Object-oriented design unit:

- Test 1 retrospective (Monday)
- OO design goals (Wednesday)
- Class extension (**Today**)
- More about class extension; refactoring (next week Monday and Wednesday)

Today:

- Basic subclassing
  - The code-reuse problem
  - Code-reuse with composition
  - Abstract classes
- Composition vs subclassing (class extension)
- Details
  - Access modification
  - Constructors
  - Overriding
  - `final`

Design goals

- Code that is easily understood, reused, and changed; code that is less likely to contain errors.
- Where should this information go? Where should this functionality go? Who should know whom?
- *Coupling* is the interconnectedness and interdependence among classes.
  - *Tight coupling* is when two classes depend on each other's implementation. Thus a change in one class's implementation affects the other.
  - *Loose coupling* is when classes communicate through well-defined and stable interfaces and thus don't depend (much) on each other's implementation.
- *Encapsulation* is the hiding of implementation details of a module from the rest of the system. "Only information about *what* a class can do should be visible to the outside, no about *how* it does it. If no other class knows how our information is stored, then we can easily change how it is stored without breaking other classes" (Barnes and Kölling, pg 207).

Design goals (continued)

- *Responsibility-driven design* is the principle that each class is "responsible for handling its own data" (Barnes and Kölling, pg 212). Data storage and manipulation should be in the the same place.
- *Reuse.* Reusing code refers to adapting classes (with instance variables and methods) for a new purpose. *White box reuse* is when the re-user can observe the what is inside the structure being reused. *Black box reuse* is when the re-user knows only what the reused structure does, not how it does it.
- *Cohesion* refers to how well a "unit of code" maps to a logical task. "A cohesive method is responsible for one, and only one, well-defined task. . . A cohesive class represents one well-defined entity" (Barnes and Kölleg, pg 219–220).

- A class declared to be *abstract* is halfway between being an interface and a class; it is a "partially written class". ("Regular" classes are thus called *concrete classes*.)
- Declare unimplemented methods *abstract*.
- Another class can *extend* an abstract class, just like (or as opposed to) implementing an interface.
- The abstract class is the *parent class* or *superclass*, and the class that extends the abstract class is a *child class* or *subclass*.
- Members in the parent class are *inherited* by the child class.
- Concrete (child) classes are obligated to implement the parent class's abstract methods, just as when implementing an interface.

Details, details...

- Abstract classes cannot be instantiated (but they can have constructors)
- There are three access modifiers that can be used on member declarations:
  - `public`: Any code in the system has access to this member
  - `private`: Only code in the class (not its subclasses) can access to this member
  - `protected`: The code in the class *and any subclass* has access to this member.
- Constructors in a child class can call constructors of the parent class using `super`.
- A child class can *override* a method that would be inherited by providing an implementation for it.
- A concrete (non-abstract) class also can be extended.
- The `final` modifier
  - A final variable cannot be modified.
  - A final method cannot be overridden
  - A final class cannot be extended

Coming up:

- **Due Fri, Feb 27.** *Do Project 3, "Homemade Linked-list Map."*
- **Due Fri, Mar 6.** *Do Project 4, "Text-based adventure game."*