6.12 Method Overloading

Methods of the *same* name can be declared in the same class, as long as they have *different* sets of parameters (determined by the number, types and order of the parameters)—this is called **method overloading**. When an overloaded method is called, the compiler selects the appropriate method by examining the number, types and order of the arguments in the call. Method overloading is commonly used to create several methods with the *same* name that perform the *same* or *similar* tasks, but on *different* types or *different* numbers of arguments. For example, `Math` methods `abs`, `min` and `max` (summarized in Section 6.3) are overloaded with four versions each:

1. One with two `double` parameters.
2. One with two `float` parameters.
3. One with two `int` parameters.
4. One with two `long` parameters.

Our next example demonstrates declaring and invoking overloaded methods. We demonstrate overloaded constructors in Chapter 8.

6.12.1 Declaring Overloaded Methods

Class `MethodOverload` (Fig. 6.10) declares two overloaded `square` methods—one calculates the square of an `int` (and returns an `int`) and one calculates the square of a `double` (and returns a `double`). Although these methods have the same name and similar parameter lists and bodies, think of them simply as *different* methods. It may help to think of the method names as "`square` of `int`" and "`square` of `double`," respectively.

Line 7 invokes method `square` with the argument `7`. Literal integer values are treated as type `int`, so the method call in line 7 invokes the version of `square` at lines 12–16 that specifies an `int` parameter. Similarly, line 8 invokes method `square` with the argument `7.5`. Literal floating-point values are treated as type `double`, so the method call in line 8 invokes the version of `square` at lines 19–23 that specifies a `double` parameter. Each method first outputs a line of text to prove that the proper method was called in each case. The values in lines 8 and 20 are displayed with the format specifier `%f`. We did not specify a precision in either case. By default, floating-point values are displayed with six digits of precision if the precision is *not* specified in the format specifier.

```
1 // Fig. 6.10: MethodOverload.java
2 // Overloaded method declarations.
3
4 public class MethodOverload {
5    // test overloaded square methods
6    public static void main(String[] args) {
7       System.out.printf("Square of integer 7 is %d
8       System.out.printf("Square of double 7.5 is %
9    }
```

```java

   // square method with int argument
   public static int square(int intValue) {
      System.out.printf("%nCalled square with int
         intValue);
      return intValue * intValue;
   }

   // square method with double argument
   public static double square(double doubleValue)
      System.out.printf("%nCalled square with doub
         doubleValue);
      return doubleValue * doubleValue;
   }
}
```

```
Called square with int argument: 7
Square of integer 7 is 49
Called square with double argument: 7.500000
Square of double 7.5 is 56.250000
```

Fig. 6.10

Overloaded method declarations.

6.12.2 Distinguishing Between Overloaded Methods

The compiler distinguishes overloaded methods by their **signatures**—a combination of the method's *name* and the *number*, *types* and *order* of its parameters, but *not* its return type. If the compiler looked only at method names during compilation, the code in Fig. 6.10 would be ambiguous—the compiler would not know how to distinguish between the two `square` methods (lines 12–16 and 19–23). Internally, the compiler uses longer method names that include the original method name, the types of each parameter and the exact order of the parameters to determine whether the methods in a class are *unique* in that class.

For example, in Fig. 6.10, the compiler might (internally) use the logical name "`square` of `int`" for the `square` method that specifies an `int` parameter and "`square` of `double`" for the `square` method that specifies a `double` parameter (the actual names the compiler uses are messier). If `method1`'s declaration begins as

```
void method1(int a, float b)
```

then the compiler might use the logical name "`method1` of `int` and `float`." If the parameters are specified as

```
void method1(float a, int b)
```

then the compiler might use the logical name "`method1` of `float` and `int`." The *order* of the parameter types is important—the compiler considers the preceding two `method1` headers to be *distinct*.

6.12.3 Return Types of Overloaded Methods

In discussing the logical names of methods used by the compiler, we did not mention the return types of the methods. *Method calls cannot be distinguished only by return type*. When two methods have the *same* signature and *different* return types, the compiler issues an error message indicating that the method is already defined in the class. Overloaded methods *can* have *different* return types if the methods have *different* parameter lists. Also, overloaded methods need *not* have the same number of parameters.

Common Programming Error 6.8

Declaring overloaded methods with identical parameter lists is a compilation error regardless of whether the return types are different.