G22.2110-003 Programming Languages - Fall 2012

Lecture 12

Thomas Wies

New York University

Review

Last lecture

- Modules

Outline

- Classes
- Encapsulation and Inheritance
- Initialization and Finalization
- Dynamic Method Binding
- Abstract Classes
- Simulating First-Class Functions

Sources:

- PLP, ch. 9
- PLP, ch. 3.6.3

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++)

A *class* is a construct which defines the data and methods associated with all of its instances (objects).

What is OOP? (part II)

The *inheritance* and *dynamic binding* ideas:

- inheritance: classes can be extended:
  - 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 a *derived* or *child* class) of A, and A is a *superclass* (or a *base* or a *parent* 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.

Information Hiding in Classes

Like modules, classes can restrict access to their data and methods.

Unlike modules, classes must take inheritance into account in their access control.

Three levels of access:

- *public*: accessible to everyone
- *protected*: accessible within the class and in any derived classes
- *private*: accessible only within the class

Members are private by default.

In JAVA, "protected" means accessible within the class, all derived subclasses, *as well as* classes declared in the same package.

A *friend* declaration allows a foreign class or subroutine access to private members.

Example in C++

```
class Point {
  double x, y;  // private data members public:
  void move (double dx, double dy) {
    x += dx;  y += dy;
  }
  virtual void draw () { ... }
};

class ColoredPoint : public Point {
  Color color;
public:
  Color getColor () { return color; }
  void draw () { ... }  // now in color!
};
```

Same thing in JAVA

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

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

  public void draw () { ... }
}

class ColoredPoint extends Point {
  private Color color;

  public Color getColor () { return color; }

  public void draw () { ... }  // now in color!
}
```

Initialization and Finalization

A *constructor* is a special class method that is automatically called to *initialize* an object at the beginning of its lifetime.

A *destructor* is a special class method that is automatically called to *finalize* an object at the end of its lifetime.

Issues:

- choosing a constructor
- references and values
- execution order
- garbage collection

Choosing a Constructor

Most OOP languages allow a class to specify more than one constructor.

- *Overloading*: In C++, Java, and C#, constructors behave like overloaded methods. They must be distinguished by their numbers and types of arguments.
- *Named constructors*: In Smalltalk and Eiffel, constructors can have different names. Code that creates an object must name a constructor explicitly.
- *Companion object*: In Scala, a class specifies no explicit constructors. Instead, classes take parameters and constructor-like functions are declared in a companion object.

References and Values

In JAVA, variables can be references to objects, but cannot contain objects as values.

As a result, every object must be created explicitly, triggering a call to the constructor.

In C++, variables can have objects as values, so it is a little more complicated to identify how and when constructors are called:

- If a variable is declared with no initial value, then the *default constructor* is called.
- If a variable is declared to be a copy of another object of the same type, the *copy constructor* is called.
- Otherwise, a constructor is called that matches the parameters passed to the variable declaration.
- Similar rules apply to objects created on the heap.

Constructor Example in C++

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

public:

  // Default constructor
  Point () : x(0), y(0) {}

  // Copy constructor
  Point (const Point& p) : x(p.x), y(p.y) {}

  // Other constructor
  Point (double xp, double yp) : x(xp), y(yp) {}
  ...
};

Point p1; // calls default constructor
Point p2(1,2); // calls last constructor
Point p3 = p2; // calls copy constructor
Point p4(p2); // same as above (syntactic variant)
```

Constructor Example in C++

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

public:

  // Default constructor
  Point () : x(0), y(0) {}

  // Copy constructor
  Point (const Point& p) : x(p.x), y(p.y) {}

  // Other constructor
  Point (double xp, double yp) : x(xp), y(yp) {}
  ...
};

Point *p1, *p2, *p3; // no calls to constructor
p1 = new Point(); // calls default constructor
p2 = new Point(*p1); // calls copy constructor
p3 = new Point(1,2); // calls last constructor
```

Constructors in Java

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

  public Point () { this.x = 0; this.y = 0; }

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

Point p1 = new Point();
Point p2 = new Point(2.0, 3.0);
Point p3 = p2; // no constructor called
```

Execution Order

How do the constructors of base classes and derived classes interact?

Execution Order

How do the constructors of base classes and derived classes interact?

Typically, we want to call the base constructor before the derived fields are initialized.

Execution Order

How do the constructors of base classes and derived classes interact?

Typically, we want to call the base constructor before the derived fields are initialized.

Both C++ and JAVA provide mechanisms for doing this.

Constructors in a base class

In C++:

```
class ColoredPoint : public Point {
  Color color;
public:
  ColoredPoint(Color c) : Point(), color(c) {}
  ColoredPoint(double x, double y, Color c)
    : Point (x, y), color(c) {}
};
```

In Java:

```
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;
  }
}
```

Destructors and Garbage Collection

When an object in C++ is destroyed, a *destructor* is called.

A destructor is typically used to release memory allocated in the constructor.

For derived classes, destructors are called in the reverse order that the constructors were called.

In languages such as JAVA that have garbage collection, there is little or no need for destructors.

However, JAVA does provide an optional *finalize* method that will be called just before an object is garbage collected.

Example of Destructors in C++

```
class String {
  char *data;
public:
  String(const char *value);
  ~String() { delete [] data; }
};

String::String(const char *value)
{
  data = new char[strlen(value) + 1];
  strcpy(data, value);
}
```

Dynamic Method Binding

A key feature of object-oriented languages is allowing an object of a derived class to be used where an object of a base class is expected.

This is called *subtype polymorphism*.

Now, consider the following code:

```
ColoredPoint *cp1 =
  new ColoredPoint (2.0, 3.0, Blue);
Point *p1 = cp1; // OK
p1->draw ();
```

Which `draw` method gets called?

Dynamic Method Binding

A key feature of object-oriented languages is allowing an object of a derived class to be used where an object of a base class is expected.

This is called *subtype polymorphism*.

Now, consider the following code:

```
ColoredPoint *cp1 =
  new ColoredPoint (2.0, 3.0, Blue);
Point *p1 = cp1; // OK
p1->draw ();
```

Which `draw` method gets called?

- If the `Point` class method is called, it is an example of *static method binding*.
- If the `ColoredPoint` class method is called, it is an example of *dynamic method binding*.

Dynamic Method Binding

What are the advantages and disadvantages of static vs dynamic method binding?

Dynamic Method Binding

What are the advantages and disadvantages of static vs dynamic method binding?

- *static* is more efficient:
  - to support dynamic method binding, an object must keep an additional pointer to a *virtual method table* (or *vtable*).
  - dynamic method binding requires additional space, as well as an additional pointer dereference when calling a method.
- *dynamic* allows a subclass to *override* the behavior of its parent, a key feature that makes inheritance much more flexible and useful.

Dynamic Method Binding

What are the advantages and disadvantages of static vs dynamic method binding?

- *static* is more efficient:
  - to support dynamic method binding, an object must keep an additional pointer to a *virtual method table* (or *vtable*).
  - dynamic method binding requires additional space, as well as an additional pointer dereference when calling a method.
- *dynamic* allows a subclass to *override* the behavior of its parent, a key feature that makes inheritance much more flexible and useful.

In C++ and C#, methods are bound statically by default.

The keyword `virtual` distinguishes a method that should be bound dynamically.

Dynamic Method Binding

What are the advantages and disadvantages of static vs dynamic method binding?

- *static* is more efficient:
  - to support dynamic method binding, an object must keep an additional pointer to a *virtual method table* (or *vtable*).
  - dynamic method binding requires additional space, as well as an additional pointer dereference when calling a method.
- *dynamic* allows a subclass to *override* the behavior of its parent, a key feature that makes inheritance much more flexible and useful.

In C++ and C#, methods are bound statically by default.

The keyword `virtual` distinguishes a method that should be bound dynamically.

In JAVA, all methods are bound dynamically. The keyword `final` distinguishes a method that should be bound statically.

Dynamic Method Binding

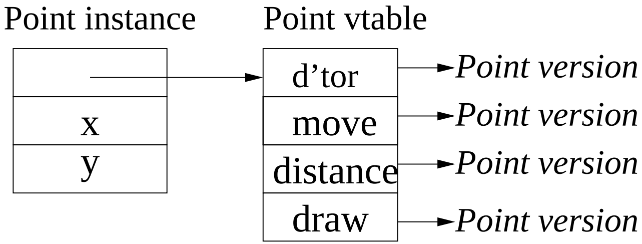
```
class Point {
  double x, y;  // private data members

public:
  // Constructors
  Point () : x(0), y(0) {}
  Point (const Point& p) : x(p.x), y(p.y) {}
  Point (double xp, double yp) : x(xp), y(yp) {}
  // Destructor
  virtual ~Point () {}

  virtual void move (double dx, double dy) {
    x += dx;  y += dy;
  }
  virtual double distance (const Point& p) {
    double xdist = x - p.x, ydist = y - p.y;
    return sqrt (xdist * xdist + ydist * ydist);
  }
  virtual void draw () { ... }
};
```

Dynamic Method Binding

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

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

  // Destructor
  ~ColoredPoint() {}

  virtual Color getColor () { return color; }

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

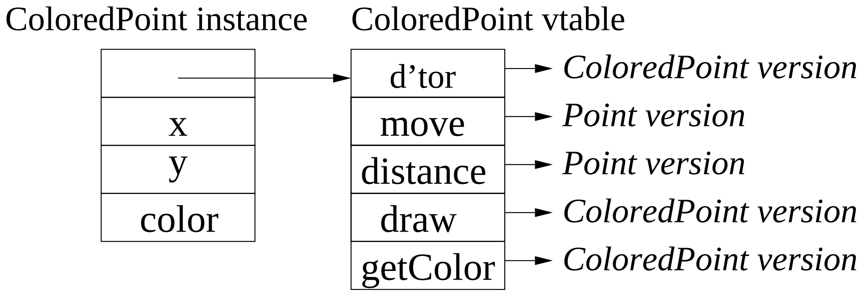
Implementation: the vtable

A typical memory layout with dynamic method binding in $\mathrm{C}{+}{+}$; using `Point` as an example:



Implementation: the vtable

For `ColoredPoint`, we have:



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

Abstract Classes

Another useful construct in object-oriented programming is the *abstract class*.

An abstract class contains at least one method which is *abstract* (called *pure virtual* in C++), meaning it has a declaration but no definition within the class.

It is not possible to declare an object of an abstract class as one of its methods has no definition.

Abstract classes are used as base classes in class hierarchies.

They are useful for defining API's when the implementation is unknown or needs to be hidden completely.

A class, all of whose methods are abstract, is called an *interface* in JAVA and C#.

Abstract Classes: Example

In C++:

```
class Drawable {
public:
  virtual void draw() = 0;
};

class Point : public Drawable {
  ...
  virtual void draw() { ... } // implementation
};
```

In Java:

```
public abstract class Drawable {
  public abstract void draw();
}

public class Point extends Drawable {
  ...
  void draw() { ... } // implementation
}
```

Class Hierarchies with Multiple Inheritance

In C++:

```
class Drawable {
public:
  virtual void draw() = 0;
};

class Resizable {
public:
  virtual void resize(double factor) = 0;
};

class Point : public Drawable {
  ...
};

class Square : public Drawable, public Resizable {
  ...
};
```

Class Hierarchies with Multiple Inheritance

In JAVA, multiple inheritance only works for abstract classes that do not have fields (called *interfaces*):

```
public interface Drawable {
  public void draw();
}

public interface Resizable {
  public void resize(double factor);
}

public Point implements Drawable {
  ...
}

public Square implements Drawable, Resizable {
  ...
}
```

Comparison of Java and C++

Java	C++
methods	virtual member functions
public/protected/private members	similar
static members	same
abstract methods	pure virtual member functions
`interface`	pure virtual class with no data members
implementation of an interface	inheritance from an abstract class

Simulating a first-class function with an object

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 balance;
   private float rate;

   Account (float b, float r) { balance = b;
                                rate = r; }

   public void deposit (float x) {
     balance = balance + x;
   }
   public void compound () {
     balance = balance * (1.0 + rate);
   }
   public float getBalance () { return balance; }
}
```

First-class functions strike back, part 2

Simulating objects using records and functions:

```
datatype account =
  AccountObj { deposit : real -> unit,
               compound : unit -> unit,
               getBalance : unit -> real }
fun Account b r =
  let val balance = ref b
      val rate = ref r
  in
    AccountObj {
      deposit = fn x => balance := !balance + x,
      compound = fn () =>
        balance := !balance * (1.0 + !rate),
      getBalance = fn () => !balance }
  end
```

`new Account(80.0,0.05)` is roughly equivalent to `Account 80.0 .05`.

OOP Pitfalls: the circle and the 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!

Pitfalls: Array subtyping

In JAVA, if class *B* is a subclass of class *A*, then JAVA considers array of *B* to be a subtype 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!  (ArrayStoreException)
```

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