

Method Syntax

Let's begin with a simple demo program that uses a method that we define. The operation is deliberately trivial to allow us to focus on the syntax. The goal is to learn how methods are defined and used. The program `DemoMethod` defines and uses a method that computes the larger of two numbers (see Figure 1).

```
1  import java.io.*;
2
3  public class DemoMethod
4  {
5      public static void main(String[] args) throws IOException
6      {
7          BufferedReader stdin =
8              new BufferedReader(new InputStreamReader(System.in), 1);
9
10         System.out.print("Enter an integer: ");
11         int x = Integer.parseInt(stdin.readLine());
12
13         System.out.print("Enter another one: ");
14         int y = Integer.parseInt(stdin.readLine());
15
16         int z = DemoMethod.largerOf(x, y);
17         System.out.println("The larger one is " + z);
18     }
19
20     public static int largerOf(int a, int b)
21     {
22         if (a > b)
23             return a;
24         else
25             return b;
26     }
27 }
```

Figure 1. `DemoMethod.java`

A sample dialogue follows:

```
PROMPT>java DemoMethod
Enter an integer: 35
Enter another one: 24
The larger one is 35
```

Obviously, this program could be written without the `largerOf()` method: It's pretty simple. However, `DemoMethod` serves our goal of introducing the syntax for defining and using methods. The very simple `largerOf()` method compares two integers and returns the larger of the two. The method is defined in lines 20-26; it is used in line 16.

The syntax in line 16 is similar to our use of pre-defined methods in previous programs, such as the `Math.sqrt()` method. The `sqrt()` method is a class method of the `Math` class. The `largerOf()` method is a class method of the `DemoFunction` class. Since the `Math` class and its methods are defined outside the program where `sqrt()` is used, the prefix "`Math.`" is

required to inform the compiler of the location of the method.¹ However, the `largerOf()` method is defined in the class `DemoMethod`, so in line 16, the prefix `"DemoMethod."` is unnecessary. The prefix is used simply to emphasize that `largerOf()` is a class method, just like `sqrt()`. The usual syntax for line 16 is

```
int x = largerOf(x, y);
```

Method Signature

The `largerOf()` method is defined in lines 20-26. The first line of the method definition is

```
public static int largerOf(int a, int b)
```

This is the method's *signature*. It contains all the information needed to use the method. The documentation for each method in Java's API contains only the method's signature and a brief description of its operation. As programmers using the method, that's all we need to know. Let's walk through the signature for the `largerOf()` method.

The reserved word `"public"` is a *modifier*; it specifies the *visibility* attribute of the method. Most Java methods are *public*, meaning they are accessible from other methods, perhaps from methods in other classes. Visibility attributes are discussed in more detail later.

The reserved word `"static"` is also a modifier; it identifies the method as a *class method* (sometimes called a *static method*). It means that the method is not called through an instance of a class but, rather, through the class itself. Understanding this is tricky, so let's review the syntax for calling a class method vs. calling an instance method. (This was discussed earlier.) The method `sqrt()` of the `Math` class is a "class method". It might be called as follows:

```
double x = 25.0
double y = Math.sqrt(x); // sqrt() is a 'class method'
```

The method `substring()` of the `String` class is an "instance method". It might be called as follows:

```
String s1 = "hello";
String s2 = s1.substring(0, 1); // substring() is an 'instance method'
```

Do you see the difference? Preceding `sqrt()` is `"Math."` which identifies the class in which `sqrt()` is defined. Preceding `substring()` is `"s1."` which is an instance variable of the `String` class. `"Math"` is the name of a class (hence "class method"), whereas `"s1"` is the name of an instance variable (hence "instance method").

The preceding discussion is put in terms of the `largerOf()` method as follows: the `largerOf()` method is a static method, or a class method, of the `DemoFunction` class.

Returning to the signature for the `largerOf()` method, the next word is the reserved word `"int"`. This identifies the return type of the method. The `largerOf()` method returns a value of type `int`. Anywhere an integer value can be used — for example, in an expression — the `largerOf()` method can be used. All Java methods must include a return type in their

¹ The `Math` class is part of the `java.lang` package. It is usually necessary to further identify the location of methods and classes with an `import` statement at the beginning of a program, for example `"import java.lang.*;"`. Since the `java.lang` package contains the basic classes required for all Java programs, its location is known by the compiler by default.

definition. In some cases, a method has nothing to return, (e.g., `println()`), and is defined to return a `void`.

Next comes the name of the method followed by a set of parentheses. The name of the method conforms to the naming conventions for Java identifiers given earlier. Within the parentheses are the arguments passed to the method. In the case of `largerOf()`, it receives two arguments of type `int`. The names of the arguments are arbitrary. They are used in the definition of the method (see lines 22-25), but they have no relationship to the names of variables in the calling program.

Immediately following the signature is the definition of the method within a set of braces. The `largerOf()` method is defined in lines 21-26. The operation is pretty simple, so we needn't dwell on the details. The reserved word "return" causes an immediate exit from the method, with control returning to the calling method. `return` is followed by an expression, a variable name, or the value to be returned to the calling program. A compile error occurs if the type of the value returned does not match the type appearing the signature (notwithstanding automatic promotion, for example, of `int` to `double`).

Let's explore the definition of class methods by "updating" several of the demo programs from previous chapters.

Count Words - Revisited

`CountWords3` (see Figure 2) is a reworked version of `CountWords2`, except the code to determine if a word contains only letters is taken out of the `main()` method and placed in a dedicated method called `isAlphaWord()`.

```

1  import java.io.*;
2  import java.util.*;
3
4  public class CountWords3
5  {
6      public static void main(String[] args) throws IOException
7      {
8          BufferedReader stdin =
9              new BufferedReader(new InputStreamReader(System.in), 1);
10
11         String line;
12         String word;
13         StringTokenizer words;
14         int count = 0;
15
16         // process lines until no more input
17         while ((line = stdin.readLine()) != null)
18         {
19             // process words in line
20             words = new StringTokenizer(line);
21             while (words.hasMoreTokens())
22             {
23                 word = words.nextToken();
24                 if (isAlphaWord(word))
25                     count++;
26             }
27         }
28         System.out.println("\n" + count + " alpha words read");
29     }
30
31     public static boolean isAlphaWord(String w)
32     {
33         for (int i = 0; i < w.length(); i++)
34         {
35             String c = w.substring(i, i + 1).toUpperCase();
36             if (c.compareTo("A") < 0 || c.compareTo("Z") > 0)
37                 return false;
38         }
39         return true;
40     }
41 }

```

Figure 2. CountWords3.java

Have a look at lines 21-26 in Figure 2 and notice how easy they are to interpret. All the "dirty work" is gone. The `isAlphaWord()` method is treated as a black box: It returns a `boolean` which is `true` if the word contains only letters of the alphabet. Since it returns a `boolean`, the method can appear anywhere a `boolean` variable can be used, for example, in a relational expression. In fact, it forms the entire relational expression in the `if` statement that determines whether or not to increment the `int` variable `count` (line 24). Recall, that a relational test is always a test for "true", so the expression

```
isAlphaWord(word)
```

is identical to

```
isAlphaWord(word) == true
```

The definition of the `isAlphaWord()` method appears in lines 31-40, and it contains what we fondly referred to earlier as "the dirty work" — in this case, the process of scanning characters in a word to determine if they are all letters. The details were discussed earlier, so we won't say more here.

Find Palindromes - Revisited

The Palindrome program presented in Chapter 4 is re-worked in Figure 3 using a method.

```
1  import java.io.*;
2  import java.util.*;
3
4  public class Palindrome2
5  {
6      public static void main(String[] args) throws IOException
7      {
8          // open keyboard for input (call it 'stdin')
9          BufferedReader stdin =
10             new BufferedReader(new InputStreamReader(System.in), 1);
11
12             // prepare to extract words from lines
13             String line;
14             StringTokenizer words;
15             String word;
16
17             // main loop, repeat until no more input
18             while ((line = stdin.readLine()) != null)
19             {
20                 // tokenize the line
21                 words = new StringTokenizer(line);
22
23                 // process each word in the line
24                 while (words.hasMoreTokens())
25                 {
26                     word = words.nextToken();
27                     if (word.length() > 2 && isPalindrome(word))
28                         System.out.println(word);
29                 }
30             }
31         }
32
33         public static boolean isPalindrome(String w)
34         {
35             int i = 0;
36             int j = w.length() - 1;
37             while (i < j)
38             {
39                 if ( ! (w.charAt(i) == w.charAt(j)) )
40                     return false;
41                 i++;
42                 j--;
43             }
44             return true;
45         }
46     }
```

Figure 3. Palindrome2.java

Note how easy it is to understand the act of checking if words of three or more characters are palindromes (see line 27). Both `word.length() > 2` and `isPalindrome(word)` are relational expressions that return boolean values. They are connected by the logical AND (`&&`) operator. If both expressions are true, the overall result is true and the word is echoed to the standard output in line 28. At the level of using the `isPalindrome()` method, we need not concern ourselves with the details: We just use the method for its intended purpose. By removing the code to check for palindromes, we have eliminated clutter in the `main()` method that otherwise obscures the overall operation.

The definition of the `isPalindrome()` method appears in lines 33-45. The code is identical to that appearing in `Palindrome.java`, except it is packaged in a class method. The method's signature (line 33) declares that the method receives a `String` argument and returns a boolean result.

Reversing Characters in a String

Now let's return to another task we met earlier. The program `StringBackwards2` is the same as `StringBackwards`, except for two small changes. First, the original string is now inputted via the keyboard, and, second, the code to reverse the characters in the string is packaged in a method (see Figure 4).

```
1  import java.io.*;
2
3  public class StringBackwards2
4  {
5      public static void main(String[] args) throws IOException
6      {
7          BufferedReader stdin =
8              new BufferedReader(new InputStreamReader(System.in), 1);
9
10         System.out.print("Enter a message string: ");
11         String s = stdin.readLine();
12         s = backwards(s);
13         System.out.print(s);
14     }
15
16     public static String backwards(String s)
17     {
18         String b = "";
19         for (int i = 0; i < s.length(); i++)
20             b = s.substring(i, i + 1) + b;
21         return b;
22     }
23 }
```

Figure 4. `StringBackwards2.java`

A sample dialogue follows: (User input is underlined.)

```
PROMPT>java StringBackwards2
Enter a message string: Java is fun!
!nuf si avaJ
```

Once again, we see a clear improvement in readability in the main program (lines 11-13) by taking a "divide-and-conquer" approach. The code to reverse the characters in a string is

packaged in the `backwards()` method. The method is defined in lines 16-22; it is used in line 12.

Note that the definition of the `backwards()` method uses an identifier, `s`, which is the same as an identifier in the main program (lines 11, 12, & 13). The two are not related. Identifiers used in the definition of methods have no relationship to identifiers of the same name in the calling program or in the definition of other methods. Have a second look at Figure 4 and make sure you are comfortable with this point. The same identifiers were used in `main()` and in `backwards()` in Figure 4 specifically to bring out this point. If the three uses of `s` in the definition of `backwards()` (lines 16, 19, & 20) are changed to, for example, `originalString`, the effect is the same. We will say more about this later in our discussion of variable scope.

We'll return to `StringBackwards2` later in our discussion of recursion and debugging.