

A public class field is essentially a global variable. The names of class fields are qualified by the unique names of the classes that contain them, however. Thus, Java does not suffer from the name collisions that can affect other languages when different modules of code define global variables with the same name.

As with class fields, *class methods* are declared with the static modifier:

```
public static double radiansToDegrees(double rads) { return rads * 180 / PI; }
```

This line declares a class method named `radiansToDegrees()`. It has a single parameter of type `double` and returns a `double` value. The body of the method is quite short; it performs a simple computation and returns the result.

Like class fields, class methods are associated with a class, rather than with an object. When invoking a class method from code that exists outside the class, you must specify both the name of the class and the method. For example:

```
// How many degrees is 2.0 radians?
double d = Circle.radiansToDegrees(2.0);
```

If you want to invoke a class method from inside the class in which it is defined, you don't have to specify the class name. However, it is often good style to specify the class name anyway, to make it clear that a class method is being invoked.

Note that the body of our `Circle.radiansToDegrees()` method uses the class field `PI`. A class method can use any class fields and class methods of its own class (or of any other class). But it cannot use any instance fields or instance methods because class methods are not associated with an instance of the class. In other words, although the `radiansToDegrees()` method is defined in the `Circle` class, it does not use any `Circle` objects. The instance fields and instance methods of the class are associated with `Circle` objects, not with the class itself. Since a class method is not associated with an instance of its class, it cannot use any instance methods or fields.

As we discussed earlier, a class field is essentially a global variable. In a similar way, a class method is a global method, or global function. Although `radiansToDegrees()` does not operate on `Circle` objects, it is defined within the `Circle` class because it is a utility method that is sometimes useful when working with circles. In many nonobject-oriented programming languages, all methods, or functions, are global. You can write complex Java programs using only class methods. This is not object-oriented programming, however, and does not take advantage of the power of the Java language. To do true object-oriented programming, we need to add instance fields and instance methods to our repertoire.

Any method not declared with the `static` keyword is an instance method. An *instance method* operates on an instance of a class (an object) instead of operating on the class itself. It is with instance methods that object-oriented programming starts to get interesting. The `Circle` class defined in Example 3-1 contains two instance methods, `area()` and `circumference()`, that compute and return the area and circumference of the circle represented by a given `Circle` object.

To use an instance method from outside the class in which it is defined, we must prefix it with a reference to the instance that is to be operated on. For example:

```
Circle c = new Circle(); // Create a Circle object; store in variable c
c.r = 2.0;                // Set an instance field of the object
double a = c.area();      // Invoke an instance method of the object
```

If you're new to object-oriented programming, that last line of code may look a little strange. We do not write:

```
a = area(c);
```

Instead, we write:

```
a = c.area();
```

This is why it is called object-oriented programming; the object is the focus here, not the function call. This small syntactic difference is perhaps the single most important feature of the object-oriented paradigm.

The point here is that we don't have to pass an argument to `c.area()`. The object we are operating on, `c`, is implicit in the syntax. Take a look at Example 3-1 again. You'll notice the same thing in the signature of the `area()` method: it doesn't have a parameter. Now look at the body of the `area()` method: it uses the instance field `r`. Because the `area()` method is part of the same class that defines this instance field, the method can use the unqualified name `r`. It is understood that this refers to the radius of whatever `Circle` instance invokes the method.

Another important thing to notice about the bodies of the `area()` and `circumference()` methods is that they both use the class field `PI`. We saw earlier that class methods can use only class fields and class methods, not instance fields or methods. Instance methods are not restricted in this way: they can use any member of a class, whether it is declared static or not.

Consider this line of code again:

```
a = c.area();
```

What's going on here? How can a method that has no parameters know what data to operate on? In fact, the `area()` method does have a parameter. All instance methods are implemented with an implicit parameter not shown in the method signature. The implicit argument is named `this`; it holds a reference to the object through which the method is invoked. In our example, that object is a `Circle`.

The implicit `this` parameter is not shown in method signatures because it is usually not needed; whenever a Java method accesses the instance fields in its class, it is implicit that it is accessing fields in the object referred to by the `this` parameter. The same is true when an instance method invokes another instance method in the same class. I said earlier that to invoke an instance method you must prepend a reference to the object to be operated on. When an instance method is invoked within another instance method in the same class, however, you don't need to specify an object. In this case, it is implicit that the method is being invoked on the `this` object.

You can use the `this` keyword explicitly when you want to make it clear that a method is accessing its own fields and/or methods. For example, we can rewrite the `area()` method to use `this` explicitly to refer to instance fields:

```
public double area() { return Circle.PI * this.r * this.r; }
```

This code also uses the class name explicitly to refer to class field `PI`. In a method this simple, it is not necessary to be explicit. In more complicated cases, however, you may find that it increases the clarity of your code to use an explicit `this` where it is not strictly required.

In some cases, the `this` keyword is required, however. For example, when a method parameter or local variable in a method has the same name as one of the fields of the class, you must use `this` to refer to the field since the field name used alone refers to the method parameter or local variable. For example, we can add the following method to the `Circle` class:

```
public void setRadius(double r) {
    this.r = r;    // Assign the argument (r) to the field (this.r)
                // Note that we cannot just say r = r
}
```

Finally, note that while instance methods can use the `this` keyword, class methods cannot. This is because class methods are not associated with objects.

Excerpts from Flanagan, D. (2005). *Java in a Nutshell*. Cambridge, MA: O'Reilly.

Instance methods are one of the key features of object-oriented programming. That doesn't mean, however, that you should shun class methods. In many cases, it is perfectly reasonable to define class methods. When working with the `Circle` class, for example, you might find that you often want to compute the area of a circle with a given radius but don't want to bother creating a `Circle` object to represent that circle. In this case, a class method is more convenient:

```
public static double area(double r) { return PI * r * r; }
```

It is perfectly legal for a class to define more than one method with the same name, as long as the methods have different parameters. Since this version of the `area()` method is a class method, it does not have an implicit `this` parameter and must have a parameter that specifies the radius of the circle. This parameter keeps it distinct from the instance method of the same name.

As another example of the choice between instance methods and class methods, consider defining a method named `bigger()` that examines two `Circle` objects and returns whichever has the larger radius. We can write `bigger()` as an instance method as follows:

```
// Compare the implicit "this" circle to the "that" circle passed
// explicitly as an argument and return the bigger one.
public Circle bigger(Circle that) {
    if (this.r > that.r) return this;
    else return that;
}
```

We can also implement `bigger()` as a class method as follows:

```
// Compare circle a to circle b and return the one with the larger radius
public static Circle bigger(Circle a, Circle b) {
    if (a.r > b.r) return a;
    else return b;
}
```

Given two `Circle` objects, `x` and `y`, we can use either the instance method or the class method to determine which is bigger. The invocation syntax differs significantly for the two methods, however:

```
Circle biggest = x.bigger(y);    // Instance method: also y.bigger(x)
Circle biggest = Circle.bigger(x, y); // Static method
```

Both methods work well, and, from an object-oriented design standpoint, neither of these methods is "more correct" than the other. The instance method is more formally object-oriented, but its invocation syntax suffers from a kind of asymmetry. In a case like this, the choice between an instance method and a class method is simply a design decision. Depending on the circumstances, one or the other will likely be the more natural choice.