Announcements

- Pretest
- Google group discussions
- Assignment 1 questions?
Object-oriented programming

- Modularity
  - Abstraction
  - Encapsulation

- Inheritance
  - Polymorphism
  - Dynamic binding
  - Abstract classes
  - Interfaces
Three related principles

- **Modularity**
  - System consists of components, each accomplishing a separate functional task.

- **Abstraction**
  - Modules described in terms of what they do, not how they do it.

- **Encapsulation**
  - Details of how module accomplishes its task are hidden from the other modules.

- **Examples:** automobile design, Cornell education.
“Wherever there is modularity there is the potential for misunderstanding: Hiding information implies a need to check communication.”

Alan Perlis, first recipient of Turing Award
Software design

- How does object-oriented programming in Java embody the principles of modularity, abstraction, and encapsulation?

- How do we avoid miscommunication?

- There is modularity at various levels. Today, we look at the level of classes/interfaces.
Classes and Objects

- **Modularity**
  - Object is a kind of module
  - Class defines a type of object

- **Abstraction**
  - Classes have well-defined interfaces that describe what objects of that class can do
  - You can use a class without knowing how it works “under the hood.” Examples: String, File, Scanner.

- **Encapsulation**
  - Classes decide how methods are implemented
  - Objects can hide data and functionality
A class’s interface is everything in it that is externally accessible (i.e., **public** methods and fields)

- **Javadocs**
  - A utility to automatically generate HTML pages that describe the interface of user defined classes
  - Built into Eclipse
  - (CODE DEMO)
Java API

- Describes the interface of all built-in classes
Encapsulation

- By hiding code and data behind its interface, a class encapsulates its “inner workings”
- Why is that good?
- Lets us change the implementation later without invalidating the code that uses the class

```
class LineSegment {
    private Point2D _p1, _p2;
    ...
    public double length() {
        return _p1.distance(_p2);
    }
}
```

```
class LineSegment {
    private Point2D _p;
    private double _length;
    private double _phi;
    ...
    public double length() {
        return _length;
    }
}
```
Public fields are usually a **bad thing**.

```java
class SmallSet {
    public int count = 0;
    ...
    public void add(Object o) {
        if (count < 10) {
            ...
        } else {
            ...
        }
    }
}
```

Anyone can change them; the class has no control.
Violating encapsulation?

- Do getters and setters violate encapsulation?

```java
public class Point2D {
    private double x;
    private double y;
    ...
    public double getX() { return x; }
    public void setX(double newX) {
        x = newX;
    }
}
```
Do getters and setters violate encapsulation?

```java
class Point2D {
    private double x;
    private double y;
    ... public double getX() { return x; }
    public void setX(double newX) {
        x = newX;
    }
}
```

```java
public class LineSegment {
    private Point2D _p1, _p2;
    ...
    public Point2D getFirstPoint() { return _p1;
    }
}
```
Inheritance

- Many ways to use (and misuse!) inheritance
- Today, focus on the mechanics with some discussion of use
Subclasses

- Subclass inherits both the **interface** and the **implementation** of the parent class
- It can change the implementation
  - method overriding
- It can extend the interface
  - add a new method
- It **cannot** restrict the interface
  - remove a method or change a method to private
- (CODE DEMO)
Inheritance and encapsulation

- private
  - Private methods and data are inherited, but cannot be modified.

- protected
  - A special designation ("public" for subclasses, "private" for all others)
  - Sometimes convenient, but can be troublesome: subclass repurposes it in ways parent class not expecting.
Polymorphism & dynamic binding

- “Many forms”
- A single reference variable of type T may refer to an object of a different type:
  - T
  - or, any subtype of T
- Type of the object may not be known until runtime
- Dynamic binding: at runtime, find the appropriate method definition
public class Drug {
    private int dosage; // mg of active ingredient
    public int getDosage() { return dosage; }
}

class Placebo extends Drug {
    public int getDosage() { return 0; }
}

Patient p = ...;
Random rand = new Random();
Drug drug = null;
if (rand.nextDouble() > 0.5) { // coin flip
    drug = new Drug();
} else {
    drug = new Placebo();
}
System.out.println("Expt log: patient " + p + " gets dosage " + drug.getDosage());
...
Power of polymorphism

Illustrated by example

```java
public class GroceryItem {
    private double price;
    public double computeTax() {
        return price * 0.05; // 5% sales tax
    }
}

class Cigarettes extends GroceryItem {
    public double computeTax() {
        return super.computeTax() + 2; // $2 “sin” tax
    }
}

// Compute total tax owed to government based on items sold.
public static double taxCollector(GroceryItem[] itemsSold) {
    double tax = 0;
    for (GroceryItem item : itemsSold) {
        tax += item.computeTax(); // defer tax calc. to item
    }
    return tax;
}
```
Power of polymorphism

- While inheritance makes it possible to develop deep hierarchies of highly specialized subtypes...
  (e.g., Shape, Circle, Polygon, Triangle, Rectangle, Square, ...)
- Some code might prefer working at a higher level of abstraction (e.g., “think” in terms of Polygon objects)
- Polymorphism makes it possible to manipulate specialized objects (Squares) as though they were generic objects (Polygons)
Abstract classes

- Define a type of object that can never be created... how is this useful ?!?  
- (CODE EXAMPLE)
Abstract classes

- Define a type of object that can never be created... how is this useful ?!?!
  - Define a common interface that all subclasses will inherit
  - Polymorphism: code can work w/ abstract class (Shape) even though objects will be of subclass types (Circle, ...)
  - Abstract methods can include some method implementations
    - Put shared code in common super class.
Multiple inheritance

- In Java, a class be a subclass of at most one parent class.
- Why?
Java interface

- A special kind of abstract class
- Three key technical differences:
  - It is just an interface: no implementations or instance data permitted
  - A class can inherit multiple interfaces (why is this okay?)
  - Java syntax (implements vs. extends)
- Differences leads to major practical differences
  - Interfaces that describe additional functionality, similar to “mixins” in C++
  - Java interfaces Comparable, Iterable
Key points from today

- Distinction between the interface (the “what”) and the implementation (the “how”)
- Polymorphism
  - Code can “think” in terms of a higher level general type, yet get specialized behavior of the lower-level types
- Concept of a Java interface