CS61B Summer 2006
Instructor: Erin Korber
Lecture 6, 5 July
Reading for tomorrow: Ch. 11 (exceptions)

1 Inheritance

- Putting common code in a *superclass*, then having *subclasses* inherit from it - they get all of its instance variables and methods.
- Subclasses can modify and augement a superclass in several ways:
  - Declaring additional instance variables
  - Declaring additional methods
  - Overriding old methods with new implementations

1.1 Designing an Inheritance Tree

- 1. See which objects will have common behaviors
  2. Design a class (the superclass) that represents that common state and behavior.
  3. For each subclass, decide if it needs behaviors specific to that subclass (adding or overriding methods).
- Using IS-A and HAS-A relationships
  - Use these to see if the inheritance heirarchy you designed makes sense.
  - If X IS-A Y, then it makes sense for X to extend Y - an X can do everything a Y can do, and possibly more.
  - If X HAS-A Y, it does NOT make sense for X to extend Y - rather, X should have a reference to Y - a Y instance variable.
  - Subclassing is transitive; if X IS-A Y and Y IS-A Z, then X IS-A Z.

1.2 The "protected" keyword

If something is `protected`, it means that subclasses can inherit the `protected` thing, even if they are outside the package of the superclass. If something is

marked `private`, even the subclasses can't see it. So if you think you might want to write a subclass of your class at some point, you may want to make some instance variables and helper methods `protected` instead of `private`.

1.3 Inheritance and Constructors

So what exactly happens when we construct an instance of a subclass? As you would expect, Java executes the subclass constructor, but first it executes the code in the superclass constructor. The subclass constructor can add to and/or modify the work done by the superclass constructor.

The zero-parameter superclass constructor is always called by default, regardless of the parameters passed to the subclass constructor. To change this default behavior, the subclass constructor can explicitly call any constructor for its superclass by using the "super" keyword.

The call to "super" must be the first statement in the constructor. If there is no explicit call to another "super" constructor, and the superclass has no zero-argument constructor, a compilation error occurs. There is no way to tell Java not to call a superclass constructor, since the subclass instance needs the superclass instance (an "inner object") to know how to behave (since it has inherited instance variables and methods from the superclass).

1.4 Preventing inheritance

- The `final` keyword can be applied to a class or method to keep it from being overridden.
- Anything `private` can't be inherited at all.
- Anything with default access can only be inherited/subclassed by something in the same package.
- A class with only `private` constructors can't be subclassed at all. (Why?)

1.5 Overriding methods

When an overridden method is called, the most specific version of the method for that object type is used. Suppose we have a hierarchy where Square extends Rectangle, which extends Shape. When invoking a method on a reference to a Square object, the JVM looks for the method first in the Square class - if it doesn't find a version written there, it will look in the

Rectangle class, and if a version is not found there, it will look in the Shape class, and so on.

Sometimes you want to override a method, yet still be able to call the method in the superclass. We can do this using "super". `super` is always a reference to the superclass object (much like `this` is always a reference to the current object). Unlike superclass constructor invocations, ordinary superclass method invocations can take place anywhere in a method.

- When overriding a method, there are some restrictions to keep in mind.
  - The access level must be the same or less restrictive. For example, you can't override a public method and make it private.
  - The arguments must be the same. (Don't get *overriding* confused with *overloading*!)
  - The return type must be the same, or a subclass of the original return type. What does "subclass" mean? Well, for that we need to talk about....

2 Polymorphism

- Now we have objects that have more than one type.
  - Every Triangle is a Shape (but not every Shape is a Triangle!)
  - So whenever we are expecting something of type Shape, we can substitute something of type Triangle.
  - We can use this to write more flexible and simpler code.
- The *reference* type can be a superclass of the actual *object type*. For example: `Shape mySquare = new Square();` So anything that extends the declared type of the reference can be assigned to it.
- So, why do we care? Because Java has *dynamic method lookup*.
  - This means that when we invoke a method that is overridden in a subclass, Java calls the right method for the *actual object*, regardless of the reference variable type.
  - There are two kinds of types in play here:
    - Static type: the type of a variable
    - Dynamic type: the class of the object the variable references

– So "dynamic method lookup" means that methods are used based on dynamic type, not static type.
– In order for dynamic method lookup to work, the method being looked up must be guaranteed to be present. For example, say SubClass extends SuperClass, and you wrote an additional method called `specialMethod` in SubClass that is not present in SuperClass. Then in the code below, the `s.specialMethod()` call will be a compile-time error, since not every object of class SuperClass has a `specialMethod()` method.

```
SubClass t = new SubClass()
t.specialMethod();            //This is fine.
SuperClass s = new SubClass();
s.specialMethod();            //Compile-time error.
```

2.1 Variable assignment and casting

```
SuperClass s;
SubClass t = new SubClass();
s = t;              // Fine
t = s;              // COMPILE-TIME error
t = (SubClass) s;   // Fine
s = new SuperClass();
t = (SubClass) s;   // RUN-TIME error: ClassCastException
```

The assignment `s = t;` works because any instance of SubClass (such as t) *is* a SuperClass, and thus can be assigned to a variable of type SuperClass. However, since the converse is not true, the assignment `t = s;` fails. We can force such an assignment to happen by *casting*, as in `t = (SubClass) s;`. This is a way of telling the complier that you have written your program to ensure that `t` will always be a SubClass. If you are wrong, you'll find out when your program terminates with a "ClassCastException" error message.

There is a boolean operator called `instanceof` that will tell you if an object is of a particular type. It is used as in: `bool b = s instanceof SubClass`, which will be true if and only if `s` references a SubClass object.

3 Abstract Classes

An abstract class is a class whose sole purpose is to be extended.

- When it doesn't make sense to have an object of a certain kind (without it being one of its subtypes), make the class abstract.
- You cannot instantiate an abstract class - the compiler forbids it.

3.1 Abstract methods

Methods can be marked abstract too. Some behaviors described in an abstract class might not make any sense outside of a specific subclass - these methods should be marked abstract.

- An abstract method has no body!
- Its purpose is to force the subclasses to implement it.
- Abstract methods can occur only in abstract classes.

4 Multiple Inheritance and Interfaces

What if you want something to inherit from more than one class at a time? For example, a Square is both a Rhombus and a Rectangle. It looks like it might be nice if it could just extend both at once, but that's not possible in Java, because of problems that occur if both superclasses define a method with the same name - which one should be used when called on the subclass?

So how do we accomodate the need to inherit from more than one thing at a time? *Interfaces.*

- like a totally abstract class - anything implementing an interface must implement all of its methods.
- A class can implement as many interfaces as you like (but it can only extend one superclass).
- Can be used as a polymorphic type (just like classes)