Chapter 10: Inheritance

Starting Out with Java: From Control Structures through Objects

Fifth Edition

by Tony Gaddis

PEARSON ALWAYS LEARNING

Troy University Resources

Chapter Topics

Chapter 10 discusses the following main 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
- Interfaces

©2013 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

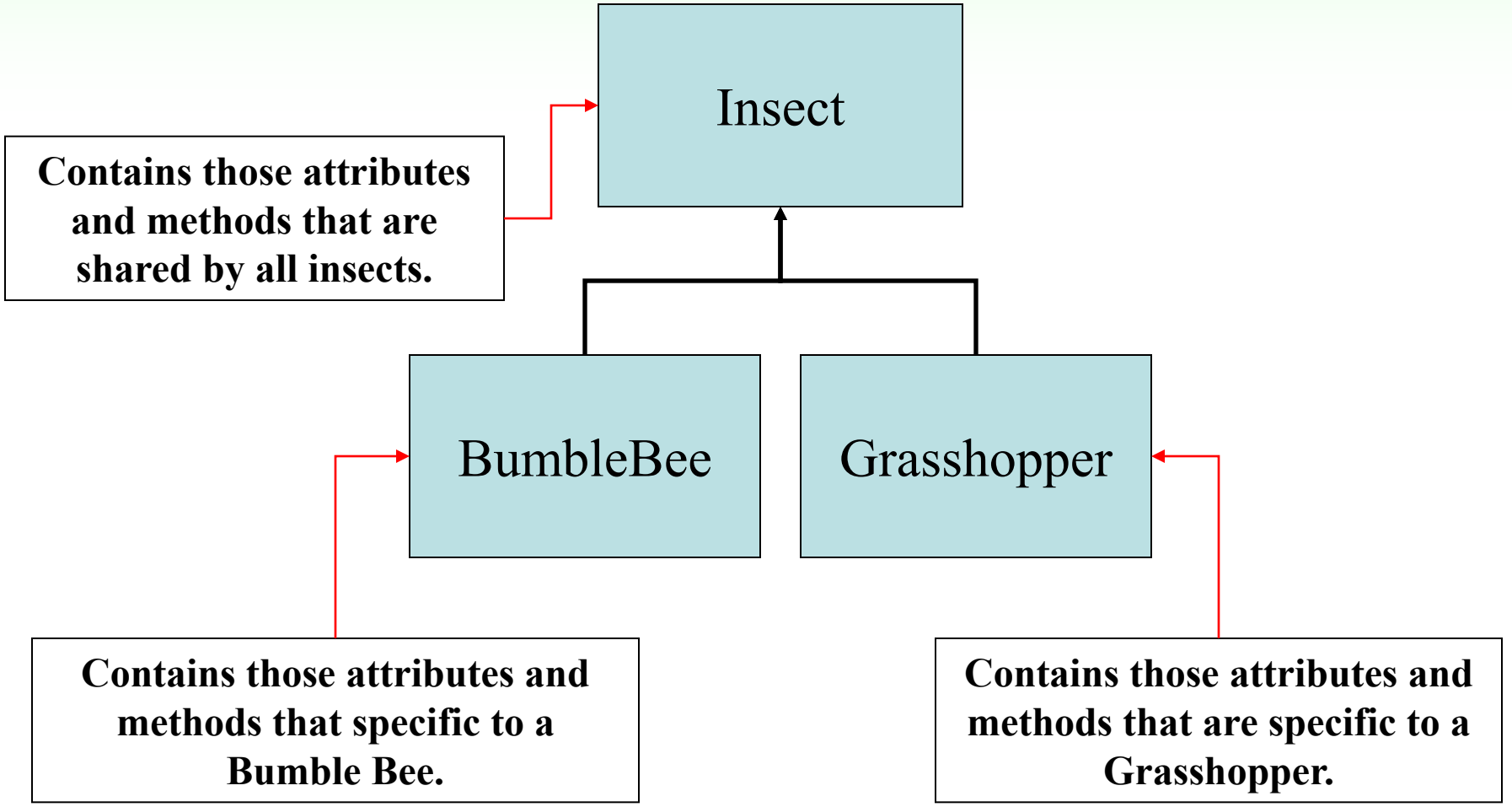
What is Inheritance?
Generalization vs. Specialization

- 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.

©2013 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

Troy University Resources

Inheritance



©2013 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

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.

©2013 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

Troy University Resources

The “is a” Relationship

- 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*.

©2013 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

Inheritance

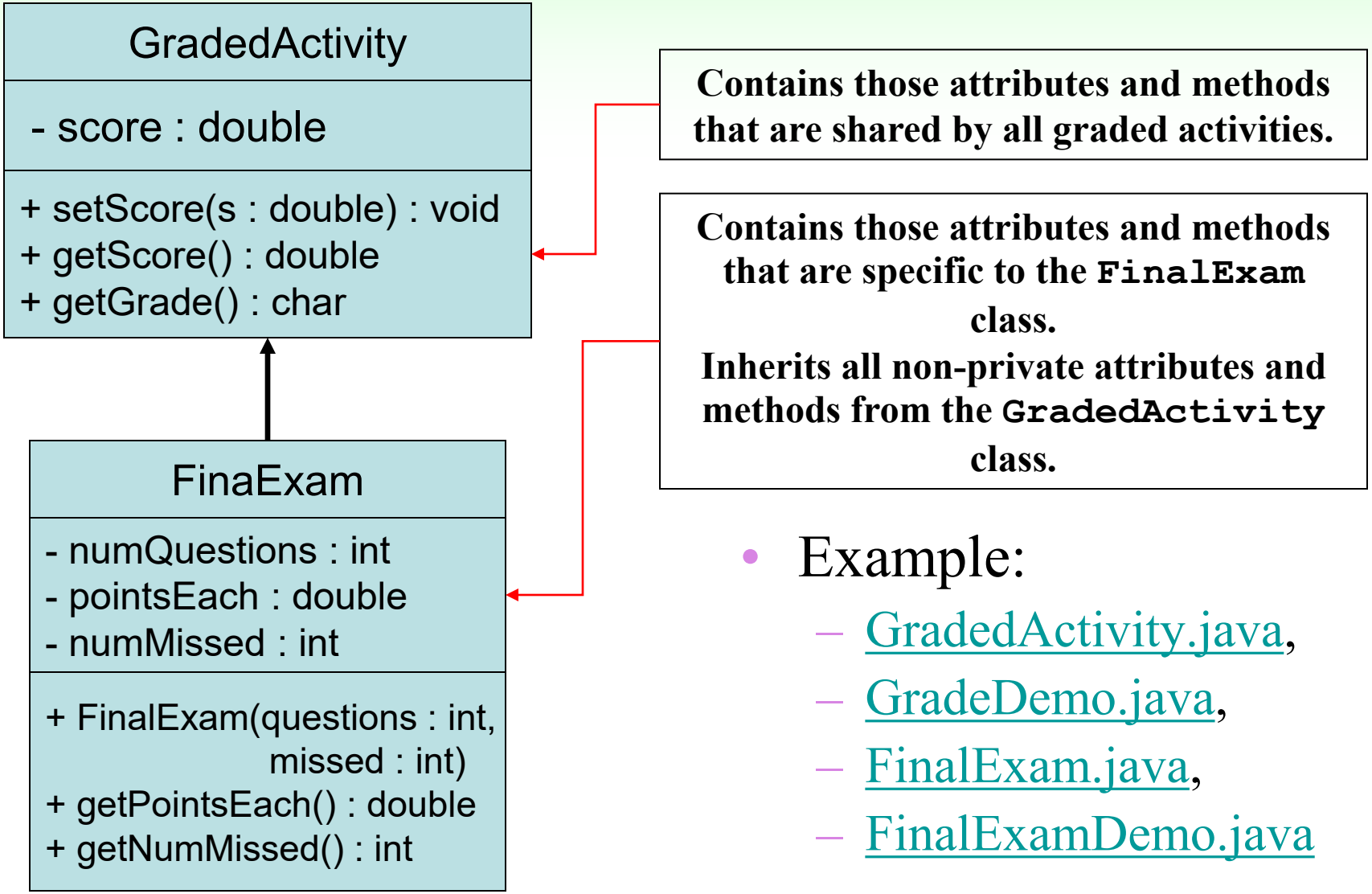
- The subclass inherits fields and methods from the superclass without any of them being rewritten.
- New fields and methods may be added to the subclass.
- The Java keyword, *extends*, is used on the class header to define the subclass.

```
public class FinalExam extends GradedActivity
```

©2013 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

Troy University Resources

The `GradedActivity` Example



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

©2013 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

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.

©2013 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

Troy University Resources

Inheritance, Fields and Methods

- 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);
```

©2013 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

Inheritance and Constructors

- Constructors are not inherited.
- When a subclass is instantiated, the superclass default constructor is executed first.
- Example:
  - SuperClass1.java
  - SubClass1.java
  - ConstructorDemo1.java

©2013 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

Troy University Resources

The Superclass's Constructor

- 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:
  - SuperClass2.java, SubClass2.java, ConstructorDemo2.java
  - Rectangle.java, Cube.java, CubeDemo.java

©2013 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

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.

©2013 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

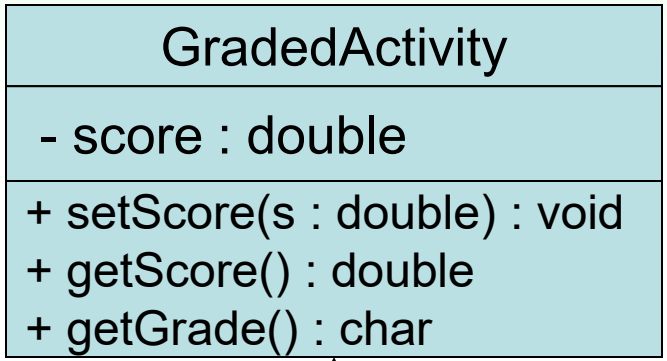
Troy University Resources

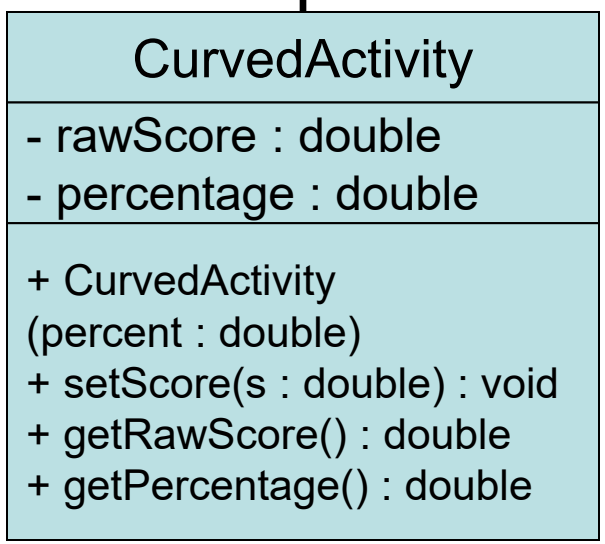
Overriding Superclass Methods

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

©2013 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

Overriding Superclass Methods





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

©2013 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

Troy University Resources

Overriding Superclass Methods

- 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.

©2013 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

Overriding Superclass Methods

- 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.

©2013 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

Troy University Resources

Overriding Superclass Methods

- 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

©2013 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

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.

©2013 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

Troy University Resources

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

©2013 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

Protected Members

- 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.

©2013 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

Troy University Resources

Access Specifiers

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

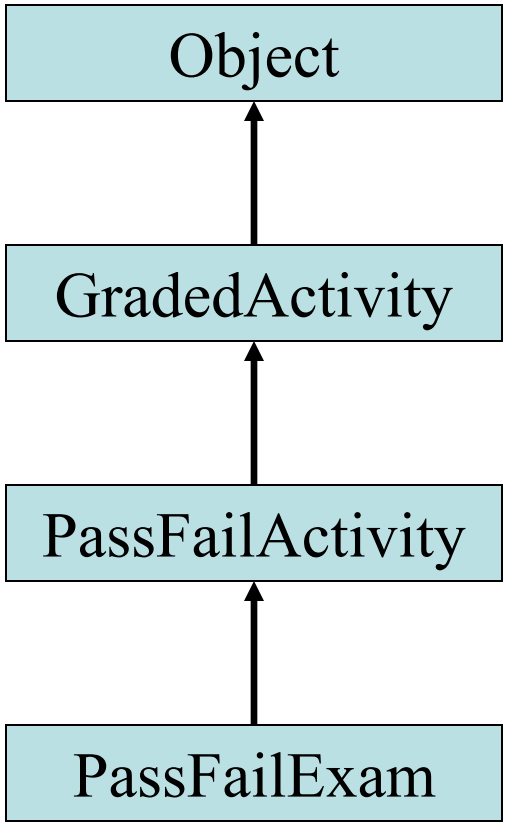
©2013 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

Chains of Inheritance

- A superclass can also be derived from another class.

Example:

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

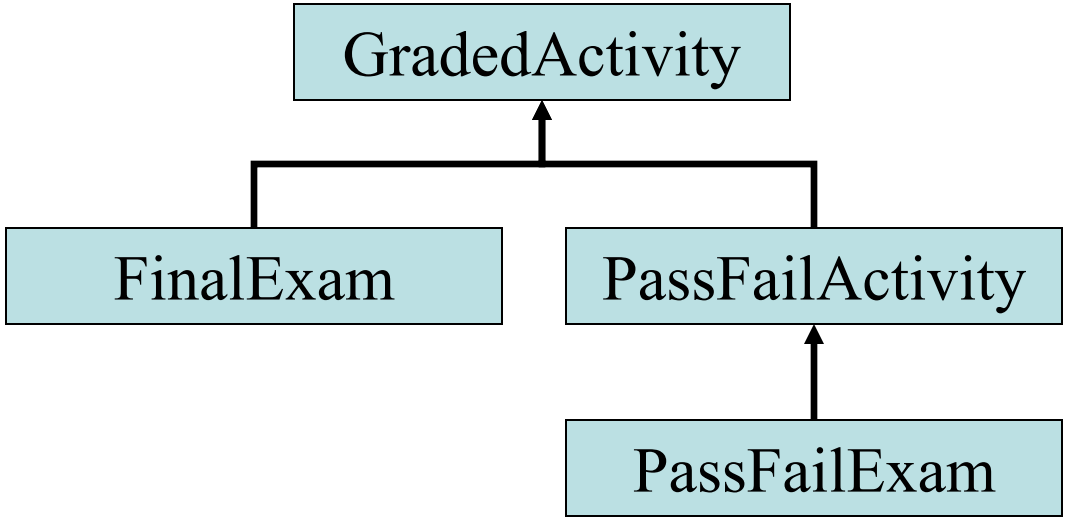


©2013 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

Troy University Resources

Chains of Inheritance

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



©2013 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

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.

©2013 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

Troy University Resources

The `Object` Class

- 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

©2013 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

Polymorphism

- A reference variable can reference objects of classes that are derived from the variable's class.

```
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.

©2013 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

Troy University Resources

Polymorphism

- 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.

©2013 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

Polymorphism

- 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!
```

©2013 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

Troy University Resources

Polymorphism and Dynamic Binding

- If the object of the subclass has overridden a method in the superclass:
  - If the variable makes a call to that method the subclass's version of the method will be run.

```
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.

©2013 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

Polymorphism

- It is the object's type, rather than the reference type, that determines which method is called.
- Example:
  - Polymorphic.java
- You cannot assign a superclass object to a subclass reference variable.

©2013 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

Abstract Classes

- An abstract class cannot be instantiated, but other classes are derived 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
```

©2013 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

Abstract Methods

- 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

©2013 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

Troy University Resources

Abstract Methods

- 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.

©2013 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

Interfaces

- An *interface* is similar to an abstract class that has all abstract methods.
  - It cannot be instantiated, and
  - all of the methods listed in an interface must be written elsewhere.
- The purpose of an interface is to specify behavior for other classes.
- An interface looks similar to a class, except:
  - the keyword `interface` is used instead of the keyword `class`, and
  - the methods that are specified in an interface have no bodies, only headers that are terminated by semicolons.

©2013 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

Troy University Resources

Interfaces

- The general format of an interface definition:

```
public interface InterfaceName
{
   (Method headers...)
}
```

- All methods specified by an interface are public by default.
- A class can implement one or more interfaces.

©2013 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

Interfaces

- If a class implements an interface, it uses the `implements` keyword in the class header.

```
public class FinalExam3 extends GradedActivity
  implements Relatable
```

- Example:
  - GradedActivity.java
  - Relatable.java
  - FinalExam3.java
  - InterfaceDemo.java

©2013 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

Fields in Interfaces

- An interface can contain field declarations:
  - all fields in an interface are treated as `final` and `static`.
- Because they automatically become `final`, you must provide an initialization value.

```
public interface Doable
{
   int FIELD1 = 1, FIELD2 = 2;
   (Method headers...)
}
```

- In this interface, `FIELD1` and `FIELD2` are `final static int` variables.
- Any class that implements this interface has access to these variables.

©2013 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

Implementing Multiple Interfaces

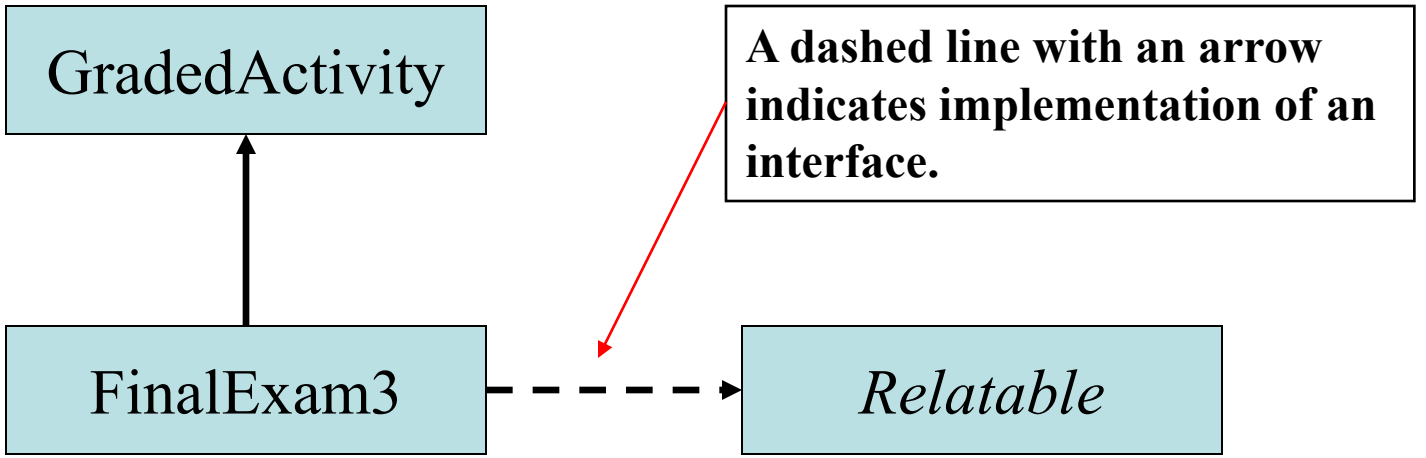
- A class can be derived from only one superclass.
- Java allows a class to implement multiple interfaces.
- When a class implements multiple interfaces, it must provide the methods specified by all of them.
- To specify multiple interfaces in a class definition, simply list the names of the interfaces, separated by commas, after the implements key word.

```
public class MyClass implements Interface1,
                                 Interface2,
                                 Interface3
```

©2013 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

Troy University Resources

Interfaces in UML



©2013 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

Polymorphism with Interfaces

- Java allows you to create reference variables of an interface type.
- An interface reference variable can reference any object that implements that interface, regardless of its class type.
- This is another example of polymorphism.
- Example:
  - RetailItem.java
  - CompactDisc.java
  - DvdMovie.java
  - PolymorphicInterfaceDemo.java

©2013 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

Troy University Resources

Polymorphism with Interfaces

- In the example code, two `RetailItem` reference variables, `item1` and `item2`, are declared.
- The `item1` variable references a `CompactDisc` object and the `item2` variable references a `DvdMovie` object.
- When a class implements an interface, an inheritance relationship known as *interface inheritance* is established.
  - a `CompactDisc` object *is a* `RetailItem`, and
  - a `DvdMovie` object *is a* `RetailItem`.

©2013 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

Polymorphism with Interfaces

- A reference to an interface can point to any class that implements that interface.
- You cannot create an instance of an interface.

```
RetailItem item = new RetailItem(); // ERROR!
```

- When an interface variable references an object:
  - only the methods declared in the interface are available,
  - explicit type casting is required to access the other methods of an object referenced by an interface reference.

©2013 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved.

Troy University Resources