Topic 5: Class Hierarchies

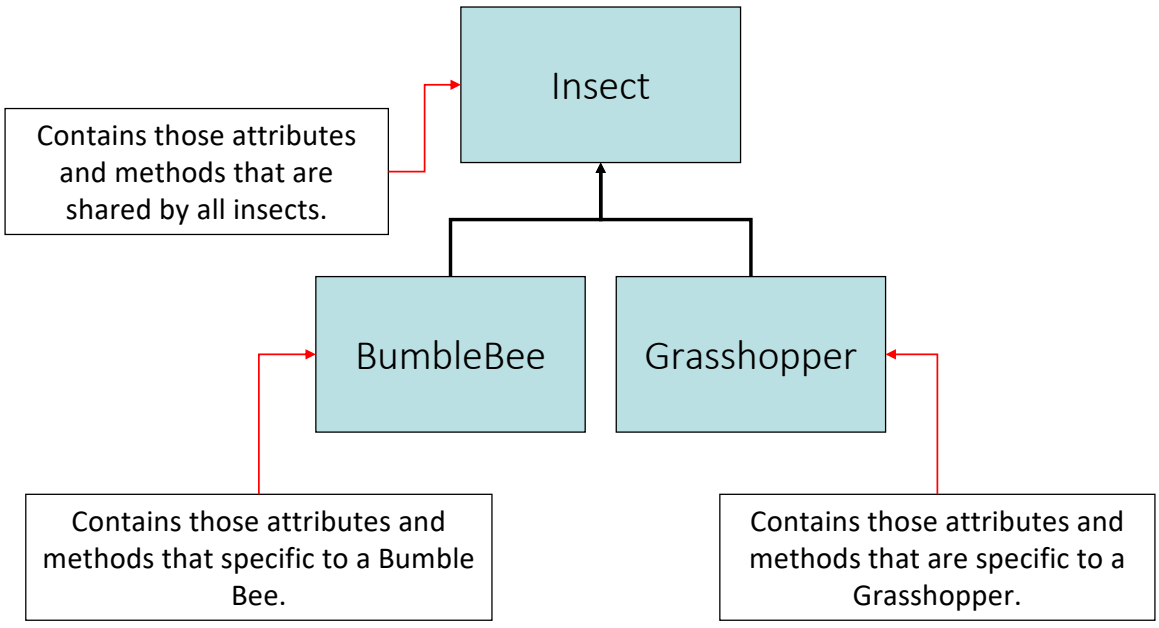
Part a: Inheritance (Ch 6)

Java Inheritance

What is Inheritance?
Generalization vs. Specialization

- Real-life objects are typically specialized versions of other more general objects.
- The term “insect” describes a very general type of creature with numerous characteristics.
- Grasshoppers and bumblebees are insects
  - They share the general characteristics of an insect.
  - However, they have special characteristics of their own.
    - grasshoppers have a jumping ability, and
    - bumblebees have a stinger.
- Grasshoppers and bumblebees are specialized versions of an insect.

Inheritance



The “is a” Relationship

- The relationship between a superclass and an inherited class is called an “is a” relationship.
  - A grasshopper “is a” insect.
  - A poodle “is a” dog.
  - A car “is a” vehicle.
- A specialized object has:
  - all of the characteristics of the general object, plus
  - additional characteristics that make it special.
- In object-oriented programming, *inheritance* is used to create an “is a” relationship among classes.

The “is a” Relationship (2)

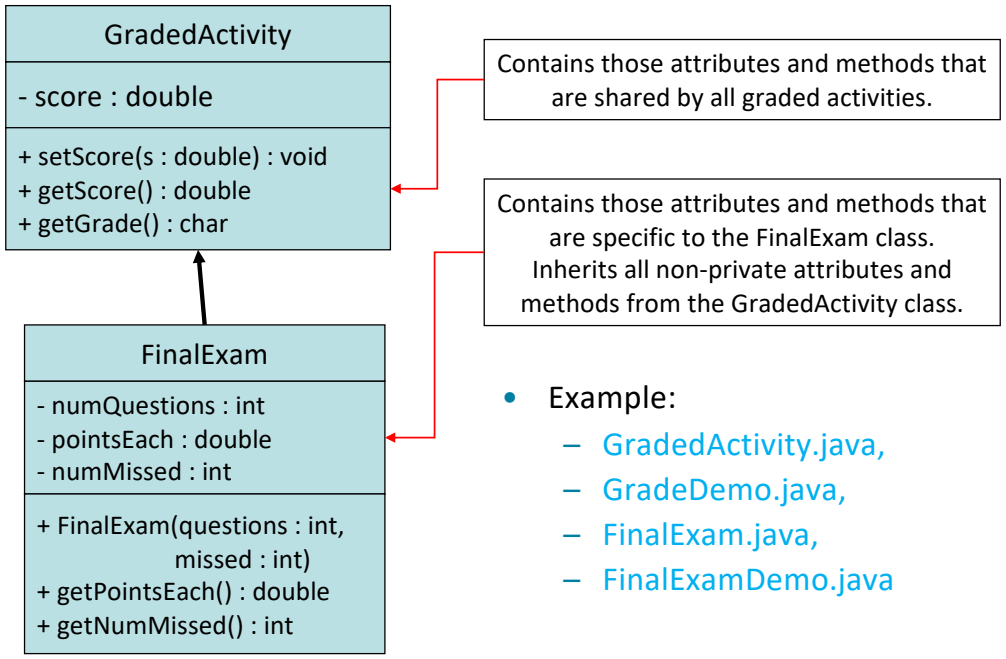
- We can ***extend*** the capabilities of a class.
- Inheritance involves a superclass and a subclass.
  - The *superclass* is the general class and
  - the *subclass* is the specialized class.
- The subclass is based on, or extended from, the superclass.
  - Superclasses are also called ***base classes***, and
  - subclasses are also called ***derived classes***.
- The relationship of classes can be thought of as *parent classes* and *child classes*.

Inheritance

- The subclass inherits fields and methods from the superclass without any of them being rewritten.
- New fields and methods may be added to the subclass.
- The Java keyword, *extends*, is used on the class header to define the subclass.

```
public class FinalExam extends GradedActivity
```

The `GradedActivity` Example



- Example:
  - GradedActivity.java,
  - GradeDemo.java,
  - FinalExam.java,
  - FinalExamDemo.java

Inheritance, Fields and Methods

- Members of the superclass that are marked *private*:
  - are not inherited by the subclass,
  - exist in memory when the object of the subclass is created
  - may only be accessed from the subclass by public methods of the superclass.
- Members of the superclass that are marked *public*:
  - are inherited by the subclass, and
  - may be directly accessed from the subclass.

Inheritance, Fields and Methods (2)

- When an instance of the subclass is created, the non-private methods of the superclass are available through the subclass object.

```
FinalExam exam = new FinalExam();
exam.setScore(85.0);
System.out.println("Score = "
                   + exam.getScore());
```

- Non-private methods and fields of the superclass are available in the subclass.

```
setScore(newScore);
```

The Superclass's Constructor

- Constructors are not inherited.
- When a subclass is instantiated, the superclass default constructor is executed first.

- The `super` keyword refers to an object's superclass.
- The superclass constructor can be explicitly called from the subclass by using the `super` keyword.
- Example:
  - SuperClass1.java, SubClass1.java, ConstructorDemo1.java
  - Rectangle.java, Cube.java, CubeDemo.java

Calling The Superclass Constructor

- If a parameterized constructor is defined in the superclass,
  - the superclass must provide a no-arg constructor, or
    - subclasses must provide a constructor, and
    - subclasses must call a superclass constructor.
- Calls to a superclass constructor must be the first java statement in the subclass constructors.
  - If missing, `super()`; automatically added as first line.

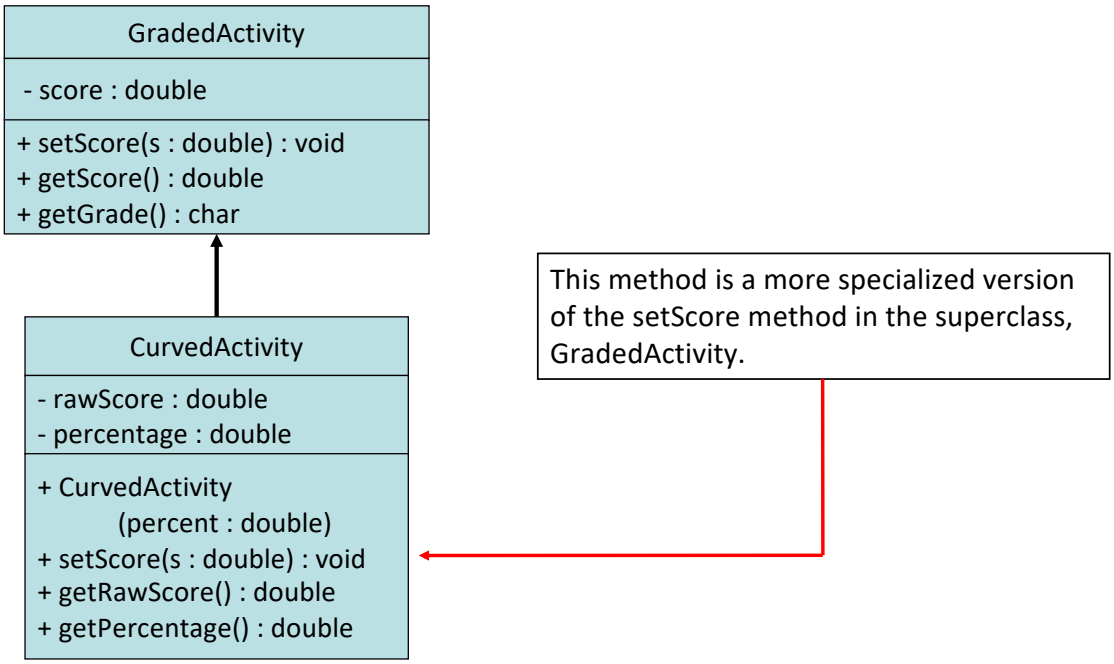
Notes about `super`

- Constructor Chaining
- Each subclass calls its superclass's constructor.
- Creates a chain of constructor calls.
- Ensures superclasses are initialized before subclasses
  - (Except if base class calls a method which is overridden in derived class.)
- Can chain to constructors of current class using `this()`

Overriding Superclass Methods

- A subclass may have a method with the same signature as a superclass method.
- The subclass method overrides the superclass method.
- Example:
  - GradedActivity.java, CurvedActivity.java, CurvedActivityDemo.java

Overriding Superclass Methods (2)



Overriding Superclass Methods (3)

- Recall that a method's *signature* consists of:
  - the method's name
  - the data types method's parameters in the order that they appear.
- A subclass method that overrides a superclass method must have the same signature as the superclass method.
- An object of the subclass invokes the subclass's version of the method, not the superclass's.
- The `@Override` annotation should be used just before the subclass method declaration.
  - This causes the compiler to display a error message if the method fails to correctly override a method in the superclass.

Overriding Superclass Methods (4)

- An subclass method can call the overridden superclass method via the super keyword.

```
super.setScore(rawScore * percentage);
```

- There is a distinction between overloading a method and overriding a method.
- Overloading is when a method has the same name as one or more other methods, but with a different signature.
- When a method overrides another method, however, they both have the same signature.

Overriding Superclass Methods (5)

- Both overloading and overriding can take place in an inheritance relationship.
- Overriding can only take place in an inheritance relationship.
- Example:
  - SuperClass3.java,
  - SubClass3.java,
  - ShowValueDemo.java

Preventing a Method from Being Overridden

- The `final` modifier will prevent the overriding of a superclass method in a subclass.

```
public final void message()
```

- If a subclass attempts to override a final method, the compiler generates an error.
- This ensures that a particular superclass method is used by subclasses rather than a modified version of it.

Protected Members

- Protected members of class:
  - may be accessed by methods in a subclass, and
  - by methods in the same package as the class.
- Java provides a third access specification, `protected`.
- A *protected* member's access is somewhere between *private* and *public*.
- Example:
  - GradedActivity.java
  - FinalExam.java

Protected Members (2)

- Using `protected` instead of `private` makes some tasks easier.
- However, any class that is derived from the class, or is in the same package, has unrestricted access to the protected member.
- It is always better to make all fields `private` and then provide `public` methods for accessing those fields.
- If no access specifier for a class member is provided, the class member is given *package access* by default.
- Any method in the same package may access the member.

Access Specifiers

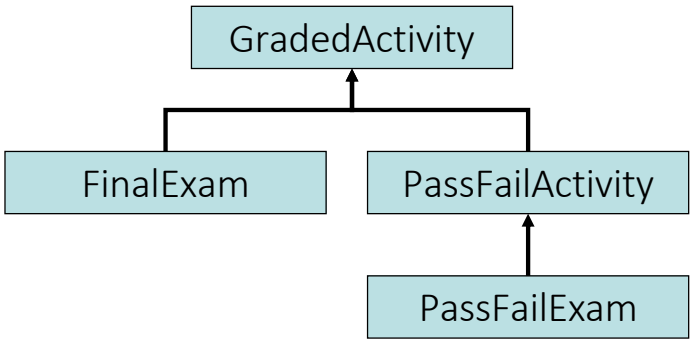
Access Modifier	Accessible to a subclass inside the same package?	Accessible to all other classes inside the same package?
default (no modifier)		
Public		
Protected		
Private		

Access Modifier	Accessible to a subclass outside the package?	Accessible to all other classes outside the package?
default (no modifier)		
Public		
Protected		
Private		

Chains of Inheritance

- A superclass can also be derived from another class.

- Classes often are depicted graphically in a *class hierarchy*.
- A class hierarchy shows the inheritance relationships between classes.



The `Object` Class

- All Java classes are directly or indirectly derived from a class named `Object`.
- `Object` is in the `java.lang` package.
- Any class that does not specify the `extends` keyword is automatically derived from the `Object` class.

```
public class MyClass
{
   // This class is derived from Object.
}
```

- Ultimately, every class is derived from the `Object` class.

The `Object` Class (2)

- Because every class is directly or indirectly derived from the `Object` class:
  - every class inherits the `Object` class's members.
    - example: `toString` and `equals`.
- In the `Object` class, the `toString` method returns a string containing the object's class name and a hash of its memory address.
- The `equals` method accepts the address of an object as its argument and returns true if it is the same as the calling object's address.
- ObjectMethods.java