6.3 `static` Methods, `static` Fields and Class `Math`

Most methods execute in response to method calls *on specific objects*. However, sometimes a method performs a task that does not depend on an object. Such a method applies to the class in which it's declared as a whole and is known as a `static` method or a **class method**. (In Section 10.10, you'll see that interfaces also may contain `static` methods.)

Classes often contain convenient `static` methods to perform common tasks. For example, recall that we used class `Math`'s `static` method `pow` to raise a value to a power in Fig. 5.6. To declare a method as `static`, place the keyword `static` before the return type in the method's declaration. For any class imported into your program, you can call the class's `static` methods by specifying the class's name, followed by a dot (`.`) and the method name, as in

```
ClassName.methodName(arguments)
```

`Math` Class Methods

We use various `Math` class methods here to present the concept of `static` methods. Class `Math` provides a collection of methods that enable you to perform common mathematical calculations. For example, you can calculate the square root of `900.0` with the `static` method call

```
Math.sqrt(900.0)
```

This expression evaluates to `30.0`. Method `sqrt` takes an argument of type `double` and returns a result of type `double`. To output the value of the preceding method call in the command window, you might write the statement

```
System.out.println(Math.sqrt(900.0));
```

In this statement, the value that `sqrt` returns becomes the argument to method `println`. There was no need to create a `Math` object before calling method `sqrt`. Also *all* `Math` class methods are `static`—therefore, each is called by preceding its name with the class name `Math` and the dot (`.`) separator.

Software Engineering Observation 6.4

Class `Math` *is part of the* `java.lang` *package, which is implicitly imported by the compiler, so it's not necessary to import class* `Math` *to use its methods.*

Method arguments may be constants, variables or expressions. If `c = 13.0`, `d = 3.0` and `f = 4.0`, then the statement

```
System.out.println(Math.sqrt(c + d * f));
```

calculates and prints the square root of `13.0 + 3.0 * 4.0 = 25.0`—namely, `5.0`. Figure 6.2 summarizes several `Math` class methods. In the figure, *x* and *y* are of type `double`.

Fig. 6.2

`Math` class methods.

Method	Description	Example

`abs(`*x*`)`	absolute value of *x*	`abs(23.7)` is `23.7`
		`abs(0.0)` is `0.0`
		`abs(-23.7)` is `23.7`

`ceil(`*x*`)`	rounds *x* to the smallest integer not less than *x*	`ceil(9.2)` is `10.0` `ceil(-9.8)` is `-9.0`
`cos(`*x*`)`	trigonometric cosine of *x* (*x* in radians)	`cos(0.0)` is `1.0`
`exp(`*x*`)`	exponential method e^x	`exp(1.0)` is `2.71828` `exp(2.0)` is `7.38906`

`floor(`*x*`)`	rounds *x* to the largest integer not greater than *x*	`floor(9.2)` is `9.0` `floor(-9.8)` is `-10.0`
`log(`*x*`)`	natural logarithm of *x* (base *e*)	`log(Math.E)` is `1.0` `log(Math.E * Math.E)` is `2.0`
`max(`*x*`, `*y*`)`	larger value of *x* and *y*	`max(2.3, 12.7)` is `12.7` `max(-2.3, -12.7)` is `-2.3`
`min(`*x*`, `*y*`)`	smaller value of *x* and *y*	`min(2.3, 12.7)` is `2.3` `min(-2.3, -12.7)` is `-12.7`
`pow(`*x*`, `*y*`)`	*x* raised to the power *y* (i.e., x^y)	`pow(2.0, 7.0)` is `128.0` `pow(9.0, 0.5)` is `3.0`
`sin(`*x*`)`	trigonometric sine of *x* (*x* in radians)	`sin(0.0)` is `0.0`
`sqrt(`*x*`)`	square root of *x*	`sqrt(900.0)` is `30.0`
`tan(`*x*`)`	trigonometric tangent of *x* (*x* in radians)	`tan(0.0)` is `0.0`

`static` Variables

Recall from Section 3.2 that each object of a class maintains its *own* copy of every instance variable of the class. There are variables for which each object of a class does *not* need its own separate copy (as you'll see momentarily). Such variables

are declared `static` and are also known as **class variables**. When objects of a class containing `static` variables are created, all the objects of that class share *one* copy of the `static` variables. Together a class's `static` variables and instance variables are known as its **fields**. You'll learn more about `static` fields in Section 8.11.

`Math` Class `static` Constants `PI` and `E`

Class `Math` declares two constants, `Math.PI` and `Math.E`, that represent *high-precision approximations* to commonly used mathematical constants:

- `Math.PI` (3.141592653589793) is the ratio of a circle's circumference to its diameter.
- `Math.E` (2.718281828459045) is the base value for natural logarithms (calculated with class `Math`'s `static` method `log`).

These constants are declared in class `Math` with the modifiers `public`, `final` and `static`. Making them `public` allows you to use them in your own classes. Any field declared with keyword `final` is *constant*—its value cannot change after the field is initialized. Making these fields `static` allows them to be accessed via the class name `Math` and a dot (`.`) separator, just as class `Math`'s methods are.

Why Is Method `main` Declared `static`?

When you execute the Java Virtual Machine (JVM) with the `java` command, the JVM attempts to invoke the `main` method of the class you specify—at this point no objects of the class have been created. Declaring `main` as `static` allows the JVM to invoke `main` without creating an instance of the class. When you execute your application, you specify its class name as an argument to the `java` command, as in

```
java ClassName argument1 argument2 …
```

The JVM loads the class specified by *ClassName* and uses that class name to invoke method `main`. In the preceding command, *ClassName* is a **command-line argument** to the JVM that tells it which class to execute. Following the *ClassName*, you can also specify a list of `String`s (separated by spaces) as command-line arguments that the JVM will pass to your application. Such arguments might be used to specify options (e.g., a filename) to run the application. Every class may contain `main`—only the `main` of the class used to execute the application is called. As you'll learn in Chapter 7, Arrays and `ArrayList`s, your application can access those command-line arguments and use them to customize the application.