CHAPTER 2

Console Input and Output

2 Console Input and Output

Don't imagine you know what a computer terminal is. A computer terminal is not some clunky old television with a typewriter in front of it. It is an interface where the mind and the body can connect with the universe and move bits of it about.

Douglas Adams, Mostly Harmless
(*the fifth volume in* The Hitchhiker's Trilogy)

console I/O

This chapter covers simple output to the screen and input from the keyboard, often called **console I/O**. Like most programming languages, Java has facilities for simple console output. In fact, we have already been using it. However, unlike other languages, Java has no simple standard console input. We will present three ways to cope with this deficiency in Java. Two are very standard and will be used throughout this book. The third way requires that you add a class to Java (written in Java of course). This third way is optional and will not be used in this book after it is explained, but many programmers find it (or something like it) to be the easiest and the cleanest way to handle keyboard input.

2.1 Screen Output

Let me tell the world.

William Shakespeare, *King Henry IV*

In this section we review `System.out.println` and present some material on formatting numeric output. As part of that material we give a brief introduction to *packages* and *import statements*. Packages are Java libraries of classes. Import statements make classes from a package available to your program.

`System.out.println`

System.out.println

We have already been using `System.out.println` for screen output. In Display 1.7, we used statements like the following to send output to the display screen:

```
System.out.println("The changed string is:");
System.out.println(sentence);
```

`System.out` is an object that is part of the Java language and `println` is a method invoked by that object. It may seem strange to spell an object name with a dot in it, but that need not concern us for now.

When you use `System.out.println` for output, the data to be output is given as an argument in parentheses, and the statement ends with a semicolon. The things you can output are strings of text in double quotes, like `"The changed string is:"`, `String` variables like `sentence`, variables of other types such as variables of type `int`, numbers like `5` or `7.3`, and almost any other object or value. If you want to output more than one thing, simply place an addition sign between the things you want to output. For example:

```
System.out.println("Answer is = " + 42
                        + " Accuracy is = " + precision);
```

If the value of `precision` is `0.01`, the output will be

```
Answer is = 42 Accuracy is = 0.01
```

Notice the space at the start of `" Accuracy is = "`. No space is added automatically.

The `+` operator used here is the concatenation operator that we discussed earlier. So, the above output statement converts the number `42` to the string `"42"`, converts the number `0.01` to the string `"0.01"`, and then forms the following string using concatenation:

```
"Answer is = 42 Accuracy is = 0.01"
```

`System.out.println` then outputs this longer string.

Every invocation of `println` ends a line of output. For example, consider the following statements:

```
System.out.println("A wet bird");
System.out.println("never flies at night.");
```

These two statements will cause the following output to appear on the screen:

```
A wet bird
never flies at night.
```

`println` Output

You can output one line to the screen using `System.out.println`. The items output can be quoted strings, variables, numbers, or almost any object you can define in Java. To output more than one item, place a plus sign between the items.

Syntax:

```
System.out.println(Item_1 + Item_2 + ... + Last_Item);
```

Example:

```
System.out.println("Welcome to Java.");
System.out.println("Elapsed time = " + time + " seconds");
```

print versus println

If you want the output from two or more output statements to place all their output on a single line, then use `print` instead of `println`. For example,

```
System.out.print("A ");
System.out.print("wet ");
System.out.println("bird");
System.out.println("never flies at night.");
```

will produce the same output as our previous example:

```
A wet bird
never flies at night.
```

Notice that a new line is not started until you use a `println`, rather than a `print`. Also notice that the new line starts *after* outputting the items specified in the `println`. This is the only difference between `print` and `println`.

println VERSUS print

The only difference between `System.out.println` and `System.out.print` is that with `println`, the *next* output goes on a *new line*, whereas with `print`, the next output will be placed on the *same line*.

EXAMPLE:

```
System.out.print("Tom ");
System.out.print("Dick ");
System.out.println("and ");
System.out.print("Harry ");
```

will produce the following output

```
Tom Dick and
Harry
```

(The output would look the same whether the last line read `print` or `println`.)

Another way to describe the difference between `print` and `println` is to note that

```
System.out.println(SomeThing);
```

is equivalent to

```
System.out.print(SomeThing + "\n");
```

Self-Test Exercises

1. Write Java statements that will cause the following to be written to the screen:

```
May the hair on your toes
grow long and curly.
```

2. What is the difference between `System.out.println` and `System.out.print`?
3. What is the output produced by the following?

```
System.out.println(2 + " " + 2);
System.out.println(2 + 2);
```

MONEY FORMATS

Using the class `NumberFormat`, you can tell Java to use the appropriate format when outputting amounts of money. The technique is illustrated in Display 2.1. Let's look at the code in the `main` method that does the formatting. First consider

```
NumberFormat moneyFormater =
                  NumberFormat.getCurrencyInstance();
```

The method invocation `NumberFormat.getCurrencyInstance()` produces an object of the class `NumberFormat` and names the object `moneyFormater`. You can use any valid identifier (other than a keyword) in place of `moneyFormater`. This object `moneyFormater` has a method named `format` that takes a floating-point number as an argument and returns a `String` value representing that number in the local currency (the default currency). For example, the invocation

```
moneyFormater.format(19.8)
```

returns the `String` value `"$19.80"`, assuming the default currency is the U.S. dollar. In Display 2.1 this method invocation occurs inside a `System.out.println` statement, but it is legal anyplace a `String` value is legal. For example, the following would be legal:

```
String moneyString = moneyFormater.format(19.8);
```

In order to make the class `NumberFormat` available to your code, you must include the following near the start of the file with your program:

```
import java.text.NumberFormat;
```

This is illustrated in Display 2.1.

The method invocation `NumberFormat.getCurrencyInstance()` produces an object that will format numbers according to the default location. In Display 2.1 we are assuming the default location is the United States and so the numbers are output as

Display 2.1 Currency Format *(Part 1 of 2)*


```
1  import java.text.NumberFormat;
2  import java.util.Locale;

3  public class CurrencyFormatDemo
4  {
5      public static void main(String[] args)
6      {
7          System.out.println("Without formatting:");

8          System.out.println(19.8);
9          System.out.println(19.81111);
10         System.out.println(19.89999);
11         System.out.println(19);
12         System.out.println();

13         System.out.println("Default location:");
14         NumberFormat moneyFormater =
15                           NumberFormat.getCurrencyInstance();

16         System.out.println(moneyFormater.format(19.8));
17         System.out.println(moneyFormater.format(19.81111));
18         System.out.println(moneyFormater.format(19.89999));
19         System.out.println(moneyFormater.format(19));
20         System.out.println();

21         System.out.println("US as location:");
22         NumberFormat moneyFormater2 =
23                           NumberFormat.getCurrencyInstance(Locale.US);

24         System.out.println(moneyFormater2.format(19.8));
25         System.out.println(moneyFormater2.format(19.81111));
26         System.out.println(moneyFormater2.format(19.89999));
27         System.out.println(moneyFormater2.format(19));
28     }

29 }
```

If you use only the default location, you do not need to import `Locale`.

Notice that this number is rounded to 19.90.

Display 2.1 Currency Format *(Part 2 of 2)*

SAMPLE DIALOGUE

```
Without formatting:
19.8
19.81111
19.89999
19

Default location:
$19.80
$19.81
$19.90
$19.00

US as location:
$19.80
$19.81
$19.90
$19.00
```

This assumes that the system is set to use U.S. as the default location. If you are not in the U.S., you will probably get the format for your local currency.

This should give you the format for U.S. currency no matter what country has been set as the default location.

U.S. dollars. On other systems, the default should be set to the local currency. If you wish, you can specify the location, and hence the local currency, by giving an argument to `NumberFormat.getCurrencyInstance`. For example, in Display 2.1 we used the constant `Locale.US` to specify that the location is the United States. The relevant line from Display 2.1 is repeated in what follows:

```
NumberFormat moneyFormater2 =
                    NumberFormat.getCurrencyInstance(Locale.US);
```

Some constants for other countries (and hence other currencies) are given in Display 2.2. However, unless your screen has the capability of displaying the currency symbol for the country whose constant you use, the output may not be as desired.

These location constants are objects of the class `Locale`. In order to make the class `Locale` and these constants available to your code, you must include the following near the start of the file with your program:

```
import java.util.Locale;
```

If you do not use any of these location constants and only use the default location, you do not need this `import` statement.

Display 2.2 Locale Constants for Currencies of Different Countries

`Locale.CANADA`	Canada (for currency the format is the same as US.)
`Locale.CHINA`	China
`Locale.FRANCE`	France
`Locale.GERMANY`	Germany
`Locale.ITALY`	Italy
`Locale.JAPAN`	Japan
`Locale.KOREA`	Korea
`Locale.TAIWAN`	Taiwan
`Locale.UK`	United Kingdom (English pound)
`Locale.US`	United States

The notation `Locale.US` may seem a bit strange, but it follows a convention that is frequently used in Java code: The constant is named `US` but we specifically want that constant named `US` that is defined in the class `Locale`. So, we use `Locale.US`. The notation `Locale.US` means the constant `US` as defined in the class `Locale`.

IMPORTING PACKAGES AND CLASSES

package

Libraries in Java are called **packages**. A package is simply a collection of classes that has been given a name and that is stored in such a way as to make it easily accessible to your Java programs. Java has a large number of standard packages that automatically come with Java. Two such packages are named `java.text` and `java.util`. In Display 2.1 we used the class `NumberFormat`, which is a member of the package `java.text`. In order to use `NumberFormat`, you must **import** the class, which we did as follows:

import

java.text

```
import java.text.NumberFormat;
```

import statement

This kind of statement is called an **import statement**. In this example, the `import` statement tells Java to look in the package `java.text` to find the definition of the class `NumberFormat`.

If you wanted to import all the classes in the `java.text` package, you would use

```
import java.text .*;
```

Then, you can use any class in the `java.text` package.

There is no loss of efficiency in importing the entire package instead of importing only the classes you use. However, many programmers find that it is an aid to documentation if they import only the classes they use, which is what we will do in this book.

Outputting Amounts of Money

To output amounts of money correctly formatted, proceed as follows:

Place the following near the start of the file containing your program:

```
import java.text.NumberFormat;
```

In your program code, create an object of the class `NumberFormat` as follows:

```
NumberFormat formaterObject =
                NumberFormat.getCurrencyInstance();
```

When outputting numbers for amounts of money, change the number to a value of type `String` using the method `formaterObject.format`, as illustrated in what follows:

```
double moneyAmount = 9.99;
System.out.println(formaterObject.format(moneyAmount));
```

The string produced by invocations such as `formaterObject.format(moneyAmount)` will add the dollar sign and ensure that there are exactly two digits after the decimal point. (This is assuming the U.S. dollar is the default currency.)

The numbers formatted in this way may be of type `double`, `int`, or `long`. You may use any (non-keyword) identifier in place of `formaterObject`. A complete example is given in Display 2.1.

The above always outputs the money amount in the default currency, which is typically the local currency. You can specify the country whose currency you want. See the text for details.

In Display 2.1 we also used the class `Locale`, which is in the `java.util` package. So, we also included the following `import` statement: `java.util`

```
import java.util.Locale;
```

There is one package that requires no `import` statement. The package `java.lang` contains classes that are fundamental to Java programming. These classes are so basic that the package is always imported automatically. Any class in `java.lang` does not need an `import` statement to make it available to your code. So, when we say that a class is in the package `java.lang`, you can simply use that class in your program without needing any `import` statement. For example, the class `String` is in the `java.lang` package and so you can use it without any `import` statement. `java.lang`

More material on packages will be given in Chapter 5.

Self-Test Exercises

4. What output is produced by the following code? (Assume a proper `import` statement has been given.)

```
NumberFormat exerciseFormater =
                    NumberFormat.getCurrencyInstance(Locale.US);
double d1 = 1.2345, d2 = 15.67890;
System.out.println(exerciseFormater.format(d1));
System.out.println(exerciseFormater.format(d2));
```

5. Suppose the class `Robot` is a part of the standard Java libraries and is in the package named `java.awt`. What `import` statement do you need to make the class `Robot` available to your program or other class?

THE `DecimalFormat` CLASS

`System.out.println` will let you output numbers but has no facilities to format the numbers. If you want to output a number in a specific format, such as having a specified number of digits after the decimal point, then you must convert the number to a string that shows the number in the desired format and then use `System.out.println` to output the string. We have seen one way to accomplish this for amounts of money. The class `DecimalFormat` provides a very versatile facility to format numbers in a variety of ways.

import

The class `DecimalFormat` is in the Java package named `java.text`. So, you must add the following (or something similar) to the beginning of the file with your program or other class that uses the class `DecimalFormat`:

```
import java.text.DecimalFormat;
```

An object of the class `DecimalFormat` has a number of different methods that can be used to produce numeral strings in various formats. In this subsection we discuss one of these methods, which is named `format`. The general approach to using the `format` method is as follows:

patterns

Create an object of the class `DecimalFormat`, using a `String` *Pattern* as follows:

```
DecimalFormat Variable_Name = new DecimalFormat(Pattern);
```

For example,

```
DecimalFormat formattingObject = new DecimalFormat("000.000");
```

Display 2.3 The DecimalFormat Class *(Part 1 of 2)*

```
1  import java.text.DecimalFormat;

2  public class DecimalFormatDemo
3  {
4      public static void main(String[] args)
5      {
6          DecimalFormat pattern00dot000 = new DecimalFormat("00.000");
7          DecimalFormat pattern0dot00 = new DecimalFormat("0.00");

8          double d = 12.3456789;
9          System.out.println("Pattern 00.000");
10         System.out.println(pattern00dot000.format(d));
11         System.out.println("Pattern 0.00");
12         System.out.println(pattern0dot00.format(d));

13         double money = 19.8;
14         System.out.println("Pattern 0.00");
15         System.out.println("$" + pattern0dot00.format(money));
16
17         DecimalFormat percent = new DecimalFormat("0.00%");

18         System.out.println("Pattern 0.00%");
19         System.out.println(percent.format(0.308));

20         DecimalFormat eNotation1 =
21            new DecimalFormat("#0.###E0");//1 or 2 digits before point
22         DecimalFormat eNotation2 =
23            new DecimalFormat("00.###E0");//2 digits before point

24         System.out.println("Pattern #0.###E0");
25         System.out.println(eNotation1.format(123.456));
26         System.out.println("Pattern 00.###E0");
27         System.out.println(eNotation2.format(123.456));

28         double smallNumber = 0.0000123456;
29         System.out.println("Pattern #0.###E0");
30         System.out.println(eNotation1.format(smallNumber));
31         System.out.println("Pattern 00.###E0");
32         System.out.println(eNotation2.format(smallNumber));
33     }
34 }
```

Display 2.3 The `DecimalFormat` Class *(Part 2 of 2)*

SAMPLE DIALOGUE

```
Pattern 00.000
12.346
Pattern 0.00
12.35
Pattern 0.00
$19.80
Pattern 0.00%
30.80%
Pattern #0.###E0
1.2346E2
Pattern 00.###E0
12.346E1
Pattern #0.###E0
12.346E-6
Pattern 00.###E0
12.346E-6
```

The number is always given, even if this requires violating the format pattern.

The method `format` of the class `DecimalFormat` can then be used to convert a floating-point number, such as one of type `double`, to a corresponding numeral `String` following the *Pattern* used to create the `DecimalFormat` object. Specifically, an invocation of `format` takes the form

Decimal_Format_Object`.format(`*Double_Expression*`)`

which returns a `String` value for a string representation of the value of *Double_Expression*. *Double_Expression* can be any expression, such as a variable or sum of variables, that evaluates to a value of type `double`.

For example, consider the following code:

```
DecimalFormat formattingObject = new DecimalFormat("000.0000");
String numeral = formattingObject.format(12.3456789);
System.out.println(numeral);
```

This produces the output

```
012.3457
```

Of course, you can use an invocation of `format`, such as `formattingObject.format(12.3456789)`, directly in `System.out.println`. So, the following produces the same output:

```
System.out.println(formattingObject.format(12.3456789));
```

The format of the string produced is determined by the *Pattern* string that was used as the argument to the constructor that created the object of the class `DecimalFormat`. For example, the pattern `"000.0000"` means that there will be three digits before the decimal point and four digits after the decimal point. Note that the result is rounded when the number of digits is less than the number of digits available in the number being formatted. If the format pattern is not consistent with the value of the number, such as a pattern that asks for two digits before the decimal point for a number like `123.456`, then the format rules will be violated so that no digits are lost.

A pattern can specify the exact number of digits before and after the decimal or it can specify minimum numbers of digits. The character `'0'` is used to represent a required digit and the character `'#'` is used to indicate an optional digit. For example, the pattern `"#0.0##"` indicates one or two digits before the decimal point and one, two, or three digits after the decimal point. The optional digit `'#'` is shown if it is a nonzero digit and is not shown if it is a zero digit. The `'#'` optional digits should go where zero placeholders would appear in a numeral string; in other words, any `'#'` optional digits precede the zero digits `'0'` before the decimal point in the pattern, and any `'#'` optional digits follow the zero digits `'0'` after the decimal point in the pattern. Use `"#0.0##"`; do not use `"0#.0##"` or `"#0.##0"`.

For example, consider the following code:

```
DecimalFormat formattingObject = new DecimalFormat("#0.0##");
System.out.println(formattingObject.format(12.3456789));
System.out.println(formattingObject.format(1.23456789));
```

This produces the output

```
12.346
1.235
```

percentages

The character `'%'` placed at the end of a pattern indicates that the number is to be expressed as a percentage. The `'%'` causes the number to be multiplied by 100 and appends a percent sign, `'%'`. Examples of this and other formatting patterns are given in Display 2.3.

E-notation

E-notation is specified by including an `'E'` in the pattern string. For example, the pattern `"00.###E0"` approximates specifying two digits before the decimal point, three or fewer digits after the decimal point, and at least one digit after the `'E'`, as in `12.346E1`. As you can see by the examples of E-notation in Display 2.3, the exact details of what E-notation string is produced can be a bit more involved than our explanation so far. Here are a couple more details:

The number of digits indicated after the `'E'` is the minimum number of digits used for the exponent. As many more digits as are needed will be used.

The **mantissa** is the decimal number before the `'E'`. The minimum number of significant digits in the mantissa (that is, the sum of the number of digits before and after the decimal point) is the *minimum* of the number of digits indicated before the

decimal point plus the *maximum* of the number of digits indicated after the decimal point. For example, `12345` formatted with `"##0.##E0"` is `"12.3E3"`.

To get a feel for how E-notation patterns work, it would pay to play with a few cases, and in any event, do not count on a very precisely specified number of significant digits.

DecimalFormat Class

Objects of the class `DecimalFormat` are used to produce strings of a specified format from numbers. As such, these objects can be used to format numeric output. The object is associated with a pattern when it is created using `new`. The object can then be used with the method `format` to create strings that satisfy the format. See Display 2.3 for examples of using the `DecimalFormat` class.

2.2 JOptionPane

As I sit looking out of a window of the building
I wish I did not have to write the instructions manual...

John Ashbery, *The Instruction Manual*

Chapters 16–18 describe the most common ways of creating windowing interfaces in Java. The class `JOptionPane` can be understood and used without needing the background given in those chapters. Thus, `JOptionPane` provides you with a way to get started doing windowing I/O immediately. This section covers enough about `JOptionPane` to get you started using it.

THE BASICS

Display 2.4 contains a demonstration Java program that uses `JOptionPane`. Following the program, we show the three windows produced by the program. The three windows are produced one at a time. The user enters a number in the text field of the first window and then clicks the `OK` button with the mouse. When the user clicks the `OK` button, the first window goes away and the second window appears. The user handles the second window in a similar way. When the user clicks the `OK` button in the second window, the second window goes away and the third window appears. When the user clicks the `OK` button in the third window, the third window goes away and that is the end of this sample program.

As an alternative to clicking the `OK` button, the user can press the Enter key, which has the exact same effect as clicking the `OK` button.

Display 2.4 **A Program Using JOptionPane** ***(Part 1 of 2)***

```
1  import javax.swing.JOptionPane;

2  public class JOptionPaneDemo
3  {
4     public static void main(String[] args)
5     {
6        String podString =
7           JOptionPane.showInputDialog("Enter number of pods:");
8        int numberOfPods = Integer.parseInt(podString);

9        String peaString =
10          JOptionPane.showInputDialog(
11                               "Enter number of peas in a pod:");
12       int peasPerPod = Integer.parseInt(peaString);

13       int totalNumberOfPeas = numberOfPods*peasPerPod;

14       JOptionPane.showMessageDialog(
15          null, "The total number of peas = " + totalNumberOfPeas);

16       System.exit(0);
17    }
18 }
```

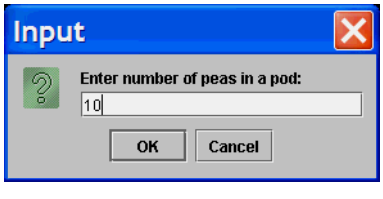
Required to tell Java where to find the code for `JOptionPane`

Needed to end the program if `JOptionPane` is used in the program

WINDOW 1

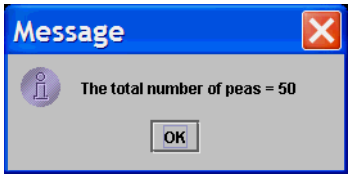
When the user clicks OK (or presses the Enter key), the window goes away and the next window (if any) is displayed.

WINDOW 2



Display 2.4 A Program Using JOptionPane *(Part 2 of 2)*

WINDOW 3



Swing

The class `JOptionPane` is in the `javax.swing` library, which is often referred to simply as the **Swing** library. You must have the following line (or something similar) at the beginning of the file with your program (or other class) that uses `JOptionPane`:

```
import javax.swing.JOptionPane;
```

showInput-Dialog

The following two lines from Display 2.4 are an invocation of the method `showInputDialog` of the class `JOptionPane`:

```
String podString =
   JOptionPane.showInputDialog("Enter number of pods:");
```

This produces a window for obtaining input. The string argument, in this case `"Enter number of pods:"`, is a prompt written in the window. This invocation of the method `showInputDialog` will produce the first window shown in Display 2.4. The user clicks her or his mouse in the text field and then types in some input. If the user does not like what she/he typed in, the user can use the Backspace key to back up and retype the input. Once the user is happy with the input in the window, the user clicks the `OK` button and the window goes away.

The method invocation

```
JOptionPane.showInputDialog("Enter number of pods:")
```

returns the input that the user typed into the text field. So, the following line from our program is an assignment statement that sets the variable `podString` so that it stores the input string that the user typed into the text field of the first window:

```
String podString =
   JOptionPane.showInputDialog("Enter number of pods:");
```

The input produced by `JOptionPane.showInputDialog` is always a string value. If you want numeric input, your program must convert the string input into a number

(or numbers). The program converts the string stored in the variable `podString` to an `int` and stores the resulting `int` value in the variable `numberOfPods` as follows:

```
int numberOfPods = Integer.parseInt(podString);
```

Integer.parseInt

The expression `Integer.parseInt` is explained in Chapter 5 in the subsection entitled "Wrapper Classes," but until you cover that material, you can simply note that this expression converts a number string to the corresponding value of type `int`.

If you are converting from a string to a number value of some type other than `int`, you use the appropriate method from Display 2.5 instead of `Integer.parseInt`.

The second widow is similar to the first one except that it prompts for a different input.

The following line from the program in Display 2.4 shows the program output, which is displayed as the third window in that display:

output window

```
JOptionPane.showMessageDialog(
    null, "The total number of peas = " + totalNumberOfPeas);
```

The part `showMessageDialog` is another method in the class `JOptionPane`. This method displays a window for showing some output. The method `showMessageDialog` has two arguments, which are separated with a comma. For simple programs, you use `null` as the first argument. The constant `null` is discussed in Chapter 5. You probably have not yet read Chapter 5, but that is not a problem. You can simply insert `null` in the first argument position. In this case it is just a placeholder. The second argument is the string that is written in the output window. The output window stays on the screen until the user clicks the `OK` button with the mouse (or alternatively presses the Enter key), and then the window disappears.

showMessage–Dialog

null

Note that you can give the output string to a `showMessageDialog` in the same way that you give an output string as an argument to `System.out.println`, that is, as a quoted string, as a variable, as a constant, or as any combination of these connected with plus signs.

The last program statement, shown in what follows, simply says that the program should end:

```
System.exit(0);
```

System.exit

`System` is a predefined Java class that is automatically provided by Java, and `exit` is a method in the class `System`. The method `exit` ends the program as soon as it is invoked. In the programs that we will write, the integer argument `0` can be any integer, but by tradition we use `0`, because, for most operating systems, `0` is used to indicate a normal ending of the program. When you write a program with a windowing interface, you always need to end the program with

```
System.exit(0);
```

The program will not end automatically when there are no more statements to execute.

Display 2.5 Methods for Converting Strings to Numbers

TYPE NAME	METHOD FOR CONVERTING
`byte`	`Byte.parseByte(`*String_To_Convert*`)`
`short`	`Short.parseShort(`*String_To_Convert*`)`
`int`	`Integer.parseInt(`*String_To_Convert*`)`
`long`	`Long.parseLong(`*String_To_Convert*`)`
`float`	`Float.parseFloat(`*String_To_Convert*`)`
`double`	`Double.parseDouble(`*String_To_Convert*`)`

To convert a value of type `String` to a value of the type given in the first column, use the method given in the second column. Each of the methods in the second column returns a value of the type given in the first column. The *String_To_Convert* must be a correct string representation of a value of the type given in the first column. For example, to convert to an `int`, the *String_To_Convert* must be a whole number (in the range of the type `int`) that is written in the usual way without any decimal point.

The Cancel Button in a showInputDialog Window

When you use the method `showInputDialog` of the class `JOptionPane`, the input has two buttons, one labeled OK and one labeled `Cancel`. What happens if the user clicks the `Cancel` button? Try it and you will see that, as we are using `JOptionPane`, the program ends with an error message that may not make sense to you yet. For now you can think of the `Cancel` button as a way to cancel (that is, end) the entire program. Later in this book you will learn how to program the `Cancel` button to do other things.

(If the `Cancel` button is clicked, then the method `showInputDialog` returns something called `null`, which we will not discuss until Chapter 5.)

multi-line output

If you want to output multiple lines using the method `JOptionPane.showMessageDialog`, then you can insert the new-line character `'\n'` into the string used as the second argument. If the string becomes too long, which it almost always does with multi-line output, then you can make each line into a separate string (ending with `'\n'`) and connect the lines with the plus sign. If the lines are long or there are very many lines, then the output window will be made larger so that it can hold all the output.

JOptionPane For Windowing Input/Output

You can use the methods `showInputDialog` and `showMessageDialog` of the class `JOptionPane` to produce input and output windows for your Java programs. When using these methods, you must include one of the following two `import` statements at the start of the file that contains your program:

```
import javax.swing.JOptionPane;
```

or

```
import javax.swing.*;
```

The syntax for statements to produce `JOptionPane` I/O windows is given below:

Syntax: (Input)

```
String_Variable = JOptionPane.showInputDialog(String_Expression);
```

Example:

```
String pinString = JOptionPane.showInputDialog("Enter your PIN:");
```

The *String_Expression* is a prompt line displayed in a window that also has both a text field in which the user can enter input and a button labeled OK. If the user types in a string and clicks the OK button, then the string that was typed in is returned by the method, and so the string typed in by the user is stored in the *String_Variable*. The window disappears when the user clicks the OK button. As an alternative to clicking the OK button, the user can press the Enter key, which has the exact same effect as clicking the OK button.

Note that when input is done in this way, all input is string input. If you want the user to input, for example, integers, then your program must convert the input string numeral to the equivalent number.

Syntax: (Output)

```
JOptionPane.showMessageDialog(null, String_Expression);
```

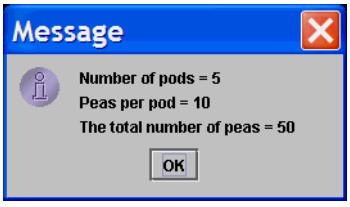
Example:

```
JOptionPane.showMessageDialog(
    null, pinString + " is not a valid PIN.");
```

The *String_Expression* is displayed in a window that has a button labeled OK. When the user clicks the OK button (or equivalently presses the Enter key), the window disappears.

One additional `JOptionPane` method that produces a window for yes/no questions is discussed in the section "Yes/No Questions with `JOptionPane` ✜."

Display 2.6 A Multi-Line Output Window



For example, the following will produce the window shown in Display 2.6, provided the value of `numberOfPods` is `5`, the value of `peasPerPod` is `10`, and the value of `totalNumberOfPeas` is `50`:

```
JOptionPane.showMessageDialog(null,
            "Number of pods = " + numberOfPods + "\n"
          + "Peas per pod = " + peasPerPod + "\n"
          + "The total number of peas = " + totalNumberOfPeas);
```

YES/NO QUESTIONS WITH `JOptionPane`

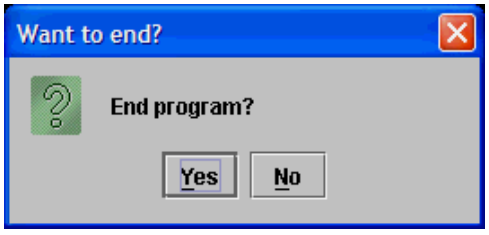
There is a version of `JOptionPane` that produces a window for asking yes/no questions of the user. The window has the question text you specify and two buttons labeled `"Yes"` and `"No"`. For example:

```
int answer =
     JOptionPane.showConfirmDialog(null,
          "End program?", "Want to end?", JOptionPane.YES_NO_OPTION);
if (answer == JOptionPane.YES_OPTION)//== tests for equality.
    System.exit(0);
else if (answer == JOptionPane.NO_OPTION)
    System.out.println("One more time");
else
    System.out.println("This is impossible");
```

This code will produce the window shown in Display 2.7. If the user clicks the `"Yes"` button, then the method invocation will return an `int` value and the window will disappear. Since the value returned was the value `JOptionPane.YES_OPTION`, the multi branch `if-else` statement will then invoke `System.exit(0)` to end the program. If the user clicks the `"No"` button, then the method invocation will return the `int` value `JOptionPane.NO_OPTION`, and then the multi branch `if-else` statement will invoke `System.out.println` to write `"One more time"` to the screen. You can see this code embedded in a complete program in the file `JOptionPaneYesNoDemo.java` on the accompanying CD.

extra code on CD

Display 2.7 A Yes/No JOptionPane Dialog Window



`JOptionPane.showConfirmDialog` returns an `int` value, but you do not want to think of it as an `int` but instead think of the returned value as the answer to a yes/no question. To facilitate this thinking the class `JOptionPane` defines names for two `int` values. The constant `JOptionPane.YES_OPTION` is the `int` returned when the `"Yes"` button is clicked. The constant `JOptionPane.NO_OPTION` is the `int` returned when the `"No"` button is clicked. Just which `int` values are named by `JOptionPane.YES_OPTION` and `JOptionPane.NO_OPTION`? It does not matter. Do not think of them as `int`s.

We have not developed enough material to allow us to fully explain the list of arguments, but we can explain most of them. Consider the argument list in our example:

```
(null, "End program?", "Want to end?", JOptionPane.YES_NO_OPTION)
```

The string `"End program?"` may be replaced by any other string and that will then be the string that appears in the windows with the `"Yes"` and `"No"` buttons. Of course, the string should normally be a yes/no question.

The string `"Want to end?"` may be replaced by any other string and that will then be the string displayed as the title of the window.

The last argument `JOptionPane.YES_NO_OPTION` indicates that you want a window with `"Yes"` and `"No"` buttons. There are other possible options but we will not discuss them here.

Why Are Some Methods Invoked with a Class Name Instead of a Calling Object?

Normally, a method invocation uses an object name as a calling object; for example, `greeting.length()`, where `greeting` is a variable of type `String`. However, when we use the methods of the class `JOptionPane`, we use the class name `JOptionPane` in place of a calling object. What is the story? Some special methods do not require a calling object and are invoked using the class name. These methods are called *static* methods and are discussed in Chapter 5. While these static methods are only a very small fraction of all methods, they are used for certain fundamental tasks, such as I/O, and so we encountered them early.

The first argument has to do with where the window is placed on the screen, but we have not developed enough material to allow you to consider the possible options. So for now, you should simply use `null`, which is a kind of default value, as the first argument.

Self-Test Exercises

6. Write some Java code that will read a line of text and then output the line with all lowercase letters changed to uppercase. Use `JOptionPane`.
7. Write some Java code that will set the value of the variable `number` equal to the number entered by a user at the keyboard. Assume that `number` is of type `int`. Use `JOptionPane`.
8. Write a complete Java program that reads two whole numbers into two variables of type `int`, and then outputs both the whole number part and the remainder when the first number is divided by the second. This can be done using the operators `/` and `%`. Use `JOptionPane`.

2.3 Console Input Using `BufferedReader`

It's a bit messy, but it works.

Walter Savitch, *Absolute Java*

You can do a kind of simple console input in Java, but it requires more material than we have covered at this stage of learning the Java language. So, we will do the console input in this section using some code that for now you can consider to be just magic formulas. In Chapters 9 and 10 we will explain these magic formulas.

THE BASICS

In Display 2.8 we have rewritten the program in Display 2.4 so that it uses simple console I/O instead of windowing I/O using `JOptionPane`.

The code in blue boxes will not be fully explained until Chapters 9 and 10. Until then you will have to simply use them as magic formulas.

To do console input, you create an object of the class `BufferedReader` using the magic formula

```
new BufferedReader(new InputStreamReader(System.in))
```

and you assign this object to a variable of type `BufferedReader`, so that the whole process is done with

```
BufferedReader keyboard =
                    new BufferedReader(new InputStreamReader(System.in));
```

Since `keyboard` is an ordinary Java variable, you may use any non-keyword identifier in place of `keyboard`.

Once you have an object of type `BufferedReader` created in this way, you can use the method `readLine` to read a line of input, as illustrated by the following line from Display 2.8:

```
String podString = keyboard.readLine();
```

Display 2.8 Console Input with BufferedReader


```
1  import java.io.BufferedReader;
2  import java.io.InputStreamReader;
3  import java.io.IOException;

4  public class BufferedReaderDemo
5  {
6     public static void main(String[] args) throws IOException
7     {
8        BufferedReader keyboard =
9                new BufferedReader(new InputStreamReader(System.in));

10       System.out.println("Enter number of pods:");
11       String podString = keyboard.readLine( );
12       int numberOfPods = Integer.parseInt(podString);

13       System.out.println("Enter number of peas in a pod:");
14       String peaString = keyboard.readLine( );
15       int peasPerPod = Integer.parseInt(peaString);

16       int totalNumberOfPeas = numberOfPods*peasPerPod;

17       System.out.print(numberOfPods + " pods and ");
18       System.out.println(peasPerPod + " peas per pod.");
19       System.out.println("The total number of peas = "
20          You do not need System.exit(0);.      + totalNumberOfPeas);
21    }
22 }
```

Magic formulas

Sample Dialogue

```
Enter number of pods:
22
Enter number of peas in a pod:
10
22 pods and 10 peas per pod.
The total number of peas = 220
```

Note that the variable `podString` is of type `String`. The method `readLine` reads an entire line of keyboard input and returns it as a value of type `String`.

`BufferedReader` has no methods for reading numbers. As was true of `JOptionPane`, with `BufferedReader` you read input as a value of type `String`. If you want a value of type `int`, you must convert the string to an `int` as in the following line from Display 2.8:

```
int numberOfPods = Integer.parseInt(podString);
```

Be sure to note that when your program does not use `JOptionPane` (or one of the other windowing interfaces we will discuss later in this text), you do not need

```
System.exit(0);
```

The program will automatically end when it runs out of statements to execute, as illustrated in Display 2.8.

You do not need to understand the magic formulas to do console input using `BufferedReader`, but it is a bit unsatisfying to leave them as a complete mystery. Before we end this subsection, let's give you at least a vague idea of what is going on in those magic formulas in Display 2.8. We have already given some discussion of the lines

```
BufferedReader keyboard =
                new BufferedReader(new InputStreamReader(System.in));
```

`System.in` looks like the input version of `System.out`. That is true, but `System.in` does not have methods as nice as those in `System.out`. In particular the object `System.in` does not have a `readLine` method. The rest of this magic formula takes `System.in` as an argument and uses it as a basis for creating an object of the class `BufferedReader`, which has the nice method `readLine`.

The three `import` statements make the three classes we use in the program available. Since all three classes imported are in the package `java.io`, you could replace those three `import` statements with the following, which imports the entire `java.io` package:

```
import java.io.*;
```

The phrase

```
throws IOException
```

tells Java that we are writing code that might "throw an exception of type `IOException`." Whenever you are using the method `readLine` of the class `BufferedReader`, there is a chance of throwing an `IOException` (whatever that is). Our code does not properly take care of `IOException` because we have not yet covered exception handling. The phrase `throws IOException` simply tells Java we know we have not taken care of exceptions, in response to which Java says "OK, if you insist, I'll let you compile this code." If we did not include the phrase, our program would not compile. Of course, we have not yet said what an exception is. Unfortunately, that will have to wait till Chapter 9.

Suffice it to say exception handling is a way of preparing in advance for things that might "go wrong."

All the details of these magic formulas will be explained in Chapters 9 and 10.

Pitfall

Input in Wrong Format

When using the method `readLine` of the class `BufferedReader` for keyboard input or using `JOptionPane` for input, your program reads a number as a string and then converts the string to a value of type `int` using `Integer.parseInt` (or using some other method to convert to a value of some other type, such as `double`). When doing numeric input in this way, the user must input the number on a line by itself with nothing else on the line, not even blanks.

If there is a chance that the user will add extra blank space before or after an input number (and there usually is such a chance), then you should use the `String` method `trim` to remove all leading or trailing blanks. For example, in Display 2.8, it would be good to replace the line

```
int numberOfPods = Integer.parseInt(podString);
```

with the following, which includes an invocation of the `trim` method:

```
int numberOfPods = Integer.parseInt(podString.trim());
```

There are ways of having multiple input numbers on the same line, but this simple technique for doing keyboard input will not handle multiple numbers on a line.

Tip

Echo Input

echoing input

You should always **echo input**. That is, you should write to the screen all input that your program receives from the keyboard. This way the user can check that he or she has entered their input correctly. For example, the following two statements from the program in Display 2.8 echo the values that were read for the number of pods and the peas per pod:

```
System.out.print(numberOfPods + " pods and ");
System.out.println(peasPerPod + " peas per pod.");
```

(We have not here directly echoed the inputs, which were the values of `podString` and `peaString`, but this gives us an even better check since it checks the converted input and so also serves as a check on the conversion from strings to numbers.)

It may seem that there is no need to echo input, since the user's input is automatically displayed on the screen as the user enters it. Why bother to write it to the screen a second time? Because the input might be incorrect even though it looks correct. For example, the user might type a comma instead of a decimal point or the letter "O" in place of a zero. Echoing the input can reveal such problems.

Keyboard Input Using BufferedReader

To do keyboard input using `BufferedReader`, you must create an object of the class `BufferedReader` using the following magic formula:

```
BufferedReader Name_Of_Object =
          new BufferedReader(new InputStreamReader(System.in));
```

You can then use the method `readLine()` with the object to read lines of input. The following reads an entire line of keyboard input and returns that line as a value of type `String`:

```
Name_Of_Object.readLine();
```

Example:

```
BufferedReader inputObject =
          new BufferedReader(new InputStreamReader(System.in));

String peopleString;
System.out.println("Enter number of people in your party:");
peopleString = inputObject.readLine();
```

This will only let your program read strings. If you want to input numbers, you must read the number as a string and convert the string to a number, as in the following example:

```
int numberOfPeople = Integer.parseInt(peopleString);
```

Techniques for obtaining numbers of other types are in Display 2.6.

More Magic Formulas

When doing keyboard input in this way, you must have the following `import` statements at the start of the file:

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
```

Alternatively, you can use the following single `import` statement in place of these three:

```
import java.io.*;
```

Also, the heading for your `main` method must have the phrase `throws IOException` added as follows:

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

For a complete example, see Display 2.8.

A PREVIEW OF THE StringTokenizer CLASS

When reading keyboard input either with `JOptionPane` or with `BufferedReader` and `readLine`, the input is always produced as a string value corresponding to a complete line of input. The class `StringTokenizer` can be used to decompose this string into words so that you can treat input as multiple items on a single line. We need the material of Chapters 3 and 4 to explain how you use `StringTokenizer` and so we will postpone discussion of the `StringTokenizer` class until Chapter 4.

Self-Test Exercises

9. In Display 2.8, can you use some name other than `keyboard` for the object of the class `BufferedReader`? For example, can you use `inputObject` as follows?

```
BufferedReader inputObject =
            new BufferedReader(new InputStreamReader(System.in));
```

   Of course, assume you also change all occurrences of `keyboard.readLine()` to `inputObject.readLine()`.

10. Write some Java code that will read a line of text and then output the line with all lowercase letters changed to uppercase. Use `BufferedReader`.

11. Write some Java code that will set the value of the variable `number` equal to the number entered by a user at the keyboard. Assume that `number` is of type `int`. Use `BufferedReader`.

12. Write a complete Java program that reads two whole numbers into two variables of type `int`, and then outputs both the whole number part and the remainder when the first number is divided by the second. This can be done using the operators `/` and `%`. Use `BufferedReader`.

13. Would it be legal to use

```
int numberOfPods =
             Integer.parseInt(keyboard.readLine());
```

   in Display 2.8 instead of the following?

```
String podString = keyboard.readLine();
int numberOfPods = Integer.parseInt(podString);
```

2.4 Input Using ConsoleIn

I get by with a little help from my friends.

Paul McCartney, *A Little Help from My Friends (song)*

The class `ConsoleIn` is a class that has methods for very robust keyboard input of strings and for each of the primitive types, such as `int` and `double`. The class is written in Java and will be presented and discussed in Chapter 9. However, you need not wait until Chapter 9 to start using the class `ConsoleIn`. In this subsection we tell you how to use this input class. The class is in the file `ConsoleIn.java` on the CD that comes with this book. To make it available to your program, place the file in the same directory (folder) as your program (or other class that uses `ConsoleIn`) and compile the class `ConsoleIn.java`.

extra code on CD

THE BASICS

You should treat the class `ConsoleIn` as if it were code that you wrote. For now, that means the file `ConsoleIn.java` (on the CD) must be copied to the directory (folder) that contains your program (or other class that uses `ConsoleIn`) and that `ConsoleIn.java` (as well as your program and other class files) must be compiled. That way, the byte-code file `ConsoleIn.class` will be in the same directory as your program (or other class that uses `ConsoleIn`). In order to use `ConsoleIn`, the file `ConsoleIn.class` must be in the same directory as the program or class that uses `ConsoleIn`. (There are other ways to make `ConsoleIn` available but they are not discussed until Chapter 5.)

ConsoleIn Is Not Part of the Java Language

The class `ConsoleIn` is not part of the Java language and does not come with the Java language. You must add the class yourself. Think of it as a class that you yourself defined. In Chapter 9 we will explain the code for `ConsoleIn`. However, you can already start using the class `ConsoleIn` for keyboard input. The code for `ConsoleIn` is in the file `ConsoleIn.java` on the CD that accompanies this book. (The code is also in Appendix 5.) When using the class `ConsoleIn`, you should have the compiled byte-code file `ConsoleIn.class` in the same directory (folder) as the program (or other class) that uses `ConsoleIn`.

Using the class `ConsoleIn` is similar to using the class `BufferedReader` except that no `import` statements are needed and you do not need to create an object of the class `ConsoleIn`. Instead you use the class name `ConsoleIn` in place of a calling object.

The class `ConsoleIn` has methods that read a piece of data from the keyboard and return that data. By placing one of these method invocations in an assignment state-

ment, your program can read from the keyboard and place the data it reads into the variable on the left-hand side of the assignment operator. For example,

```
int amount = ConsoleIn.readLineInt();
```

will read in one integer and make that integer the value of the variable `amount`. The method `readLineInt` does not use any arguments. That is why there is nothing in the parentheses after the name `readLineInt`, but still you must include the parentheses.

readLineInt

The method `readLineInt` expects the user to input one integer (of type `int`) on a line by itself, possibly with space before or after it. If the user inputs anything else, then an error message will be output to the screen and the user will be asked to reenter the input. Input is read only after the user starts a new line. So nothing happens until the user presses the Enter key. This is illustrated in Display 2.9.

readLine-Double

What if you want to read in a number of some type other than `int`? The methods `readLineByte`, `readLineShort`, `readLineLong`, `readLineFloat`, and `readLineDouble` work in the exact same way, except that they read in values of type `byte`, `short`, `long`, `float`, and `double`, respectively. For example, the following will read a single number of type `double` and store that value in the variable `measurement`:

```
double measurement = ConsoleIn.readLineDouble();
```

You can use the method `readLineNonwhiteChar` to read the first nonwhitespace character on a line:

```
char symbol = ConsoleIn.readLineNonwhiteChar();
```

whitespace

Whitespace characters are all characters that print as white space if you output them to paper (or to the screen). The only whitespace characters you are likely to be concerned with at first are the space (blank) character, the new-line character, and the tab character.

There is a slight difference between `readLineNonwhiteChar` and the methods that read a single number. For the methods `readLineInt` and `readLineDouble`, the input number must be on a line with nothing before or after the number, except possibly white space. The method `readLineNonwhiteChar` allows anything to be on the line after the first nonwhitespace character, but ignores the rest of the line. This way, when the user enters a word like `yes`, `readLineNonwhiteChar` can read the first letter, like `'y'`, and ignore the rest of the word `yes`.

If you want to read in an entire line, you would use the method `readLine` (without any `Int` or `Double` or such at the end). For example,

```
String sentence = ConsoleIn.readLine();
```

reads in one line of input and places that string in the variable `sentence`. The method `readLine` of the class `ConsoleIn` behaves the same as the method `readLine` of the class `BufferedReader`.

The methods in the class `ConsoleIn` are described in the box entitled “Input Using `ConsoleIn`.”

Self-Test Exercises

14. Write a Java statement that will set the value of the variable `number` equal to the number typed in at the keyboard. Assume that `number` is of type `int` and that the input is entered on a line by itself. Use `ConsoleIn`.

Display 2.9 Console Input with ConsoleIn


```
1  public class ConsoleInDemo                    The file ConsoleIn.class must be in the
2  {                                             same directory as this program.
3     public static void main(String[] args)
4     {
5        System.out.println("Enter number of pods:");
6        int numberOfPods = ConsoleIn.readLineInt();

7        System.out.println("Enter number of peas in a pod:");
8        int peasPerPod = ConsoleIn.readLineInt();

9        int totalNumberOfPeas = numberOfPods*peasPerPod;

10       System.out.print(numberOfPods + " pods and ");
11       System.out.println(peasPerPod + " peas per pod.");
12       System.out.println("The total number of peas = "
13                                        + totalNumberOfPeas);
14    }
            You do not need
15 }        System.exit(0);.
```

Sample Dialogue

```
Enter number of pods:
22.0
Input number is not in correct format.
The input number must be
a whole number written as an
ordinary numeral, such as 42.
Do not include a plus sign.
Minus signs are OK.
Try again.
Enter a whole number
22
Enter number of peas in a pod:
10
22 pods and 10 peas per pod.
The total number of peas = 220
```

Input Using ConsoleIn

You can use methods of the class `ConsoleIn` to read a value of a primitive type or of type `String` that has been entered at the keyboard. The input must be on a line by itself. In most cases, leading or trailing whitespace characters are ignored. If the input is entered in an incorrect format, then an error message is issued and the user is asked to reenter the input line.

Syntax:

```
Byte_Variable = ConsoleIn.readLineByte();
Short_Variable = ConsoleIn.readLineShort();
Int_Variable = ConsoleIn.readLineInt();
Long_Variable = ConsoleIn.readLineLong();
Float_Variable = ConsoleIn.readLineFloat();
Double_Variable = ConsoleIn.readLineDouble();
Boolean_Variable = ConsoleIn.readLineBoolean();
Char_Variable = ConsoleIn.readLineNonwhiteChar();
String_Variable = ConsoleIn.readLine();
```

Example:

```
int count;
count = ConsoleIn.readLineInt();
long bigOne;
bigOne = ConsoleIn.readLineLong();
float increment;
increment = ConsoleIn.readLineFloat();
double distance;
distance = ConsoleIn.readLineDouble();
char letter;
letter = ConsoleIn.readLineNonwhiteChar();
String wholeLine;
wholeLine = ConsoleIn.readLine();
```

15. Write a Java statement that will set the value of the variable `amount` equal to the number typed in at the keyboard. Assume that `amount` is of type `double` and that the input is entered on a line by itself. Use `ConsoleIn`.
16. Write a Java statement that will set the value of the variable `answer` equal to the first non-whitespace character typed in at the keyboard. The rest of the line of input is discarded. The variable `answer` is of type `char`. Use `ConsoleIn`.
17. What are the whitespace characters?
18. Is the class `ConsoleIn` part of the Java language (or does the programmer have to define the class)?
19. Write some Java code for your program that will read a line of text and then output the line with all lowercase letters changed to uppercase. Use `ConsoleIn`.

Chapter Summary

- You can use `System.out.println` for simple console output.
- You can use `NumberFormat.getCurrencyInstance()` to produce an object that can convert numbers to strings that show the number as a correctly formatted currency amount, for example, by adding a dollar sign and having exactly two digits after the decimal point.
- You can use the class `DecimalFormat` to output numbers using almost any format you desire.
- You can use `JOptionPane` for windowing I/O.
- You can use `BufferedReader` for simple console input.
- As an optional alternative to `BufferedReader`, you can use the class `ConsoleIn` for keyboard input. The class `ConsoleIn` is not in the Java standard libraries, but it is written in Java and is provided on the CD accompanying this book.

Answers to Self-Test Exercises

1. ```
   System.out.println("May the hair on your toes");
   System.out.println("grow long and curly.");
   ```

2. `System.out.println` ends a line of input, so the next output goes on the next line. With `System.out.print`, the next output goes on the same line.

3. ```
   2 2
   4
   ```

   Note that

   ```
   2 + " " + 2
   ```

   contains a string, namely `" "`. So, Java knows it is supposed to produce a string. On the other hand, `2 + 2` contains only integers. So, Java thinks `+` denotes addition in this second case and produces the value `4` to be output.

4. ```
   $1.23
   $15.68
   ```

5. Either

   ```
   import java.awt.Robot;
   ```

   or

   ```
   import java.awt.*;
   ```
   ```

6.
```
String rawString =
         JOptionPane.showInputDialog("Enter some text:");
String uppercaseString = rawString.toUpperCase();
JOptionPane.showMessageDialog(
          null, "In uppercase:\n" + uppercaseString);
```

7.
```
String numberString =
         JOptionPane.showInputDialog("Enter a number:");
int number = Integer.parseInt(numberString);
```

8.
```
import javax.swing.JOptionPane;

public class JOptionPaneExercise
{
   public static void main(String[] args)
   {
      String numeratorString =
         JOptionPane.showInputDialog("Enter numerator:");
      int numerator = Integer.parseInt(numeratorString);

      String denominatorString =
         JOptionPane.showInputDialog("Enter denominator:");
      int denominator =
              Integer.parseInt(denominatorString);

      JOptionPane.showMessageDialog(null,
         numerator + " divided by " + denominator
        + " is " + numerator/denominator
        + "\nwith a remainder of " + (numerator % denominator);

      System.exit(0);
   }
}
```

9. Yes, it is a variable name and so can be any legal identifier that is not a keyword.

10.
```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class BufferedReaderExercise
{
   public static void main(String[] args) throws IOException
   {
      BufferedReader console = new BufferedReader(
                          new InputStreamReader(System.in));
```

```
        System.out.println("Enter a line of text:");
        String line = console.readLine();
        String uppercaseLine = line.toUpperCase();
        System.out.println("In all uppercase that is:");
        System.out.println(uppercaseLine);
    }
}
```

11.
```
BufferedReader console = new BufferedReader(
                         new InputStreamReader(System.in));

System.out.println("Enter a whole number:");
String numberString = console.readLine();
int number = Integer.parseInt(numberString);
```

12.
```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class BufferedReaderExercise2
{
   public static void main(String[] args) throws IOException
   {
      BufferedReader console = new BufferedReader(
                                  new InputStreamReader(System.in));

      System.out.println(
                  "Enter two whole numbers on two lines:");
      String numeratorString = console.readLine();
      int numerator = Integer.parseInt(numeratorString);

      String denominatorString = console.readLine();
      int denominator =
                     Integer.parseInt(denominatorString);

      System.out.println(numerator + " divided by " + denominator
                + " is " + (numerator / denominator)
                + "\nwith a remainder of " + (numerator % denominator));
   }
}
```

13. Yes, it would be legal, and the resulting program would be equivalent to the one in Display 2.8.

14.
```
number = ConsoleIn.readLineInt();
```

15.
```
amount = ConsoleIn.readLineDouble();
```

16. `answer = ConsoleIn.readLineNonwhiteChar();`
17. The whitespace characters are the blank, the tab, the new-line character, `'\n'`, and all other characters that print as white space on (white) paper.
18. The class `ConsoleIn` is not part of the Java language. The programmer (like you) is supposed to define the class `ConsoleIn`. To make your life easier, we have defined it for you.
19.
```
System.out.println("Enter a line of text:");
String line = ConsoleIn.readLine();
String uppercaseString = line.toUpperCase();

System.out.println("In all uppercase that is:");
System.out.println(uppercaseString);
```

Programming Projects

1. Write a program that will read in two integers and output their sum, difference, and product. Do two versions: one version uses `JOptionPane` for I/O, the other uses `BufferedReader` for I/O. If you covered `ConsoleIn`, do a third version using `ConsoleIn`.
2. An automobile is used for commuting purposes. Write a program that will take as input the distance of the commute in miles, the automobile's fuel consumption rate in miles per gallon, and the price of a gallon of gas. The program then outputs the cost of the commute. Do two versions: one version uses `JOptionPane` for I/O, the other uses `BufferedReader` for I/O. If you covered `ConsoleIn`, do a third version using `ConsoleIn`.

3. The straight-line method for computing the yearly depreciation in value D for an item is given by the formula

$$D = \frac{P - S}{Y}$$

where P is the purchase price, S is the salvage value, and Y is the number of years the item is used. Write a program that takes as input the purchase price of an item, the expected number of years of service, and the expected salvage value. The program then outputs the yearly depreciation for the item. Do two versions: one version uses `JOptionPane` for I/O; the other uses `BufferedReader` for I/O. If you covered `ConsoleIn`, do a third version using `ConsoleIn`.

4. (This is a version with input of an exercise from Chapter 1.) A government research lab has concluded that an artificial sweetener commonly used in diet soda pop will cause death in laboratory mice. A friend of yours is desperate to lose weight but cannot give up soda pop. Your friend wants to know how much diet soda pop it is possible to drink without dying as a result. Write a program to supply the answer. The input to the program is the amount of artificial sweetener needed to kill a mouse, the weight of the mouse, and the desired weight

of the dieter. Assume that diet soda contains 1/10th of one percent artificial sweetener. Use a named constant for this fraction. You may want to express the percent as the `double` value `0.001`. Do two versions: one version uses `JOptionPane` for I/O, the other uses `BufferedReader` for I/O. If you covered `ConsoleIn`, do a third version using `ConsoleIn`.

5. Write a program that determines the change to be dispensed from a vending machine. An item in the machine can cost between 25 cents and a dollar, in 5-cent increments (25, 30, 35, . . ., 90, 95, or 100), and the machine accepts only a single dollar bill to pay for the item. For example, a possible sample dialog might be

```
Enter price of item
(from 25 cents to a dollar, in 5-cent increments): 45

You bought an item for 45 cents and gave me a dollar,
so your change is
   2 quarters,
   0 dimes, and
   1 nickel.
```

   The sample dialog is for `BufferedReader` or `ConsoleIn` for input and `System.out.println` for output. However, you may use `JOptionPane` if you prefer.

6. Write a program that reads in a line of text and then outputs that line of text first in all uppercase letters and then in all lowercase letters. Do two versions: one version uses `JOptionPane` for I/O, the other uses `BufferedReader` for I/O. If you covered `ConsoleIn`, do a third version using `ConsoleIn`.

7. (This is a version with input of an exercise from Chapter 1.) Write a program that reads in a line of text and then outputs that line of text with the first occurrence of `"hate"` changed to `"love"`. For example, a possible sample dialog might be

```
Enter a line of text.
I hate you.
I have rephrased that line to read:
I love you.
```

   You can assume that the word `"hate"` occurs in the input. If the word `"hate"` occurs more than once in the line, your program will replace only the first occurrence of `"hate"`. The sample dialog is for `BufferedReader` or `ConsoleIn` for input and `System.out.println` for output. However, you may use `JOptionPane` if you prefer.