynamic dispatching

```
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, cp2, and p3 affected

p1.display();    // Point's display
cp1.display();   // ColoredPoint's display
p3.display();    // ColoredPoint's display
```

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.

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.

Interface Comparable

```
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.
```

Classes in C++

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 () { ... }
};
```

Extending a class

```
class ColoredPoint : public Point {
  Color color;

public:

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

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

  virtual Color getColor () { return color; }

  virtual void display () { ... }  // now in color!
};
```

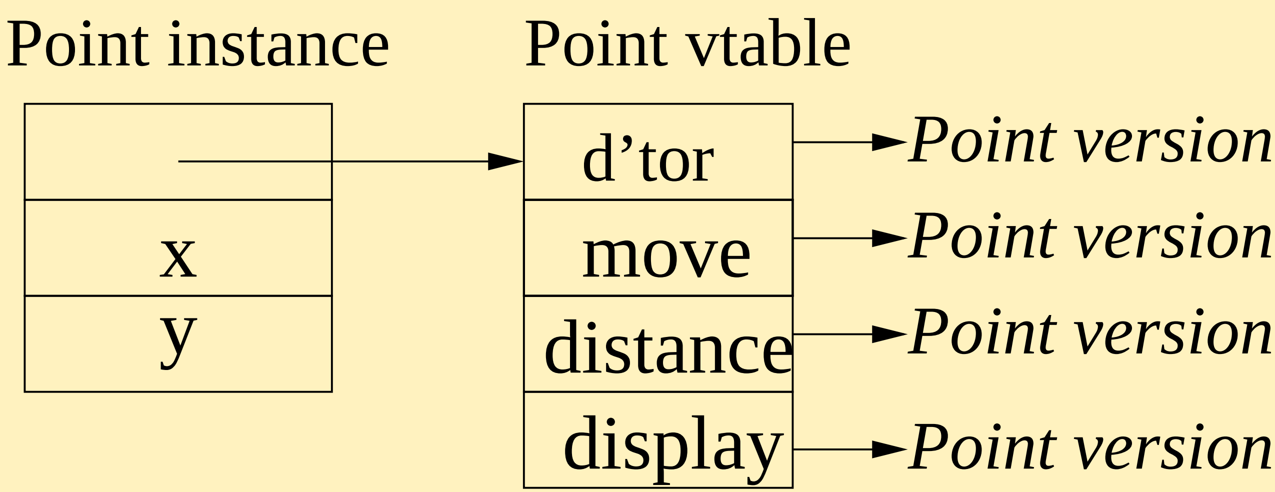
Dynamic dispatching

```
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
```

Implementation: the vtable

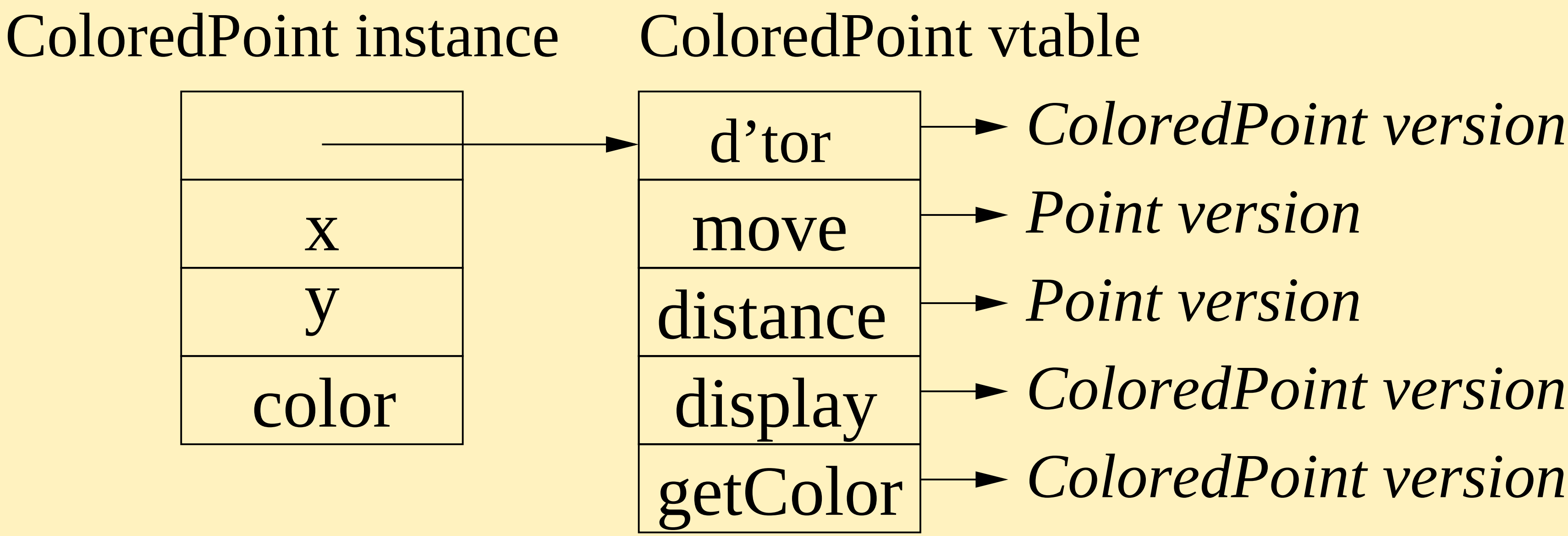
- vtables : used to determine which class' method to invoke
- `virtual` method means: "use the subclass version" (including all descendant subclasses)
- `virtual` methods are placed in the vtable
- One vtable per class

A typical implementation of a class in C++; using `Point` as an example:



An extended vtable

For `ColoredPoint`, we have:



Non-virtual member functions are never put in the vtable

Coercion

C++ coerces non-primitives using copy constructors.

```
class Foo {
    int value;

    public:
    Foo (int i) : value(i) {}
};

void bar (Foo f) {}

int main()
{
    bar(42); // Equivalent to bar(Foo(42));
}
```

If coercion is not intended, use keyword `explicit`:

```
explicit Foo(int i) : value(i) {}
```

Now this is illegal:

```
bar(42); //compiler error, but explicit cast OK: bar(Foo(42))
```

Java/C++ Comparison

Java	C++
methods	virtual member functions
public/protected/private members	similar
static members	same
abstract methods	pure virtual member functions
`final` methods	same
`interface`	pure virtual class with no data members
implementation of an interface	virtual inheritance
auto default constructors	same
not used	copy constructors
not supported	method deletion

Simulating first-class functions

A simple first-class function:

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

The corresponding C++ class:

```
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)`

First-class functions strike back

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 + theRate);
   }
   public float balance () { return theBalance; }
}
```

First-class functions strike back

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 vs. inheritance

ML datatypes and OO inheritance organize data and routines in orthogonal ways:

	data variants	**data operations**
datatypes	all together/closed	scattered/open
classes	scattered/open	all together/closed

datatypes	easy to add new operations
	harder to add new variants
classes	easy to add new variants
	harder to add new operations

OOP Pitfalls: circle & ellipse

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`?

Circles and ellipses

```
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!

C++: `Circle::resize (double x, double y) = delete;`

Pitfalls: Array subclassing

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":

```
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.

Pitfalls: Cross-cutting concerns

OOP inheritance hierarchies tend to force concerns (i.e., logically related groupings of functionality) to be vertical.

We want to reuse concerns, but placing them in a traditional OOP hierarchy can cause problems:

- we may not want the concern to be available to all subclasses
- the concern may need to be available *across* inheritance hierarchies
- lack of "is-a" relationship violates the OO contract (e.g. Car is *not* a Logger)
- concern logic may be spread out—not "all in one place"

We refer to such concerns as *cross-cutting concerns*.

Examples: logging, caching, persistence, security, data validation.

Aspect-Oriented Programming (AOP) is a paradigm, orthogonal to OOP, intended to properly separate and weave cross-cutting concerns into a non-AOP application.

Aspect-Oriented Programming

There are several key AOP concepts:

- **Aspect**: a module encapsulating a concern.
- **Join point**: place in the code we want to insert an aspect.
- **Advice**: code to execute at a particular join point.
- **Pointcut**: set of join points.

Pointcut:

```
call(void *.deposit(double)) ||
call(void *.withdraw(double))
```

Advice:

```
before(): withdraw() {
    System.out.println("about␣to␣withdraw␣money");
}
after(): deposit() {
    System.out.println("just␣deposited␣money");
}
```

Introspection, reflection, and typeless programming

```
public void DoSomething (Object thing) {
  // what can be do with a generic object?
  if (thing instanceof Gizmo) {
    // we know the methods in class Gizmo
    ....
```

`instanceof` requires an accessible run-time descriptor in the object.

Reflection is a general programming model that relies on run-time representations of aspects of the computation that are usually not available to the programmer.

More common in dynamically typed languages, e.g., Smalltalk and Common LISP.

Reflection and metaprogramming

Given an object at run-time, it is possible to obtain:

- its class
- its fields (data members) as strings
- the classes of its fields
- the methods of its class, as strings
- the types of the methods

It is then possible to construct calls to these methods.

- This is possible because the JVM provides a high- level representation of a class, with embedded strings that allow almost complete disassembly.

It is also possible to *change* the program by modifying the above.

- Functional languages support this indirectly: continuations
- Can be abused. Other uses: malware

Reflection classes

- java.lang.Class

```
Class.getMethods ()    // returns array
                       //  of method objects
Class.getConstructor (Class[] parameterTypes)
  // returns the constructor with those parameters
```

- java.lang.reflect.Array

```
Array.NewInstance (Class componentType,
                   int length)
```

- java.lang.reflect.Field
- java.lang.reflect.Method

Example: look for and invoke method `doSomething` in some instance `foo`:

```
Method m = foo.getClass().getMethod("doSomething", null);
m.invoke(foo, null);
```