Review

Last week

- Modules
Outline

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

Sources:

PLP, 9
Barrett. Lecture notes, Fall 2008.
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 common levels of access:

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

In C++, members are private by default.

In JAVA, members are **package private** by default, that is, they are accessible within the package. Furthermore, “protected” means accessible within the class and the package.

In C++, a **friend** declaration allows a foreign class or subrouting access to private members.
Example in C++

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

class ColoredPoint : public Point {
    Color color;
public:
    Color getColor () { return color; }
    void display () { ... } // now in color (e.g., green)!
};
```
class Point {
    private double x, y;  // private data members

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

    public void display () { ... }
}

class ColoredPoint extends Point {
    private Color color;

    public Color getColor () { return color; }

    public void display () { ... }  // now in color (still green)!
}
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, OBJECTIVE-C and EIFFEL, constructors can have different names. Code that creates an object must name a constructor explicitly. To put it differently, in these languages, constructors are just a *convention*.

To illustrate constructors as a convention, compare a Java object creation

```
// Allocate memory and initialize object.
Point p = new Point();
```

with one in Objective-C

```
// Allocate memory and initialize object,
// this time with regular method calls.
Point *p = [[Point alloc] init];
```
References and Values

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

As a result, every object has reference semantics and 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
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?*

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++:

```cpp
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:

```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 may be called just before an object is garbage collected. It only “may” be called since a Java virtual machine shutdown does not process pending finalizer calls.
Example Destructor 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:

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

*Which display 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?

- **static** is more efficient:
  - To support dynamic method binding, an object must keep an additional pointer to a `virtual method table` (or `vtable`). Even worse, SMALLTALK, OBJECTIVE-C, and most modern scripting languages require more complex data structures for dynamic method binding.
  - dynamic method binding requires additional space, as well as an additional pointer dereference when calling a method (and more for SMALLTALK, OBJECTIVE-C, RUBY, etc.).

- **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 non-private, non-static methods are bound dynamically.
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 display () { ... }
};
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 display () { ... }  // now in color!
};
A typical memory layout with dynamic method binding in C++; using `Point` as an example:
Implementation: the vtable

For `ColoredPoint`, we have:

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

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++:

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

In JAVA:

```java
abstract class DrawableObject {
    public abstract void draw();
}
```
Comparison of JAVA and 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</td>
<td>similar</td>
</tr>
<tr>
<td>members</td>
<td>same</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>interface</td>
<td>pure virtual class with no data members</td>
</tr>
<tr>
<td>implementation of an</td>
<td>inheritance from an abstract class</td>
</tr>
<tr>
<td>interface</td>
<td></td>
</tr>
</tbody>
</table>
Simulating a first-class function with an object

A simple first-class function:

```plaintext
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)`. 
First-class functions strike back

A simple unsuspecting object (in JAVA, for variety):

```java
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:

\[
\text{(define (Account b r)}
\]
\[
\text{(let ((theBalance b) (theRate r)})}
\]
\[
\text{(lambda (method)}
\]
\[
\text{(case method}
\]
\[
\text{(deposit)}
\]
\[
\text{(lambda (x) (set! theBalance}
\]
\[
\text{(+ theBalance x)))}
\]
\[
\text{(compound)}
\]
\[
\text{(set! theBalance (* theBalance}
\]
\[
\text{(+ 1.0 theRate))))}
\]
\[
\text{(balance)}
\]
\[
\text{theBalance)))))}
\]

\text{new Account(100.0, 0.05) is roughly equivalent to (Account 100.0 0.05).}
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).


- 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 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 $A$:

```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. JAVa catches the type error at runtime, at the cost of a dynamic type check for every reference assignment to an array slot.
Things I Am Not Telling

- Prototype-based inheritance: instead of extending a class, a new object simply specifies its \textit{prototype}, i.e., another object that has suitable methods. See \textsc{JavaScript}.

- Multiple inheritance: instead of extending one class, a new class may inherit from several classes.
  - C++, of course, supports the full complexity of multiple inheritance for classes.
  - \textsc{Java} and C# only support multiple inheritance for interfaces.
  - So-called mixins and traits are an attempt to provide more expressivity than interfaces (no method bodies) but without the problems of multiple inheritance (duplicate instance data, i.e., the \textit{diamond problem}).

- Family inheritance: instead of extending one class, extend an entire group of related classes — typically, only found in research languages.

- Object adaptation: instead of subclassing, extend all instances in place. Typically only found in research languages, but see \textsc{Objective-C}.
Endianess

The order of how the bytes belonging to a larger number are stored in memory.

- **Big Endian**: the byte with the *biggest* impact is stored at the smallest address. E.g., 0C0A0F0E is stored as 0C0A0F0E. Example architectures include the 68000 and PowerPC running Mac OS.

- **Little Endian**: the byte with the least impact is stored at the smallest address. E.g., 0C0A0F0E is stored as 0E0F0A0C. Example architectures include the x86 and 6502.

- **Network Byte Order**: same as big endian.

- **Bi-Endian**: processor architectures such as the ARM and PowerPC that can be switched between big and little endian.

To convert between network and host byte order in C, use *ntohs*, *ntohl*, *htons*, and *htonl*.

This slide builds on (dis)information available on Wikipedia.