**Interfaces**

There are a number of situations in software engineering when it is important for disparate groups of programmers to agree to a "contract" that spells out how their software interacts. Each group should be able to write their code without any knowledge of how the other group's code is written. Generally speaking, *interfaces* are such contracts.

For example, imagine a futuristic society where computer-controlled robotic cars transport passengers through city streets without a human operator. Automobile manufacturers write software (Java, of course) that operates the automobile—stop, start, accelerate, turn left, and so forth. Another industrial group, electronic guidance instrument manufacturers, make computer systems that receive GPS (Global Positioning System) position data and wireless transmission of traffic conditions and use that information to drive the car.

The auto manufacturers must publish an industry-standard interface that spells out in detail what methods can be invoked to make the car move (any car, from any manufacturer). The guidance manufacturers can then write software that invokes the methods described in the interface to command the car. Neither industrial group needs to know *how* the other group's software is implemented. In fact, each group considers its software highly proprietary and reserves the right to modify it at any time, as long as it continues to adhere to the published interface.

**Interfaces in Java**

In the Java programming language, an *interface* is a reference type, similar to a class, that can contain *only* constants, method signatures, and nested types. There are no method bodies. Interfaces cannot be instantiated—they can only be *implemented* by classes or *extended* by other interfaces. Extension is discussed later in this lesson.

Defining an interface is similar to creating a new class:

public interface OperateCar {

// constant declarations, if any

// method signatures

// An enum with values RIGHT, LEFT

int turn(Direction direction,

double radius,

double startSpeed,

double endSpeed);

int changeLanes(Direction direction,

double startSpeed,

double endSpeed);

int signalTurn(Direction direction,

boolean signalOn);

int getRadarFront(double distanceToCar,

double speedOfCar);

int getRadarRear(double distanceToCar,

double speedOfCar);

......

// more method signatures

}

Note that the method signatures have no braces and are terminated with a semicolon.

To use an interface, you write a class that *implements* the interface. When an instantiable class implements an interface, it provides a method body for each of the methods declared in the interface. For example,

public class OperateBMW760i implements OperateCar {

// the OperateCar method signatures, with implementation --

// for example:

int signalTurn(Direction direction, boolean signalOn) {

// code to turn BMW's LEFT turn indicator lights on

// code to turn BMW's LEFT turn indicator lights off

// code to turn BMW's RIGHT turn indicator lights on

// code to turn BMW's RIGHT turn indicator lights off

}

// other members, as needed -- for example, helper classes not

// visible to clients of the interface

}

In the robotic car example above, it is the automobile manufacturers who will implement the interface. Chevrolet's implementation will be substantially different from that of Toyota, of course, but both manufacturers will adhere to the same interface. The guidance manufacturers, who are the clients of the interface, will build systems that use GPS data on a car's location, digital street maps, and traffic data to drive the car. In so doing, the guidance systems will invoke the interface methods: turn, change lanes, brake, accelerate, and so forth.

**Interfaces as APIs**

The robotic car example shows an interface being used as an industry standard *Application Programming Interface (API)*. APIs are also common in commercial software products. Typically, a company sells a software package that contains complex methods that another company wants to use in its own software product. An example would be a package of digital image processing methods that are sold to companies making end-user graphics programs. The image processing company writes its classes to implement an interface, which it makes public to its customers. The graphics company then invokes the image processing methods using the signatures and return types defined in the interface. While the image processing company's API is made public (to its customers), its implementation of the API is kept as a closely guarded secret—in fact, it may revise the implementation at a later date as long as it continues to implement the original interface that its customers have relied on.

**Interfaces and Multiple Inheritance**

Interfaces have another very important role in the Java programming language. Interfaces are not part of the class hierarchy, although they work in combination with classes. The Java programming language does not permit multiple inheritance (inheritance is discussed later in this lesson), but interfaces provide an alternative.

In Java, a class can inherit from only one class but it can implement more than one interface. Therefore, objects can have multiple types: the type of their own class and the types of all the interfaces that they implement. This means that if a variable is declared to be the type of an interface, its value can reference any object that is instantiated from any class that implements the interface. This is discussed later in this lesson, in the section titled "Using an Interface as a Type."

# Defining an Interface

An interface declaration consists of modifiers, the keyword interface, the interface name, a comma-separated list of parent interfaces (if any), and the interface body. For example:

public interface GroupedInterface extends Interface1, Interface2, Interface3 {

// constant declarations

// base of natural logarithms

double E = 2.718282;

// method signatures

void doSomething (int i, double x);

int doSomethingElse(String s);

}

The public access specifier indicates that the interface can be used by any class in any package. If you do not specify that the interface is public, your interface will be accessible only to classes defined in the same package as the interface.

An interface can extend other interfaces, just as a class can extend or subclass another class. However, whereas a class can extend only one other class, an interface can extend any number of interfaces. The interface declaration includes a comma-separated list of all the interfaces that it extends.

## The Interface Body

The interface body contains method declarations for all the methods included in the interface. A method declaration within an interface is followed by a semicolon, but no braces, because an interface does not provide implementations for the methods declared within it. All methods declared in an interface are implicitly public, so the public modifier can be omitted.

An interface can contain constant declarations in addition to method declarations. All constant values defined in an interface are implicitly public, static, and final. Once again, these modifiers can be omitted.

**Implementing an Interface**

To declare a class that implements an interface, you include an implements clause in the class declaration. Your class can implement more than one interface, so the implements keyword is followed by a comma-separated list of the interfaces implemented by the class. By convention, the implements clause follows the extends clause, if there is one.

**A Sample Interface, Relatable**

Consider an interface that defines how to compare the size of objects.

public interface Relatable {

// this (object calling isLargerThan)

// and other must be instances of

// the same class returns 1, 0, -1

// if this is greater // than, equal

// to, or less than other

public int isLargerThan(Relatable other);

}

If you want to be able to compare the size of similar objects, no matter what they are, the class that instantiates them should implement Relatable.

Any class can implement Relatable if there is some way to compare the relative "size" of objects instantiated from the class. For strings, it could be number of characters; for books, it could be number of pages; for students, it could be weight; and so forth. For planar geometric objects, area would be a good choice (see the RectanglePlus class that follows), while volume would work for three-dimensional geometric objects. All such classes can implement the isLargerThan() method.

If you know that a class implements Relatable, then you know that you can compare the size of the objects instantiated from that class.

**Implementing the Relatable Interface**

Here is the Rectangle class that was presented in the [Creating Objects](http://docs.oracle.com/javase/tutorial/java/javaOO/objectcreation.html) section, rewritten to implement Relatable.

public class RectanglePlus

implements Relatable {

public int width = 0;

public int height = 0;

public Point origin;

// four constructors

public RectanglePlus() {

origin = new Point(0, 0);

}

public RectanglePlus(Point p) {

origin = p;

}

public RectanglePlus(int w, int h) {

origin = new Point(0, 0);

width = w;

height = h;

}

public RectanglePlus(Point p, int w, int h) {

origin = p;

width = w;

height = h;

}

// a method for moving the rectangle

public void move(int x, int y) {

origin.x = x;

origin.y = y;

}

// a method for computing

// the area of the rectangle

public int getArea() {

return width \* height;

}

// a method required to implement

// the Relatable interface

public int isLargerThan(Relatable other) {

**RectanglePlus otherRect**

**= (RectanglePlus)other;**

if (this.getArea() < otherRect.getArea())

return -1;

else if (this.getArea() > otherRect.getArea())

return 1;

else

return 0;

}

}

Because RectanglePlus implements Relatable, the size of any two RectanglePlus objects can be compared.

**Note:** The isLargerThan method, as defined in the Relatable interface, takes an object of type Relatable. The line of code, shown in bold in the previous example, casts other to a RectanglePlus instance. Type casting tells the compiler what the object really is. Invoking getArea directly on the other instance (other.getArea()) would fail to compile because the compiler does not understand that other is actually an instance of RectanglePlus.

# Using an Interface as a Type

When you define a new interface, you are defining a new reference data type. You can use interface names anywhere you can use any other data type name. If you define a reference variable whose type is an interface, any object you assign to it *must* be an instance of a class that implements the interface.

As an example, here is a method for finding the largest object in a pair of objects, for *any* objects that are instantiated from a class that implements Relatable:

public Object findLargest(Object object1, Object object2) {

Relatable obj1 = (Relatable)object1;

Relatable obj2 = (Relatable)object2;

if ((obj1).isLargerThan(obj2) > 0)

return object1;

else

return object2;

}

By casting object1 to a Relatable type, it can invoke the isLargerThan method.

If you make a point of implementing Relatable in a wide variety of classes, the objects instantiated from *any* of those classes can be compared with the findLargest() method—provided that both objects are of the same class. Similarly, they can all be compared with the following methods:

public Object findSmallest(Object object1, Object object2) {

Relatable obj1 = (Relatable)object1;

Relatable obj2 = (Relatable)object2;

if ((obj1).isLargerThan(obj2) < 0)

return object1;

else

return object2;

}

public boolean isEqual(Object object1, Object object2) {

Relatable obj1 = (Relatable)object1;

Relatable obj2 = (Relatable)object2;

if ( (obj1).isLargerThan(obj2) == 0)

return true;

else

return false;

}

These methods work for any "relatable" objects, no matter what their class inheritance is. When they implement Relatable, they can be of both their own class (or superclass) type and a Relatable type. This gives them some of the advantages of multiple inheritance, where they can have behavior from both a superclass and an interface.

# Rewriting Interfaces

Consider an interface that you have developed called DoIt:

public interface DoIt {

void doSomething(int i, double x);

int doSomethingElse(String s);

}

Suppose that, at a later time, you want to add a third method to DoIt, so that the interface now becomes:

public interface DoIt {

void doSomething(int i, double x);

int doSomethingElse(String s);

boolean didItWork(int i, double x, String s);

}

If you make this change, all classes that implement the old DoIt interface will break because they don't implement the interface anymore. Programmers relying on this interface will protest loudly.

Try to anticipate all uses for your interface and to specify it completely from the beginning. Given that this is often impossible, you may need to create more interfaces later. For example, you could create a DoItPlus interface that extends DoIt:

public interface DoItPlus extends DoIt {

boolean didItWork(int i, double x, String s);

}

Now users of your code can choose to continue to use the old interface or to upgrade to the new interface.

# Summary of Interfaces

An interface defines a protocol of communication between two objects.

An interface declaration contains signatures, but no implementations, for a set of methods, and might also contain constant definitions.

A class that implements an interface must implement all the methods declared in the interface.

An interface name can be used anywhere a type can be used.

## Questions

1. What methods would a class that implements the java.lang.CharSequence interface have to implement?
2. What is wrong with the following interface?
3. public interface SomethingIsWrong {
4. void aMethod(int aValue){
5. System.out.println("Hi Mom");
6. }
7. }
8. Fix the interface in question 2.
9. Is the following interface valid?
10. public interface Marker {
11. }

## Exercises

1. Write a class that implements the CharSequence interface found in the java.lang package. Your implementation should return the string backwards. Select one of the sentences from this book to use as the data. Write a small main method to test your class; make sure to call all four methods.
2. Suppose you have written a time server that periodically notifies its clients of the current date and time. Write an interface the server could use to enforce a particular protocol on its clients.

# 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, the class itself *must* be declared abstract, as in:

public abstract class GraphicObject {

// declare fields

// declare non-abstract 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, the subclass must also be declared abstract.

**Note:** All of the methods in an *interface* (see the [Interfaces](http://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html) section) are *implicitly* abstract, so the abstract modifier is not used with interface methods (it could be—it's just not necessary).

## Abstract Classes versus Interfaces

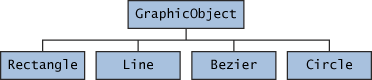
Unlike interfaces, abstract classes can contain fields that are not static and final, and they can contain implemented methods. Such abstract classes are similar to interfaces, except that they provide a partial implementation, leaving it to subclasses to complete the implementation. If an abstract class contains *only* abstract method declarations, it should be declared as an interface instead.

Multiple interfaces can be implemented by classes anywhere in the class hierarchy, whether or not they are related to one another in any way. Think of Comparable or Cloneable, for example.

By comparison, abstract classes are most commonly subclassed to share pieces of implementation. A single abstract class is subclassed by similar classes that have a lot in common (the implemented parts of the abstract class), but also have some differences (the abstract methods).

## 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 know how 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, as shown in the following figure.



Classes Rectangle, Line, Bezier, and Circle inherit from 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:

abstract class GraphicObject {

int x, y;

...

void moveTo(int newX, int newY) {

...

}

abstract void draw();

abstract void resize();

}

Each non-abstract subclass of GraphicObject, such as Circle and Rectangle, must provide implementations for the draw and resize methods:

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](http://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html), 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 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.

.