Inheritance

Reading:

• Java Tutorial:  
  http://docs.oracle.com/javase/tutorial/java/landl/subclasses.html
• Textbook: Chapter 7 and 8
Inheritance

In the Java language, classes can be derived from other classes, thereby inheriting fields and methods from those classes.

**Definitions:** A class that is derived from another class is called a **subclass** (also a derived class, extended class, or child class).

The class from which the subclass is derived is called a **superclass** (also a base class or a parent class).

Excepting **Object**, which has no superclass, every class has one and only one direct superclass (single inheritance).

In the absence of any other explicit superclass, every class is implicitly a subclass of **Object**.

Classes can be derived from classes that are derived from classes that are derived from classes, and so on, and ultimately derived from the topmost class, **Object**. Such a class is said to be descended from all the classes in the inheritance chain stretching back to **Object**.
The idea of inheritance is simple but powerful: **When you want to create a new class and there is already a class that includes some of the code that you want, you can derive your new class from the existing class.**

In doing this, you can reuse the fields and methods of the existing class without having to write (and debug!) them yourself.

**A subclass inherits all the members (fields, methods, and nested classes) from its superclass.**

Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.
The Java Platform Class Hierarchy

The **Object** class, defined in the `java.lang` package, defines and implements behavior common to all classes—including the ones that you write.

In the Java platform, many classes derive directly from **Object**, other classes derive from some of those classes, and so on, forming a hierarchy of classes.

All Classes in the Java Platform are Descendants of Object

At the top of the hierarchy, **Object** is the most general of all classes. Classes near the bottom of the hierarchy provide more specialized behavior.
An Example of Inheritance

Here is the sample code for a possible implementation of a Bicycle class that was presented in the Classes and Objects lesson:

```java
public class Bicycle {

    // the Bicycle class has three fields
    public int cadence;
    public int gear;
    public int speed;

    // the Bicycle class has one constructor
    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }
}
// the Bicycle class has four methods
public void setCadence(int newValue) {
    cadence = newValue;
}

public void setGear(int newValue) {
    gear = newValue;
}

public void applyBrake(int decrement) {
    speed -= decrement;
}

public void speedUp(int increment) {
    speed += increment;
}

A class declaration for a MountainBike class that is a subclass of Bicycle might look like this:

public class MountainBike extends Bicycle {

    // the MountainBike subclass adds one field
    public int seatHeight;

    // the MountainBike subclass has one constructor
    public MountainBike(int startHeight, int startCadence, int startSpeed, int startGear) {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
    }

    // the MountainBike subclass adds one method
    public void setHeight(int newValue) {
        seatHeight = newValue;
    }
}
MountainBike inherits all the fields and methods of Bicycle and adds the field seatHeight and a method to set it.

Except for the constructor, it is as if you had written a new MountainBike class entirely from scratch, with four fields and five methods.

What You Can Do in a Subclass

A subclass inherits all of the public and protected members of its parent, no matter what package the subclass is in.

If the subclass is in the same package as its parent, it also inherits the package-private members of the parent.

You can use the inherited members as is, replace them, hide them, or supplement them with new members:

- The inherited fields can be used directly, just like any other fields.
- You can declare a field in the subclass with the same name as the one in the superclass, thus hiding it (not recommended).
- You can declare new fields in the subclass that are not in the superclass.
- The inherited methods can be used directly as they are.
• You can write a new *instance* method in the subclass that has the same signature as the one in the superclass, thus *overriding* it.
• You can write a new *static* method in the subclass that has the same signature as the one in the superclass, thus *hiding* it.
• You can declare new methods in the subclass that are not in the superclass.
• You can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword *super*.

**Private Members in a Superclass**

A subclass does not inherit the private members of its parent class.

However, if the superclass has public or protected methods for accessing its private fields, these can also be used by the subclass.

A nested class has access to all the private members of its enclosing class—both fields and methods.

Therefore, a public or protected nested class inherited by a subclass has **indirect access** to all of the private members of the superclass.
Casting Objects

public MountainBike myBike = new MountainBike();

Here myBike is of type MountainBike.

MountainBike is descended from Bicycle and Object. Therefore,

a MountainBike is a Bicycle and is also an Object,

and it can be used wherever Bicycle or Object objects are called for.

The reverse is not necessarily true:

a Bicycle may be a MountainBike, but it isn't necessarily.

Similarly, an Object may be a Bicycle or a MountainBike, but it isn't necessarily.
Casting shows the use of an object of one type in place of another type, among the objects permitted by inheritance and implementations. For example, if we write

```java
Object obj = new MountainBike();
```

then `obj` is both an `Object` and a `MountainBike` (until such time as `obj` is assigned another object that is not a `MountainBike`). **This is called implicit casting.**

If, on the other hand, we write

```java
MountainBike myBike = obj;
```

**We would get a compile-time error** because `obj` is not known to the compiler to be a `MountainBike`.

However, we can tell the compiler that we promise to assign a `MountainBike` to `obj` by explicit casting:

```java
MountainBike myBike = (MountainBike)obj;
```

This cast inserts a **runtime check** that `obj` is assigned a `MountainBike` so that the compiler can safely assume that `obj` is a `MountainBike`. If `obj` is not a `MountainBike` at runtime, an exception will be thrown.
Note: You can make a logical test as to the type of a particular object using the instanceof operator. This can save you from a runtime error owing to an improper cast. For example:

```java
if (obj instanceof MountainBike) {
    MountainBike myBike = (MountainBike)obj;
}
```

Here the instanceof operator verifies that obj refers to a MountainBike so that we can make the cast with knowledge that there will be no runtime exception thrown.

---

Multiple Inheritance of State, Implementation, and Type

*Multiple inheritance of state*, which is the ability to inherit fields from multiple classes.

What if methods or constructors from different superclasses instantiate the same field?

*Java programming language does not permit you to extend more than one class.*

*Multiple inheritance of implementation vs Multiple inheritance of state?*
Problem with default methods?

An object can have multiple types: the type of its own class and the types of all the interfaces that the class implements (*multiple inheritance of type*).

### Overriding and Hiding Methods

<table>
<thead>
<tr>
<th>Defining a Method with the Same Signature as a Superclass's Method</th>
<th>Superclass Instance Method</th>
<th>Superclass Static Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subclass Instance Method</td>
<td>Overrides</td>
<td>Generates a compile-time error</td>
</tr>
<tr>
<td>Subclass Static Method</td>
<td>Generates a compile-time error</td>
<td>Hides</td>
</tr>
</tbody>
</table>

### Instance Methods

The overriding method has the same name, number and type of parameters, and return type as the method that it overrides.

An overriding method can also return a subtype of the type returned by the overridden method. This subtype is called a *covariant return type*. 
When overriding a method, you might want to use the `@Override` annotation that instructs the compiler that you intend to override a method in the superclass. If, for some reason, the compiler detects that the method does not exist in one of the superclasses, then it will generate an error. For more information on `@Override`, see [Annotations](#).

### Static Methods

If a subclass defines a static method with the same signature as a static method in the superclass, then the method in the subclass *hides* the one in the superclass.

The distinction between hiding a static method and overriding an instance method has important implications:

- The version of the overridden instance method that gets invoked is the one in the subclass.
- The version of the hidden static method that gets invoked depends on whether it is invoked from the superclass or the subclass.

```java
public class Animal {
    public static void testClassMethod() {
        System.out.println("The static method in Animal");
    }
    public void testInstanceMethod() {
        System.out.println("The instance method in Animal");
    }
}

public class Cat extends Animal {
    public static void testClassMethod() {
        System.out.println("The static method in Cat");
    }
    public void testInstanceMethod() {
        System.out.println("The instance method in Cat");
    }
}
```
public static void main(String[] args) {
    Cat myCat = new Cat();
    Animal myAnimal = myCat;
    Animal.testClassMethod();
    myAnimal.testInstanceMethod();
}

What is the output?
Output:

The static method in Animal
The instance method in Cat

As promised, the version of the hidden static method that gets invoked is the one in the superclass, and the version of the overridden instance method that gets invoked is the one in the subclass.

Interface Methods

Default methods and abstract methods in interfaces are inherited like instance methods.

What if the supertypes of a class or interface provide multiple default methods with the same signature?

The Java compiler follows inheritance rules to resolve the name conflict:
Rule 1: Instance methods are preferred over interface default methods:

```java
public class Horse {
    public String identifyMyself() {
        return "I am a horse.";
    }
}

public interface Flyer {
    default public String identifyMyself() {
        return "I am able to fly.";
    }
}

public interface Mythical {
    default public String identifyMyself() {
        return "I am a mythical creature.";
    }
}

public class Pegasus extends Horse implements Flyer, Mythical {
    public static void main(String... args) {
        Pegasus myApp = new Pegasus();
        System.out.println(myApp.identifyMyself());
    }
}

The method Pegasus.identifyMyself returns the string I am a horse.
```
Rule 2: Methods that are already overridden by other candidates are ignored.

This circumstance can arise when supertypes share a common ancestor.

```java
public interface Animal {
    default public String identifyMyself() {
        return "I am an animal.";
    }
}

public interface EggLayer extends Animal {
    default public String identifyMyself() {
        return "I am able to lay eggs.";
    }
}

public interface FireBreather extends Animal {
}

public class Dragon implements EggLayer, FireBreather {
    public static void main (String... args) {
        Dragon myApp = new Dragon();
        System.out.println(myApp.identifyMyself());
    }
}

The method Dragon.identifyMyself returns the string I am able to lay eggs.
```
Rule 3: If two or more independently defined default methods conflict, or a default method conflicts with an abstract method, then the Java compiler produces a compiler error. You must explicitly override the supertype methods.

```java
public interface OperateCar {
    // ...
    default public int startEngine(EncryptedKey key) {
        // Implementation
    }
}

public interface FlyCar {
    // ...
    default public int startEngine(EncryptedKey key) {
        // Implementation
    }
}
```

A class that implements both `OperateCar` and `FlyCar` must override the method `startEngine`. You could invoke any of the default implementations with the `super` keyword.

```java
public class FlyingCar implements OperateCar, FlyCar {
    // ...
    public int startEngine(EncryptedKey key) {
        FlyCar.super.startEngine(key);
        OperateCar.super.startEngine(key);
    }
}
```

The name preceding `super` (in this example, `FlyCar` or `OperateCar`) must refer to a direct superinterface that defines or inherits a default for the invoked method.

**Note:** You can use the `super` keyword to invoke a default method in both classes and interfaces.
Rule 4: Inherited instance methods from classes can override abstract interface methods. Consider the following interfaces and classes:

```java
public interface Mammal {
    String identifyMyself();
}
public class Horse {
    public String identifyMyself() {
        return "I am a horse.";
    }
}
public class Mustang extends Horse implements Mammal {
    public static void main(String... args) {
        Mustang myApp = new Mustang();
        System.out.println(myApp.identifyMyself());
    }
}
```

The method `Mustang.identifyMyself` returns the string "I am a horse."

The class `Mustang` inherits the method `identifyMyself` from the class `Horse`, which overrides the abstract method of the same name in the interface `Mammal`.

**Note:** Static methods in interfaces are never inherited.
Modifiers

The access specifier for an overriding method can allow more, but not less, access than the overridden method.

For example, a protected instance method in the superclass can be made public, but not private, in the subclass.

You will get a compile-time error if you attempt to change an instance method in the superclass to a static method in the subclass, and vice versa.
Polymorphism

The dictionary definition of *polymorphism* refers to a principle in biology in which an organism or species can have many different forms or stages. This principle can also be applied to object-oriented programming and languages like the Java language.

Subclasses of a class can define their own unique behaviors and yet share some of the same functionality of the parent class.

Suppose our `bicycle` class has the following method:

```java
public void printDescription(){
    System.out.println("Bike is in gear " + this.gear + " with a cadence of " + this.cadence + " and travelling at a speed of " + this.speed + ". ");
}
```

Now...

```java
public class MountainBike extends Bicycle {
    private String suspension;

    public MountainBike(
        int startCadence,
        int startSpeed,
        int startGear,
        String suspensionType)
    {
        super(startCadence, startSpeed, startGear);
        this.setSuspension(suspensionType);
    }

    public String getSuspension(){
        return this.suspension;
    }
}
```
public void setSuspension(String suspensionType) {
    this.suspension = suspensionType;
}

public void printDescription() {
    super.printDescription();
    System.out.println("The " + "MountainBike has a" +
                      getSuspension() + " suspension.");
}

Note the overridden printDescription method. In addition to the information provided before, additional data about the suspension is included to the output.

public class RoadBike extends Bicycle{
    // In millimeters (mm)
    private int tireWidth;

    public RoadBike(int startCadence, 
                    int startSpeed, 
                    int startGear, 
                    int newTireWidth){
        super(startCadence, startSpeed, startGear);
        this.setTireWidth(newTireWidth);
    }

    public int getTireWidth(){
        return this.tireWidth;
    }

    public void setTireWidth(int newTireWidth){
        this.tireWidth = newTireWidth;
    }

    public void printDescription(){
        super.printDescription();
        System.out.println("The RoadBike" + " has " +
                            getTireWidth() + " MM tires.");
    }
Note that once again, the `printDescription` method has been overridden. This time, information about the tire width is displayed.

To summarize, there are three classes: `Bicycle`, `MountainBike`, and `RoadBike`. The two subclasses override the `printDescription` method and print unique information.

Here is a test program that creates three `Bicycle` variables. Each variable is assigned to one of the three bicycle classes. Each variable is then printed.

```java
public class TestBikes {
    public static void main(String[] args) {
        Bicycle bike01, bike02, bike03;

        bike01 = new Bicycle(20, 10, 1);
        bike02 = new MountainBike(20, 10, 5, "Dual");
        bike03 = new RoadBike(40, 20, 8, 23);

        bike01.printDescription();
        bike02.printDescription();
        bike03.printDescription();
    }
}
```
The following is the output from the test program:

**Bike is in gear 1 with a cadence of 20 and travelling at a speed of 10.**

**Bike is in gear 5 with a cadence of 20 and travelling at a speed of 10.**
The MountainBike has a Dual suspension.

**Bike is in gear 8 with a cadence of 40 and travelling at a speed of 20.**
The RoadBike has 23 MM tires.

The Java virtual machine (JVM) calls the appropriate method for the object that is referred to in each variable. It does not call the method that is defined by the variable's type.

This behavior is referred to as **virtual method invocation** and demonstrates an aspect of the important polymorphism features in the Java language.

## Hiding Fields

**Within a class, a field that has the same name as a field in the superclass hides the superclass's field, even if their return types are different.**

Within the subclass, the field in the superclass cannot be referenced by its simple name. Instead, the field must be accessed through `super`.

Generally hiding fields are not recommended since it makes code difficult to read.
Using the Keyword super
Accessing Superclass Members

If your method overrides one of its superclass's methods, you can invoke the overridden method through the use of the keyword super.

You can also use super to refer to a hidden field (although hiding fields is discouraged). Consider this class, Superclass:

```java
public class Superclass {
    public void printMethod() {
        System.out.println("Printed in Superclass.");
    }
}
```

Here is a subclass, called Subclass, that overrides printMethod():

```java
public class Subclass extends Superclass {
    // overrides printMethod in Superclass
    public void printMethod() {
        super.printMethod();
        System.out.println("Printed in Subclass");
    }
    public static void main(String[] args) {
        Subclass s = new Subclass();
        s.printMethod();
    }
}
```

Compiling and executing Subclass prints the following:

Printed in Superclass.
Printed in Subclass
Subclass Constructors

The following example illustrates how to use the super keyword to invoke a superclass's constructor.

```java
public MountainBike(int startHeight,
                     int startCadence,
                     int startSpeed,
                     int startGear) {
    super(startCadence, startSpeed, startGear);
    seatHeight = startHeight;
}
```

Invocation of a superclass constructor must be the first line in the subclass constructor.

The syntax for calling a superclass constructor is

```java
super();
```

or:

```java
super(parameter list);
```

With super(), the superclass no-argument constructor is called. With super(parameter list), the superclass constructor with a matching parameter list is called.

---

Note: If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass. If the super class does not have a no-argument constructor, you will get a compile-time error. Object does have such a constructor, so if Object is the only superclass, there is no problem.
If a subclass constructor invokes a constructor of its superclass, either explicitly or implicitly, you might think that there will be a whole chain of constructors called, all the way back to the constructor of Object.

In fact, this is the case. It is called constructor chaining, and you need to be aware of it when there is a long line of class descent.
Object as a Superclass

The `Object` class, in the `java.lang` package, sits at the top of the class hierarchy tree.

Every class is a descendant, direct or indirect, of the `Object` class.

**Every class you use or write inherits the instance methods of `Object`**.

You need not use any of these methods, but, if you choose to do so, you may need to override them with code that is specific to your class. The methods inherited from `Object` that are discussed in this section are:

- `protected Object clone() throws CloneNotSupportedException`
  
  Creates and returns a copy of this object.

- `public boolean equals(Object obj)`
  
  Indicates whether some other object is "equal to" this one.

- `protected void finalize() throws Throwable`
  
  Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.

- `public final Class getClass()`
  
  Returns the runtime class of an object.

- `public int hashCode()`
  
  Returns a hash code value for the object.

- `public String toString()`
  
  Returns a string representation of the object.

The `notify`, `notifyAll`, and `wait` methods of `Object` all play a part in synchronizing the activities of independently running threads in a program, (will be discussed in Java threads). There are five of these methods:
- public final void notify()
- public final void notifyAll()
- public final void wait()
- public final void wait(long timeout)
- public final void wait(long timeout, int nanos)

**The clone() Method**

If a class, or one of its superclasses, implements the `Cloneable` interface, you can use the `clone()` method to create a copy from an existing object. To create a clone, you write:

```java
aCloneableObject.clone();
```

Object's implementation of this method checks to see whether the object on which `clone()` was invoked implements the `Cloneable` interface. If the object does not, the method throws a `CloneNotSupportedException` exception. Exception handling will be covered in a later lesson. For the moment, you need to know that `clone()` must be declared as

```java
protected Object clone() throws CloneNotSupportedException
```

or:

```java
public Object clone() throws CloneNotSupportedException
```

if you are going to write a `clone()` method to override the one in `Object`.

The simplest way to make your class cloneable is to add `implements Cloneable` to your class's declaration. Then your objects can invoke the `clone()` method.
For some classes, the default behavior of Object's clone() method works just fine. If, however, an object contains a reference to an external object, say ObjExternal, you may need to override clone() to get correct behavior. Otherwise, a change in ObjExternal made by one object will be visible in its clone also.

The equals() Method

The equals() method compares two objects for equality and returns true if they are equal.

The equals() method provided in the Object class uses the identity operator (==) to determine whether two objects are equal. For primitive data types, this gives the correct result. For objects, however, it does not.

The equals() method provided by Object tests whether the object references are equal—that is, if the objects compared are the exact same object.

To test whether two objects are equal in the sense of equivalency (containing the same information), you must override the equals() method. Here is an example of a Book class that overrides equals():

```java
public class Book {
    ...
    
    public boolean equals(Object obj) {
        if (obj instanceof Book)
            return ISBN.equals((Book)obj.getISBN());
        else
            return false;
    }

```
Consider this code that tests two instances of the Book class for equality:

```java
// Swing Tutorial, 2nd edition
Book firstBook = new Book("0201914670");
Book secondBook = new Book("0201914670");
if (firstBook.equals(secondBook)) {
  System.out.println("objects are equal");
} else {
  System.out.println("objects are not equal");
}
```

You should always override the `equals()` method if the identity operator is not appropriate for your class.

---

**Note:** If you override `equals()`, you must override `hashCode()` as well.

---

**The finalize() Method**

The Object class provides a callback method, `finalize()`, that *may be* invoked on an object when it becomes garbage. Object's implementation of `finalize()` does nothing—you can override `finalize()` to do cleanup, such as freeing resources.

The `finalize()` method *may be* called automatically by the system (by the garbage collector), but when it is called, or even if it is called, is uncertain.
The getClass() Method

You cannot override getClass.

The getClass() method returns a class object, which has methods you can use to get information about the class, such as its name (getSimpleDateFormat()), its superclass (getSuperclass()), and the interfaces it implements (getInterfaces()).

For example, the following method gets and displays the class name of an object:

```java
void printClassName(Object obj) {
    System.out.println("The object's" + " class is " +
                        obj.getClass().getSimpleName());
}
```

The Class class, in the java.lang package, has a large number of methods (more than 50). For example, you can test to see if the class is an annotation (isAnnotation()), an interface (isInterface()), or an enumeration (isEnum()). You can see what the object's fields are (getFields()) or what its methods are (getMethods()), and so on.
The hashCode() Method

The value returned by `hashCode()` is the object's hash code, which is the object's memory address in hexadecimal.

By definition, if two objects are equal, their hash code must also be equal.

If you override the `equals()` method, you change the way two objects are equated and `Object`'s implementation of `hashCode()` is no longer valid. Therefore, if you override the `equals()` method, you must also override the `hashCode()` method as well.

An example from Wikipedia:

```java
public class Employee {
    int employeeId;
    String name;
    Department dept;

    // other methods would be in here

    @Override
    public int hashCode() {
        int hash = 1;
        hash = hash * 17 + employeeId;
        hash = hash * 31 + name.hashCode();
        hash = hash * 13 + (dept == null ? 0 : dept.hashCode());
        return hash;
    }
}
```
The toString() Method

You should always consider overriding the toString() method in your classes.

The Object's toString() method returns a String representation of the object, which is very useful for debugging.

The String representation for an object depends entirely on the object, which is why you need to override toString() in your classes.

You can use toString() along with System.out.println() to display a text representation of an object, such as an instance of Book:

System.out.println(firstBook.toString());

which would, for a properly overridden toString() method, print something useful, like this:

Writing Final Classes and Methods

You use the `final` keyword in a method declaration to indicate that the method cannot be overridden by subclasses.

You might wish to make a method final if it has an implementation that should not be changed and it is critical to the consistent state of the object.

For example:

```java
class ChessAlgorithm {
    enum ChessPlayer { WHITE, BLACK }
    ...
    final ChessPlayer getFirstPlayer() {
        return ChessPlayer.WHITE;
    }
    ...
}
```

Methods called from constructors should generally be declared final. If a constructor calls a non-final method, a subclass may redefine that method with surprising or undesirable results.

Note that you can also declare an entire class final. A class that is declared final cannot be subclassed. This is particularly useful, for example, when creating an immutable class like the `String` class.
Abstract Methods and Classes

An abstract class is a class that is declared abstract—it may or may not include abstract methods.

Abstract classes cannot be instantiated, but they can be subclassed.

An abstract method is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

abstract void moveTo(double deltaX, double deltaY);

If a class includes abstract methods, then the class itself must be declared abstract, as in:

```java
public abstract class GraphicObject {
    // declare fields
    // declare nonabstract methods
    abstract void draw();
}
```

When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, then the subclass must also be declared abstract.

Note: Methods in an interface (see the Interfaces section) that are not declared as default or static are implicitly abstract, so the abstract modifier is not used with interface methods. (It can be used, but it is unnecessary.)
Abstract Classes Compared to Interfaces

Abstract classes are similar to interfaces. You cannot instantiate them, and they may contain a mix of methods declared with or without an implementation.

However, with abstract classes, you can declare fields that are not static and final, and define public, protected, and private concrete methods.

With interfaces, all fields are automatically public, static, and final, and all methods that you declare or define (as default methods) are public.

In addition, you can extend only one class, whether or not it is abstract, whereas you can implement any number of interfaces.

Which should you use, abstract classes or interfaces?

- **Consider using abstract classes if any of these statements apply to your situation:**
  - You want to share code among several closely related classes.
  - You expect that classes that extend your abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private).
  - You want to declare non-static or non-final fields. This enables you to define methods that can access and modify the state of the object to which they belong.

- **Consider using interfaces if any of these statements apply to your situation:**
  - You expect that unrelated classes would implement your interface. For example, the interfaces `Comparable` and `Cloneable` are implemented by many unrelated classes.
  - You want to specify the behavior of a particular data type, but not concerned about who implements its behavior.
  - You want to take advantage of multiple inheritance of type.
An Abstract Class Example

In an object-oriented drawing application, you can draw circles, rectangles, lines, Bezier curves, and many other graphic objects.

These objects all have certain states (for example: position, orientation, line color, fill color) and behaviors (for example: moveTo, rotate, resize, draw) in common.

Some of these states and behaviors are the same for all graphic objects (for example: position, fill color, and moveTo).

Others require different implementations (for example, resize or draw).

This is a perfect situation for an abstract superclass.

Classes Rectangle, Line, Bezier, and Circle Inherit from GraphicObject
abstract class GraphicObject {
    int x, y;
    ...
    void moveTo(int newX, int newY) {
        ...
    }
    abstract void draw();
    abstract void resize();
}

class Circle extends GraphicObject {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}

class Rectangle extends GraphicObject {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}
When an Abstract Class Implements an Interface

In the section on Interfaces, it was noted that a class that implements an interface must implement all of the interface's methods.

It is possible, however, to define a class that does not implement all of the interface's methods, provided that the class is declared to be abstract. For example,

```java
abstract class X implements Y {
    // implements all but one method of Y
}

class XX extends X {
    // implements the remaining method in Y
}
```

In this case, class X must be abstract because it does not fully implement Y, but class XX does, in fact, implement Y.