# SUPPLEMENTAL CHAPTER 6
# Writing Methods in a Non-Object-Oriented Environment

## Objectives

- Write programs with several methods in the same class.
- Understand scope, persistence, and use of the `static` modifier.
- Understand different ways to terminate method execution.
- Understand a method's type.
- Learn how to pass data into and out of methods.
- Understand the behavior of arguments and parameters.
- Learn how to trace execution through method calls and returns.
- Learn how to overload a method name.
- Use `private` to hide subordinate helper methods.

## Outline

## S6.1 Introduction

Chapter 5 served as a bridge from basic programming language constructs (variables, assignments, operators, `if` statements, loops, etc.) to modular concepts. There we focused primarily on one important aspect of modularization – calling prebuilt methods from your `main` method. The Interlude between Chapter 5 and Chapter 6 showed you how to do more than just call prebuilt methods. You learned how to write your own methods that can be called from `main`. The purpose of writing methods in addition to the `main` method is to partition your programs into separate modules, where each module solves a particular subtask. That sort of partitioning makes large programs easier to work with. Making large programs easier to work with is very important because today's computers use lots of very large programs!

In this chapter, we start with an overview of basic method concepts. We then lead you through the design and implementation of a simple program that implements two methods – `main` and a method that's called from `main`. This complements what you learned in the Interlude. It provides another example of the use of method *parameters*, which receive data from a method call. It also illustrates use of a separate set of *local variables*, which store data that's hidden within a particular method. You'll learn how to adapt the previously described tracing technique to a program with multiple methods. You'll learn how to use the same name for multiple methods that perform only slightly different tasks. And finally you'll learn how to delegate subtasks to hidden subordinate methods.

## S6.2 Methods Overview

A method is an isolated chunk of code that accomplishes one task. Often, the task is small, like calculating the area of something. But the task can also be big, like playing a game of tic-tac-toe. To implement a big task, write the task's `main` method first and postpone the implementation of subtask details by calling other methods to perform the subtasks.

Here are some of the benefits of partitioning a program into separate methods:

- Readability – If you move some of the details from a big method into one or more smaller methods, that will make the original method smaller. And as long as the method partitioning makes sense, having smaller methods will lead to programs that are more readable (i.e., easier to understand).

- Reusability – If a particular subtask needs to be performed at more than one place within a large method and you implement the subtask with its own method, that's an example of code re-use because you're using the same code in more than one place. Code re-use is beneficial because it means that you don't have to re-think or re-enter a task's code more than one time.

- Maintainability – If a particular subtask needs to be performed at more than one place within a program and you implement the subtask with its own method, that can make the program easier to maintain. Suppose the subtask's method needs to be fixed or enhanced. You will have to make the change in just one place. On the other hand, if you implement the subtask with redundant code (i.e., you insert the complete code at multiple places within your program) and the code needs to be fixed or enhanced, you will have to make the change in multiple places. That can lead to inconsistencies and errors.

- Encapsulation – A method can define local variables that hide the method's internal data from the outside world. Such data hiding is beneficial because it reduces the danger of having activity in one part of a program accidentally interfere with data in another part of the program.

## S6.3 Example UsernamePasswordEntry Program

Suppose you want a program to prompt the user for two inputs − a username and a password. After each input, the program should check to see whether the user-entered data meets certain criteria. The input username must have at least 4 characters, and the characters must be either letters or numbers. The input password must have at least 6 characters, the characters must be either letters or numbers, and at least one of the characters must be a number. If the user-entered data is invalid, the program should generate an error message and then re-prompt the user to enter valid input. The program should repeat that process until the user-entered data is valid.

If the user's input conforms to the required constraints, the program should produce output like this:

Sample session:

```
Your username must be alphanumeric with at least 4 characters.
Username: ksebelius
Your password must be alphanumeric with at least 6 characters and
at least 1 digit.
Password: windpwr2

Your username is "ksebelius"
Your password is "windpwr2"
```

If the user enters a valid username and invalid passwords, the program should produce output like this:

Sample session:

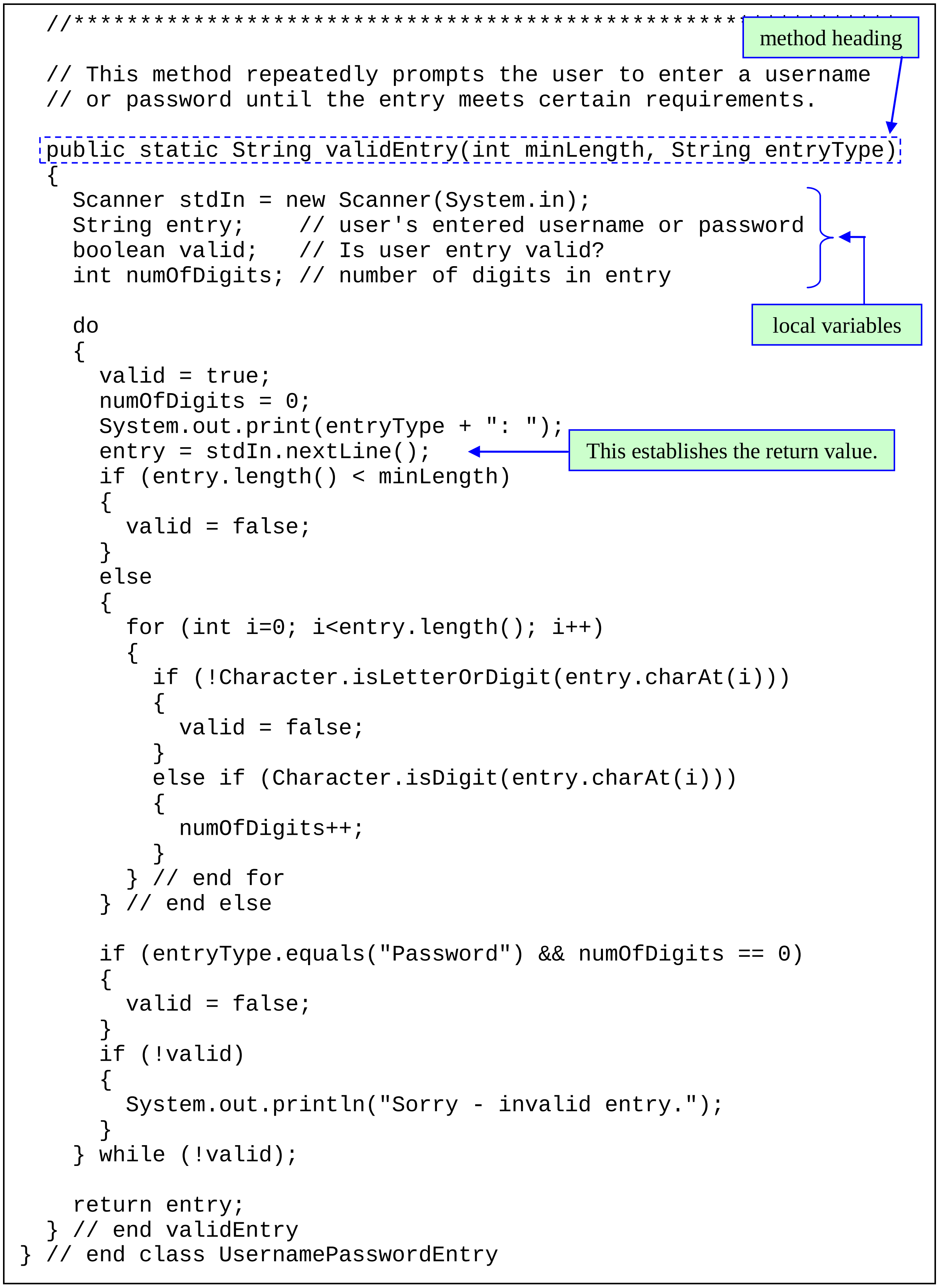
```
Your username must be alphanumeric with at least 4 characters.
Username: ksebelius
Your password must be alphanumeric with at least 6 characters and
at least 1 digit.
Password: wind2
Sorry - invalid entry.
Password: windpwrTwo
Sorry - invalid entry.
Password: windpwr2

Your username is "ksebelius"
Your password is "windpwr2"
```

Figures S6.1a and S6.1b show the UsernamePasswordEntry program, which implements the functionality described above. The program's `main` method prints the first, third, and fourth lines of the above output and the last two lines of the above output. But it delegates other output, input, and input validation operations to a subordinate `validEntry` method. In particular, note how the program avoids code duplication by calling the same subordinate method, `validEntry` for username entry and password entry. In the first subordinate method call, the calling arguments specify the minimum number of characters, 4, and the prompting word, "Username." In the second subordinate method call, the calling arguments specify the minimum number of characters, 6, and the prompting word, "Password." These different arguments make the subordinate method behave in an appropriately different way for each of the two tasks.

```
/*******************************************************************
* UsernamePasswordEntry.java
* Dean & Dean
*
* This program prompts the user to enter valid username and
* password values.
*******************************************************************/

import java.util.Scanner;

public class UsernamePasswordEntry
{
  public static void main(String args[])
  {
    String username;    }      local variables
    String password;    }

    System.out.println("Your username must be alphanumeric" +
      " with at least 4 characters.");
    username = validEntry(4, "Username");          method call

    System.out.println("Your password must be alphanumeric" +
      " with at least 6 characters and\nat least 1 digit.");
    password = validEntry(6, "Password");          method call

    System.out.println("\nYour username is \"" + username + "\"");
    System.out.println("Your password is \"" + password + "\"");
  } // end main
```

**Figure S6.1a** UsernamePasswordEntry program – part A

Figure S6.1b contains the remainder of the UsernamePasswordEntry program – the subordinate `validEntry` method. The method prompts the user to enter a value. The first time the method is called, the prompt will be for the username. The second time the method is called, the prompt will be for the password. The user entry is then checked to see if it contains the minimum number of characters and if each of its characters is either a letter or a digit. For passwords, the method checks one more thing – whether the user entry contains at least one digit. If all those criteria are met, then the `validEntry` method returns the user entry back to the `main` method. Otherwise, the `validEntry` method warns the user and loops back for another user entry attempt.



```
  //**************************************************************

  // This method repeatedly prompts the user to enter a username
  // or password until the entry meets certain requirements.

  public static String validEntry(int minLength, String entryType)
  {
    Scanner stdIn = new Scanner(System.in);
    String entry;    // user's entered username or password
    boolean valid;   // Is user entry valid?
    int numOfDigits; // number of digits in entry

    do
    {
      valid = true;
      numOfDigits = 0;
      System.out.print(entryType + ": ");
      entry = stdIn.nextLine();
      if (entry.length() < minLength)
      {
        valid = false;
      }
      else
      {
        for (int i=0; i<entry.length(); i++)
        {
          if (!Character.isLetterOrDigit(entry.charAt(i)))
          {
            valid = false;
          }
          else if (Character.isDigit(entry.charAt(i)))
          {
            numOfDigits++;
          }
        } // end for
      } // end else

      if (entryType.equals("Password") && numOfDigits == 0)
      {
        valid = false;
      }
      if (!valid)
      {
        System.out.println("Sorry - invalid entry.");
      }
    } while (!valid);

    return entry;
  } // end validEntry
} // end class UsernamePasswordEntry
```

**Figure S6.1b** UsernamePasswordEntry program – part B

Note the description at the top of Figure S6.1b. Proper coding conventions suggest that above each method, you should insert a blank line (the white space at the bottom of Figure S6.1a), a line of asterisks, another blank line, a description of the method, and another blank line. The blank lines and asterisks serve to separate different methods. The method description helps someone who's reading your program to quickly get an idea of what's going on.

## Method Heading

The method itself consists of a *heading* and a *body*. Note `validEntry`'s heading in Figure S6.1b. Also note `validEntry`'s body, which consists of everything below the heading down to the method's closing brace:

```
} // end validEntry
```

Let's now examine the `validEntry` method heading in depth. The `public` modifier establishes the method's accessibility. Accessibility can be either `public` or `private`. Given the standard definitions of the words "public" and "private," you can probably surmise that a `public` method is easier to access than a `private` method. If you declare a method to be `public`, the method can be accessed from anywhere (from within the method's class, and also from outside the method's class). When you want a method to perform a local task only, you should declare it to be `private`. We'll discuss `private` methods in more detail in a later section.

In the UserPasswordEntry program above, the `validEntry` method performs a local task, so it could have been declared `private`. But we chose to make it `public` so that, as an option, it could also be accessed from the outside world – from a different class. In other words, since the `validEntry` method is `public`, you can use it like you use the `public` Java API methods described in Chapter 5.

Note the `static` modifier in `validEntry`'s heading. The `static` modifier establishes the method as a class method. As you might recall, class methods are associated with an entire class, not a particular instance of a class. And that makes class methods easier to work with. If we had omitted the `static` modifier in `validEntry`'s heading, then extra work would be necessary in order to use the method. Specifically, you'd have to instantiate a `UserPasswordEntry` object and use that object to call `validEntry`. You'll get to that sort of thing later on, when you jump into the deep end of the object-oriented-programming pool. Since `validEntry` is a class method, you can call it easily, as shown in this code fragment from Figure S6.1a:

```
username = validEntry(4, "Username");
```

You might recall that with class methods found in the `Math` class, you call them by prefacing the call with the method's class name, `Math`. For example:

```
futureValue = presentValue * Math.pow((1 + interestRate), years);
```

In Figure S6.1a's `main` method, you could preface `validEntry`'s method call with `validEntry`'s class name, `UserPasswordEntry`, but it's unnecessary. Why? Because `main` and `validEntry` are in the same class.

Note the third word in `validEntry`'s heading – `String`. `String` is the method's return type, and it specifies the type of data that the method will return. We'll discuss the return operation in a later section.

Finally, note `minLength` and `entryType` inside the parentheses in `validEntry`'s method heading. Those are *parameters*. Each parameter definition consists of the parameter's type followed by the parameter's name. Parameters are in charge of receiving and storing argument values that come from method calls. Here's the first call to `validEntry` , copied from Figure S6.1a:

```
 username = validEntry(4, "Username");
```

The argument 4 gets passed to the parameter `minLength`, and the argument `"Username"` gets passed to the parameter `entryType`.

## S6.4 Local Variables

A *local variable* is a variable that's declared and used "locally" inside a method or in a `for` loop header. As you perhaps now realize, all the variables we defined in prior chapters were local variables. They were all declared within `main` methods, so they were all local variables within the `main` method, i.e., they were accessible only from within the `main` method. We didn't bother to explain the term "local variable" until now because we did not create any other methods besides `main`, and the idea of a variable being local to `main` wouldn't have made much sense.

### Scope

A local variable has *local scope* – it can be used only from the point at which the variable is declared to the end of the variable's *block*. A variable's block is established by the closest pair of braces that enclose the variable's declaration. Most of the time, you should declare a method's local variables at the top of the method's body. The scope of such variables is then the entire body of the method.

`for` loop index variables are local variables, but they are special. Their scope rule is slightly different from what is described above. As you know from Chapter 4, you should normally declare a `for` loop's index variable within the `for` loop's header. The scope of such a variable is the `for` loop's header plus the `for` loop's body.

Method parameters are usually not referred to as "local variables," but they are very similar to local variables in that they are declared and used "locally" inside a method. As with local variables, the scope of a method's parameters is limited to within the body of that method.

The UsernamePasswordEntry program in Figures S6.1a and S6.1b has two variables that are local to the `main` method (`username` and `password`), four variables that are local to the `validEntry` method (`entry`, `valid`, `numOfDigits`, and `stdIn`), and two parameters that are local to the `validEntry` method (`minLength` and `entryType`).

Frequently, local variables are not initialized when they are declared. That's the case for all of the local variables in the Figures S6.1a and S6.1b, except for `stdIn`. The initial value of any local variable that is not specifically initialized is *garbage*. Garbage means that the variable's value is unknown – it's whatever just happens to be in memory at the time the variable is created. All local variables must be given non-garbage values before they can be accessed.

If a program's logic is such that there might possibly be an attempt to access a variable that contains garbage, the compiler will generate a compilation error. For example, what might happen if the `valid = true;` statement were removed from the top of the `do` loop header in Figure S6.1b? It's possible that when the program runs, the user might enter a valid value right off the bat.[1] In that case, none of the subsequent `valid = false;` assignment statements would get executed and `valid` would retain its original garbage value. Then when the `if (!valid)` line is reached, there would be an attempt to access a variable that contains garbage. The compiler is able to recognize this potential problem. So if the `valid = true;` statement were removed from the top of the `do` loop header in Figure S6.1b, the compiler would generate this error message:

```
…\UsernamePasswordEntry1.java:71: variable valid might not have been
initialized
      if (!valid)
```

### Local Variable Persistence

OK, let's say you do initialize a local variable. How long will it *persist*? A local variable (or parameter) persists only within its scope and only for the current duration of the method (or `for` loop) in which it is defined. The next time the method (or `for` loop) is called, the local variable's value resets to garbage or the value given to it by initialization.

## S6.5 Returning a Value

In this section, we discuss in depth the process of returning a value from a method. Since the primary purpose of Figure S6.1b's `validEntry` method is to return a validated user entry, we'll use that method as the vehicle for our discussion. Recall that the `validEntry` method heading, copied below for your convenience, includes a `String` return type:

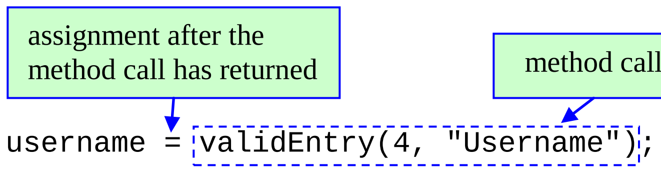
```
public static String validEntry(int minLength, String entryType)
```

The `String` return type must match the type of the value that's returned from the method. That is indeed the case because near the top of the `validEntry` method, the local variable `entry` is declared to be a `String`. And at the bottom of the `validEntry` method, `entry`'s value is returned by virtue of this statement:

```
return entry;
```

The `return` statement passes a copy of the `entry` value back to the place where the method was called. Figure S6.1a's `main` method calls the `validEntry` method two times. Here's the first `validEntry` method call:

[1] When something occurs "right off the bat," it occurs immediately. The idiom originates from baseball. Just after a batter hits a baseball, the baseball will be "right off the bat" and it will tend to be fast. Such fastness implies immediacy.



```
username = validEntry(4, "Username");
```

After the `validEntry` method has completed its execution, the Java Virtual Machine (JVM) returns the validated username and effectively substitutes that username for the `validEntry(4, "username")` method call in the `main` method. To perform a mental trace, imagine that the method call is overlaid by the returned value. So if the user enters "ksebelius" for a username, then "ksebelius" is returned, and you can replace the `validEntry` method call by the value "ksebelius". Since the method call is embedded inside an assignment statement, the method call's returned value, "ksebelius", is then assigned to the `username` variable.

## S6.6 Method Calls Without Data Transfer

### `void` Methods

For a long time, you've used this heading for your `main` methods:

```
  public static void main(String args[])
```

Note that `main`'s return type is `void`. That means `main` returns nothing, and we say that it's a "`void` method." In this section, you'll learn about `void` method details by examining the MemorizePasswordEntry program. The program is similar to the UsernamePasswordEntry program in that it handles user input for a password, but it doesn't bother with a username or input validation. The MemorizePasswordEntry program contains two new methods, whose headings are:

```
  public static void printInstructions()
  public static void memorizePassword(String password)
```

To exercise these two new methods, we insert the following two statements at the end of Figure S6.2a's `main` method:

```
  printInstructions();
  memorizePassword(password);
```

Since these methods are `void` methods and return nothing, their method calls appear on lines by themselves. Remember how the `validEntry` method calls were embedded inside assignment statements? That wouldn't work with `printInstructions` and `memorizePassword` because they return nothing and you can't assign nothing in an assignment statement.

```
/*******************************************************************
* MemorizePasswordEntry.java
* Dean & Dean
*
* This program prompts the user to enter a password and
* then attempts to get the user to memorize it.
*******************************************************************/

import java.util.Scanner;

public class MemorizePasswordEntry
{
  public static void main(String args[])
  {
    Scanner stdIn = new Scanner(System.in);
    String password;

    System.out.print("Password: ");
    password = stdIn.nextLine();
    System.out.println("Your password is \"" + password + "\"");
    printInstructions();
    memorizePassword(password);
  } // end main
```

new method calls

**Figure S6.2a** MemorizePasswordEntry program – part A

The `printInstructions` method appears in Figure S6.2b. Since its method heading parentheses are empty, the `printInstructions` method defines no parameters, and a method call passes no data in. The `printInstructions` method simply prints this message:

```
Memorize your password, and after you have it memorized,
press Enter to continue.
```

Since `printInstructions` is a `void` method, no data passes out. Note that there is no `return` statement. When there is no `return` statement, a method automatically returns after execution of its last statement.

```
//**************************************************************

// This method explains what to do with the password entry.

public static void printInstructions()
{
  Scanner stdIn = new Scanner(System.in);

  System.out.println(
    "\nMemorize your password, and after you have it" +
    " memorized,\npress Enter to continue.");
  stdIn.nextLine(); // Wait for user input. Then discard it.
} // end printInstructions
```

automatic return after last statement

**Figure S6.2b** MemorizePasswordEntry program – part B

## Empty `return` Statement

For methods with a `void return` type, it's legal to have an *empty* `return` statement. The empty `return` statement looks like this:

```
return;
```

The empty `return` statement does what you'd expect. It terminates the current method and causes control to be passed back to the calling module at the point that immediately follows the method call.

Note the empty `return` statement inside of Figure S6.2c's `memorizePassword` method. The purpose of `memorizePassword` is to ensure that the user has memorized his/her password. This is done by receiving the password as a `password` parameter, making the previously entered password scroll off the display screen by printing 40 blank lines in a loop, and then prompting the user to enter the memorized password. If the user's entry equals the `password` parameter's value, the program prints "Perfect!" and executes the empty `return` statement. If the user's entry does not equal the `password` parameter's value, the program prints the original password, asks the user to try again to memorize it, and repeats the loop.

The `do` loop's closing condition, `while (true)`, ensures that the loop continues. So what keeps the program from running on forever? What stops the looping is the `return` statement that's buried inside the loop. When the `return` statement executes, control bypasses the closing `while (true);` and returns to MemorizePasswordEntry's `main` method.

```
  //**************************************************************

  // This method ensures that user has memorized his/her password.

  public static void memorizePassword(String password)
  {
    Scanner stdIn = new Scanner(System.in);
    String confirmationPassword; // user's memorization attempt

    do
    {
      // Scroll original password entry off the display screen.
      for (int i=0; i<40; i++)
      {
        System.out.println();
      }

      System.out.print("Enter your memorized password: ");
      confirmationPassword = stdIn.nextLine();

      if (confirmationPassword.equals(password))
      {
        System.out.println("Perfect!");
        return;
      }
      else
      {
        System.out.println(
          "Nope. \"" + password + "\" is your password.");
        System.out.println(
          "After you have it memorized, press Enter to continue.");
        stdIn.nextLine(); // Wait for user input. Then discard it.
      }
    } while (true);
  } // end memorizePassword
} // end class MemorizePasswordEntry
```

empty `return` statement buried inside a loop

**Figure S6.2c** MemorizePasswordEntry program – part C

When a method includes conditional branching (as in an `if` statement or `switch` statement), it's possible to return from more than one place in the method. In such cases, all returns must match the type specified in the method heading. It would be illegal to have an empty `return` statement and a non-empty `return` statement in the same method. Why? Empty and non-empty `return` statements have different return types (`void` for an empty `return` statement and some other type for a non-empty `return` statement). There is no way to specify a type in the heading that simultaneously matches two different return types.

The empty `return` statement is a helpful statement in that it provides an easy way to exit quickly from a `void` method. However, it does not provide unique functionality. Code that uses an empty `return` statement can always be replaced by code that has no `return` statements. For example, Figure S6.3's `memorizePassword` method is functionally equivalent to Figure S6.2c's `memorizePassword` method, but Figure S6.3's version has no `return` statement. It implements the return the old-fashioned way – by allowing the loop to terminate via the loop's termination condition and allowing the method to finish executing. When the method finishes executing, control will automatically return to where the method was called. Notice how the loop terminates by replacing the original `do` loop's closing condition, `while (true)`, with this:

```
} while (!confirmationPassword.equals(password));
```

```
//*************************************************************

// This method ensures that user has memorized his/her password.

public static void memorizePassword(String password)
{
  Scanner stdIn = new Scanner(System.in);
  String confirmationPassword; // user's memorization attempt

  do
  {
    // Scroll original password entry off the display screen.
    for (int i=0; i<40; i++)
    {
      System.out.println();
    }

    System.out.print("Enter your memorized password: ");
    confirmationPassword = stdIn.nextLine();

    if (confirmationPassword.equals(password))
    {
      System.out.println("Perfect!");
    }
    else
    {
      System.out.println(
        "Nope. \"" + password + "\" is your password.");
      System.out.println(
        "After you have it memorized, press Enter to continue.");
      stdIn.nextLine(); // Wait for user input. Then discard it.
    }
  } while (!confirmationPassword.equals(password));
} // end memorizePassword
```

automatic return when loop terminates

**Figure S6.3** Alternate way to return from memorizePassword method

## `return` Statement Within a Loop

Programmers in industry often are asked to maintain (fix and improve) other people's code. In doing that, they often find themselves having to examine the loops and, more specifically, the loop termination conditions in the program they're working on. Therefore, it's important that loop termination conditions be clear. Normally, loop termination conditions appear in the standard loop-condition location. For `while` loops, that's at the top, for `do` loops that's at the bottom, and for `for` loops, that's in the header's second component. However, a `return` statement inside a loop results in a loop termination condition that's not in a standard location. For example, in our first `memorizePassword` method (in Figure S6.2c), the `return` statement is inside an `if` statement and the loop termination condition is consequently "hidden" in the `if` statement's condition.

In the interest of maintainability, you should use restraint when considering the use of a `return` statement inside a loop. Based on the context, if inserting `return` statements inside a loop improves clarity, then feel free to insert. However, if it simply makes the coding chores easier and it does not add clarity, then don't insert. So, which `memorizePassword` implementation is better – the empty `return` version or the `return`-less version? In general, we prefer the `return`-less version for maintainability reasons. However, because the code in both of our `memorizePassword` methods is relatively simple, it doesn't make much of a difference here.

## S6.7 Argument Passing

You've worked with argument passing for quite a while now, but there are a few more details that you should be aware of. In this section, we discuss the pass-by-value scheme for passing arguments, and we also discuss using the same name versus using different names for argument-parameter pairs.

### Example

Let's dig into the details of argument passing by examining another program, ConfirmPasswordEntry. The purpose of ConfirmPasswordEntry is to determine whether the user has memorized his/her entered password. There's no loop this time (as there is in the MemorizePasswordEntry program), so if the user guesses wrong, that's it, there are no second guesses.

See Figure S6.4a, which shows the program's `main` method. Within the `main` method, note the call to `confirmPassword`. That method call is embedded in the condition part of an `if` statement. That should make sense when you realize that the `confirmPassword` method has a return type of `boolean`. As shown in Figure S6.4b, the `confirmPassword` method returns `true` if the user guesses the password correctly and returns `false` otherwise. The returned `true` or `false` value is then used as the condition in main's `if` statement.

```
/*******************************************************************
* ConfirmPasswordEntry.java
* Dean & Dean
*
* This program prompts the user to enter a password and
* then determines whether the user has memorized it.
*******************************************************************/

import java.util.Scanner;

public class ConfirmPasswordEntry
{
  public static void main(String args[])
  {
    Scanner stdIn = new Scanner(System.in);
    String password;

    System.out.print("Password: ");
    password = stdIn.nextLine();
    if (confirmPassword(password))
    {
      System.out.println("Perfect!");
    }
    else
    {
      System.out.println("Nope. \"" + password + "\" is your" +
        " password. Try harder to memorize it!");
    }
  } // end main
```

method call embedded in an `if` statement

This displays the value of `main`'s local variable password.

**Figure S6.4a** ConfirmPasswordEntry program – part A

```
  //***************************************************************

  // This method returns true or false depending on whether the
  // user has memorized his/her password.

  public static boolean confirmPassword(String password)
  {
    Scanner stdIn = new Scanner(System.in);
    String originalPassword = password;

    System.out.println("Your password is \"" + password + "\"");
    System.out.println(
      "After you have it memorized, press Enter to continue.");
    stdIn.nextLine(); // Wait for user input. Then discard it.

    // Scroll original password entry off the display screen.
    for (int i=0; i<40; i++)
    {
      System.out.println();
    }

    System.out.print("Enter your memorized password: ");
    password = stdIn.nextLine();

    if (password.equals(originalPassword))
    {
      return true;
    }
    else
    {
      return false;
    }
  } // end confirmPassword
} // end class ConfirmPasswordEntry
```

The `password` parameter receives the `password` argument's value.

This might change the value of `confirmPassword`'s password.

Sample Session:

```
Password: uwmpanther4
Your password is "uwmpanther4"
After you have it memorized, press Enter to continue.

<40 blank lines>

Enter your memorized password: panther4
Nope. "uwmpanther4" is your password. Try harder to memorize it!
```

initial value of local variable in `main`

This changes the value of `confirmPassword`'s parameter.

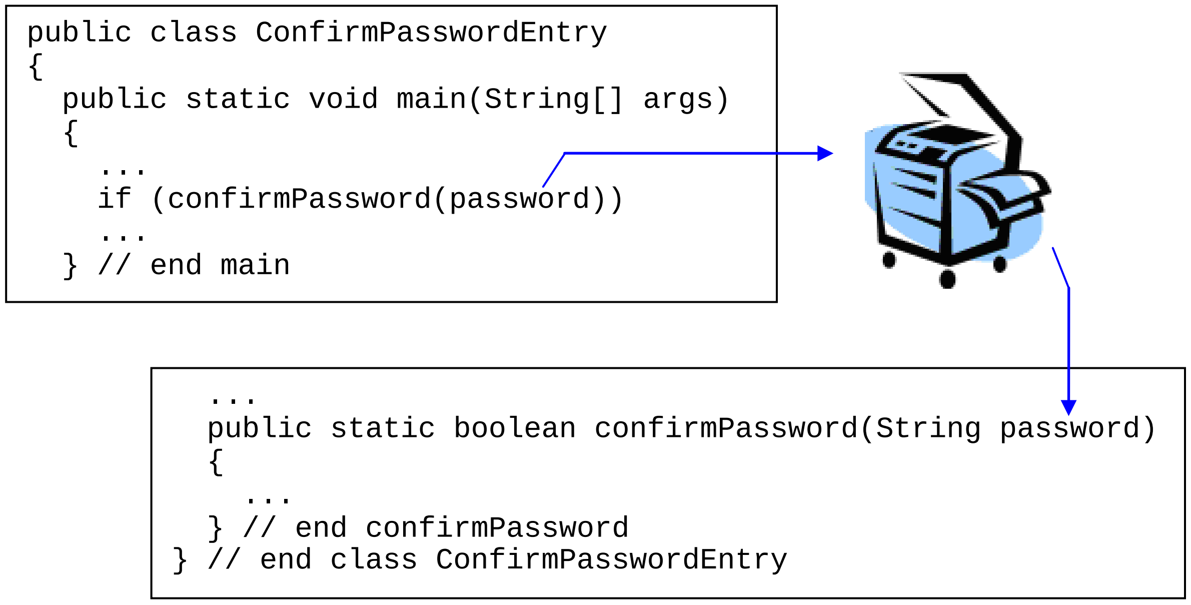
Final value of local variable in `main` – it did not change!

**Figure S6.4b** ConfirmPasswordEntry program – part B

ConfirmPasswordEntry's `main` method calls the `confirmPassword` method with an argument named `password`. The `password` argument holds the user's original password entry. As part of the method call process, `main`'s `password` argument value gets assigned to the `confirmPassword` method's `password` parameter, as shown in Figure S6.4b. The `confirmPassword` method then makes the previously entered password scroll off the display screen by printing 40 blank lines in a loop. The `confirmPassword` method then prompts the user to guess the memorized password. Here's the key – the user's guess gets assigned into the `password` parameter. What happens to the corresponding `password` argument variable in the `main` method if the `password` parameter's value changes? Will there be a simultaneous change in `main`'s `password` argument variable? Because the `password` argument and the `password` parameter are distinct variables, the `password` variable in `main` does not change with the `password` variable in `confirmPassword`. So as shown in Figure S6.4b's sample session, when `main` prints its version of `password` at the end of the program, it prints the unchanged original password value, "uwmpanther4", not the "panther4" value stored in `confirmPassword`'s `password` variable.

## Pass-by-Value

We say that Java uses *pass-by-value* for its argument-passing scheme. As illustrated by Figure S6.5, pass-by-value means that the JVM passes a copy of the argument's value (not the argument itself) to the parameter. Changing the copy does not change the original.



```
public class ConfirmPasswordEntry
{
  public static void main(String[] args)
  {
    ...
    if (confirmPassword(password))
    ...
  } // end main
```

```
    ...
    public static boolean confirmPassword(String password)
    {
      ...
    } // end confirmPassword
  } // end class ConfirmPasswordEntry
```

**Figure S6.5** Pass-by-value means a copy of the argument's value goes to the corresponding

Notice that the calling method's argument is called `password` and the `confirmPassword` method's parameter is called `password` also. Does using the same name make them the same thing? No! They are separate variables separately encapsulated in separate blocks of code. Because these two variables are in separate blocks of code, there is no conflict, and it's OK to give them the same name. Using the same name is natural because these two variables describe the same kind of thing. When names are in different blocks, you don't have to worry about whether they are the same or not. That's the beauty of encapsulation. Big programs would be horrible nightmares if you were prohibited from using the same name in different blocks of code.

### Same Name Versus Different Names for Argument-Parameter Pairs

So far in this chapter, all of our programs have used the same name for argument-parameter pairs because we want to encourage you to take advantage of encapsulation and use the most natural and reasonable names whenever you can. But we don't want you to think that argument and parameter names must be the same always. When it's more natural and reasonable to use different names for an argument-parameter pair, then use different names. The only requirement is that the argument's type must match the parameter's type. For example, suppose you felt that "originalPassword" would be a more descriptive name for the parameter in `confirmPassword`'s heading. To implement that feeling and implement a functionally equivalent `confirmPassword` method, all you'd have to do is replace the method's first four lines with these four lines:

```
public static boolean confirmPassword(String originalPassword)
{
  Scanner stdIn = new Scanner(System.in);
  String password; // user's memorization attempt
```

The compiler would have no problem with the `originalPassword` parameter in `confirmPassword` and the `password` argument in `main` having different names because the parameter and argument would be the same type – `String`.

In the above code fragment, `password` is a local variable. As such, it would not interfere with `main`'s `password` variable. When the user's password guess gets assigned into `confirmPassword`'s `password` variable (as shown in Figure S6.4b), `main`'s `password` variable would not be affected. Yeah!

## S6.8 Tracing a Program with More than One Method

To reinforce what you've learned so far, we'll trace a simplified version of our UsernamePasswordEntry program. Remember the tracing procedure we used in prior chapters? It worked fine for programs with only one method – the `main` method. But for multiple methods, you'll need to keep track of which method you're in. In addition, you'll need to keep track of parameters and local variables in subordinate methods. This requires a more elaborate trace table.

The first step in performing a trace is to generate a copy of the code that includes line numbers. These line numbers provide the necessary connection between individual trace events and the code statements that caused those events. Figure S6.6 contains the UserPasswordEntryTrace program with line numbers.

```
 1  /****************************************************************
 2  * UsernamePasswordEntryTrace.java
 3  * Dean & Dean
 4  *
 5  * This program prompts the user to enter mimimum-length
 6  * username and password values.
 7  ****************************************************************/
 8
 9  import java.util.Scanner;
10
11  public class UsernamePasswordEntryTrace
12  {
13    public static void main(String args[])
14    {
15      String username;
16      String password;
17
18      username = validEntry(4, "username");
19      password = validEntry(6, "password");
20      System.out.println("\nYour username is \"" + username + "\"");
21      System.out.println("Your password is \"" + password + "\"");
22    } // end main
23
24    //*************************************************************
25
26    // This method repeatedly prompts the user to enter a username
27    // or password until the entry is a minimum length.
28
29    public static String validEntry(int minLength, String entryType)
30    {
31      String entry;    // user's entered username or password
32      Scanner stdIn = new Scanner(System.in);
33
34      do
35      {
36        System.out.print(
37          "Enter a " + entryType + " with at least " +
38          minLength + " characters: ");
39        entry = stdIn.nextLine();
40        if (entry.length() < minLength)
41        {
42          System.out.println("Sorry - too short.");
43        }
44      } while (entry.length() < minLength);
45
46      return entry;
47    } // end validEntry
48  } // end class UsernamePasswordEntryTrace
```

**Figure S6.6** Numbered UsernamePasswordEntryTrace program

## Trace Setup

Figure 6.7 shows the setup. As with previous traces, the input goes in the top-left corner. To illustrate the validation loop's functionality, we have inserted an invalid password entry before the final valid password entry.

**input**
ksebelius
wind
windpwr2

| UsernamePasswordEntryTrace | | | | | | |
|---|---|---|---|---|---|---|
| | main | | validEntry | | | |
| ***line#*** | **username** | **password** | **minLength** | **entryType** | **entry** | **output** |

**Figure S6.7** Trace setup for the UsernamePasswordEntryTrace program

Unlike traces in previous chapters, the trace-table headings now require more than one line. The first heading line has class names. Because both methods are `public`, they could be in different classes (with separate line numbers for each class), but we elected to put them both in the same `UserNamePasswordEntryTrace` class. Under each class name heading, there's a heading for each of that class's methods. Under each method-name heading, there's a heading for each of that method's parameters and local variables. The two entries under `main` are `main`'s two local variables (`username` and `password`). The first two entries under `validEntry` are its parameters (`minLength` and `entryType`), and the third entry under `validEntry` is its local variable (`entry`). The wide column on the right is for output.

The `validEntry` method also defines another local variable, `stdIn`, but we don't need to trace it because it uses Java API components. They've already been traced and tested thoroughly by the implementers of the Java programming language, and you can assume they work properly.

## Trace Execution

Using Figure S6.7's trace setup as a starting point, we'll walk you through the key sections of the trace shown in Figure S6.8. When declaring a local variable (as on lines 15 and 16 in Figure S6.6), enter the code line number and initial values in appropriate columns, using "?" if not initialized. When calling a method (as on lines 18 and 19), enter the code line number at which the call occurs, and under the method's parameter headings, enter the passed in parameter values. When printing output (as on lines 20 and 21), enter the code line number and the output's literal and variable values in the output column.

| | | | | | | |
|---|---|---|---|---|---|---|
| **input** | | | | | | |
| ksebelius | | | | | | |
| wind | | | | | | |
| windpwr2 | | | | | | |
| | | | | | | |
| UsernamePasswordEntryTrace | | | | | | |
| | main | | validEntry | | | |
| ***line#*** | **username** | **password** | **minLength** | **entryType** | **entry** | **output** |
| *15* | ? | | | | | |
| *16* | | ? | | | | |
| *18* | | | 4 | username | | |
| *31* | | | | | ? | |
| *36* | | | | | | Enter a username with at least 4 characters: |
| *39* | | | | | ksebelius | |
| *18* | ksebelius | | | | | |
| *19* | | | 6 | password | | |
| *31* | | | | | ? | |
| *36* | | | | | | Enter a password with at least 6 characters: |
| *39* | | | | | wind | |
| *42* | | | | | | Sorry – too short. |
| *36* | | | | | | Enter a password with at least 6 characters: |
| *39* | | | | | windpwr2 | |
| *19* | | windpwr2 | | | | |
| *20* | | | | | | Your username is "ksebelius" |
| *21* | | | | | | Your password is "windpwr2" |

**Figure S6.8** Completed trace of UsernamePasswordEntry program

When making an assignment, enter the code line number, and for the variable to which the assignment is made, enter the assigned value. Similarly, when reading input (as on line 39) enter the code line number, and for the variable getting the input, copy the value from the next item in the input list at the upper left hand corner of the trace table. (To keep track of where you are in the input list, you might want to put a check mark after each input item as you use it.)

Except at the start of a `for` loop (where you need to initialize and update the `for` loop index variable), don't make entries at branch points. Just look at the data already in the trace table to decide where to go next. For example, at the `if` condition on code line 40, ask yourself, "Is the length of the string in the `entry` variable less than the value stored in `minLength`?" If yes, go to code line 42. If no, go to code line 44.

To indicate a `return` from a method, go to a new row on the trace table and draw a heavy horizontal line under all the parameters and local variables in that method. This heavy underline says these variables are wiped out after the `return`. On that same row in the trace table, enter the code line number of the point in the calling method to which control returns.[2] In our UserPasswordEntryTrace program, that's

[2] If calling and called methods are in separate classes, each class will probably have its own *line #* column. Then if the called method returned from an explicit `return` statement, you would also enter the line number of this `return` statement in the called method's class's *line #* column. This can help you untangle poorly structured returns.

code line 18 after the first `validEntry` call and code line 19 after the second `validEntry` call. If there is a return value (i.e., the called method is not a `void` method) on this same row in the trace table, enter the return value under the variable to which the return value is assigned. In our example, after the first `validEntry` call, on code line 18 the returned `ksebelius` value is assigned to `username`. After the second `validEntry` call, on code line 19 the returned `windpwr2` value is assigned to `password`.

Experience with this long-form tracing technique will make it easier for you to understand what an automated debugger in an *Integrated Development Environment* (IDE) is telling you. As you step through a program that's running in debug mode under the control of an IDE debugger, when you get to a method call, you have two choices. You can "step into" and go through all the statements in the called method, like we do in Figure S6.8, or you can "step over" and just see what happens after the method returns. If our trace table had stepped over both `validEntry` method calls, it would have had a total of only eight rows, for code lines 15, 16, 18, 18, 19, 19, 20, and 21. In a typical debugging activity, you will use a combination of stepping over and stepping into.

## S6.9 Overloaded Methods

Up until this point, all of the methods we defined for a given class have had unique names. But if you think back to some of the Java API methods presented in Chapter 5, you'll recall that there were several examples where the same name (`abs`, `max`, `min`) was used to identify more than one method in the same class (the `Math` class). This section will show you how to do this in classes you write.

### What Are Overloaded Methods?

*Overloaded methods* are two or more methods in the same class that use the same name. Since they use the same name, the compiler needs something else besides the name to distinguish them. Parameters to the rescue! To make two overloaded methods distinguishable, you define them with different parameters. More specifically, you define them with a different number of parameters or different types of parameters. The combination of a method's name, the number of its parameters, and the types of its parameters is called the method's *signature*. Each distinct method has a distinct signature. Could these three lines be used as headings for three overloaded `findMaximum` methods?

```
int findMaximum(int a, int b, int c)
double findMaximum(double a, double b, double c)
double findMaximum(double a, double b, double c, double d)
```

Yes, they are a legal overloading of the `findMaximum` method name, because each heading is distinguishable in terms of number and types of parameters. How about the next two lines – could a `findAverage` method name be overloaded in this way?

```
int findAverage(int a, int b, int c)
double findAverage(int x, int y, int z)
```

No. These are not distinguishable methods, because they have the same signature – same method names and same number and types of parameters. Since these two methods are not distinguishable, if you try to include these two method headings in the same class, the compiler will think you're defining the same method twice. And that will make the compiler irritable. Be prepared for it to snarl back at you with a “duplicate definition” compile-time error message.

Note that the above `findAverage` method headings have different return types. You might think that the different return types indicate different signatures. Not true. The return type is not part of the signature, so you cannot use just a different return type to distinguish overloaded methods.

## Benefit of Overloaded Methods

When should you use overloaded methods? When there's a need to perform essentially the same task with different parameters. For example, the methods associated with the above `findMaximum` headings perform essentially the same basic task – they calculate the maximum value from a given list of numbers. But they perform the task on different sets of parameters. Given that situation, overloaded methods are a perfect fit.

Note that using overloaded methods is never an absolute requirement. As an alternative, you can always use different method names to distinguish different methods. So why are the above `findMaximum` method headings better than the below method headings?

```
int findMaximumOf3Ints(int a, int b, int c)
double findMaximumOf3Doubles(double a, double b, double c)
double findMaximumOf4Doubles(double a, double b, double c, double d)
```

As these examples suggest, using different method names is cumbersome. With only one method name, the name can be simple. As a programmer, wouldn't you prefer to use and remember just one simple name rather than several cumbersome names?

## Example

Suppose you want methods that compute the distance between two points in space, where the space might have either one dimension (be along a line) or two dimensions (be on a plane). In the future, you might want to add similar methods to determine distance between two points in three dimensions (in a volume) or perhaps even more than three dimensions (in “hyperspace”). In principal, you can get what you need immediately and also be ready for the future by settling on one name for all possibilities, `findDistance`. Then let the dimensionality establish the number of parameters.

The one dimensional method could have a heading like this:

```
public static double findDistance(double x1, double x2)
```

In this one-dimensional method, `x1` gives the position of the first point, and `x2` gives the position of the second point.

The two dimensional method could have a heading like this:

```
public static double findDistance(
    double x1, double y1, double x2, double y2)
```

In this two-dimensional method, `x1` and `y1` give the position of the first point, and `x2` and `y2` give the position of the second point.

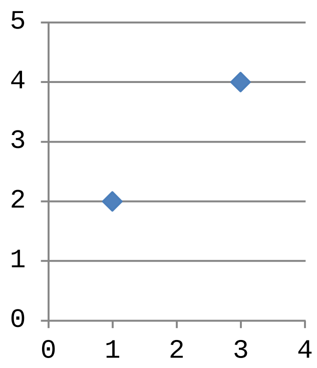
These two `findDistance` methods perform essentially the same task – they determine the distance between two points. And they do it with only a slight variation – the dimensionality of the space in which the points exist. That's why we use the same name and “overload” that name.

Here's an example of what a program running these overloaded methods might do:

Sample Session:

```
Enter x position of point #1: 1.0
Enter y position of point #1: 2.0
Enter x position of point #2: 3.0
Enter y position of point #2: 4.0
1-D horizontal separation = 2.0
1-D vertical separation = 2.0
2-D distance = 2.8284271247461903
```

Here's a picture showing this example's location of the two points in the x-y plane (x is horizontal and y is vertical):



With equal horizontal and vertical separations, the straight-line distance should equal the separation in either direction times $\sqrt{2}$ = 1.414…; and since 2.0 x 1.414… = 2.828…, it does.

Figures S6.9a and S6.9b contain a program that defines and uses the desired overloaded methods to generate the above results. Figure S6.9a's `main` method makes two calls to the two-parameter version of `findDistance` to compute the separations in the horizontal (x) and vertical(y) directions. Then it calls the four-parameter version to determine the direct distance between the two points. Figure S6.9b presents straightforward implementations of the overloaded `findDistance` methods.

```
/*************************************************************
* Distance.java
* Dean & Dean
*
* These methods compute distance between two points.
*************************************************************/

import java.util.Scanner;

public class Distance
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    double x1;     // x position of point #1
    double y1;     // y position of point #1
    double x2;     // x position of point #2
    double y2;     // y position of point #2
    double dx, dy; // distances in x and y directions

    System.out.print("Enter x position of point #1: ");
    x1 = stdIn.nextDouble();
    System.out.print("Enter y position of point #1: ");
    y1 = stdIn.nextDouble();
    System.out.print("Enter x position of point #2: ");
    x2 = stdIn.nextDouble();
    System.out.print("Enter y position of point #2: ");
    y2 = stdIn.nextDouble();
    System.out.println(
      "1-D horizontal separation = " + findDistance(x1, x2));
    System.out.println(
      "1-D vertical separation = " + findDistance(y1, y2));
    System.out.println(
      "2-D distance = " + findDistance(x1, y1, x2, y2));
  } // end main
```

**Figure S6.9a** Distance program – part A:  main method

```
  //*****************************************************

  // This method computes distance in one dimension

  public static double findDistance(double x1, double x2)
  {
    return Math.abs(x2 - x1);
  } // end findDistance in 1-D

  //*****************************************************

  // This method computes distance in two dimensions

  public static double findDistance(
    double x1, double y1, double x2, double y2)
  {
    double dx = x2 - x1;
    double dy = y2 - y1;
    return Math.sqrt(dx * dx + dy * dy);
  } // end findDistance in 2-D
} // end class Distance
```

**Figure S6.9b** Distance program – part B: overloaded `findDistance` methods

## Calling an Overloaded Method from Within Another Overloaded Method

Look at the two implementations of the overloaded `findDistance` methods in Figure S6.9b. In the two-dimensional (4-parameter) version, the local variables, `dx` and `dy`, are initialized with a mathematical operation that is similar to the mathematical operation used to implement the one-dimensional (2-parameter) version of the method. This suggests calling the 2-parameter method inside the 4-parameter method, like this:

```
  public static double findDistance(
    double x1, double y1, double x2, double y2)
  {
    double dx = findDistance(x1, x2);
    double dy = findDistance(y1, y2);
    return Math.sqrt(dx * dx + dy * dy);
  } // end findDistance in 2-D
```

This variation of the 4-parameter `findDistance` method produces exactly the same results as the 4-parameter `findDistance` method in Figure S6.9b. The latter version shows that an overloaded method can call another overloaded method. Additional method calls make the latter implementation slightly less efficient, but one might consider it more elegant because it eliminates code redundancy.

## Program Evolution

The ability to overload a method name promotes graceful program evolution, because it corresponds to how natural language regularly overloads the meanings of words. For example, the first version of a program might define a one-parameter version of some particular method.  Later, when you decide to enhance your program, it's easier for your existing users if you minimize the new things they have to learn. You let them either keep using the original method or switch to the improved method, which has an additional parameter that enables selection among several options. When they want to use the improved method, all they have to remember is the original method name with another parameter that provides options. That's an almost obvious variation, and it's easier to remember than a different method name. It's certainly easier than being forced to learn a new method name for the old task – which would be a necessary cost of upgrading if method overloading were not available.

## S6.10  Helper Methods

As problems get bigger, it becomes more and more necessary to partition them into sub-problems, each of which has a manageable size. We started doing this in Chapter 5 when our `main` method called on some of Java's API methods for help. In this chapter, we've been writing programs with multiple methods, where the `main` method calls one or more subordinate methods. In a broad sense, any method that is called by another method is a "helper method" because the called method helps the calling method. But most programmers prefer a narrower definition for a helper method. We'll now explain the narrower definition….

Up to this point, all methods you've used have been `public`; that is, they've been defined with the `public` access modifier. These `public` methods are part of the class's *interface*, because they handle the communication between the class and the outside world. Sometimes, you'll want to create a method that is not part of the interface. Instead it just supports the operation of other non-`main` methods within its own class. Those types of methods are commonly referred to as *helper methods*, and they can be recognized by their use of the `private` access modifier.

To further your understanding of helper methods, let's use them to write an improved version of the `validEntry` method. Glance back at Figure S6.1b, and you'll see the somewhat cumbersome code that comprises the original `validEntry` method. Now take a look at the new and improved `validEntry` method in Figure S6.10a.

```
/*****************************************************************
* ValidatePasswordEntry.java
* Dean & Dean
*
* This program prompts the user to enter a valid password.
*****************************************************************/

import java.util.Scanner;

public class ValidatePasswordEntry
{
  public static void main(String args[])
  {
    String password;

    System.out.println("Your password must be alphanumeric" +
      " with at least 6 characters and\nat least 1 digit.");
    password = validEntry(6);
    System.out.println("\nYour password is \"" + password + "\"");
  } // end main

  //**************************************************************

  // This method repeatedly prompts the user to enter a username
  // until the entry is valid.

  public static String validEntry(int minLength)
  {
    Scanner stdIn = new Scanner(System.in);
    String entry;    // user's entered password
    boolean valid;   // Is user entry valid?

    do
    {
      valid = false;
      System.out.print("Password: ");
      entry = stdIn.nextLine();

      if ((entry.length() >= minLength) &&
          alphanumeric(entry) && atLeastOneDigit(entry))
      {
        valid = true;
      }
      else
      {
        System.out.println("Sorry - invalid entry.");
      }
    } while (!valid);

    return entry;
  } // end validEntry
```

calls to helper methods

**Figure S6.10a** ValidatePasswordEntry program – part A

In the interest of keeping things simple, the new `validEntry` method handles just passwords (and not usernames), so there's no need for the original method's `entryType` parameter. But other than that minor difference, the new `validEntry` method is functionally equivalent to the original. The new `validEntry` method is quite a bit shorter than the original, so how can it be functionally equivalent? The key is moving some of `validEntry`'s coding chores to helper methods. The original `validEntry` method included a loop that checked whether the given entry consisted of all alphanumeric characters and whether the given entry contained at least one digit. In the new `validEntry` method, those two tasks are implemented with calls to helper methods. Specifically, note the `alphanumeric(entry)` and `atLeastOneDigit(entry)` method calls in `validEntry`'s `if` statement, copied here for your convenience:

```
if ((entry.length() >= minLength) &&
    alphanumeric(entry) && atLeastOneDigit(entry))
```

That code is fairly easy to understand, and it improves the overall readability of the `validEntry` method.

Now take a look at the helper methods themselves in Figure S6.10b. There's no special syntax for a helper method other than the `private` access modifier in a helper method's heading. The `alphanumeric` method receives a string parameter named `s`, loops through each character in `s`, and returns `false` if and when it finds a character that is not a letter or a digit. Otherwise it returns `true`. The `atLeastOneDigit` method receives a string parameter named `s`, loops through each character in `s`, and returns `true` if and when it finds a character that is a digit. Otherwise it returns `false`. The new `validEntry` method is less efficient than the original `validEntry` method because of the two method calls and the two loops (the original `validEnty` method required just one loop). However, this small, unnoticeable decrease in efficiency is offset by the improvement in readability.

```
  //***************************************************************

  // Return true if the given string is all letters and digits.

  private static boolean alphanumeric(String s)
  {
    for (int i=0; i<s.length(); i++)
    {
      if (!Character.isLetterOrDigit(s.charAt(i)))
      {
        return false;
      }
    } // end for

    return true;
  } // end alphanumeric

  //***************************************************************

  // Return true if the given string has at least 1 digit.

  private static boolean atLeastOneDigit(String s)
  {
    for (int i=0; i<s.length(); i++)
    {
      if (Character.isDigit(s.charAt(i)))
      {
        return true;
      }
    } // end for

    return false;
  } // end atLeastOneDigit
} // end class ValidatePasswordEntry
```

`private` access modifer

**Figure S6.10b** ValidatePasswordEntry program – part B