Object as a Superclass ← **Abstract Methods and Classes** → That's the end of the series!

Abstract Methods and Classes

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:

```
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 example of an abstract class in the JDK is `AbstractMap`, which is part of the Collections Framework. Its subclasses (which include `HashMap`, `TreeMap`, and `ConcurrentHashMap` share many methods (including `get()`, `put()`, `isEmpty()`, `containsKey()`, and `containsValue()`) that `AbstractMap` defines.

An example of a class in the JDK that implements several interfaces is `HashMap`, which implements the interfaces `Serializable`, `Cloneable`, and `Map<K, V>`. By reading this list of interfaces, you can infer that an instance of `HashMap` (regardless of the developer or company who implemented the class) can be cloned, is serializable (which means that it can be converted into a byte stream; see the section Serializable Objects), and has the functionality of a map. In addition, the `Map<K, V>` interface has been enhanced with many default methods such as `merge()` and `forEach()` that older classes that have implemented this interface do not have to define.

Note that many software libraries use both abstract classes and interfaces; the `HashMap` class implements several interfaces and also extends the abstract class `AbstractMap`.

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).

All GraphicObjects must be able to draw or resize themselves; they just differ in how they do it. This is a perfect situation for an abstract superclass. You can take advantage of the similarities and declare all the graphic objects to inherit from the same abstract parent object, for example, `GraphicObject`.

First, you declare an abstract class, `GraphicObject`, to provide member variables and methods that are wholly shared by all subclasses, such as the current position and the `moveTo()` method. `GraphicObject` also declares abstract methods for methods, such as `draw()` or `resize()`, that need to be implemented by all subclasses but must be implemented in different ways. The `GraphicObject` class can look something like this:

```java
abstract class GraphicObject {
    int x, y;
    ...
    void moveTo(int newX, int newY) {
        ...
    }
    abstract void draw();
    abstract void resize();
}
```

Each nonabstract subclass of `GraphicObject`, such as `Circle` and `Rectangle`, must provide implementations for the `draw()` and `resize()` methods:

```java
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,

```
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`.

Class Members

An abstract class may have `static` fields and `static` methods. You can use these `static` members with a class reference (for example, `AbstractClass.staticMethod()`) as you would with any other class.

Last update: *September 14, 2021*

Object as a Superclass

🡐

Abstract Methods and Classes

🡒

That's the end of the series!

Home > Tutorials > Inheritance > Abstract Methods and Classes