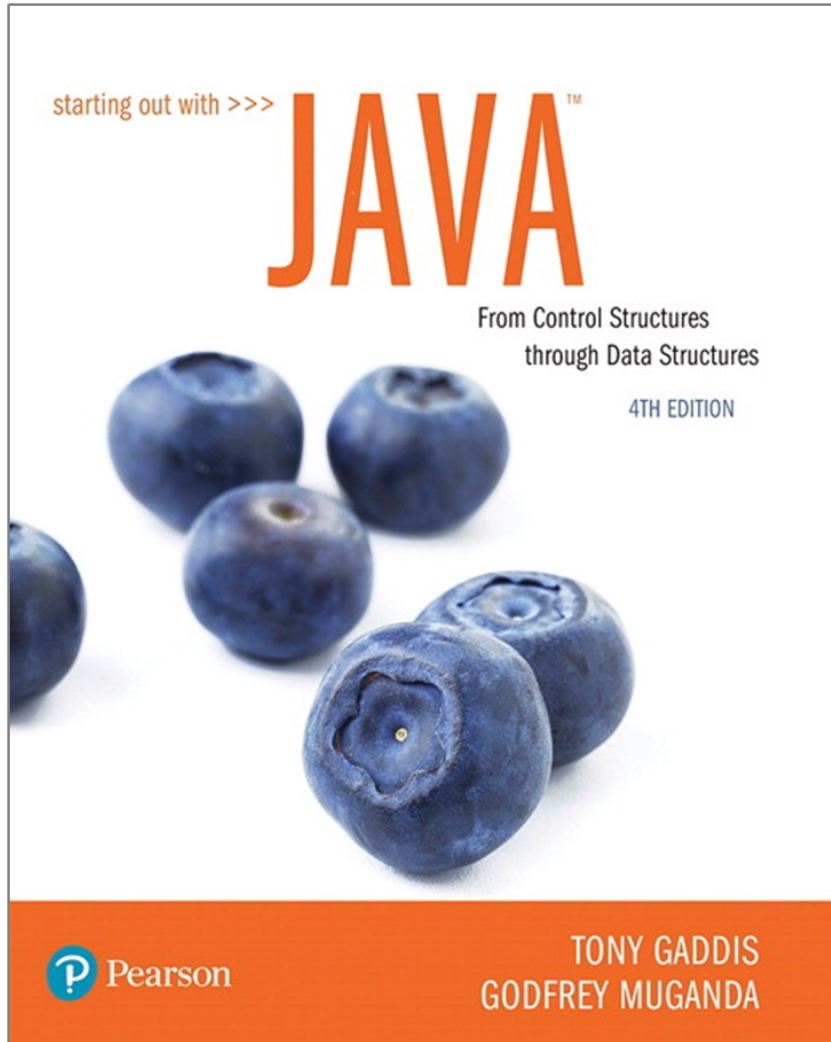


STARTING OUT WITH JAVA™

4th Edition



Chapter 10

Inheritance

Chapter Topics

Chapter 10 discusses the following main topics:

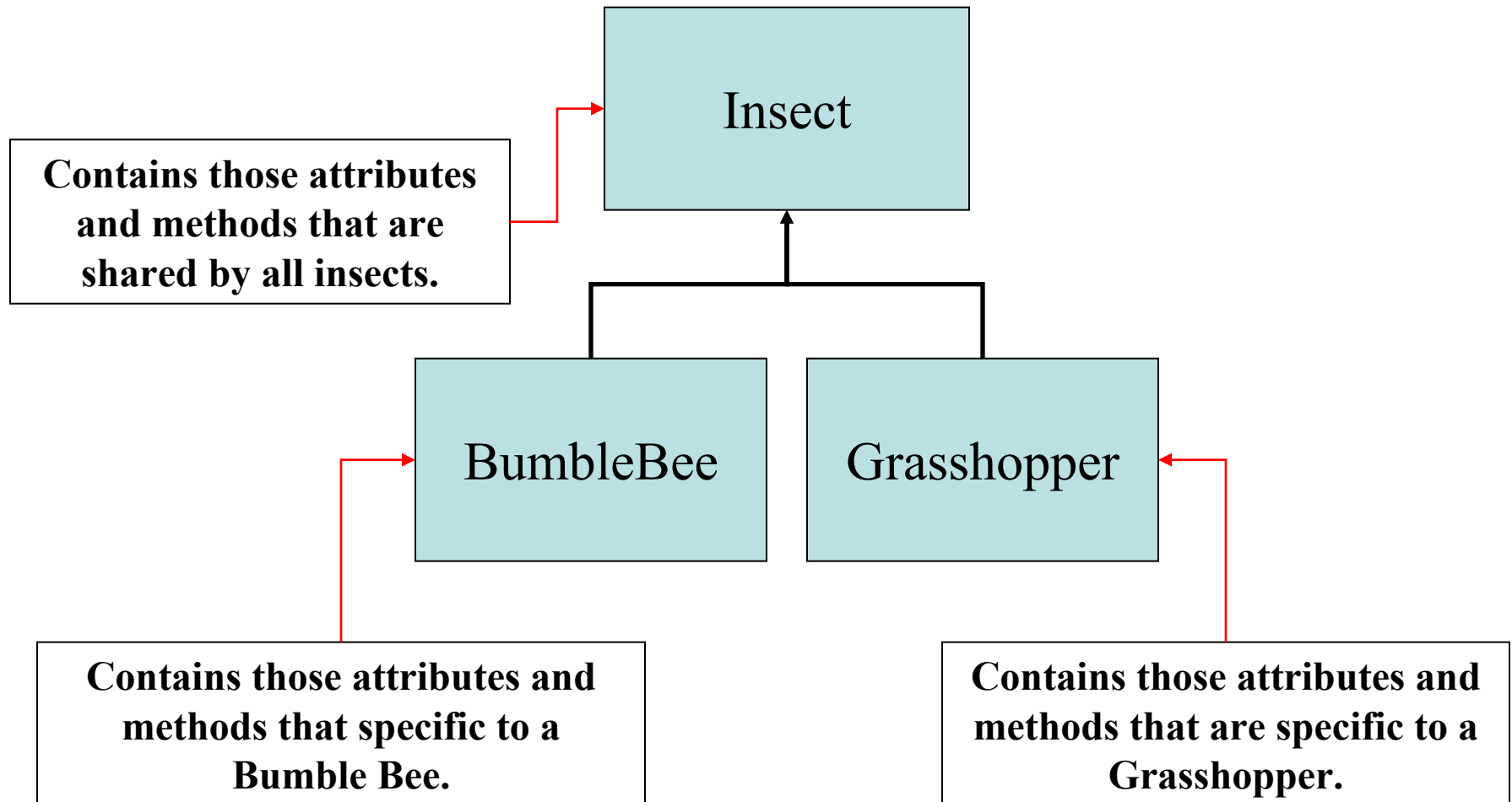
- What Is Inheritance?
- Calling the Superclass Constructor
- Overriding Superclass Methods
- Protected Members
- Chains of Inheritance
- The `Object` Class
- Polymorphism
- Abstract Classes and Abstract Methods
- Interfaces
- Anonymous Classes
- Functional Interfaces and Lambda Expressions

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 (1 of 2)

- 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 of 2)

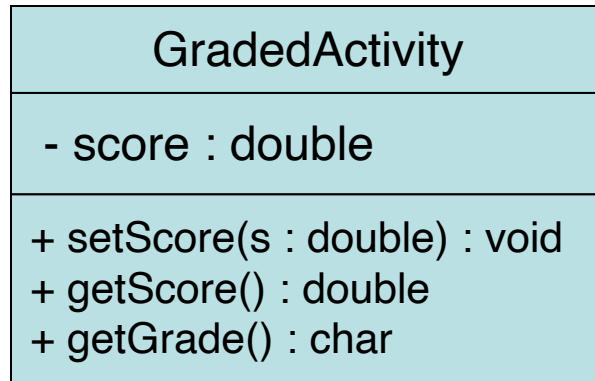
- We can *extend* the capabilities of a class.
- Inheritance involves a superclass and a subclass.
 - The *superclass* is the general class and
 - the *subclass* is the specialized class.
- The subclass is based on, or extended from, the superclass.
 - Superclasses are also called *base classes*, and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

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.

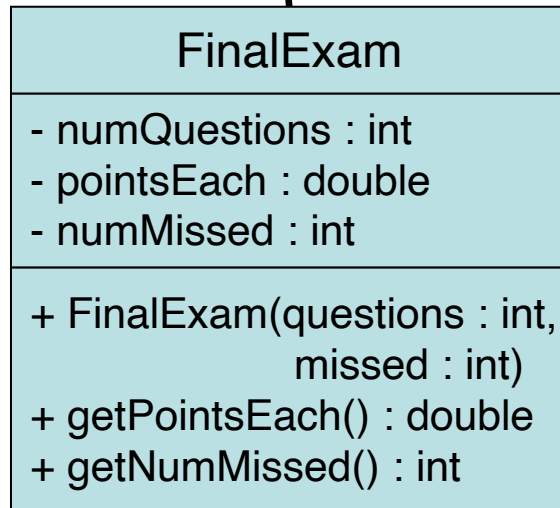
```
public class FinalExam extends  
    GradedActivity
```

The GradedActivity Example



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.



- Example:
 - [GradedActivity.java](#),
 - [GradeDemo.java](#),
 - [FinalExam.java](#),
 - [FinalExamDemo.java](#)

Inheritance, Fields and Methods (1 of 2)

- 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 of 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);
```

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](#)

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](#)

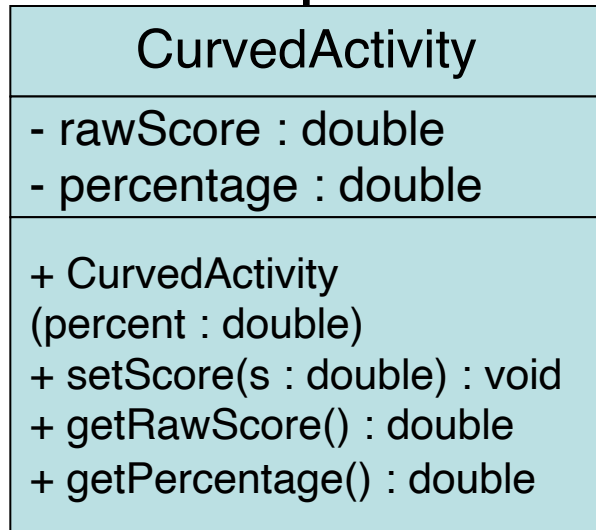
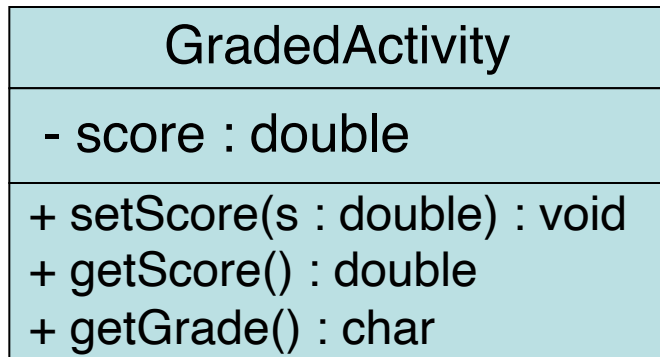
Calling The Superclass Constructor

- 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.

Overriding Superclass Methods (1 of 5)

- 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](#)

Overriding Superclass Methods (2 of 5)



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



Overriding Superclass Methods (3 of 5)

- 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 of 5)

- 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 of 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 (1 of 2)

- 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 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)	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

Chains of Inheritance (1 of 2)

- A superclass can also be derived from another class.

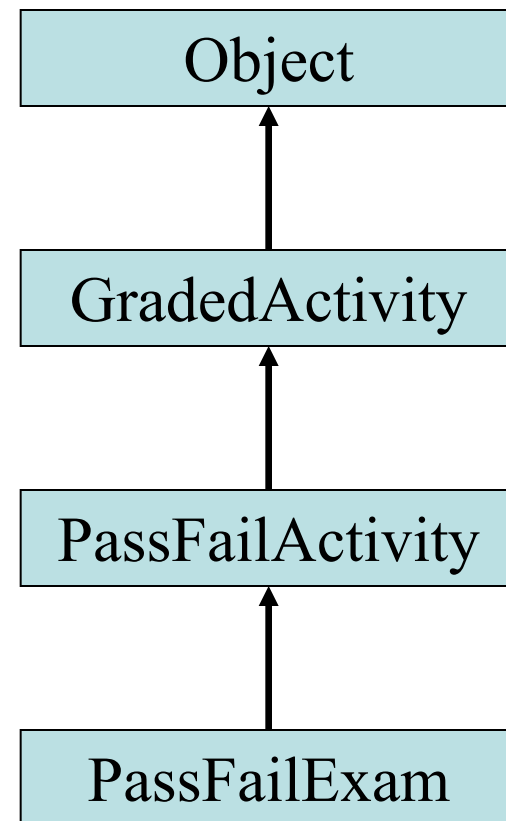
Example:

[GradedActivity.java](#)

[PassFailActivity.java](#)

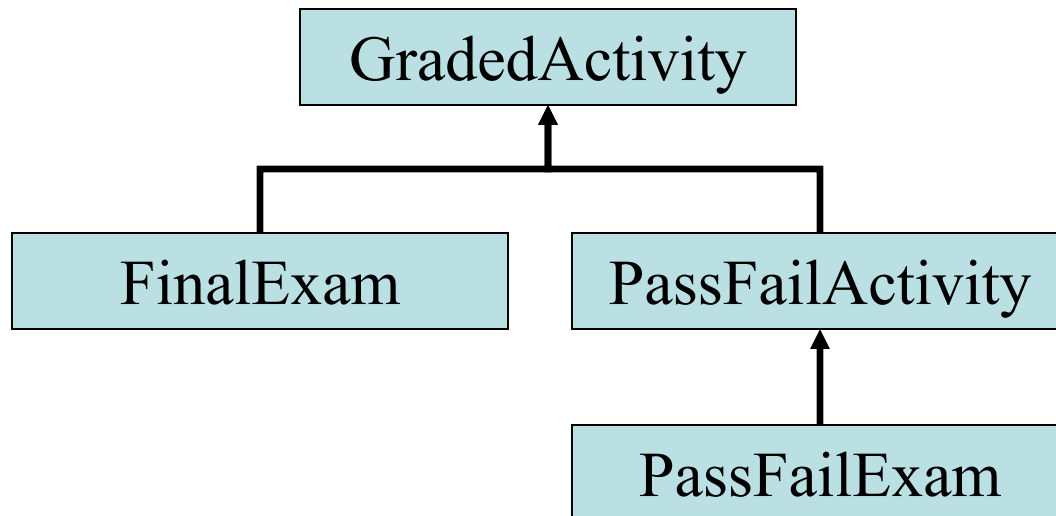
[PassFailExam.java](#)

[PassFailExamDemo.java](#)



Chains of Inheritance (2 of 2)

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



The Object Class (1 of 2)

- 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 of 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.
- Example: [ObjectMethods.java](#)

Polymorphism (1 of 4)

- 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.

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

- 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.

Polymorphism (4 of 4)

- 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.

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
```


Abstract Methods (1 of 2)

- 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 Methods (2 of 2)

- 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.

Interfaces (1 of 3)

- 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.
- It is often said that an interface is like a “contract,” and when a class implements an interface it must adhere to the contract.
- 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.

Interfaces (2 of 3)

- 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.

Interfaces (3 of 3)

- 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](#)

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...)
}
```

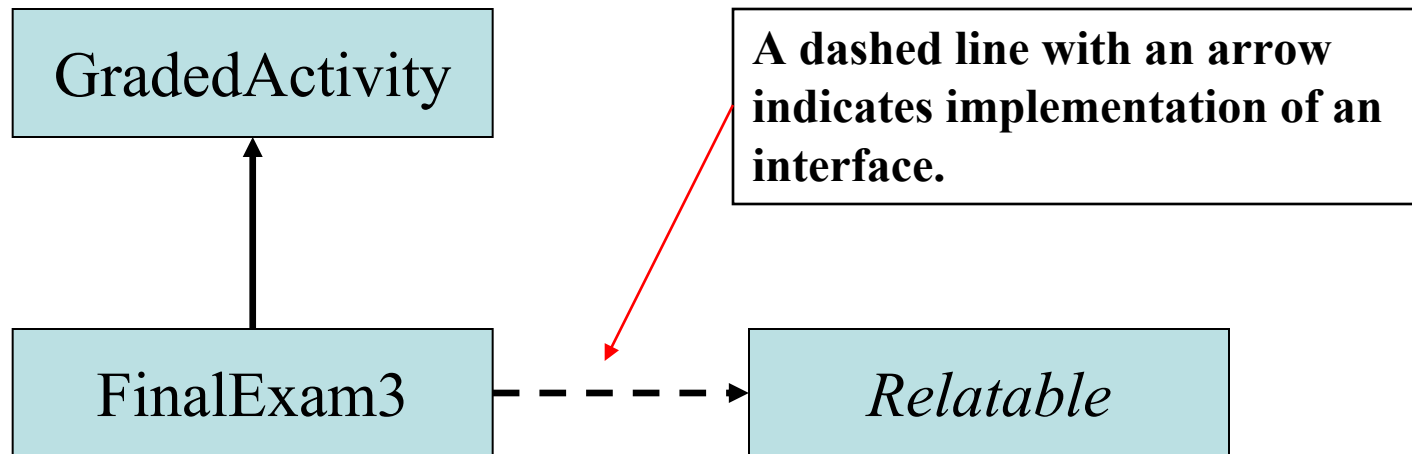
- In this interface, `FIELD1` and `FIELD2` are `final static int` variables.
- Any class that implements this interface has access to these variables.

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
```

Interfaces in UML



Polymorphism with Interfaces (1 of 3)

- 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](#)

Polymorphism with Interfaces (2 of 3)

- 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`.

Polymorphism with Interfaces (3 of 3)

- 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.

Default Methods

- Beginning in Java 8, interfaces may have *default methods*.
- A default method is an interface method that has a body.
- You can add new methods to an existing interface without causing errors in the classes that already implement the interface.
- Example:
 - [Displayable.java](#)
 - [Person.java](#)
 - [InterfaceDemoDefaultMethod.java](#)

Anonymous Inner Classes

- An inner class is a class that is defined inside another class.
- An anonymous inner class is an inner class that has no name.
- An anonymous inner class must implement an interface, or extend another class.
- Useful when you need a class that is simple, and to be instantiated only once in your code.
- Example:
 - [IntCalculator.java](#)
 - [AnonymousClassDemo.java](#)

Functional Interfaces and Lambda Expressions

- A functional interface is an interface that has one abstract method.
- A lambda expression can be used to create an object that implements the interface, and overrides its abstract method.
- In Java 8, these features work together to simplify code, particularly in situations where you might use anonymous inner classes.
- Example:
 - [LambdaDemo.java](#)
 - [LambdaDemo2.java](#)

Copyright



This work is protected by United States copyright laws and is provided solely for the use of instructions in teaching their courses and assessing student learning. dissemination or sale of any part of this work (including on the World Wide Web) will destroy the integrity of the work and is not permitted. The work and materials from it should never be made available to students except by instructors using the accompanying text in their classes. All recipients of this work are expected to abide by these restrictions and to honor the intended pedagogical purposes and the needs of other instructors who rely on these materials.