Inheritance

Lecture 9a

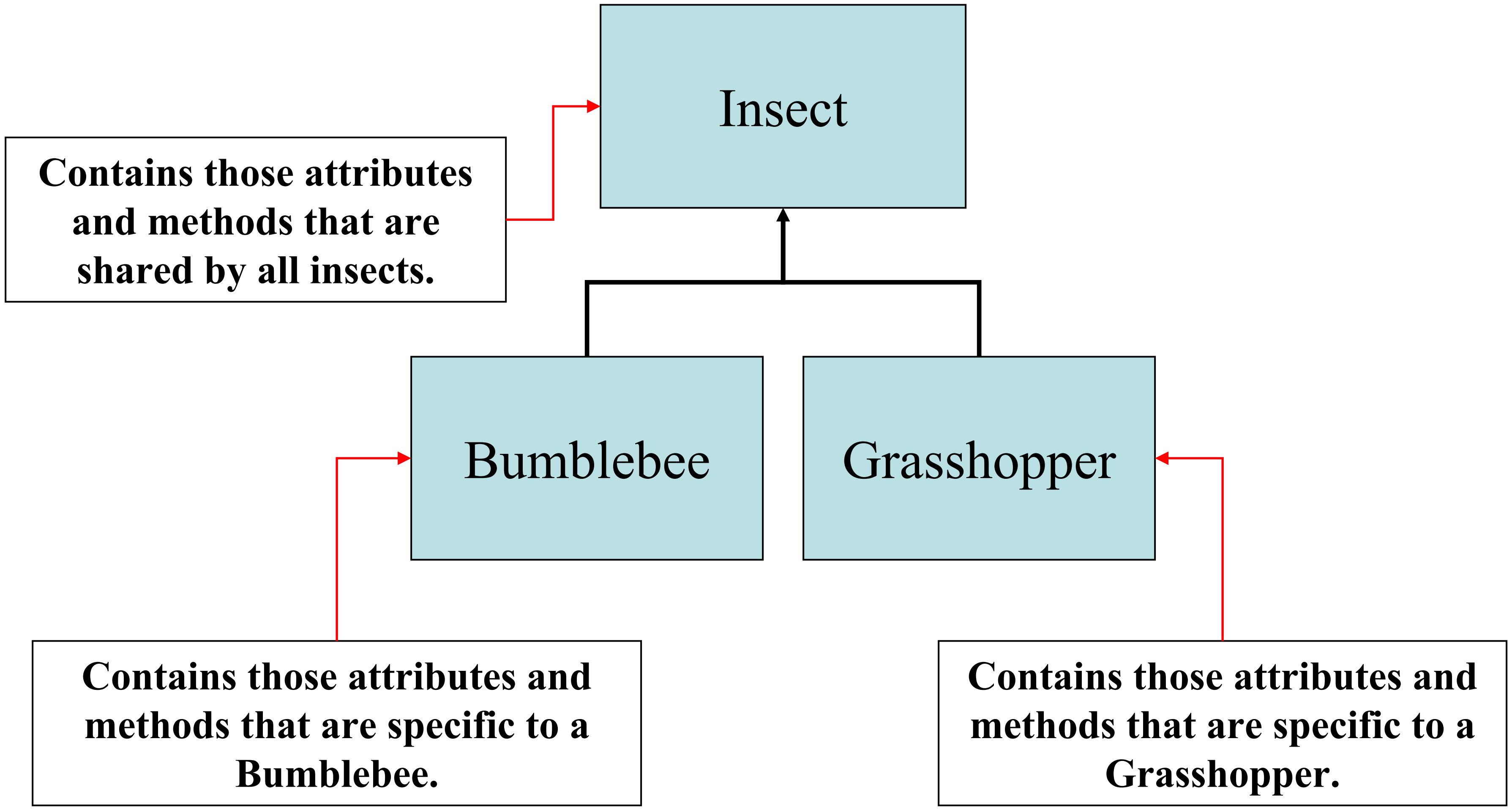
Topics

- What Is Inheritance?
- Calling the Superclass Constructor
- Overriding Superclass Methods
- Protected Members
- Chains of Inheritance
- The `Object` Class
- Polymorphism
- Abstract Classes and Abstract Methods

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


Insect
Bumblebee
Grasshopper
Contains those attributes and methods that are shared by all insects.
Contains those attributes and methods that are specific to a Bumblebee.
Contains those attributes and methods that are specific to a Grasshopper.

The “is a” Relationship (1 of 2)

- The relationship between a superclass and an inherited class is called an “is a” relationship.
  - A grasshopper “is an” insect.
  - A poodle “is a” dog.
  - A car “is a” vehicle.
- A specialized object has:
  - all 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 of 2)

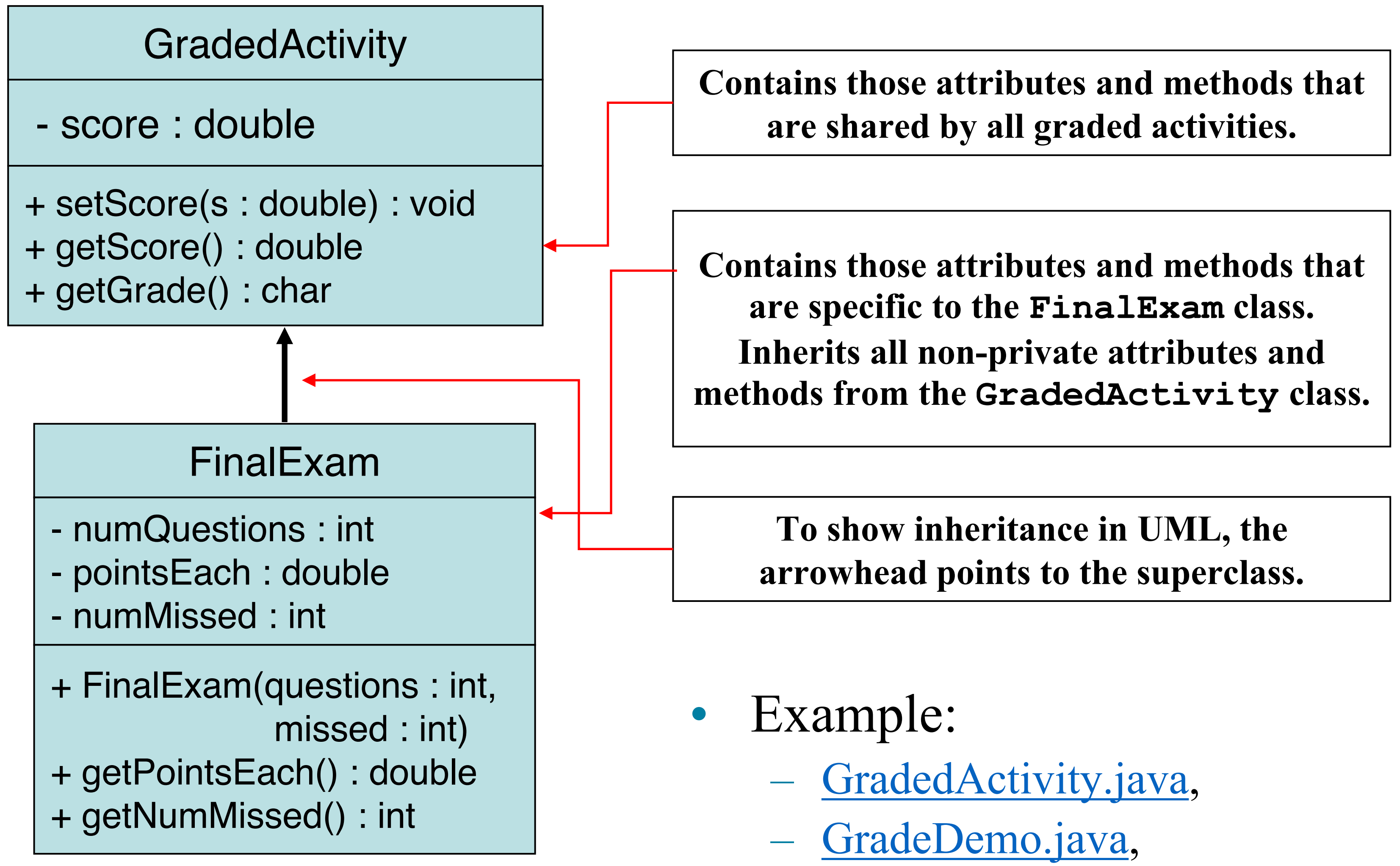
- We can *extend* the capabilities of a class.
- Inheritance involves a superclass and a subclass.
  - The *superclass* is the general class and
  - the *subclass* is the specialized class.
- The subclass is based on, or extended from, the superclass.
  - Superclasses are also called *base classes*,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

Extending a class

- 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.

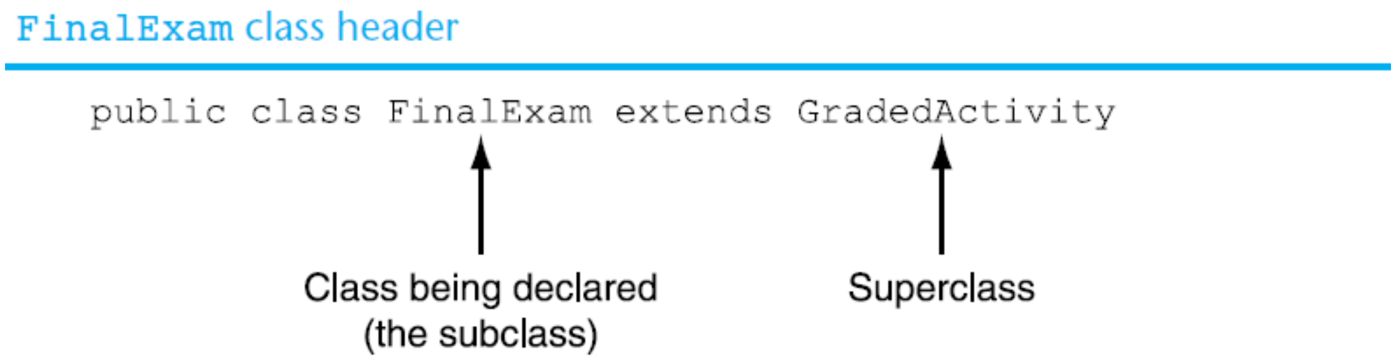
```
public class Grasshopper extends Inset
```

The `GradedActivity` Example


GradedActivity
- score : double
+ setScore(s : double) : void
+ getScore() : double
+ getGrade() : char
FinalExam
- numQuestions : int
- pointsEach : double
- numMissed : int
+ FinalExam(questions : int, missed : int)
+ getPointsEach() : double
+ getNumMissed() : int
Contains those attributes and methods that are shared by all graded activities.
Contains those attributes and methods that are specific to the FinalExam class. Inherits all non-private attributes and methods from the GradedActivity class.
To show inheritance in UML, the arrowhead points to the superclass.

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

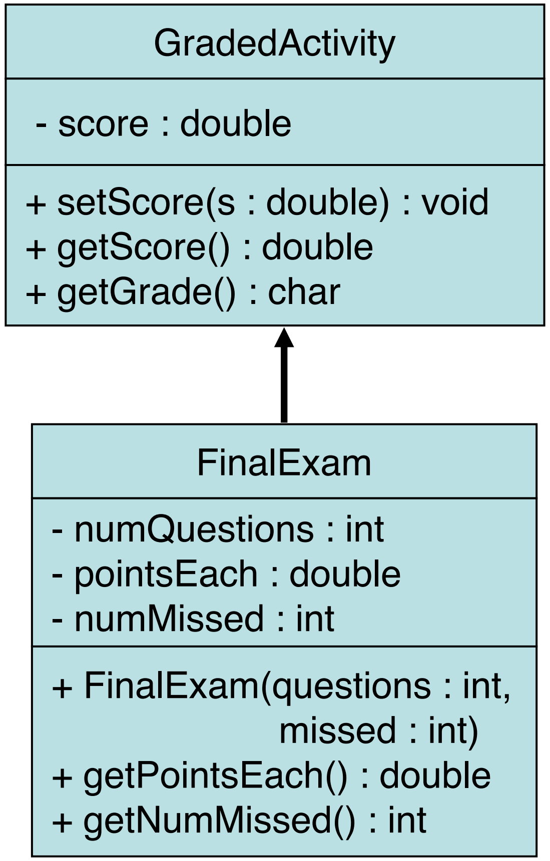
The `GradedActivity` Example


FinalExam class header
public class FinalExam extends GradedActivity
Class being declared (the subclass)
Superclass

FinalExam is a GradeActivity

The `GradedActivity` Example

- Because the FinalExam class extends the GradedActivity class, it inherits all the non-private members of the GradedActivity class. Here is a list of the members of the FinalExam class.



Fields:	
`int numQuestions;`	Declared in `FinalExam`
`double pointsEach;`	Declared in `FinalExam`
`int numMissed;`	Declared in `FinalExam`
`Methods:`	
`Constructor`	Declared in `FinalExam`
`getPointsEach`	Declared in `FinalExam`
`getNumMissed`	Declared in `FinalExam`
`setScore`	Inherited from `GradedActivity`
`getScore`	Inherited from `GradedActivity`
`getGrade`	Inherited from `GradedActivity`

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

Inheritance, Fields and Methods (1 of 3)

- 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 by 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 of 3)

- 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.

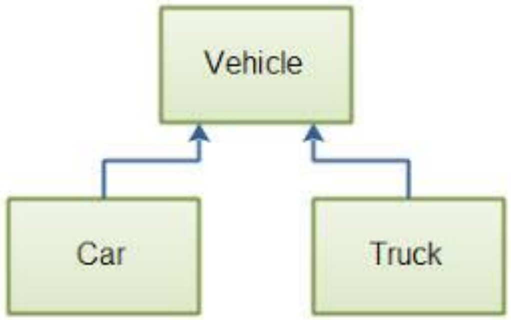
```
setScore(newScore);
```

Inheritance, Fields and Methods (3 of 3)

Point to ponder #1:

Is it possible for a superclass to call a subclass's method?

No, in an inheritance relationship, the subclass inherits members from the superclass, not the other way around.



Car and Truck inherit members from Vehicle, but Vehicle does not inherit from Car or Truck.

Inheritance and Constructors

- Constructors are not inherited.
- When a subclass is instantiated, an instance of its parent class is created implicitly, and the superclass default constructor is executed first.
- Example:
  - SuperClass1.java
  - SubClass1.java
  - ConstructorDemo1.java

The Superclass's Constructor

- The `super` keyword refers to an object's superclass (reference variable) and can be used to access members of the superclass.
- The superclass constructor can be explicitly called from the subclass by using the `super` keyword.
- Example:
  - SuperClass2.java,
  - SubClass2.java, ConstructorDemo2.java
  - Rectangle.java, Cube.java, CubeDemo.java

Calling The Superclass Constructor

- The super statement that calls the superclass constructor may be written only in the subclass’s constructor, not in any other method.
- The super statement that calls the superclass constructor must be the first statement in the subclass’s constructor, since it must execute before the code in the subclass’s constructor executes.
- If a subclass constructor does not explicitly call a superclass constructor, Java will automatically call the superclass’s default constructor, or no-arg constructor, just before the code in the subclass’s constructor executes. This is equivalent to:

```
super();
```

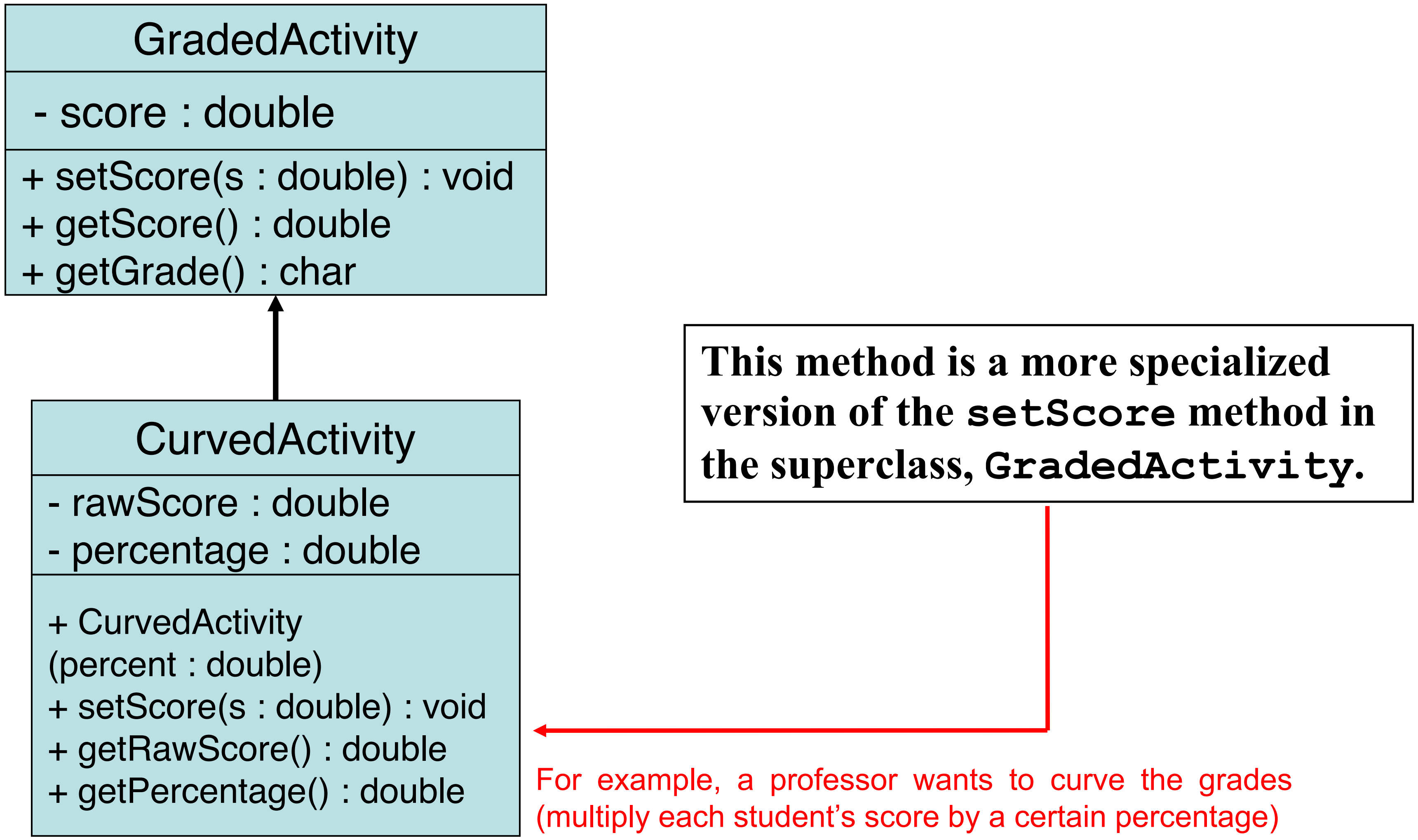
Calling The Superclass Constructor

- If a superclass does not have a default and a no-arg constructor, then a class that inherits from it must call one of the constructors that the superclass does have.
- If it does not, an error will result when the subclass is compiled.

Overriding Superclass Methods (1 of 6)

- A subclass may have a method with the same signature as a superclass method.
- The subclass method overrides the superclass method with a more suitable behavior.
- This is known as *method overriding*.

Overriding Superclass Methods (2 of 6)



This method is a more specialized version of the `setScore` method in the superclass, `GradedActivity`.

For example, a professor wants to curve the grades (multiply each student's score by a certain percentage)

Overriding Superclass Methods (3 of 6)

- 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 an error message if the method fails to correctly override a method in the superclass.

- Example:
  - GradedActivity.java, CurvedActivity.java, CurvedActivityDemo.java

Overriding Superclass Methods (4 of 6)

- Point to ponder #2:

What is the 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 of 6)

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

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

Overriding Superclass Methods (6 of 6)

- Point to ponder #3:

Both overloading and overriding can take place in an inheritance relationship?

Yes.

Overriding can happen in the same class?

No, it must happen only in an inheritance relationship.

- Example:
  - SuperClass3.java,
  - SubClass3.java,
  - ShowValueDemo.java

Preventing a Method from Being Overridden (1 of 2)

- 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.

Preventing a Method from Being Overridden (2 of 2)

- Point to ponder #1:

What is the goal of making a method final in the superclass?

This ensures that a particular superclass method is used by subclasses rather than a modified version of it.

Abstract Classes and Methods (1 of 6)

- An abstract class cannot be instantiated, but other classes can inherit from it.
- An *Abstract class* serves as a superclass for other classes.
- The abstract class represents the generic or abstract form of all the classes that are derived from it.
- A class becomes abstract when you place the abstract key word in the class definition.

```
public abstract class ClassName
```

Abstract Classes and Methods (2 of 6)

- An abstract method has no body and must be overridden in a subclass.
- An *abstract method* is a method that appears in a superclass but expects to be overridden in a subclass.
- An abstract method has only a header and no body.

```
AccessSpecifier abstract ReturnType MethodName(ParameterList);
```

- Example:
  - Student.java, CompSciStudent.java, CompSciStudentDemo.java

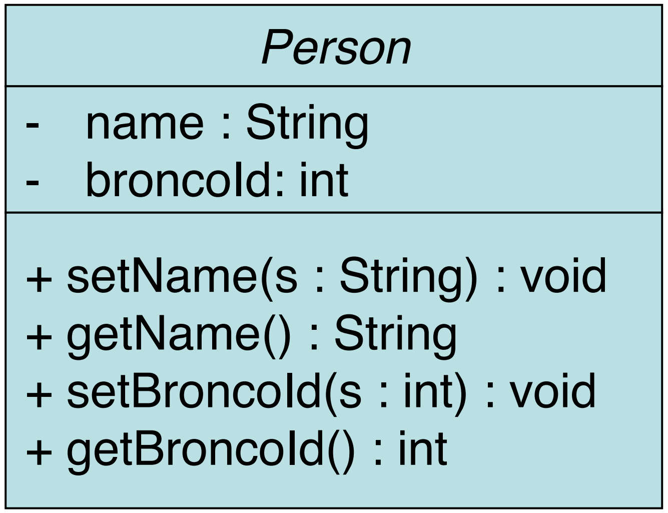
Abstract Classes and Methods (3 of 6)

- Notice that the key word `abstract` appears in the header, and that the header ends with a semicolon.

```
public abstract void setValue(int value);
```

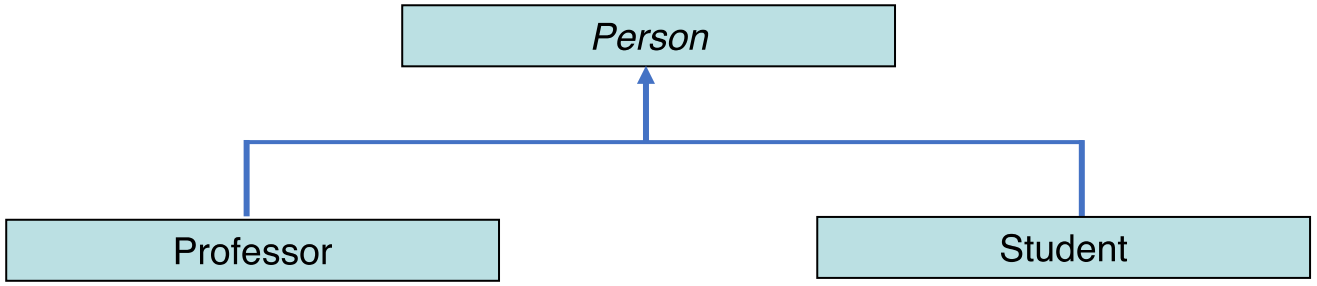
- Any class that contains an abstract method is automatically abstract.
- If a subclass fails to override an abstract method, a compiler error will result.
- Abstract methods are used to ensure that a subclass implements the method.

Abstract Classes and Methods (4 of 6)



- Abstract classes are drawn like regular classes in UML, except the name of the class and the names of abstract methods are shown in italics. For example, Person is an abstract class.

Abstract Classes and Methods (5 of 6)



- Point to ponder #5:

A class that is not abstract is what type of class?

A concrete class.

Inheritance

Lecture 9b

Topics

- What Is Inheritance?
- Calling the Superclass Constructor
- Overriding Superclass Methods
- Protected Members
- Chains of Inheritance
- The `Object` Class
- Polymorphism
- Abstract Classes and Abstract Methods

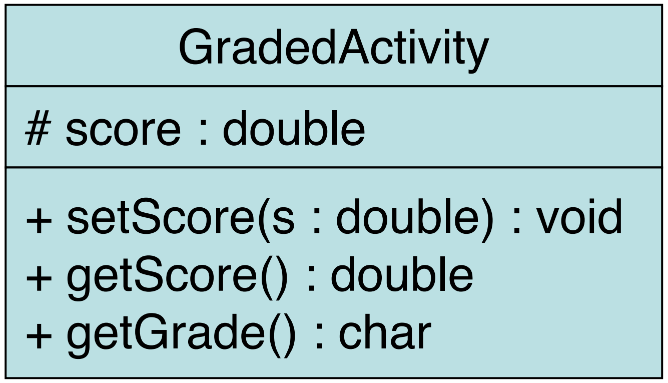
Protected Members (1 of 3)

- 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:
  - GradedActivity2.java
  - FinalExam2.java
  - ProtectedDemo.java

Protected Members (2 of 3)

- 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.

Protected Members (3 of 3)

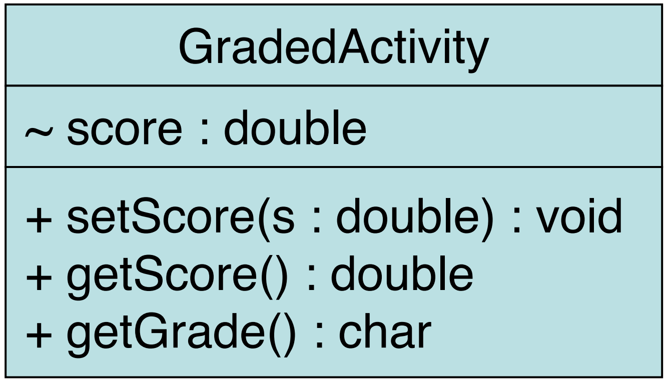


- Protected class members may be denoted in a UML diagram with the `#` symbol. E.g., the `score` field denoted as protected in GradedActivity class.

Default Access Modifier (package access)

- If you do not provide an access specifier for a class member, the class member is given package access by default. This means that any method in the same package may access the member. Here is an example:

```
public class Circle
{
double radius;
int centerX, centerY;
(Method definitions follow . . .)
}
```



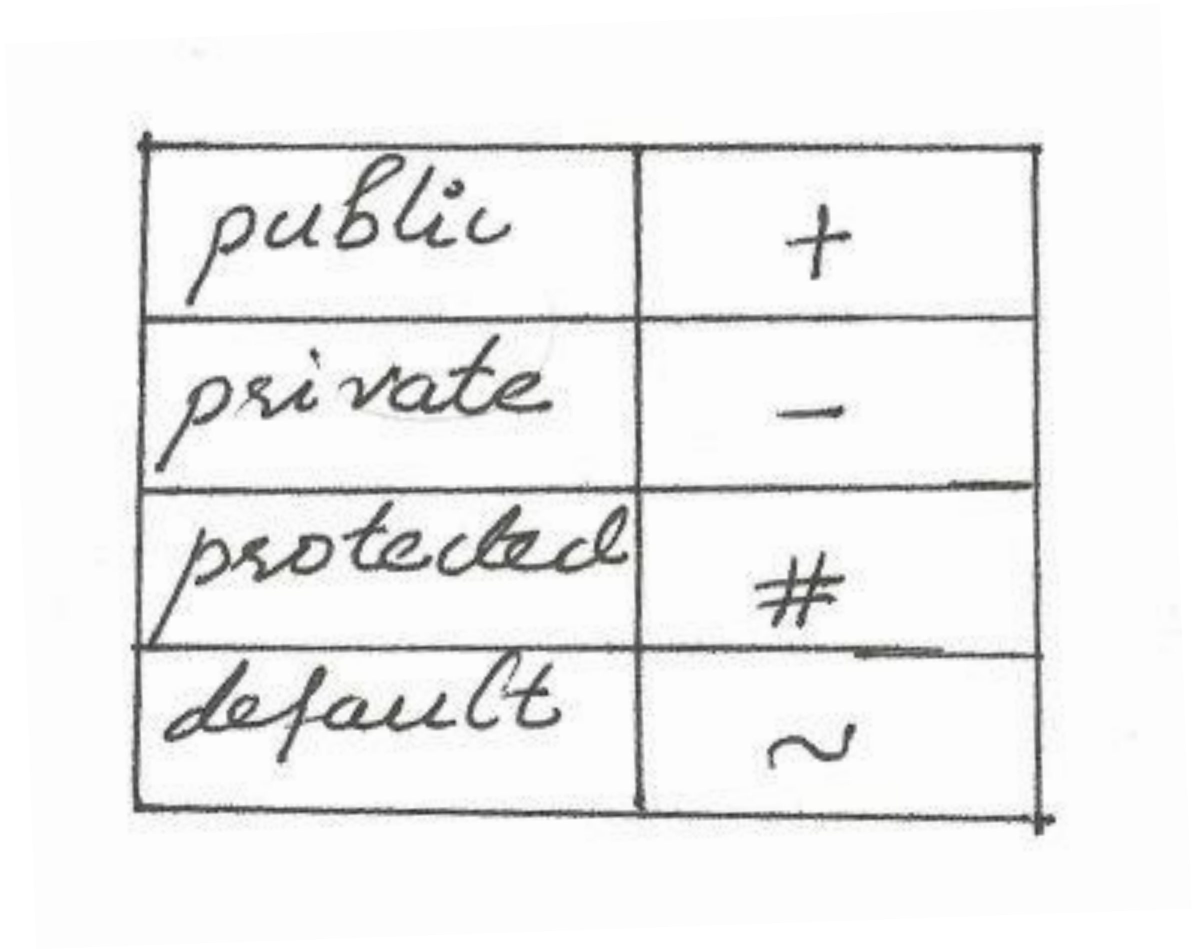
- Package access may be denoted in a UML diagram with the ~ symbol.

Access Specifiers (1 of 2)

Access Modifier	Accessible to a subclass inside the same package?	Accessible to all other classes inside the same package?
default (no modifier)	Yes	Yes
public	Yes	Yes
protected	Yes	Yes
private	No	No

Access Modifier	Accessible to a subclass outside the package?	Accessible to all other classes outside the package?
default (no modifier)	No	No
public	Yes	Yes
protected	Yes	No
private	No	No

Access Specifiers (2 of 2)

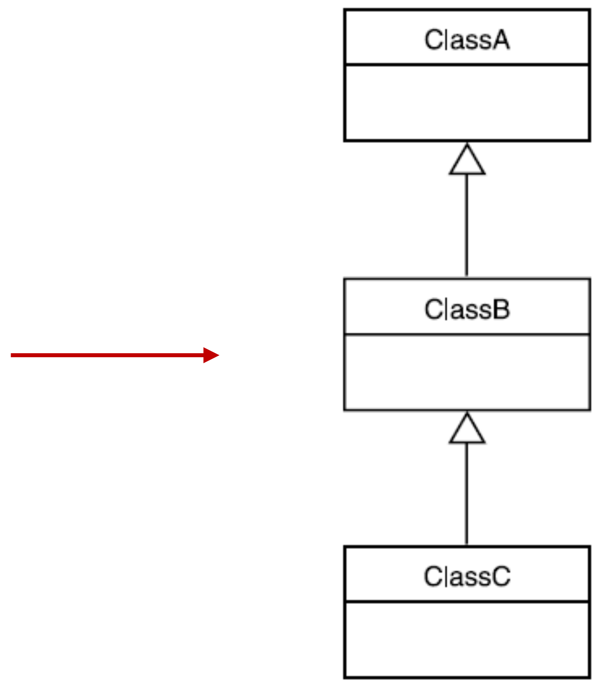


public	+
private	−
protected	#
default	~

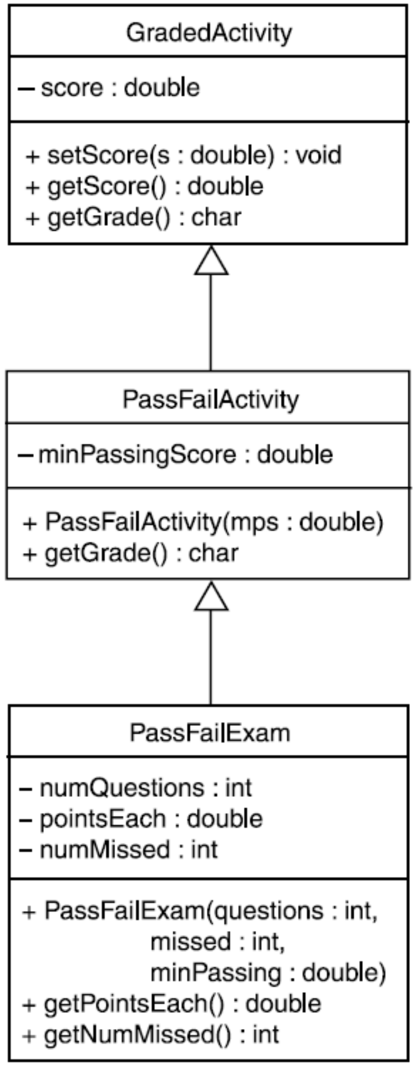
Chains of Inheritance (1 of 3)

- A superclass can also inherit from another class.

ClassC inherits ClassB's members, including the ones that ClassB inherited from ClassA.



Chains of Inheritance (2 of 3)

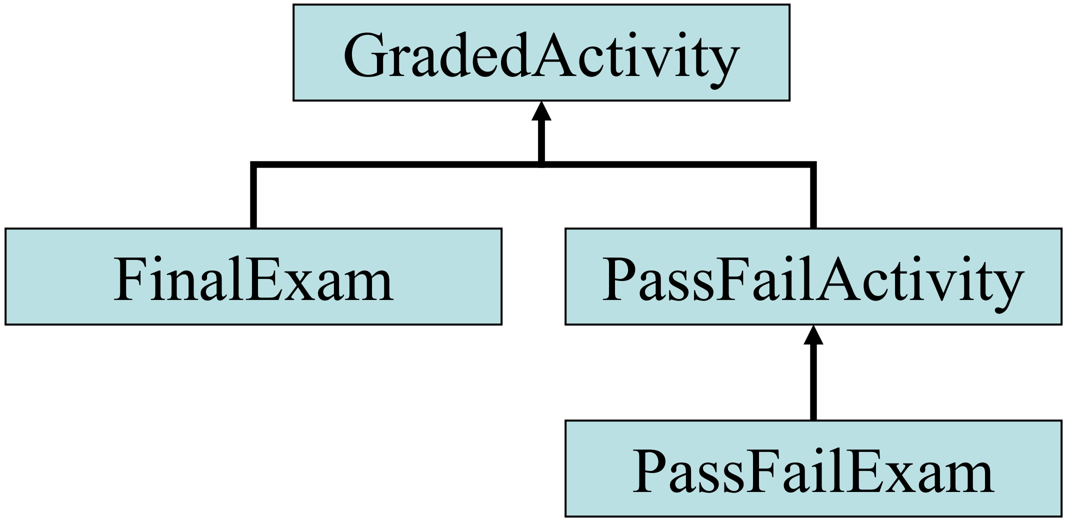


Example:

- GradedActivity.java
- PassFailActivity.java
- PassFailExam.java
- PassFailExamDemo.java

Chains of Inheritance (3 of 3)

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



The `Object` Class (1 of 4)

- 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.

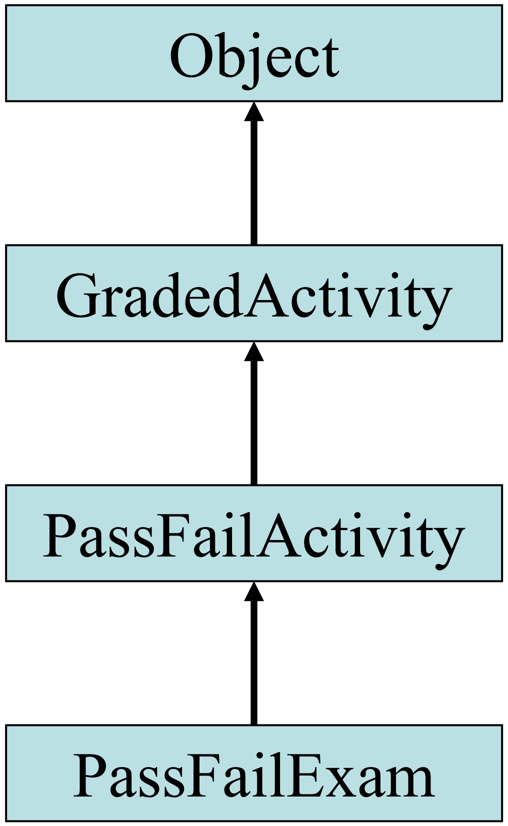
```
public class MyClass extends Object
{
  // This class is derived from Object.
}
```

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

The `Object` Class (2 of 4)

- 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.
- Example: ObjectMethods.java

The `Object` Class (3 of 4)



The `Object` Class (4 of 4)

- Point to ponder #4:

When you specify a toString() method for a given class, what you are doing in term of inheritance??

Overriding the toString() method inherited from the Object class.

Polymorphism (1 of 4)

- A superclass reference variable can reference objects of a subclasses.

```
GradedActivity exam;
```

- We can use the exam variable to reference a `GradedActivity` object.

```
exam = new GradedActivity();
```

- The `GradedActivity` class is also used as the superclass for the `FinalExam` class.
- An object of the `FinalExam` class ***is a*** `GradedActivity` object.

Polymorphism (2 of 4)

- A `GradedActivity` variable can be used to reference a `FinalExam` object.

```
GradedActivity exam = new FinalExam(50, 7);
```

- This statement creates a `FinalExam` object and stores the object's address in the exam variable.
- This is an example of polymorphism.
- The term *polymorphism* means the ability to take many forms.
- In Java, a reference variable is *polymorphic* because it can reference objects of types different from its own, as long as those types are subclasses of its type.

Polymorphism (3 of 4)

- Other legal polymorphic references:

```
GradedActivity exam1 = new FinalExam(50, 7);
GradedActivity exam2 = new PassFailActivity(70);
GradedActivity exam3 = new PassFailExam(100, 10, 70);
```

- The `GradedActivity` class has three methods: `setScore`, `getScore`, and `getGrade`.
- A `GradedActivity` variable can be used to call only those three methods.

```
GradedActivity exam = new PassFailExam(100, 10, 70);
System.out.println(exam.getScore()); // This works.
System.out.println(exam.getGrade()); // This works.
System.out.println(exam.getPointsEach()); // ERROR!
```

Polymorphism and Dynamic Binding

- When a superclass variable references a subclass object that has overridden a method in the superclass, the subclass's version of the method will be run when it is called.

```
GradedActivity exam = new PassFailActivity(60);
exam.setScore(70);
System.out.println(exam.getGrade());
```

- Java performs *dynamic binding* or *late binding* when a variable contains a polymorphic reference.
- The Java Virtual Machine determines at runtime which method to call, depending on the type of object that the variable references.

Polymorphism (4 of 4)

- It is the object’s type, rather than the reference type, that determines which method is called.
- Example:
  - Polymorphic.java

The instanceof Operator (1 of 2)

- The instanceof method can be used to determine whether an object is an instance of a particular class.

```
GradedActivity activity = new GradedActivity();
if (activity instanceof GradedActivity)
   System.out.println("Yes, activity is a GradedActivity.");
else
   System.out.println("No, activity is not a GradedActivity.");
```

This code will display "Yes, activity is a GradedActivity."

The instanceof Operator (2 of 2)

- The instanceof operator understands the “is-a” relationship that exists when a class inherits from another class.

```
FinalExam exam = new FinalExam(20, 2);
if (exam instanceof GradedActivity)
   System.out.println("Yes, exam is a GradedActivity.");
else
   System.out.println("No, exam is not a GradedActivity.");
```

Even though the object referenced by `exam` is a `FinalExam` object, this code will display "Yes, exam is a GradedActivity." The instanceof operator returns true because `FinalExam` is a subclass of `GradedActivity`.