CHAPTER 2

Using Data

In this chapter, you will:

- Declare and use constants and variables
- Use integer data types
- Use the `boolean` data type
- Use floating-point data types
- Use the `char` data type
- Use the `Scanner` class to accept keyboard input
- Use the `JOptionPane` class to accept GUI input
- Perform arithmetic
- Understand type conversion

Unless noted otherwise, all images are © 2014 Cengage Learning

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Declaring and Using Constants and Variables

A data item is **constant** when its value cannot be changed while a program is running. For example, when you include the following statement in a Java class, the number 459 is a constant:

```
System.out.println(459);
```

Every time an application containing the constant 459 is executed, the value 459 is displayed. Programmers refer to the number 459 as a **literal constant** because its value is taken literally at each use. The number 459 is also a **numeric constant** as opposed to a character or string constant. Additionally, it is an **unnamed constant** as opposed to a named one, because no identifier is associated with it.

Instead of using constant data, you can set up a data item to be variable. A **variable** is a named memory location that can store a value. A variable can hold only one value at a time, but the value it holds can change. For example, if you create a variable named `ovenTemperature`, it might hold 0 when the application starts, later be altered to hold 350, and still later be altered to hold 400.

Whether a data item is variable or constant, in Java it always has a data type. An item's **data type** describes the type of data that can be stored there, how much memory the item occupies, and what types of operations can be performed on the data. Java provides for eight primitive types of data. A **primitive type** is a simple data type. The eight types are described in Table 2-1. Later in this chapter, you will learn more specific information about several of these data types.

Keyword	Description
`byte`	Byte-length integer
`short`	Short integer
`int`	Integer
`long`	Long integer
`float`	Single-precision floating point
`double`	Double-precision floating point
`char`	A single character
`boolean`	A Boolean value (`true` or `false`)

Table 2-1 Java primitive data types

The eight data types in Table 2-1 are called *primitive* because they are simple and uncomplicated. Primitive types also serve as the building blocks for more complex data types, called **reference types**, which hold memory addresses. The classes you will begin creating in Chapter 3 are examples of reference types, as is the `Scanner` class you will use later in this chapter.

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Declaring Variables

A **variable declaration** is a statement that reserves a named memory location and includes the following:

- A data type that identifies the type of data that the variable will store
- An identifier that is the variable's name
- An optional assignment operator and assigned value, if you want a variable to contain an initial value
- An ending semicolon

You name variables using the same naming rules as you do for legal class identifiers. Basically, variable names must start with a letter and cannot be a reserved keyword. You must declare a variable before you can use it. You can declare a variable at any point before you use it, but it is common practice to declare variables first in a method and to place executable statements after the declarations. Java is a **strongly typed language**, or one in which each variable has a well-defined data type that limits the operations you can perform with it; strong typing implies that all variables must be declared before they can be used.

Variable names conventionally begin with lowercase letters to distinguish them from class names. However, as with class names, a program can compile without error even if names are constructed unconventionally. Beginning an identifier with a lowercase letter and capitalizing subsequent words within the identifier is a style known as **camel casing**. An identifier such as `lastName` resembles a camel because of the uppercase "hump" in the middle.

For example, the following declaration creates a variable of type `int` named `myAge` and assigns it an initial value of 25;

```
int myAge = 25;
```

This declaration is a complete, executable statement, so it ends with a semicolon. The equal sign (=) is the **assignment operator**. Any value to the right of the equal sign is assigned to the variable on the left of the equal sign. An assignment made when you declare a variable is an **initialization**; an assignment made later is simply an **assignment**. Thus, the first statement that follows is an initialization, and the second is an assignment:

```
int myAge = 25;
myAge = 42;
```

You declare a variable just once, but you might assign new values to it any number of times.

Note that an expression with a literal to the left of the assignment operator (such as `25 = myAge`) is illegal. The assignment operator has right-to-left associativity. **Associativity** refers to the order in which values are used with operators. The associativity of every operator is either right-to-left or left-to-right. An identifier that can appear on the left side of an assignment operator sometimes is referred to as an **lvalue**. A numeric constant like 25 is not an lvalue; it is only an **rvalue**, or an item that can appear only on the right side of an assignment operator. A variable can be used as an lvalue or an rvalue, but a literal number can only be an rvalue.

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

When you declare a variable within a method but do not assign a value to it, it is an **uninitialized variable**. For example, the following variable declaration declares a variable of type `int` named `myAge`, but no value is assigned at the time of creation:

```
int myAge;
```

An uninitialized variable contains an unknown value called a **garbage value**. Java protects you from inadvertently using the garbage value that is stored in an uninitialized variable. For example, if you attempt to display garbage or use it as part of a calculation, you receive an error message stating that the variable might not have been initialized.

When you learn about creating classes in the chapter *Using Methods, Classes, and Objects*, you will discover that variables declared in a class, but outside any method, are automatically initialized for you.

You can declare multiple variables of the same type in separate statements. You also can declare two (or more) variables of the same type in a single statement by separating the variable declarations with a comma, as shown in the following statement:

```
int myAge = 25, yourAge = 19;
```

By convention, programmers declare most variables in separate statements. You might declare multiple variables in the same statement only if they are closely related. Remember that even if a statement occupies multiple lines, the statement is not complete until the semicolon is reached.

You can declare as many variables in a statement as you want, as long as the variables are the same data type. However, if you want to declare variables of different types, you must use a separate statement for each type.

Declaring Named Constants

A variable is a named memory location for which the contents can change. If a named location's value should not change during the execution of a program, you can create it to be a **named constant**. A named constant is also known as a **symbolic constant**. A named constant is similar to a variable in that it has a data type, a name, and a value. A named constant differs from a variable in several ways:

- In its declaration statement, the data type of a named constant is preceded by the keyword **`final`**.
- A named constant can be assigned a value only once, and then it can never be changed. Usually you initialize a named constant when you declare it; if you do not initialize the constant at declaration, it is known as a **blank `final`**, and you can assign a value later. You can assign a value to a `final` constant only once, and you must assign a value before the constant is used.
- Although it is not a requirement, named constants conventionally are given identifiers using all uppercase letters, using underscores as needed to separate words.

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

For example, each of the following defines a conventionally named constant:

```
final int NUMBER_OF_DEPTS = 20;
final double PI = 3.14159;
final double TAX_RATE = 0.015;
final string COMPANY = "ABC Manufacturing";
```


You can use each of these named constants anywhere you can use a variable of the same type, except on the left side of an assignment statement after the first value has been assigned. In other words, after they receive their initial values, named constants are rvalues.

A constant always has the same value within a program, so you might wonder why you cannot use the actual, literal value. For example, why not use the unnamed constant *20* when you need the number of departments in a company rather than going to the trouble of creating the `NUMBER_OF_DEPTS` named constant? There are several good reasons to use the named constant rather than the literal one:

- The number 20 is more easily recognized as the number of departments if it is associated with an identifier. Using named constants makes your programs easier to read and understand. Some programmers refer to the use of a literal numeric constant, such as 20, as using a **magic number**—a value that does not have immediate, intuitive meaning or a number that cannot be explained without additional knowledge. For example, you might write a program that uses the value 7 for several purposes, so you might use constants such as `DAYS_IN_WEEK` and `NUM_RETAIL_OUTLETS` that both hold the value 7 but more clearly describe the purpose. Avoiding magic numbers helps provide internal documentation for your programs.
- If the number of departments in your organization changes, you would change the value of `NUMBER_OF_DEPTS` at one location within your program—where the constant is defined—rather than searching for every use of 20 to change it to a different number. Being able to make the change at one location saves you time and prevents you from missing a reference to the number of departments.
- Even if you are willing to search for every instance of 20 in a program to change it to the new department number value, you might inadvertently change the value of one instance of 20 that is being used for something else, such as a payroll deduction value.
- Using named constants reduces typographical errors. For example, if you must include 20 at several places within a program, you might inadvertently type 10 or 200 for one of the instances, and the compiler will not recognize the mistake. However, if you use the identifier `NUMBER_OF_DEPTS`, the compiler will ensure that you spell it correctly.
- When you use a named constant in an expression, it stands out as separate from a variable. For example, in the following arithmetic statement, it is easy to see which elements are variable and which are constant because the constants have been named conventionally:

```
double payAmount = hoursWorked * STD_PAY_RATE -
   numDependents * DEDUCTION;
```

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Although many programmers use named constants to stand for most of the constant values in their programs, many make an exception when using 0 or 1.

The Scope of Variables and Constants

A data item's **scope** is the area in which it is visible to a program and in which you can refer to it using its simple identifier. A variable or constant is in scope from the point it is declared until the end of the block of code in which the declaration lies. A **block of code** is the code contained between a set of curly braces. So, if you declare a variable or constant within a method, it can be used from its declaration until the end of the method unless the method contains multiple sets of curly braces. Then, a data item is usable only until the end of the block that holds the declaration.

In the chapter *Using Methods, Classes, and Objects*, you will start to create classes that contain multiple sets of curly braces. In the chapter *More Object Concepts*, you will learn some techniques for using variables that are not currently in scope.

Concatenating Strings to Variables and Constants

You can display a variable or a constant in a `print()` or `println()` statement alone or in combination with a string. For example, the `NumbersPrintln` class shown in Figure 2-1 declares an integer `billingDate`, which is initialized to 5. In the first shaded statement, the value of `billingDate` is sent alone to the `print()` method; in the second shaded statement, `billingDate` is combined with, or **concatenated** to, a `String`. In Java, when a numeric variable is concatenated to a `String` using the plus sign, the entire expression becomes a `String`. In Figure 2-1, `print()` and `println()` method calls are used to display different data types, including a `String`, a number, and a concatenated `String`. The output of the application shown in Figure 2-1 appears in Figure 2-2.

```
public class NumbersPrintln
{
   public static void main(String[] args)
   {
      int billingDate = 5;
      System.out.print("Bills are sent on day ");
      System.out.print(billingDate);
      System.out.println(" of the month");
      System.out.println("Next bill: October " +
         billingDate);
   }
}
```

Figure 2-1 `NumbersPrintln` class

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

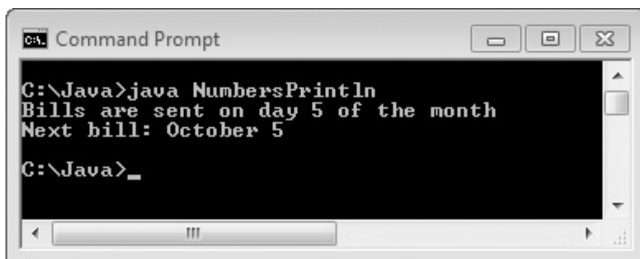


Figure 2-2 Output of `NumbersPrintln` application

Later in this chapter, you will learn that a plus sign (+) between two numeric values indicates an addition operation. However, when you place a string on one or both sides of a plus sign, concatenation occurs. In Chapter 1, you learned that *polymorphism* describes the feature of languages that allows the same word or symbol to be interpreted correctly in different situations based on the context. The plus sign is polymorphic in that it indicates concatenation when used with strings but addition when used with numbers.

When you concatenate a `String` with numbers, the entire expression is a `String`. Therefore, the expression `"A" + 3 + 4` results in the `String` `"A34"`. If your intention is to create the `String` `"A7"`, then you could add parentheses to write `"A" + (3 + 4)`.

The program in Figure 2-1 uses the command line to display values, but you also can use a dialog box. Recall from Chapter 1 that you can use the `showMessageDialog()` method with two arguments: `null`, which indicates the box should appear in the center of the screen, and the `String` to be displayed in the box. Figure 2-3 shows a `NumbersDialog` class that uses the `showMessageDialog()` method twice to display an integer declared as `creditDays` and initialized to 30. In each shaded statement in the class, the numeric variable is concatenated to a `String`, making the entire second argument a `String`. In the first shaded statement, the concatenated `String` is an empty `String` (or **null `String`**), created by typing a set of quotes with nothing between them. The application produces the two dialog boxes shown in Figures 2-4 and 2-5. The first dialog box shows just the value 30; after it is dismissed by clicking OK, the second dialog box appears.

```
import javax.swing.JOptionPane;
public class NumbersDialog
{
   public static void main(String[] args)
   {
      int creditDays = 30;
      JOptionPane.showMessageDialog(null, "" + creditDays);
      JOptionPane.showMessageDialog
         (null, "Every bill is due in " + creditDays + " days");
   }
}
```

Figure 2-3 `NumbersDialog` class

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

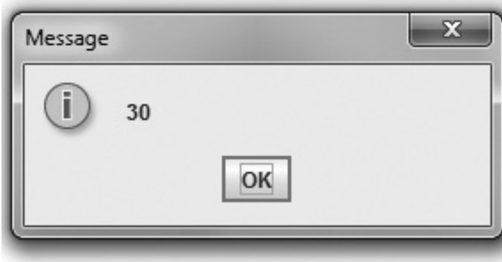


Figure 2-4 First dialog box created by `NumbersDialog` application

Figure 2-5 Second dialog box created by `NumbersDialog` application

Pitfall: Forgetting That a Variable Holds One Value at a Time

Each constant can hold only one value for the duration of its program; each variable can hold just one value at a time. Suppose you have two variables, `x` and `y`, and `x` holds 2 and `y` holds 10. Suppose further that you want to switch their values so that `x` holds 10 and `y` holds 2. You cannot simply make an assignment such as `x = y` because then both variables will hold 10, and the 2 will be lost. Similarly, if you make the assignment `y = x`, then both variables will hold 2, and the 10 will be lost. The solution is to declare and use a third variable, as in the following sequence of events:

```
int x = 2, y = 10, z;
z = x;
x = y;
y = z;
```

In this example, the third variable, `z`, is used as a temporary holding spot for one of the original values. The variable `z` is assigned the value of `x`, so `z` becomes 2. Then the value of `y`, 10, is assigned to `x`. Finally, the 2 held in `z` is assigned to `y`. The extra variable is used because as soon as you assign a value to a variable, any value that was previously in the memory location is gone.

Watch the video *Declaring Variables and Constants.*

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

TWO TRUTHS & A LIE

Declaring and Using Constants and Variables

1. A variable is a named memory location that you can use to store a value; it can hold only one value at a time, but the value it holds can change.
2. An item's data type determines what legal identifiers can be used to describe variables and whether the variables can occupy memory.
3. A variable declaration is a statement that reserves a named memory location and includes a data type, an identifier, an optional assignment operator and assigned value, and an ending semicolon.

The false statement is #2. An item's data type describes the type of data that can be stored, how much memory the item occupies, and what types of operations can be performed on the data. The data type does not alter the rules for a legal identifier, and the data type does not determine whether variables can occupy memory—all variables occupy memory.

You Do It

Declaring and Using a Variable

In this section, you write an application to work with a variable and a constant.

1. Open a new document in your text editor. Create a class header and an opening and closing curly brace for a new class named `DataDemo` by typing the following:

```
public class DataDemo
{
}
```

2. Between the curly braces, indent a few spaces and type the following `main()` method header and its curly braces:

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

(continues)

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

(continued)

3. Between the `main()` method's curly braces, type the following variable declaration:

```
int aWholeNumber = 315;
```

4. Type the following output statements. The first displays a string that includes a space before the closing quotation mark and leaves the insertion point for the next output on the same line. The second statement displays the value of `aWholeNumber` and then advances to a new line.

```
System.out.print("The number is ");
System.out.println(aWholeNumber);
```

5. Save the file as **DataDemo.java**.
6. Up to this point in the book, every `print()` and `println()` statement you have seen has used a `String` as an argument. When you added the last two statements to the `DataDemo` class, you wrote a `println()` statement that uses an `int` as an argument. As a matter of fact, there are many different versions of `print()` and `println()` that use different data types. Go to the Java Web site (***www.oracle.com/technetwork/java/index.html***), select **Java APIs**, and then select **Java SE 7**. Scroll through the list of **All Classes**, and select **PrintStream**; you will recall from Chapter 1 that `PrintStream` is the data type for the `out` object used with the `println()` method. Scroll down to view the list of methods in the **Method Summary**, and notice the many versions of the `print()` and `println()` methods, including ones that accept a `String`, an `int`, a `long`, and so on. In the last two statements you added to this program, one used a method version that accepts a `String` and the other used a method version that accepts an `int`.
7. Compile the file from the command line by typing **javac DataDemo.java**. If necessary, correct any errors, save the file, and then compile again.
8. Execute the application from the command line by typing **java DataDemo**. The command window output is shown in Figure 2-6.

Figure 2-6 Output of the `DataDemo` application

(continues)

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

(continued)

Trying to Use an Uninitialized Variable

In this section you see what happens when a variable is uninitialized.

1. In the `DataDemo` class, remove the assignment operator and the initialization value of the `aWholeNumber` variable so the declaration becomes:

```
int aWholeNumber;
```

2. Save the class and recompile it. An error message appears as shown in Figure 2-7. Notice that the declaration statement does not generate an error because you can declare a variable without initializing it. However, the `println()` statement generates the error message because in Java, you cannot display an uninitialized variable.

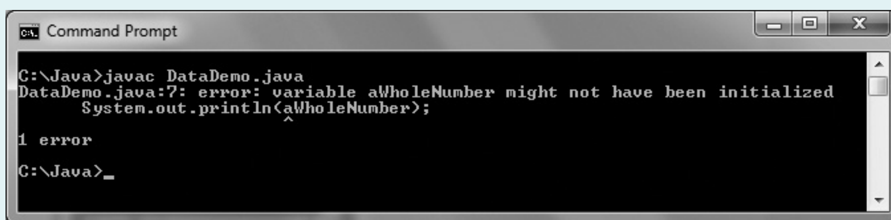


Figure 2-7 Error message generated when a variable is not initialized

3. Modify the `aWholeNumber` declaration so that the variable is again initialized to 315. Compile the class, and execute it again.

Adding a Named Constant to a Program

In this section you add a named constant to the `DataDemo` program.

1. After the declaration of the `aWholeNumber` variable in the `DataDemo` class, insert a new line in your program and type the following constant declaration:

```
final int STATES_IN_US = 50;
```

2. Following the last `println()` statement in the existing program, add a new statement to display a string and the constant. In this case, you concatenate the string and the numeric value, so the `println()` method call uses the version that accepts a `String` argument.

```
System.out.println("The number of states is " +
   STATES_IN_US);
```

(continues)

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

(continued)

3. Save the program, and then compile and execute it. The output appears in Figure 2-8.

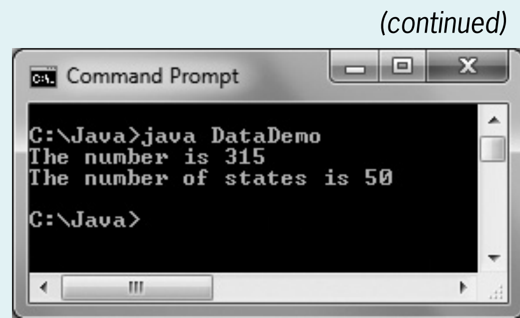


Figure 2-8 Output of `DataDemo` program after recent changes

Learning About Integer Data Types

In Java, you can use variables of types `byte`, `short`, `int`, and `long` to store (or hold) integers; an **integer** is a whole number without decimal places.

The **`int`** data type is the most commonly used integer type. A variable of type `int` can hold any whole number value from –2,147,483,648 to +2,147,483,647. When you assign a value to an `int` variable, you do not type any commas or periods; you type only digits and an optional plus or minus sign to indicate a positive or negative integer.

The legal integer values are -2^{31} through $2^{31}-1$. These are the highest and lowest values that you can store in four bytes of memory, which is the size of an `int` variable.

The types **`byte`**, **`short`**, and **`long`** are all variations of the integer type. The `byte` and `short` types occupy less memory and can hold only smaller values; the `long` type occupies more memory and can hold larger values. Table 2-2 shows the upper and lower value limits for each of these types. In other programming languages, the format and size of primitive data types might depend on the platform on which a program is running. In contrast, Java consistently specifies the size and format of its primitive data types.

Type	Minimum Value	Maximum Value	Size in Bytes
`byte`	–128	127	1
`short`	–32,768	32,767	2
`int`	–2,147,483,648	2,147,483,647	4
`long`	–9,223,372,036,854,775,808	9,223,372,036,854,775,807	8

Table 2-2 Limits on integer values by type

It is important to choose appropriate types for the variables you will use in an application. If you attempt to assign a value that is too large for the data type of the variable, the compiler issues an

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

error message, and the application does not execute. If you choose a data type that is larger than you need, you waste memory. For example, a personnel application might use a `byte` variable for number of dependents (because a limit of 127 is more than enough), a `short` for hours worked in a month (because 127 isn't enough), and an `int` for an annual salary (because even though a limit of 32,000 might be large enough for your salary, it isn't enough for the CEO's).

Some famous glitches have occurred because programmers did not pay attention to the limits of various data types. For example, a hospital computer system in Washington, D.C., used the equivalent of a `short` to count days elapsed since January 1, 1900. The system collapsed on the 32,768th day (which was in 1989), requiring manual operations for a lengthy period.

If an application uses a literal constant integer, such as 932, the number is an `int` by default. If you need to use a constant higher than 2,147,483,647, you must follow the number with the letter *L* to indicate `long`. For example, the following statement stores a number that is greater than the maximum limit for the `int` type.

```
long mosquitosInTheNorthWoods = 2444555888L;
```

You can type either an uppercase or lowercase L after the digits to indicate the `long` type, but the uppercase *L* is preferred to avoid confusion with the number 1. You don't need any special notation to store a numeric constant in an `int`, `byte`, or a `short`.

Because integer constants, such as 18, are type `int` by default, the examples in this book almost always declare a variable as type `int` when the variable's purpose is to hold a whole number. That is, even if the expected value is less than 128, such as `hoursWorkedToday`, this book will declare the variable to be an `int`. If you are writing an application in which saving memory is important, you might choose to declare the same variable as a `byte`.

Saving memory is seldom an issue for an application that runs on a PC. However, when you write applications for small devices with limited memory, like phones, conserving memory becomes more important.

TWO TRUTHS & A LIE

Learning About Integer Data Types

1. A variable of type `int` can hold any whole number value from approximately negative two billion to positive two billion.
2. When you assign a value to an `int` variable, you do not type any commas; you type only digits and an optional plus or minus sign to indicate a positive or negative integer.
3. You can use the data types `byte` or `short` to hold larger values than can be accommodated by an `int`.

The false statement is #3. You use a `long` if you know you will be working with very large values; you use a `byte` or a `short` if you know a variable will need to hold only small values.

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

You Do It

Working with Integers

In this section you work more with integer values.

1. Open a new file in your text editor, and create a shell for an `IntegerDemo` class as follows:

```
public class IntegerDemo
{
}
```

2. Between the curly braces, indent a few spaces and write the shell for a `main()` method as follows:

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

3. Within the `main()` method, create four declarations, one each for the four integer data types.

```
int anInt = 12;
byte aByte = 12;
short aShort = 12;
long aLong = 12;
```

4. Next, add four output statements that describe and display each of the values. The spaces are included at the ends of the string literals so that the values will be aligned vertically when they are displayed.

```
System.out.println("The int is   " + anInt);
System.out.println("The byte is  " + aByte);
System.out.println("The short is " + aShort);
System.out.println("The long is  " + aLong);
```

5. Save the file as **IntegerDemo.java**. Then compile and execute it. Figure 2-9 shows the output. All the values are legal sizes for each data type, so the program compiles and executes without error.

(continues)

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

(continued)

Figure 2-9 Output of the `IntegerDemo` program

6. Change each assigned value in the application from 12 to **1234**, and then save and recompile the program. Figure 2-10 shows the error message generated because 1234 is too large to be placed in a `byte` variable. The message, "possible loss of precision", means that if the large number had been inserted into the small space, the accuracy of the number would have been compromised.

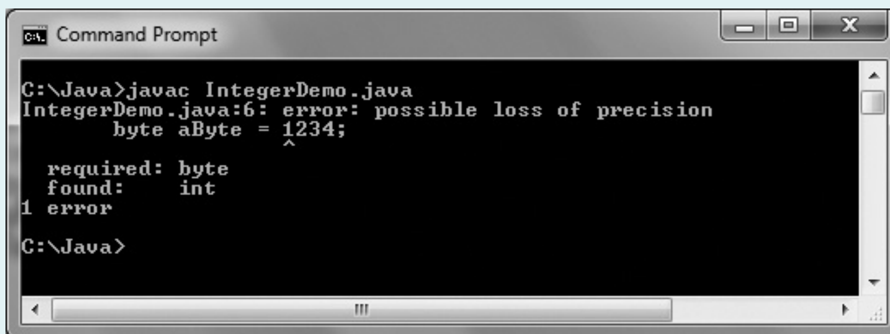


Figure 2-10 Error message generated when a value that is too large is assigned to a `byte` variable

7. Change the value of `aByte` back to 12. Change the value of `aShort` to **123456**. Save and recompile the program. Figure 2-11 shows the result. The error message "possible loss of precision" is the same as when the `byte` value was invalid, but the error indicates that the problem is now with the `short` variable.

(continues)

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

(continued)

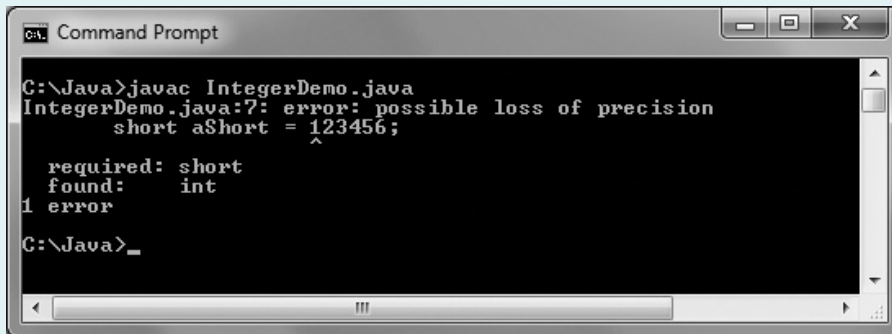


Figure 2-11 Error message generated when a value that is too large is assigned to a `short` variable

8. Change the value of the `short` variable to **12345**, and then save and compile the program. Now, the program compiles without error. Execute the program and confirm that it runs as expected.

9. At the Java Web site (*www.oracle.com/technetwork/java/index.html*), examine the list of `println()` methods in the `PrintStream` class. Although you can find versions that accept `String`, `int`, and `long` arguments, you cannot find ones that accept `byte` or `short` values. Yet, the `println()` statements in the latest version of the program work correctly. The reason has to do with *type conversion*, which you will learn about later in this chapter.

10. Replace the value of `aLong` with **1234567890987654321**. Save the program and compile it. Figure 2-12 shows the error message that indicates that the integer number is too large. The message does not say that the value is too big for a `long` type variable. Instead, it means that the literal constant was evaluated and found to be too large to be a default `int` before any attempt was made to store it in the `long` variable.

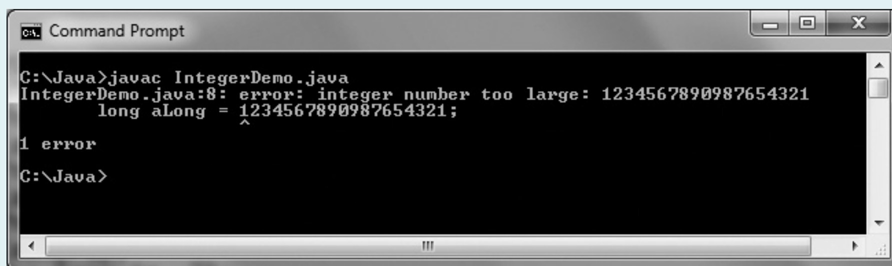


Figure 2-12 Error message generated when a value that is too large is assigned to a `long` variable

(continues)

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

(continued)

11. Remedy the problem by adding an ***L*** to the end of the `long` numeric value. Now, the constant is the correct data type that can be assigned to the `long` variable. Save, compile, and execute the program; it executes successfully.

12. Watch out for errors that occur when data values are acceptable for a data type when used alone but together might produce arithmetic results that are out of range. To demonstrate, add the following declaration at the end of the current list of variable declarations in the `IntegerDemo` program:

```
int anotherInt = anInt * 10000000;
```

13. At the end of the current list of output statements, add another output statement so that you can see the result of the arithmetic:

```
System.out.println("Another int is " + anotherInt);
```

Save, compile, and execute the program. The output appears in Figure 2-13. Although 1234 and 10000000 are both acceptable `int` values, their product is out of range for an `int`, and the resulting `int` does not appear to have been calculated correctly. Because the arithmetic result was too large, some information about the value has been lost, including the result's sign. If you see such unreasonable results in your programs, you need to consider using different data types for your values.

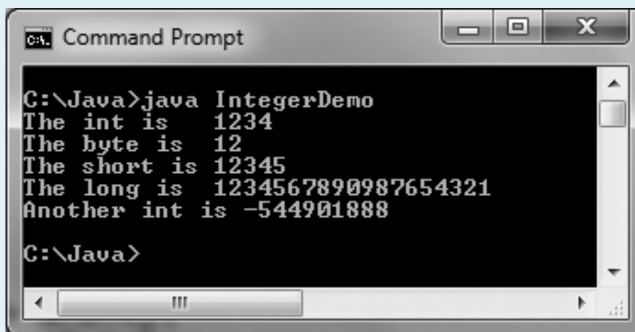


Figure 2-13 Output of the modified `IntegerDemo` program with an out-of-range integer

Using the boolean Data Type

Boolean logic is based on true-or-false comparisons. Whereas an `int` variable can hold millions of different values (at different times), a **boolean variable** can hold only one of two values—`true` or `false`. The following statements declare and assign appropriate values to Boolean variables:

```
boolean isItPayday = false;
boolean areYouBroke = true;
```

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

You also can assign values based on the result of comparisons to Boolean variables. Java supports six relational operators that are used to make comparisons. A **relational operator** compares two items; it is sometimes called a **comparison operator**. An expression that contains a relational operator has a Boolean value. Table 2-3 describes the relational operators.

When you use *Boolean* as an adjective, as in *Boolean operators*, you usually begin with an uppercase *B* because the data type is named for Sir George Boole, the founder of symbolic logic, who lived from 1815 to 1864. The Java data type `boolean`, however, begins with a lowercase *b*.

Operator	Description	True Example	False Example
<	Less than	3 < 8	8 < 3
>	Greater than	4 > 2	2 > 4
==	Equal to	7 == 7	3 == 9
<=	Less than or equal to	5 <= 5	8 <= 6
>=	Greater than or equal to	7 >= 3	1 >= 2
!=	Not equal to	5 != 6	3 != 3

Table 2-3 Relational operators

When you use any of the operators that have two symbols (==, <=, >=, or !=), you cannot place any whitespace between the two symbols. You also cannot reverse the order of the symbols. That is, =<, =>, and =! are all invalid operators.

Legal declaration statements might include the following statements, which compare two values directly:

```
boolean isSixBigger = (6 > 5);
 // Value stored would be true
boolean isSevenSmallerOrEqual = (7 <= 4);
 // Value stored would be false
```

Although you can use any legal identifier for Boolean variables, they are easily identified as Boolean if you use a form of *to be* (such as *is* or *are*) as part of the variable name, as in `isSixBigger`.

The Boolean expressions are more meaningful when variables (that have been assigned values) are used in the comparisons, as in the following examples. In the first statement, the `hours` variable is compared to a constant value of 40. If the `hours` variable is not greater than 40, the expression evaluates to `false`. In the second statement, the `income` variable must be greater than 100000 for the expression to evaluate to `true`. In the third statement, two variables are compared to determine the value of `isFirstScoreHigher`.

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

```
boolean isOvertimePay = (hours > 40);
boolean isTaxBracketHigh = (income > 100000);
boolean isFirstScoreHigher = (score1 > score2);
```

Boolean expressions will become far more useful to you when you learn about decision making and looping in Chapters 5 and 6.

TWO TRUTHS & A LIE

Using the `boolean` Data Type

1. A Boolean variable can hold only one of two values—`true` or `false`.
2. Java supports six relational operators that are used to make comparisons: =, <, >, =<, =>, and =!.
3. An expression that contains a relational operator has a Boolean value.

The false statement is #2. The six relational operators used to make comparisons are == (two equal signs), <, >, <= (the less-than sign precedes the equal sign), >= (the greater-than sign precedes the equal sign), and != (the exclamation point precedes the equal sign).

Learning About Floating-Point Data Types

A **floating-point** number contains decimal positions. Java supports two floating-point data types: `float` and `double`. A **`float`** data type can hold floating-point values of up to six or seven significant digits of accuracy. A **`double`** data type requires more memory than a `float` and can hold 14 or 15 significant digits of accuracy. The term **significant digits** refers to the mathematical accuracy of a value. For example, a `float` given the value 0.324616777 is displayed as 0.324617 because the value is accurate only to the sixth decimal position. Table 2-4 shows the minimum and maximum values for each floating-point data type. Notice that the maximum value for a `double` is 3.4 * 10 to the 38th power, which means 3.4 times 10 with 38 trailing zeros—a very large number.

A `float` given the value 324616777 is displayed as 3.24617e+008, which means approximately 3.24617 times 10 to the 8th power, or 324617000. The `e` in the displayed value stands for *exponent*; the +008 means the true decimal point is eight positions to the right of where it is displayed, indicating a very large number. (A negative number would indicate that the true decimal point belongs to the left, indicating a very small number.) This format is called **scientific notation**. The large value contains only six significant digits.

A programmer might choose to store a value as a `float` instead of a `double` to save memory. However, if high levels of accuracy are needed, such as in graphics-intensive software, the programmer might choose to use a `double`, opting for high accuracy over saved memory.

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Type	Minimum	Maximum	Size in Bytes
`float`	$-3.4 * 10^{38}$	$3.4 * 10^{38}$	4
`double`	$-1.7 * 10^{308}$	$1.7 * 10^{308}$	8

Table 2-4 Limits on floating-point values

A value stored in a `double` is a **double-precision floating-point number**; a value in a `float` is a **single-precision floating-point number**.

Just as an integer constant, such as 18, is a value of type `int` by default, a floating-point constant, such as 18.23, is a `double` by default. To indicate that a floating-point numeric constant is a `float`, you can type the letter *F* after the number, as in the following:

```
float pocketChange = 4.87F;
```

You can type either a lowercase or an uppercase *F*. You also can type *D* (or *d*) after a floating-point constant to indicate it is a `double`, but even without the *D*, the value will be stored as a `double` by default. Floating-point numbers can be imprecise, as you will see later in this chapter.

TWO TRUTHS & A LIE

Learning About Floating-Point Data Types

1. Java supports two floating-point data types: `float` and `double`. The `double` data type requires more memory and can hold more significant digits.
2. A floating-point constant, such as 5.6, is a `float` by default.
3. As with integers, you can perform the mathematical operations of addition, subtraction, multiplication, and division with floating-point numbers.

The false statement is #2. A floating-point constant, such as 5.6, is a `double` by default.

Using the `char` Data Type

You use the **`char`** data type to hold any single character. You place constant character values within single quotation marks because the computer stores characters and integers differently. For example, the following are typical character declarations:

```
char middleInitial = 'M';
char gradeInChemistry = 'A';
char aStar = '*';
```

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Some programmers prefer to pronounce *char* as *care* because it represents the first syllable in the word *character*. Others prefer to pronounce the word as *char* to rhyme with *car*. You should use the preferred pronunciation in your organization.

A character can be any letter—uppercase or lowercase. It might also be a punctuation mark or digit. A character that is a digit is represented in computer memory differently than a numeric value represented by the same digit. For example, the following two statements are legal:

```
char aCharValue = '9';
int aNumValue = 9;
```

If you display each of these values using a `println()` statement, you see a 9. However, only the numeric value, `aNumValue`, can be used to represent the value 9 in arithmetic statements.

A numeric constant can be stored in a character variable and a character that represents a number can be stored in a numeric variable. For example, the following two statements are legal, but unless you understand their meanings, they might produce undesirable results:

```
char aCharValue = 9;
int aNumValue = '9';
```

If these variables are displayed using `println()` statements, then the resulting output is a blank for `aCharValue` and the number 57 for `aNumValue`. The unexpected values are Unicode values. Every computer stores every character it uses as a number; every character is assigned a unique numeric code using Unicode. Table 2-5 shows some Unicode decimal values and their character equivalents. For example, the character *A* is stored using the value 65, and the character *B* is stored using the value 66. Appendix B contains more information on Unicode.

Dec	Char	Dec	Char	Dec	Char	Dec	Char
0	nul	32		64	@	96	`
1	soh^A	33	!	65	A	97	a
2	stx^B	34	“	66	B	98	b
3	etx^C	35	#	67	C	99	c
4	eot^D	36	$	68	D	100	d
5	enq^E	37	%	69	E	101	e
6	ask^F	38	&	70	F	102	f
7	bel^G	39	‘	71	G	103	g
8	bs^H	40	(	72	H	104	h
9	ht^I	41	)	73	I	105	i
10	lf^J	42	*	74	J	106	j

Table 2-5 Unicode values 0 through 127 and their character equivalents (*continues*)

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

(continued)

Dec	Char	Dec	Char	Dec	Char	Dec	Char
11	vt^K	43	+	75	K	107	k
12	ff^L	44	,	76	L	108	l
13	cr^M	45	-	77	M	109	m
14	so^N	46	.	78	N	110	n
15	si^O	47	/	79	O	111	o
16	dle^P	48	0	80	P	112	p
17	dc1^Q	49	1	81	Q	113	q
18	dc2^R	50	2	82	R	114	r
19	dc3^S	51	3	83	S	115	s
20	dc4^T	52	4	84	T	116	t
21	nak^U	53	5	85	U	117	u
22	syn^V	54	6	86	V	118	v
23	etb^W	55	7	87	W	119	w
24	can^X	56	8	88	X	120	x
25	em^Y	57	9	89	Y	121	y
26	sub^Z	58	:	90	Z	122	z
27	esc	59	;	91	[	123	{
28	fs	60	<	92	\	124	|
29	gs	61	=	93	]	125	}
30	rs	62	>	94	^	126	~
31	us	63	?	95	_	127	del

Table 2-5 Unicode values 0 through 127 and their character equivalents

A variable of type `char` can hold only one character. To store a string of characters, such as a person's name, you must use a data structure called a `String`. In Java, **`String`** is a built-in class that provides you with the means for storing and manipulating character strings. Unlike single characters, which use single quotation marks, string constants are written between double quotation marks. For example, the expression that stores the name *Audrey* as a string in a variable named `firstName` is:

```
String firstName = "Audrey";
```

You will learn more about strings and the `String` class in the chapter *Characters, Strings, and the `StringBuilder`*.

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

You can store any character—including nonprinting characters such as a backspace or a tab—in a `char` variable. To store these characters, you can use an **escape sequence**, which always begins with a backslash followed by a character—the pair represents a single character. For example, the following code stores a newline character and a tab character in the `char` variables `aNewLine` and `aTabChar`:

```
char aNewLine = '\n';
char aTabChar = '\t';
```

In the declarations of `aNewLine` and `aTabChar`, the backslash and character pair acts as a single character; the escape sequence serves to give a new meaning to the character. That is, the literal characters in the preceding code have different values from the "plain" characters `'n'` or `'t'`. Table 2-6 describes some common escape sequences that you can use with command window output in Java.

Escape Sequence	Description
\b	Backspace; moves the cursor one space to the left
\t	Tab; moves the cursor to the next tab stop
\n	Newline or linefeed; moves the cursor to the beginning of the next line
\r	Carriage return; moves the cursor to the beginning of the current line
\"	Double quotation mark; displays a double quotation mark
\'	Single quotation mark; displays a single quotation mark
\\	Backslash; displays a backslash character

Table 2-6 Common escape sequences

When you display values within `JOptionPane` dialog boxes rather than in a command window, the escape sequences `'\n'` (newline), `'\"'` (double quote), and `'\\'` (backslash) operate as expected within a `JOptionPane` object, but `'\t'`, `'\b'`, and `'\r'` do not work in the GUI environment.

When you want to produce console output on multiple lines in the command window, you have two options: You can use the newline escape sequence, or you can use the `println()` method multiple times. For example, Figures 2-14 and 2-15 both show classes that produce the same output: "Hello" on one line and "there" on another. The version you choose to use is up to you. The example in Figure 2-14 is more efficient—from a typist's point of view because the text `System.out.println` appears only once, and from the compiler's point of view because the `println()` method is called only once. The example in Figure 2-15, however, might be easier to read and understand. When programming in Java, you will find occasions when each of these approaches makes sense.

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

```
public class HelloThereNewLine
{
   public static void main(String[] args)
   {
      System.out.println("Hello\nthere");
   }
}
```

Figure 2-14 `HelloThereNewLine` class

```
public class HelloTherePrintlnTwice
{
   public static void main(String[] args)
   {
      System.out.println("Hello");
      System.out.println("there");
   }
}
```

Figure 2-15 `HelloTherePrintlnTwice` class

The `println()` method uses the local platform's line terminator character, which might or might not be the newline character `\n`.

TWO TRUTHS & A LIE

Using the char Data Type

1. You use the `char` data type to hold any single character; you place constant character values within single quotation marks.
2. To store a string of characters, you use a data structure called a `Text`; string constants are written between parentheses.
3. An escape sequence always begins with a backslash followed by a character; the pair represents a single character.

The false statement is #2. To store a string of characters, you use a data structure called a `String`; string constants are written between quotation marks.

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

You Do It

Working with the char Data Type

In the steps in this section, you create an application that demonstrates some features of the char data type.

1. Create the shells for a class named CharDemo and its main() method as follows:

```
public class CharDemo
{
    public static void main(String[] args)
    {
    }
}
```

2. Between the curly braces for the main() method, declare a char variable, and provide an initialization value:

```
char initial = 'A';
```

3. Add two statements. The first displays the variable, and the second demonstrates some escape sequence characters.

```
System.out.println(initial);
System.out.print("\t\"abc\\def\bghi\n\njkl");
```

4. Save the file as **CharDemo.java**, and then compile and execute it. Figure 2-16 shows the output. The first line of output contains the value of the char variable. The next line starts with a tab created by the escape sequence \t. The tab is followed by a quotation mark produced by the escape sequence \". Then abc is displayed, followed by the next escape sequence that produces a slash. The next series of characters to display is def, but because those letters are followed by a backspace escape sequence, the f is overridden by ghi. After ghi, two newline escape sequences provide a double-spaced effect. Finally, the last three characters jkl are displayed.

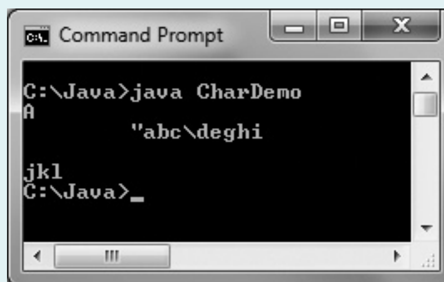


Figure 2-16 Output of the CharDemo program

5. Modify, recompile, and execute the CharDemo program as many times as you like until you can accurately predict what will be displayed when you use various combinations of characters and escape sequences.

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Using the Scanner Class to Accept Keyboard Input

Although you can assign values to variables you declare, programs typically become more useful when a user can supply different values for variables each time a program executes. In Chapter 1, you learned how to display output on the monitor using the `System.out` property. `System.out` refers to the standard output device, which usually is the monitor. To create interactive programs that accept input from a user, you can use `System.in`, which refers to the **standard input device** (normally the keyboard).

You have learned that you can use the `print()` and `println()` methods to display many data types; for example, you can use them to display a `double`, `int`, or `String`. The `System.in` object is not as flexible; it is designed to read only bytes. That's a problem, because you often want to accept data of other types. Fortunately, the designers of Java have created a class named `Scanner` that makes `System.in` more flexible.

To create a `Scanner` object and connect it to the `System.in` object, you write a statement similar to the following:

```
Scanner inputDevice = new Scanner(System.in);
```

The portion of the statement to the left of the assignment operator, `Scanner inputDevice`, declares an object of type `Scanner` with the programmer-chosen name `inputDevice`, in exactly the same way that `int x;` declares an integer with the programmer-chosen name `x`.

The portion of the statement to the right of the assignment operator, `new Scanner(System.in)`, creates a `Scanner` object that is connected to the `System.in` property. In other words, the created `Scanner` object is connected to the default input device. The keyword `new` is required by Java; you will use it whenever you create objects that are more complex than the simple data types.

In the chapter *More Object Concepts*, you will learn that the second part of the `Scanner` declaration calls a special method called a constructor that is part of the prewritten `Scanner` class. You also will learn more about the Java keyword `new` in the next two chapters.

The assignment operator in the `Scanner` declaration statement assigns the value of the new object—that is, its memory address—to the `inputDevice` object in the program.

The `Scanner` class contains methods that retrieve values from an input device. Each retrieved value is a **token**, which is a set of characters that is separated from the next set by whitespace. Most often, this means that data is accepted when a user presses the Enter key, but it could also mean that a token is accepted after a space or tab. Table 2-7 summarizes some of the most useful methods that read different data types from the default input device. Each retrieves a value from the keyboard and returns it if the next token is the correct data type.

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Method	Description
nextDouble()	Retrieves input as a double
nextInt()	Retrieves input as an int
nextLine()	Retrieves the next line of data and returns it as a String
next()	Retrieves the next complete token as a String
nextShort()	Retrieves input as a short
nextByte()	Retrieves input as a byte
nextFloat()	Retrieves input as a float. Note that when you enter an input value that will be stored as a float, you do not type an *F*. The *F* is used only with constants coded within a program.
nextLong()	Retrieves input as a long. Note that when you enter an input value that will be stored as a long, you do not type an *L*. The *L* is used only with constants coded within a program.

Table 2-7 Selected Scanner class methods

The Scanner class does not contain a nextChar() method. To retrieve a single character from the keyboard, you can use the nextLine() method and then use the charAt() method. The chapter *Characters, Strings, and the StringBuilder* provides more details about the charAt() method.

Figure 2-17 contains a program that uses two of the Scanner class methods. The program reads a string and an integer from the keyboard and displays them. The Scanner class is used in the four shaded statements in the figure.

- The first shaded statement is import java.util.Scanner;. This statement imports the package necessary to use the Scanner class.
- The second shaded statement declares a Scanner object named inputDevice.
- The third shaded statement uses the nextLine() method to retrieve a line of text from the keyboard and store it in the name variable.
- The last shaded statement uses the nextInt() method to retrieve an integer from the keyboard and store it in the age variable.

Figure 2-18 shows a typical execution of the program.

Java programmers would say that the Scanner methods *return* the appropriate value. That also means that the value of the method is the appropriate value, and that you can assign the returned value to a variable, display it, or use it in other legal statements. In the chapter *Using Methods, Classes, and Objects*, you will learn how to write your own methods that return values.

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

```
import java.util.Scanner;
public class GetUserInfo
{
   public static void main(String[] args)
   {
      String name;
      int age;
      Scanner inputDevice = new Scanner(System.in);
      System.out.print("Please enter your name >> ");
      name = inputDevice.nextLine();
      System.out.print("Please enter your age >> ");
      age = inputDevice.nextInt();
      System.out.println("Your name is " + name +
         " and you are " + age + " years old.");
   }
}
```

Repeating as output what a user has entered as input is called **echoing the input**. Echoing input is a good programming practice; it helps eliminate misunderstandings when the user can visually confirm what was entered.

Figure 2-17 The `GetUserInfo` class

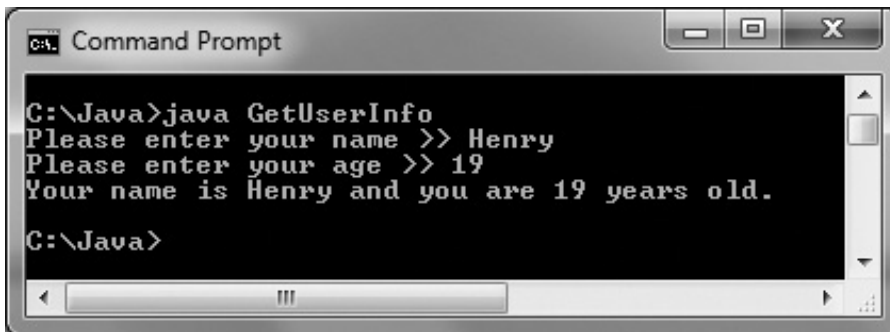


Figure 2-18 Typical execution of the `GetUserInfo` program

If you use any of the `Scanner` methods and the next token cannot be converted to the right data type, you receive an error message. For example, the program in Figure 2-17 uses `nextInt()` to retrieve `age`, so if the user entered a noninteger value for `age`, such as the `double` 19.5 or the `String` `"nineteen"`, an error would occur. You will learn how to recover from this type of error in the chapter *Exception Handling*, but for now, you will have to trust the user to enter the correct data type.

The literal `Strings` contained in the `print()` statements that appear before each input statement in Figure 2-17 are examples of prompts. A **prompt** is a message displayed for the user that requests and describes input. Interactive programs would work without prompts, but they would not be as user-friendly. Each prompt in the `GetUserInfo` class ends with two greater-than signs and a space. This punctuation is not required; it just separates the words in the prompt from the user's input value on the screen, improving readability. You might prefer to use a series of periods, several dashes, or just a few spaces.

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

It is legal to write a single prompt that requests multiple input values—for example, "Please enter your age, area code, and zip code." The user could then enter the three values separated with spaces, tabs, or Enter key presses. The values would then be interpreted as separate tokens and could be retrieved with three separate `nextInt()` method calls. However, asking a user to enter multiple values often leads to mistakes. This book will follow the practice of using a separate prompt for each input value required.

Pitfall: Using `nextLine()` Following One of the Other `Scanner` Input Methods

You can encounter a problem when you use one of the numeric `Scanner` class retrieval methods or the `next()` method before you use the `nextLine()` method. Consider the program in Figure 2-19. It is identical to the one in Figure 2-17, except that the user is asked for an age before being asked for a name. (See shading.) Figure 2-20 shows a typical execution.

```
import java.util.Scanner;
public class GetUserInfo2
{
   public static void main(String[] args)
   {
      String name;
      int age;
      Scanner inputDevice = new Scanner(System.in);
      System.out.print("Please enter your age >> ");
      age = inputDevice.nextInt();
      System.out.print("Please enter your name >> ");
      name = inputDevice.nextLine();
      System.out.println("Your name is " + name +
         " and you are " + age + " years old.");
   }
}
```

Don't Do It
If you accept numeric input prior to string input, the string input is ignored unless you take special action.

Figure 2-19 The `GetUserInfo2` class

```
Command Prompt
C:\Java>java GetUserInfo2
Please enter your age >> 28
Please enter your name >> Your name is  and you are 28 years old.

C:\Java>
```

Figure 2-20 Typical execution of the `GetUserInfo2` program

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

In Figure 2-20, the user is prompted correctly for an age. However, after the user enters an age and the prompt for the name is displayed, the program does not pause to let the user enter a name. Instead, the program proceeds directly to the output statement, which does not contain a valid name, as you can see in Figure 2-20.

When you type characters using the keyboard, they are stored temporarily in a location in memory called the **keyboard buffer**. The keyboard buffer sometimes is called the **type-ahead buffer**. All keystrokes are stored in the keyboard buffer, including the Enter key. The problem occurs because of a difference in the way the `nextLine()` method and the other `Scanner` retrieval methods work:

- The `Scanner` methods `next()`, `nextInt()`, and `nextDouble()` retrieve the next token in the buffer up to the next whitespace, which might be a space, tab, or Enter key.
- The `nextLine()` method reads all data up to the Enter key character.

So, in the execution of the program in Figure 2-20, the user is prompted for an age, types 28, and presses Enter. The call to the `nextInt()` method retrieves the 28 and leaves the Enter key press in the input buffer. Then the name prompt is displayed and the call to `nextLine()` retrieves the waiting Enter key before the user can type a name.

The solution to the problem is simple. After any `next()`, `nextInt()`, or `nextDouble()` call, you can add an extra `nextLine()` method call that will retrieve the abandoned Enter key character. Then, no matter what type of input follows, the program will execute smoothly. Figure 2-21 shows a program that contains just one change from Figure 2-19—the addition of the shaded statement that retrieves the abandoned Enter key character from the input buffer. Although you could assign the Enter key to a character variable, there is no need to do so. When you accept an entry and discard it without using it, programmers say that the entry is **consumed**. Figure 2-21 shows that the call to `nextInt()` accepts the integer, the first call to `nextLine()` accepts the Enter key that follows the integer entry, and the second `nextLine()` call accepts both the entered name and the Enter key that follows it. Figure 2-22 shows that the revised program executes correctly.

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

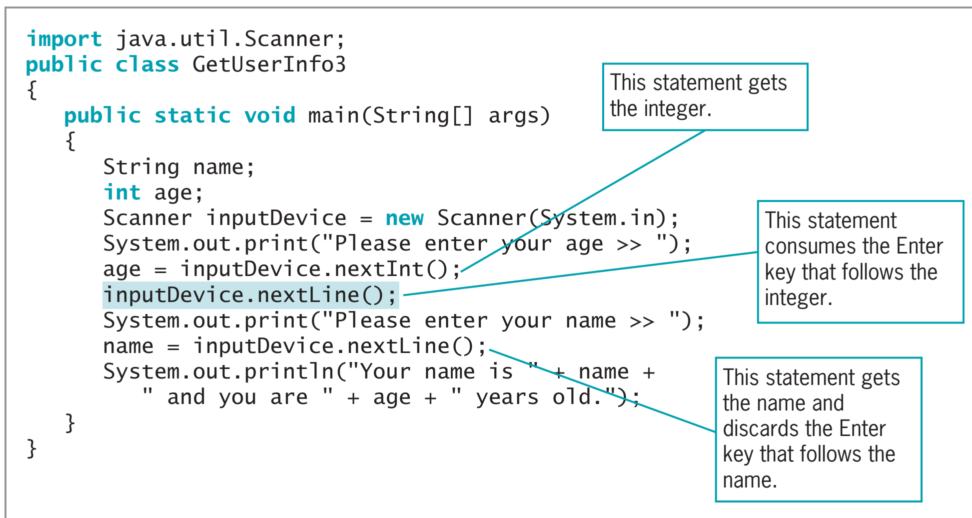


Figure 2-21 The `GetUserInfo3` class

Figure 2-22 Typical execution of the `GetUserInfo3` program

When you write programs that accept user input, there is a risk that the user will enter the wrong type of data. For example, if you include a `nextInt()` method call in your program, but the user types an alphabetic character, an error will occur, and your program will stop running. You will learn to handle this type of error later in this book.

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

TWO TRUTHS & A LIE

Using the Scanner Class to Accept Keyboard Input

1. `System.in` refers to the standard input device, which normally is the keyboard.
2. `System.in` is more flexible than `System.out` because it can read all the basic Java data types.
3. When a user types data followed by the Enter key, the Enter key character is left in the keyboard buffer after `Scanner` class methods retrieve the other keystrokes.

The false statement is #2. `System.in` is not as flexible as `System.out`. `System.out` can display various data types, but `System.in` is designed to read only bytes.

You Do It

Accepting User Input

In the next steps you create a program that accepts user input.

1. Open the **IntegerDemo.java** file you created in a "You Do It" section earlier in this chapter. Change the class name to **`IntegerDemoInteractive`**, and save the file as **IntegerDemoInteractive.java**.
2. As the first line in the file, insert an `import` statement that will allow you to use the `Scanner` class:

```
import java.util.Scanner;
```

3. Remove the assignment operator and the assigned values from each of the four numeric variable declarations.
4. Following the numeric variable declarations, insert a `Scanner` object declaration:

```
Scanner input = new Scanner(System.in);
```

5. Following the variable declarations, insert a prompt for the integer value, and an input statement that accepts the value, as follows:

```
System.out.print("Please enter an integer >> ");
anInt = input.nextInt();
```

(continues)

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

(continued)

6. Then add similar statements for the other three variables:

```
System.out.print("Please enter a byte integer >> ");
aByte = input.nextByte();
System.out.print("Please enter a short integer >> ");
aShort = input.nextShort();
System.out.print("Please enter a long integer >> ");
aLong = input.nextLong();
```

7. Save the file, and then compile and execute it. Figure 2-23 shows a typical execution. Execute the program a few more times, using different values each time and confirming that the correct values have been accepted from the keyboard.

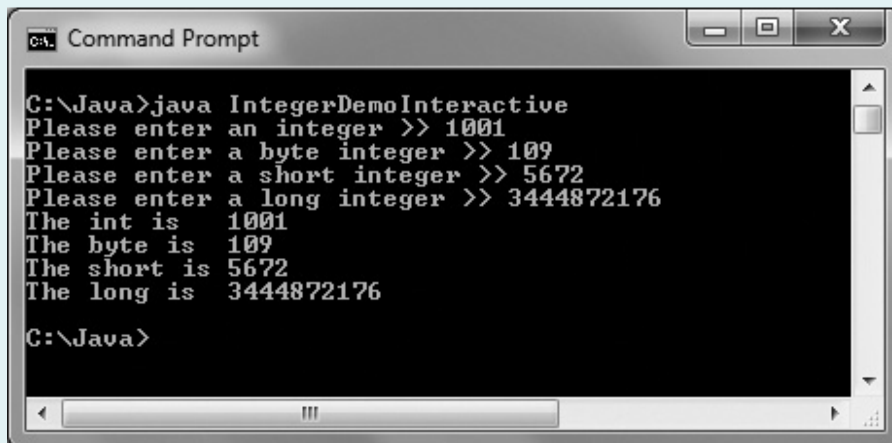


Figure 2-23 Typical execution of the `IntegerDemoInteractive` program

Adding `String` Input

Next, you add `String` input to the `IntegerDemoInteractive` program.

1. Change the class name of the `IntegerDemoInteractive` program to **`IntegerDemoInteractiveWithName`**, and immediately save the file as **IntegerDemoInteractiveWithName.java**.

2. Add a new variable with the other variable declarations as follows:

```
String name;
```

3. After the last input statement (that gets the value for `aLong`), add three statements that prompt the user for a name, accept the name, and use the name as follows:

```
System.out.print("Please enter your name >> ");
name = input.nextLine();
System.out.println("Thank you, " + name);
```

(continues)

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

(continued)

4. Save the file, and compile and execute it. Figure 2-24 shows a typical execution. You can enter the numbers, but when the prompt for the name appears, you are not given the opportunity to respond. Instead, the string "`Thank you`", including the ending comma and space, is output immediately, and the program ends. This output is incorrect because the input statement that should retrieve the name from the keyboard instead retrieves the Enter key that was still in the keyboard buffer after the last numeric entry.

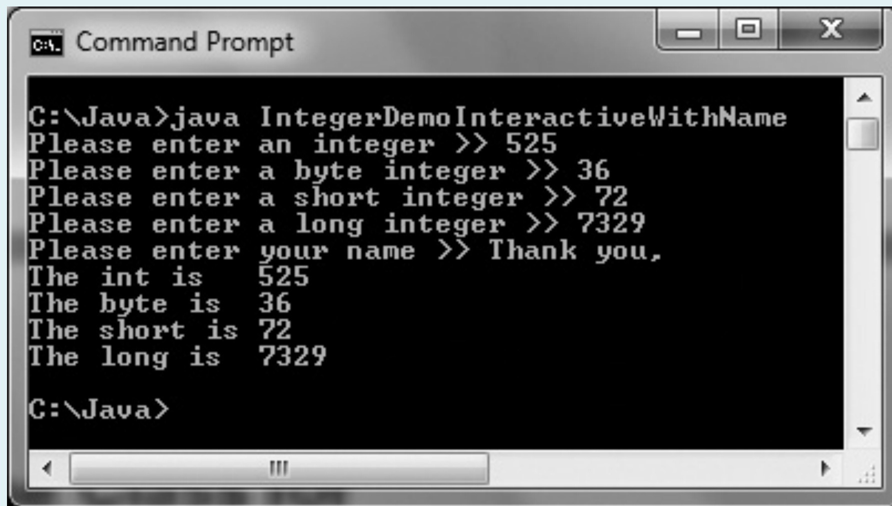


Figure 2-24 Typical execution of incomplete `IntegerDemoInteractiveWithName` application that does not accept a name

5. To fix the problem, insert an extra call to the `nextLine()` method just before the statement that accepts the name. This call will consume the Enter key. You do not need an assignment operator with this statement, because there is no need to store the Enter key character.

```
input.nextLine();
```

(continues)

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

(continued)

6. Save, compile, and execute the program. Figure 2-25 shows a typical successful execution.

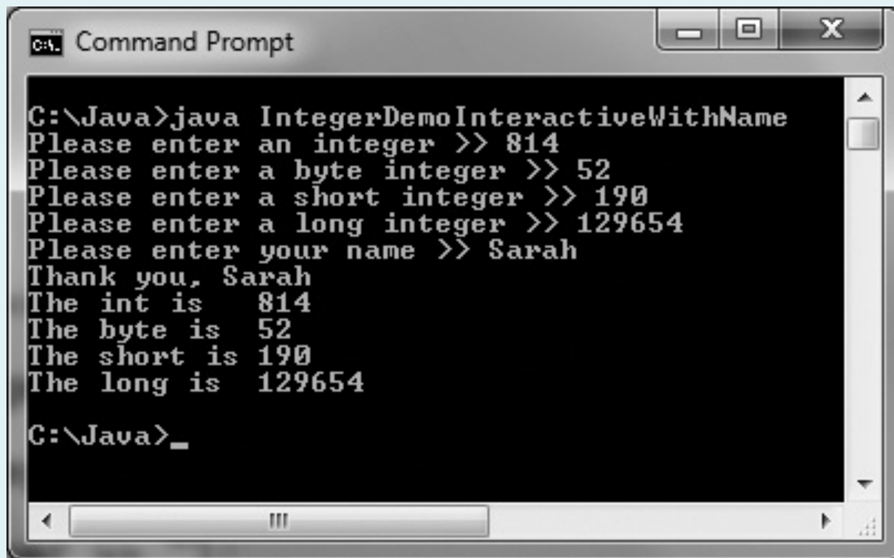


Figure 2-25 Typical successful execution of `IntegerDemoInteractiveWithName` application

Using the `JOptionPane` Class to Accept GUI Input

In Chapter 1, you learned how to display output at the command line and how to create GUI message boxes to display `String` objects. Earlier in this chapter, you learned to accept input from the keyboard at the command line. You also can accept input in a GUI dialog box using the `JOptionPane` class.

Two dialog boxes that can be used to accept user input are:

- `InputDialog`—Prompts the user for text input
- `ConfirmDialog`—Asks the user a question, providing buttons that the user can click for Yes, No, and Cancel responses

Using Input Dialog Boxes

An **input dialog box** asks a question and provides a text field in which the user can enter a response. You can create an input dialog box using the **`showInputDialog()` method**. Six versions of this method are available, but the simplest version uses a single argument that is the prompt

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

you want to display within the dialog box. The `showInputDialog()` method returns a `String` that represents a user's response; this means that you can assign the `showInputDialog()` method to a `String` variable and the variable will hold the value that the user enters.

For example, Figure 2-26 shows an application that creates an input dialog box containing a prompt for a first name. When the user executes the application, types "William", then clicks the OK button or presses Enter on the keyboard, the response `String` will contain "William". In the application in Figure 2-26, the response is concatenated with a welcoming message and displayed in a message dialog box. Figure 2-27 shows the dialog box containing a user's response, and Figure 2-28 shows the resulting output message box.

```
import javax.swing.JOptionPane;
public class HelloNameDialog
{
   public static void main(String[] args)
   {
      String result;
      result = JOptionPane.showInputDialog(null, "What is your name?");
      JOptionPane.showMessageDialog(null, "Hello, " + result + "!");
   }
}
```

Figure 2-26 The `HelloNameDialog` class

Figure 2-27 Input dialog box of the `HelloNameDialog` application

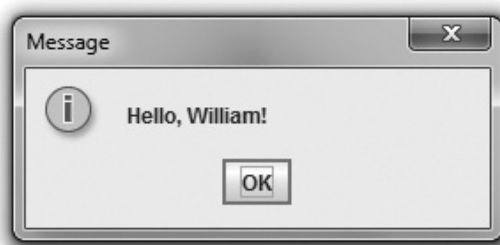


Figure 2-28 Output of the `HelloNameDialog` application

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Within the `JOptionPane` class, one version of the `showInputDialog()` method allows the programmer flexibility in controlling the appearance of the input dialog box. The version of `showInputDialog()` that requires four arguments can be used to display a title in the dialog box title bar and a message that describes the type of dialog box. The four arguments to `showInputDialog()` include:

- The parent component, which is the screen component, such as a frame, in front of which the dialog box will appear. If this argument is null, the dialog box is centered on the screen.
- The message the user will see before entering a value. Usually this message is a `String`, but it actually can be any type of object.
- The title to be displayed in the title bar of the input dialog box.
- A class field describing the type of dialog box; it can be one of the following:

  `ERROR_MESSAGE`, `INFORMATION_MESSAGE`, `PLAIN_MESSAGE`, `QUESTION_MESSAGE`, or `WARNING_MESSAGE`.

For example, when the following statement executes, it displays the input dialog box shown in Figure 2-29.

```
JOptionPane.showInputDialog(null,
   "What is your area code?",
   "Area code information",
   JOptionPane.QUESTION_MESSAGE);
```

Note that the title bar displays "Area code information," and the dialog box shows a question mark icon.

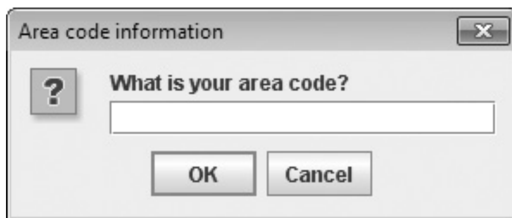


Figure 2-29 An input dialog box with a `String` in the title bar and a question mark icon

The `showInputDialog()` method returns a `String` object, which makes sense when you consider that you might want a user to type any combination of keystrokes into the dialog box. However, when the value that the user enters is intended to be used as a number, as in an arithmetic statement, the returned `String` must be converted to the correct numeric type. Later in this chapter, you will learn how to change primitive data from one data type to another. However, the techniques you will learn work only with primitive data types—`double`, `int`, `char`, and so on—not with class objects (that are reference types) such as a `String`. To convert a `String` to an `integer` or `double`, you must use methods from the built-in Java classes `Integer` and `Double`. Each primitive type in Java has a corresponding

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

class contained in the java.lang package; like most classes, the names of these classes begin with uppercase letters. These classes are called **type-wrapper classes**. They include methods that can process primitive type values.

Figure 2-30 shows a SalaryDialog application that contains two String objects—wageString and dependentsString. Two showInputDialog() methods are called, and the answers are stored in the declared Strings. The shaded statements in Figure 2-30 show how the Strings are converted to numeric values using methods from the type-wrapper classes Integer and Double. The double value is converted using the Double.parseDouble() method, and the integer is converted using the Integer.parseInt() method. Figure 2-31 shows a typical execution of the application.

Remember that in Java, the reserved keyword static means that a method is accessible and usable even though no objects of the class exist. You can tell that the method Double.parseDouble() is a static method, because the method name is used with the class name Double—no object is needed. Similarly, you can tell that Integer.parseInt() is also a static method.

The term **parse** means to break into component parts. Grammarians talk about "parsing a sentence"—deconstructing it so as to describe its grammatical components. Parsing a String converts it to its numeric equivalent.

```
import javax.swing.JOptionPane;
public class SalaryDialog
{
   public static void main(String[] args)
   {
       String wageString, dependentsString;
       double wage, weeklyPay;
       int dependents;
       final double HOURS_IN_WEEK = 37.5;
       wageString = JOptionPane.showInputDialog(null,
          "Enter employee's hourly wage", "Salary dialog 1",
          JOptionPane.INFORMATION_MESSAGE);
       weeklyPay = Double.parseDouble(wageString) *
          HOURS_IN_WEEK;
       dependentsString = JOptionPane.showInputDialog(null,
          "How many dependents?", "Salary dialog 2",
          JOptionPane.QUESTION_MESSAGE);
       dependents = Integer.parseInt(dependentsString);
       JOptionPane.showMessageDialog(null, "Weekly salary is $" +
          weeklyPay + "\nDeductions will be made for " +
          dependents + " dependents");
    }
}
```

Figure 2-30 The SalaryDialog class

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

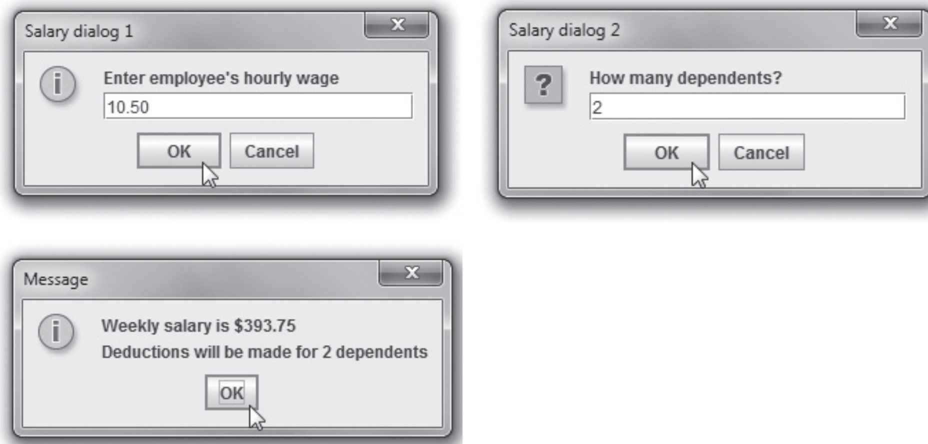


Figure 2-31 Sample execution of the `SalaryDialog` application

Using Confirm Dialog Boxes

Sometimes, the input you want from a user does not have to be typed from the keyboard. When you present simple options to a user, you can offer buttons that the user can click to confirm a choice. A **confirm dialog box** displays the options Yes, No, and Cancel; you can create one using the **`showConfirmDialog()` method** in the `JOptionPane` class. Four versions of the method are available; the simplest requires a parent component (which can be null) and the `String` prompt that is displayed in the box. The `showConfirmDialog()` method returns an integer containing one of three possible values: `JOptionPane.YES_OPTION`, `JOptionPane.NO_OPTION`, or `JOptionPane.CANCEL_OPTION`. Figure 2-32 shows an application that asks a user a question. The shaded statement displays the dialog box shown in Figure 2-33 and stores the user's response in the integer variable named `selection`.

```
import javax.swing.JOptionPane;
public class AirlineDialog
{
   public static void main (String[] args)
   {
      int selection;
      boolean isYes;
      selection = JOptionPane.showConfirmDialog(null,
         "Do you want to upgrade to first class?");
      isYes = (selection == JOptionPane.YES_OPTION);
      JOptionPane.showMessageDialog(null,
         "You responded " + isYes);
   }
}
```

Figure 2-32 The `AirlineDialog` class

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

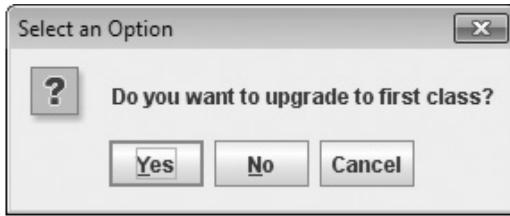


Figure 2-33 The confirm dialog box displayed by the `AirlineDialog` application

After a value is stored in `selection`, a Boolean variable named `isYes` is set to the result when `selection` and `JOptionPane.YES_OPTION` are compared. If the user has selected the Yes button in the dialog box, this variable is set to `true`; otherwise, the variable is set to `false`. Finally, the true or false result is displayed; Figure 2-34 shows the result when a user clicks the Yes button in the dialog box.

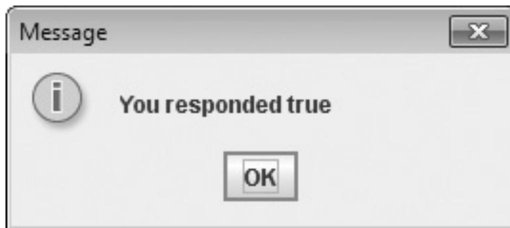


Figure 2-34 Output of `AirlineDialog` application when user clicks Yes

You can also create a confirm dialog box with five arguments, as follows:

- The parent component, which can be null
- The prompt message
- The title to be displayed in the title bar
- An integer that indicates which option button will be shown (It should be one of the class variables `YES_NO_CANCEL_OPTION` or `YES_NO_OPTION`.)
- An integer that describes the kind of dialog box (It should be one of the class variables `ERROR_MESSAGE`, `INFORMATION_MESSAGE`, `PLAIN_MESSAGE`, `QUESTION_MESSAGE`, or `WARNING_MESSAGE`.)

When the following statement is executed, it displays a confirm dialog box, as shown in Figure 2-35:

```
JOptionPane.showConfirmDialog(null,
      "A data input error has occurred. Continue?",
      "Data input error", JOptionPane.YES_NO_OPTION,
       JOptionPane.ERROR_MESSAGE);
```

Note that the title bar displays “Data input error,” the Yes and No buttons appear, and the dialog box shows the error message, “A data input error has occurred. Continue?” It also displays the octagonal ERROR_MESSAGE icon.

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Figure 2-35 Confirm dialog box with title, Yes and No buttons, and error icon

Confirm dialog boxes provide more practical uses when your applications can make decisions based on the users' responses. In the chapter *Making Decisions*, you will learn how to make decisions within programs.

TWO TRUTHS & A LIE

Using the `JOptionPane` Class to Accept GUI Input

1. You can create an input dialog box using the `showInputDialog()` method; the method returns a `String` that represents a user's response.
2. You can use methods from the Java classes `Integer` and `Double` when you want to convert a dialog box's returned values to numbers.
3. A confirm dialog box can be created using the `showConfirmDialog()` method in the `JOptionPane` class; a confirm dialog box displays the options Accept, Reject, and Escape.

The false statement is #3. A confirm dialog box displays the options Yes, No, and Cancel.

Watch the video *Getting Input*.

Performing Arithmetic

Table 2-8 describes the five **standard arithmetic operators** that you use to perform calculations with values in your programs. A value used on either side of an operator is an **operand**. For example, in the expression 45 + 2, the numbers 45 and 2 are operands. The arithmetic operators are examples of **binary operators**, so named because they require two operands.

You will learn about the Java shortcut arithmetic operators in the chapter *Looping*.

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Operator	Description	Example
+	Addition	45 + 2, the result is 47
–	Subtraction	45 – 2, the result is 43
*	Multiplication	45 * 2, the result is 90
/	Division	45.0 / 2, the result is 22.5 45 / 2, the result is 22 (not 22.5)
%	Remainder (modulus)	45 % 2, the result is 1 (that is, 45 / 2 = 22 with a remainder of 1)

Table 2-8 Arithmetic operators

The operators / and % deserve special consideration. Java supports two types of division:

- **Floating-point division** occurs when either or both of the operands are floating-point values. For example, 45.0 / 2 is 22.5.
- **Integer division** occurs when both of the operands are integers. The result is an integer, and any fractional part of the result is lost. For example, the result of 45 / 2 is 22. As another example, 39 / 5 is 7 because 5 goes into 39 seven whole times; 38 / 5, 37 / 5, 36 / 5, and 35 / 5 all evaluate to 7.

The percent sign is the **remainder operator**. The remainder operator is most often used with two integers, and the result is an integer with the value of the remainder after division takes place. For example, the result of 45 % 2 is 1 because 2 "goes into" 45 twenty-two times with a remainder of 1. Other examples of remainder operations include the following:

- 39 % 5 is 4 because 5 goes into 39 seven times with a remainder of 4.
- 20 % 3 is 2 because when 20 is divided by 3, the remainder is 2. 36 % 4 is 0 because there is no remainder when 4 is divided into 36.

Note that when you perform paper-and-pencil division, you divide first to determine a remainder. In Java, you do not need to perform a division operation before you can perform a remainder operation. A remainder operation can stand alone.

Although the remainder operator is most often used with integers, it is legal but less often useful to use the operator with floating-point values. In Java, when you use the % operator with floating-point values, the result is the remainder from a rounded division.

The remainder operator is also called the **modulus operator**, or sometimes just **mod**. Mathematicians would argue that *remainder* is the better term because in Java, the result of using the remainder operator can be negative, but in mathematics, the result of a modulus operation can never be negative.

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Associativity and Precedence

When you combine mathematical operations in a single statement, you must understand both associativity and precedence. The associativity of arithmetic operators with the same precedence is left-to-right. In a statement such as `answer = x + y + z;`, the `x` and `y` are added first, producing a temporary result, and then `z` is added to the temporary sum. After the sum is computed, the result is assigned to `answer`.

Operator precedence refers to the rules for the order in which parts of a mathematical expression are evaluated. The multiplication, division, and remainder operators have the same precedence. Their precedence is higher than that of the addition and subtraction operators. Addition and subtraction have the same precedence, and their precedence is lower than that of the other operators. In other words, expressions are evaluated from left to right, and multiplication, division, and remainder always take place prior to addition or subtraction. Table 2-9 summarizes the precedence of the arithmetic operators.

Operators	Descriptions	Relative Precedence
* / %	Multiplication, division, remainder	Higher
+ –	Addition, subtraction	Lower

Table 2-9 Relative precedence of arithmetic operators

For example, the following statement assigns 14 to `result`:

```
int result = 2 + 3 * 4;
```

The multiplication operation (3 * 4) occurs before adding 2. You can override normal operator precedence by putting the operation to perform first in parentheses. The following statement assigns 20 to `result`:

```
int result = (2 + 3) * 4;
```

The addition within the parentheses takes place first, and then the intermediate result (5) is multiplied by 4. When multiple pairs of parentheses are used in a statement, the innermost expression surrounded by parentheses is evaluated first. For example, the value of the following expression is 46:

```
2 * (3 + (4 * 5))
```

First, 4 * 5 evaluates to 20, and then 3 is added, giving 23. Finally, the value is multiplied by 2, giving 46.

Remembering that *, /, and % have the same precedence is important in arithmetic calculations. These operations are performed from left to right, regardless of the order in which they appear. For example, the value of the following expression is 9:

```
25 / 8 * 3
```

First, 25 is divided by 8. The result is 3 because with integer division, you lose any remainder. Then 3 is multiplied by 3, giving 9. If you assumed that multiplication was performed before division, you would calculate an incorrect answer.

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

You will learn more about operator precedence in the chapter *Making Decisions*.

Writing Arithmetic Statements Efficiently

You can make your programs operate more efficiently if you avoid unnecessary repetition of arithmetic statements. For example, suppose you know the values for an employee's hourly pay and pay rate and you want to compute state and federal withholding tax based on known rates. You could write two statements as follows:

```
stateWithholding = hours * rate * STATE_RATE;
federalWithholding = hours * rate * FED_RATE;
```

With this approach, you perform the multiplication of `hours * rate` twice. It is more efficient to perform the calculation once, as follows:

```
grossPay = hours * rate;
stateWithholding = grossPay * STATE_RATE;
federalWithholding = grossPay * FED_RATE;
```

The time saved is very small, but these savings would be more important if the calculation was more complicated or if it was repeated many times in a program. As you think about the programs you write, remain on the lookout for ways to improve efficiency by avoiding duplication of operations.

Pitfall: Not Understanding Imprecision in Floating-Point Numbers

Integer values are exact, but floating-point numbers frequently are only approximations. For example, when you divide 1.0 by 3.0, the mathematical result is 0.3333333…, with the *3*s continuing infinitely. No matter how many decimal places you can store, the result is only an approximation. Even values that don't repeat indefinitely in our usual numbering system, such as 0.1, cannot be represented precisely in the binary format used by computers. Imprecision leads to several problems:

Appendix B provides a more thorough explanation of numbering systems and why fractional values cannot be represented accurately.

- When you produce floating-point output, it might not look like what you expect or want.
- When you make comparisons with floating-point numbers, the comparisons might not be what you expect or want.

For example, Figure 2-36 shows a class in which an answer is computed as 2.20 – 2.00. Mathematically, the result should be 0.20. But, as the output in Figure 2-37 shows, the result

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

is calculated as a value that is slightly more than 0.20, and when `answer` is compared to 0.20, the result is false.

```
public class ImprecisionDemo
{
   public static void main(String[] args)
   {
       double answer = 2.20 - 2.00;
       boolean isEqual = answer == 0.20;
       System.out.println("answer is " + answer);
       System.out.println("isEqual is " + isEqual);
   }
}
```

Figure 2-36 The `ImprecisionDemo` program

```
Command Prompt
C:\Java>java ImprecisionDemo
answer is 0.20000000000000018
isEqual is false

C:\Java>
```

Figure 2-37 Execution of the `ImprecisionDemo` program

For now, you might choose to accept the slight imprecisions generated when you use floating-point numbers. However, if you want to eliminate the imprecisions, you can use one of several techniques to round values. Appendix C contains directions on how to round numbers and how to format a floating-point number so it displays the desired number of decimal positions.

Several movies have used the fact that floating-point numbers are not precise as a plot element. For example, in the movies *Superman III* and *Office Space*, thieves round currency values and divert the remaining fractions of cents to their own accounts.

Watch the video *Arithmetic*.

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

TWO TRUTHS & A LIE

Performing Arithmetic

1. The arithmetic operators are examples of unary operators, which are so named because they perform one operation at a time.
2. In Java, operator precedence dictates that multiplication, division, and remainder always take place prior to addition or subtraction in an expression.
3. Floating-point arithmetic might produce imprecise results.

The false statement is #1. The arithmetic operators are examples of binary operators, which are so named because they require two operands.

You Do It

Using Arithmetic Operators

In these steps, you create a program that uses arithmetic operators.

1. Open a new file in your text editor, and type the `import` statement needed for interactive input with the `Scanner` class:

```
import java.util.Scanner;
```

2. Type the class header and its curly braces for a class named `ArithmeticDemo`. Within the class's curly braces, enter the `main()` method header and its braces.

```
public class ArithmeticDemo
{
   public static void main(String[] args)
   {
   }
}
```

3. With the `main()` method, declare five `int` variables that will be used to hold two input values and their sum, difference, and average:

```
int firstNumber;
int secondNumber;
int sum;
int difference;
int average;
```

(continues)

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

(continued)

4. Also declare a `Scanner` object so that keyboard input can be accepted.

```
Scanner input = new Scanner(System.in);
```

5. Prompt the user for and accept two integers:

```
System.out.print("Please enter an integer >> ");
firstNumber = input.nextInt();
System.out.print("Please enter another integer >> ");
secondNumber = input.nextInt();
```

6. Add statements to perform the necessary arithmetic operations:

```
sum = firstNumber + secondNumber;
difference = firstNumber - secondNumber;
average = sum / 2;
```

7. Display the three calculated values:

```
System.out.println(firstNumber + " + " +
   secondNumber + " is " + sum);
System.out.println(firstNumber + " - " +
   secondNumber + " is " + difference);
System.out.println("The average of " + firstNumber +
   " and " + secondNumber + " is " + average);
```

8. Save the file as **ArithmeticDemo.java**, and then compile and execute it. Enter values of your choice. Figure 2-38 shows a typical execution. Notice that because integer division was used to compute the average, the answer is an integer.

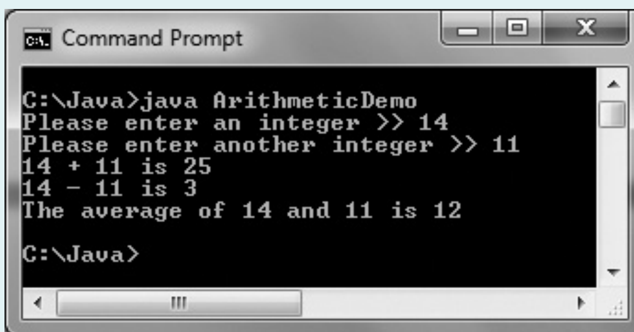


Figure 2-38 Typical execution of `ArithmeticDemo` application

9. Execute the program multiple times using various integer values and confirm that the results are accurate.

(continues)

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

(continued)

Performing Floating-Point Arithmetic

Next, you will modify the `ArithmeticDemo` application to work with floating-point values instead of integers.

1. Within the `ArithmeticDemo` application, change the class name to **`ArithmeticDemo2`**, and immediately save the file as **ArithmeticDemo2.java**. Change all the variables' data types to `double`. Change the two prompts to request `double` values, and change the two calls to the `nextInt()` method to `nextDouble()`. Save, compile, and execute the program again. Figure 2-39 shows a typical execution. Notice that the average calculation now includes decimal places.

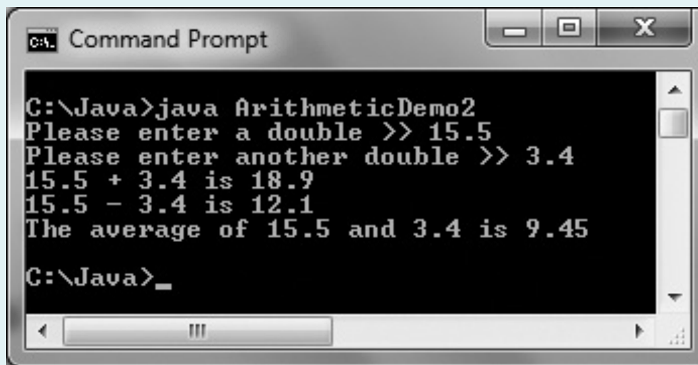


Figure 2-39 Typical execution of the `ArithmeticDemo2` application

2. Rerun the program, experimenting with various input values. Some of your output might appear with imprecisions similar to those shown in Figure 2-40. If you are not satisfied with the slight imprecisions created when using floating-point arithmetic, you can round or change the display of the values, as discussed in Appendix C.

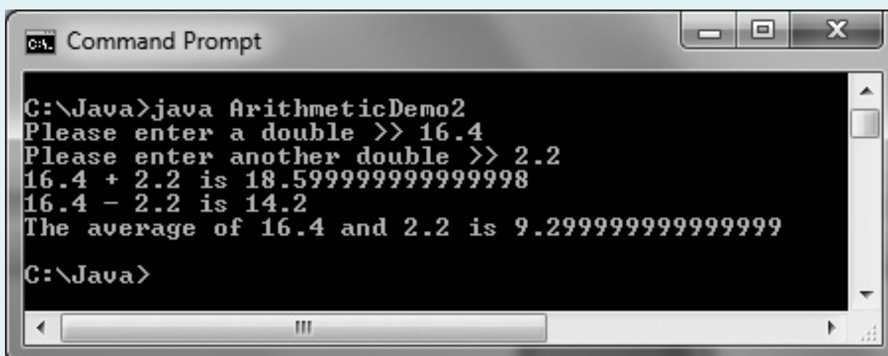


Figure 2-40 Another typical execution of the `ArithmeticDemo2` application

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Understanding Type Conversion

When you perform arithmetic with variables or constants of the same type, the result of the operation retains the same type. For example, when you divide two `int`s, the result is an `int`, and when you subtract two `double`s, the result is a `double`. Often, however, you might want to perform mathematical operations on operands with unlike types. The process of converting one data type to another is **type conversion**. Java performs some conversions for you automatically or implicitly, but other conversions must be requested explicitly by the programmer.

Automatic Type Conversion

When you perform arithmetic operations with operands of unlike types, Java chooses a unifying type for the result. The **unifying type** is the type to which all operands in an expression are converted so that they are compatible with each other. Java performs an **implicit conversion**; that is, it automatically converts nonconforming operands to the unifying type. Implicit conversions also are called **promotions**. Figure 2-41 shows the order for establishing unifying types between values.

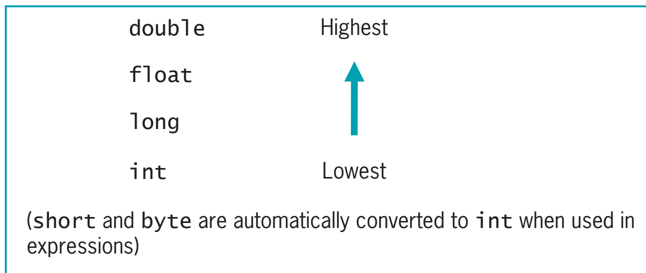


Figure 2-41 Order for establishing unifying data types

When two unlike types are used in an expression, the unifying type is the one that is higher in the list in Figure 2-41. In other words, when an operand that is a type lower on the list is combined with a type that is higher, the lower-type operand is converted to the higher one. For example, the addition of a `double` and an `int` results in a `double`, and the subtraction of a `long` from a `float` results in a `float`.

Boolean values cannot be converted to another type. In some languages, such as C++, Boolean values are actually numbers. However, this is not the case in Java.

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

For example, assume that an `int`, `hoursWorked`, and a `double`, `payRate`, are defined and then multiplied as follows:

```
int hoursWorked = 37;
double payRate = 6.73;
double grossPay = hoursWorked * payRate;
```


The result of the multiplication is a `double` because when a `double` and an `int` are multiplied, the `int` is promoted to the higher-ranking unifying type `double`—the type that is higher in the list in Figure 2-41. Therefore, assigning the result to `grossPay` is legal.

The following code will not compile because `hoursWorked` times `payRate` is a `double`, and Java does not allow the loss of precision that occurs if you try to store the calculated `double` result in an `int`.

```
int hoursWorked = 37;
double payRate = 6.73;
int grossPay = hoursWorked * payRate;
```

The data types `char`, `short`, and `byte` all are promoted to `int` when used in statements with unlike types. If you perform a calculation with any combination of `char`, `short`, and `byte` values, the result is an `int` by default. For example, if you add two `byte`s, the result is an `int`, not a `byte`.

Explicit Type Conversions

You can purposely override the unifying type imposed by Java by performing a type cast. **Type casting** forces a value of one data type to be used as a value of another type. To perform a type cast, you use a **cast operator**, which is created by placing the desired result type in parentheses. Using a cast operator is an **explicit conversion**. The cast operator is followed by the variable or constant to be cast. For example, a type cast is performed in the following code:

```
double bankBalance = 189.66;
float weeklyBudget = (float) (bankBalance / 4);
  // weeklyBudget is 47.415, one-fourth of bankBalance
```

The cast operator is more completely called the **unary cast operator**. Unlike a binary operator that requires two operands, a **unary operator** uses only one operand. The unary cast operator is followed by its operand.

In this example, the `double` value `bankBalance` is divided by the integer 4, and the result is a `double`. Then the `double` result is converted to a `float` before it is stored in `weeklyBudget`.

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Without the conversion, the statement that assigns the result to `weeklyBudget` would not compile. Similarly, a cast from a `float` to an `int` occurs in this code segment:

```
float myMoney = 47.82f;
int dollars = (int) myMoney;
  // dollars is 47, the integer part of myMoney
```

In this example, the `float` value `myMoney` is converted to an `int` before it is stored in the integer variable named `dollars`. When the `float` value is converted to an `int`, the decimal place values are lost. The cast operator does not permanently alter any variable's data type; the alteration is only for the duration of the current operation.

The word *cast* is used in a similar fashion when referring to molding metal, as in *cast iron*. In a Java arithmetic cast, a value is "molded" into a different type.

It is easy to lose data when performing a cast. For example, the largest `byte` value is 127 and the largest `int` value is 2,147,483,647, so the following statements produce distorted results:

```
int anOkayInt = 200;

byte aBadByte = (byte)anOkayInt;
```

A `byte` is constructed from eight *1*s and *0*s, or binary digits. The first binary digit, or bit, holds a 0 or 1 to represent positive or negative. The remaining seven bits store the actual value. When the integer value 200 is stored in the `byte` variable, its large value consumes the eighth bit, turning it to a 1, and forcing the `aBadByte` variable to appear to hold the value –72, which is inaccurate and misleading.

You do not need to perform a cast when assigning a value to a higher unifying type. For example, when you write a statement such as the following, Java automatically promotes the integer constant 10 to be a `double` so that it can be stored in the `payRate` variable:

```
double payRate = 10;
```

However, for clarity, if you want to assign 10 to `payRate`, you might prefer to write the following:

```
double payRate = 10.0;
```

The result is identical whether you assign the literal `double` 10.0 or the literal `int` 10 to the `double` variable.

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

TWO TRUTHS & A LIE

Understanding Type Conversion

1. When you perform arithmetic operations with operands of unlike types, you must make an explicit conversion to a unifying type.
2. Summing a `double`, `int`, and `float` results in a `double`.
3. You can explicitly override the unifying type imposed by Java by performing a type cast; type casting forces a value of one data type to be used as a value of another type.

The false statement is #1. When you perform arithmetic operations with operands of unlike types, Java performs an implicit conversion to a unifying type.

You Do It

Implicit and Explicit Casting

In this section you explore the concepts of the unifying types and casting.

1. Open the **ArithmeticDemo.java** file that uses integer values to calculate a sum, difference, and average. Change the class name to **`ArithmeticDemo3`**, and immediately save the file as **ArithmeticDemo3.java**.
2. In the previous version of the program, the average was calculated without decimal places because when two integers are divided, the result is an integer. To compute a more accurate average, change the data type for the average variable from `int` to **`double`**.
3. Save, compile, and execute the program. As the sample execution in Figure 2-42 shows, the program compiles and executes, but the average is still not accurate. The average of 20 and 19 is calculated to be just 19 because the decimal portion of the arithmetic result is lost. The program executes because the result of an integer divided by an integer is an integer, and when the integer is assigned to the `double`, automatic type conversion takes place.

(continues)

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

(continued)

Figure 2-42 Typical execution of `ArithmeticDemo3` application

4. Change the statement that computes the average to include a cast as follows:

```
average = (double) sum / 2;
```

5. Save, compile, and execute the program. As shown in Figure 2-43, now the program displays a more accurate average. The integer `sum` has been cast to a `double`, and when the `double` is divided by the integer, the result is a `double`, which is then assigned to `average`.

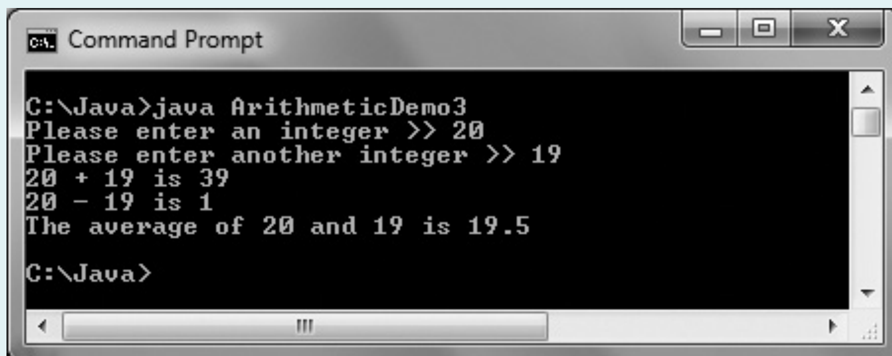


Figure 2-43 Typical execution of `ArithmeticDemo3` application after addition of a cast operation for the average

6. Change the statement that computes the average to include a second set of parentheses, as follows:

```
average = (double) (sum / 2);
```

(continues)

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

(continued)

7. Save, compile, and execute the program. Now, the fractional portion of the result is omitted again. That's because the result of `sum / 2` is calculated first, and the result is an integer. Then the whole-number result is cast to a `double` and assigned to a `double`—but the fractional part of the answer was already lost and casting is too late. Remove the newly added parentheses, save the program, compile it, and execute it again to confirm that the fractional part of the answer is reinstated.

8. As an alternative to the explicit cast in the division statement in the `ArithmeticDemo` program, you could write the average calculation as follows:

```
average = sum / 2.0;
```

   In this calculation, when the integer `sum` is divided by the `double` constant 2.0, the result is a `double`. The result then does not require any cast to be assigned to the `double average` without loss of data. Try this in your program.

9. Go to the Java Web site (***www.oracle.com/technetwork/java/index.html***), select **Java APIs**, and then select **Java SE 7**. Scroll through the list of **All Classes**, and select **PrintStream**, which is the data type for the `out` object used with the `println()` method. Scroll down to view the list of methods in the Method Summary. As you did in a previous exercise, notice the many versions of the `print()` and `println()` methods, including ones that accept a `String`, an `int`, and a `long`. Notice, however, that no versions accept a `byte` or a `short`. That's because when a `byte` or `short` is sent to the `print()` or `println()` method, it is automatically promoted to an `int`, so that version of the method is used.

Don't Do It

- Don't mispronounce "integer." People who are unfamiliar with the term often say "interger," inserting an extra *r*.
- Don't attempt to assign a literal constant floating-point number, such as 2.5, to a `float` without following the constant with an uppercase or lowercase *F*. By default, constant floating-point values are `double`s.
- Don't try to use a Java keyword as an identifier for a variable or constant. Table 1-1 in Chapter 1 contains a list of Java keywords.
- Don't attempt to assign a constant value under –2,147,483,648 or over +2,147,483,647 to a `long` variable without following the constant with an uppercase or lowercase *L*. By default, constant integers are `int`s, and a value under –2,147,483,648 or over 2,147,483,647 is too large to be an `int`.
- Don't assume that you must divide numbers as a step to determining a remainder; the remainder operator (%) is all that's needed.

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

- Don't try to use a variable or named constant that has not yet been assigned a value.
- Don't forget to consume the Enter key after numeric input using the `Scanner` class when a `nextLine()` method call follows.
- Don't forget to use the appropriate `import` statement when using the `Scanner` or `JOptionPane` class.
- Don't forget precedence rules when you write statements that contain multiple arithmetic operations. For example, `score1 + score2 / 2` does not compute the average of two scores. Instead, it adds half of `score2` to `score1`. To compute the average, you would write `(score1 + score2) / 2`.
- Don't forget that integer division results in an integer, dropping any fractional part. For example, 1 / 2 is not equal to 0.5; it is equal to 0.
- Don't forget that extra parentheses can change the result of an operation that includes casting.
- Don't forget that floating-point numbers are imprecise.
- Don't attempt to assign a constant decimal value to an integer using a leading 0. For example, if you declare `int num = 021;` and then display `num`, you will see 17. The leading 0 indicates that the value is in base 8 (octal), so its value is two *8*s plus one *1*. In the decimal system, 21 and 021 mean the same thing, but not in Java.
- Don't use a single equal sign (=) in a Boolean comparison for equality. The operator used for equivalency is composed of two equal signs (==).
- Don't try to store a string of characters, such as a name, in a `char` variable. A `char` variable can hold only a single character.
- Don't forget that when a `String` and a numeric value are concatenated, the resulting expression is a string. For example, `"X" + 2 + 4` results in `"X24"`, *not* `"X6"`. If you want the result to be `"X6"`, you can use the expression `"X" + (2 + 4)`.

Key Terms

Constant describes values that cannot be changed during the execution of an application.

A **literal constant** is a value that is taken literally at each use.

A **numeric constant** is a number whose value is taken literally at each use.

An **unnamed constant** has no identifier associated with it.

A **variable** is a named memory location that you can use to store a value.

An item's **data type** describes the type of data that can be stored there, how much memory the item occupies, and what types of operations can be performed on the data.

A **primitive type** is a simple data type. Java's primitive types are `byte`, `short`, `int`, `long`, `float`, `double`, `char`, and `boolean`.

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Reference types are complex data types that are constructed from primitive types.

A **variable declaration** is a statement that reserves a named memory location.

A **strongly typed language** is one in which each variable has a well-defined type that limits the operations you can perform with it; strong typing implies that variables must be declared before they can be used.

Camel casing is a style in which an identifier begins with a lowercase letter and subsequent words within the identifier are capitalized.

The **assignment operator** is the equal sign (=); any value to the right of the equal sign is assigned to the variable on the left of the equal sign.

An **initialization** is an assignment made when you declare a variable.

An **assignment** is the act of providing a value for a variable.

Associativity refers to the order in which operands are used with operators.

An **lvalue** is an expression that can appear on the left side of an assignment statement.

An **rvalue** is an expression that can appear only on the right side of an assignment statement.

An **uninitialized variable** is one that has not been assigned a value.

A **garbage value** is the unknown value stored in an uninitialized variable.

A **named constant** is a named memory location whose value cannot change after it is assigned.

A **symbolic constant** is a named constant.

The keyword **`final`** precedes named constant declarations.

A **blank `final`** is a `final` variable that has not yet been assigned a value.

A **magic number** is a value that does not have immediate, intuitive meaning or a number that cannot be explained without additional knowledge. Unnamed constants are magic numbers.

The **scope** of a data item is the area in which it is visible to a program and in which you can refer to it using its simple identifier.

A **block of code** is the code contained between a set of curly braces.

Concatenated describes values that are attached end to end.

A **null `String`** is an empty `String` created by typing a set of quotes with nothing between them.

An **integer** is a whole number without decimal places.

The **`int`** data type is used to declare variables and constants that store integers in the range of –2,147,483,648 to +2,147,483,647.

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

The **byte** data type holds very small integers, from –128 to 127.

The **short** data type holds small integers, from –32,768 to 32,767.

The **long** data type holds very large integers, from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

A **boolean variable** can hold only one of two values—true or false.

A **relational operator** compares two items; an expression that contains a relational operator has a Boolean value.

A **comparison operator** is another name for a relational operator.

A **floating-point** number contains decimal positions.

A **float** data type can hold a floating-point value of up to six or seven significant digits of accuracy.

A **double** data type can hold a floating-point value of up to 14 or 15 significant digits of accuracy.

Significant digits refers to the mathematical accuracy of a value.

Scientific notation is a display format that more conveniently expresses large or small numeric values; a multidigit number is converted to a single-digit number and multiplied by 10 to a power.

A **double-precision floating-point number** is stored in a double.

A **single-precision floating-point number** is stored in a float.

The **char** data type is used to hold any single character.

String is a built-in Java class that provides you with the means for storing and manipulating character strings.

An **escape sequence** begins with a backslash followed by a character; the pair represents a single character.

The **standard input device** normally is the keyboard.

A **token** is a unit of data separated with whitespace.

A **prompt** is a message that requests and describes user input.

Echoing the input means to repeat the user's entry as output so the user can visually confirm the entry's accuracy.

The **keyboard buffer** is a small area of memory where keystrokes are stored before they are retrieved into a program.

The **type-ahead buffer** is the keyboard buffer.

To **consume** an entry is to retrieve and discard it without using it.

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

An **input dialog box** asks a question and provides a text field in which the user can enter a response.

The **`showInputDialog()` method** creates an input dialog box.

Type-wrapper classes, contained in the `java.lang` package, include methods that can process primitive type values.

To **parse** means to break into component parts.

A **confirm dialog box** displays the options Yes, No, and Cancel; you can create one using the **`showConfirmDialog()` method** in the `JOptionPane` class.

Standard arithmetic operators are used to perform calculations with values.

An **operand** is a value used in an arithmetic statement.

Binary operators require two operands.

Floating-point division is the operation in which two values are divided and either or both are floating-point values.

Integer division is the operation in which two values are divided and both are integers; the result contains no fractional part.

The **remainder operator** is the percent sign; when it is used with two integers, the result is an integer with the value of the remainder after division takes place.

The **modulus operator**, sometimes abbreviated as **mod**, is an alternate name for the remainder operator.

Operator precedence is the rules for the order in which parts of a mathematical expression are evaluated.

Type conversion is the process of converting one data type to another.

A **unifying type** is a single data type to which all operands in an expression are converted.

An **implicit conversion** is the automatic transformation of one data type to another.

Promotion is an implicit conversion.

Type casting forces a value of one data type to be used as a value of another type.

A **cast operator** performs an explicit type conversion; it is created by placing the desired result type in parentheses before the expression to be converted.

An **explicit conversion** is the data type transformation caused using a cast operator.

The **unary cast operator** is a more complete name for the cast operator that performs explicit conversions.

A **unary operator** uses only one operand.

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Chapter Summary

- Variables are named memory locations in which programs store values; the value of a variable can change. You must declare all variables you want to use in a program by providing a data type and a name. Java provides for eight primitive types of data: `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, and `short`. A named constant is a memory location that holds a value that can never be changed; it is preceded by the keyword `final`.
- A variable of type `int` can hold any whole number value from –2,147,483,648 to +2,147,483,647. The types `byte`, `short`, and `long` are all variations of the integer type.
- A `boolean` type variable can hold a `true` or `false` value. Java supports six relational operators: >, <, ==, >=, <=, and !=.
- A floating-point number contains decimal positions. Java supports two floating-point data types: `float` and `double`.
- You use the `char` data type to hold any single character. You type constant character values in single quotation marks. You type `String` constants that store more than one character between double quotation marks. You can store some characters using an escape sequence, which always begins with a backslash.
- You can use the `Scanner` class and the `System.in` object to accept user input from the keyboard. Several methods are available to convert input to usable data, including `nextDouble()`, `nextInt()`, and `nextLine()`.
- You can accept input using the `JOptionPane` class. The `showInputDialog()` method returns a `String`, which must be converted to a number using a type-wrapper class before you can use it as a numeric value.
- There are five standard arithmetic operators: +, –, *, /, and %. Operator precedence is the order in which parts of a mathematical expression are evaluated. Multiplication, division, and remainder always take place prior to addition or subtraction in an expression. Right and left parentheses can be added within an expression when exceptions to this rule are required. When multiple pairs of parentheses are added, the innermost expression surrounded by parentheses is evaluated first.
- When you perform mathematical operations on unlike types, Java implicitly converts the variables to a unifying type. You can explicitly override the unifying type imposed by Java by performing a type cast.

Review Questions

1. When data cannot be changed after a class is compiled, the data is ________.

   a. constant　　c. volatile
   b. variable　　d. mutable

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

2. Which of the following is not a primitive data type in Java?
   a. `boolean`
   b. `byte`
   c. `int`
   d. `sector`

3. Which of the following elements is not required in a variable declaration?
   a. a type
   b. an identifier
   c. an assigned value
   d. a semicolon

4. The assignment operator in Java is ______________.
   a. =
   b. ==
   c. :=
   d. ::

5. Assuming you have declared `shoeSize` to be a variable of type `int`, which of the following is a valid assignment statement in Java?
   a. `shoeSize = 9;`
   b. `shoeSize = 9.5;`
   c. `shoeSize = '9';`
   d. `shoeSize = "nine";`

6. Which of the following data types can store a value in the least amount of memory?
   a. `short`
   b. `long`
   c. `int`
   d. `byte`

7. A `boolean` variable can hold ______________.
   a. any character
   b. any whole number
   c. any decimal number
   d. the value `true` or `false`

8. The value 137.68 can be held by a variable of type ______________.
   a. `int`
   b. `float`
   c. `double`
   d. Two of the preceding answers are correct.

9. An escape sequence always begins with a(n) ______________.
   a. e
   b. forward slash
   c. backslash
   d. equal sign

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

10. Which Java statement produces the following output?

```
w
xyz
```

    a. `System.out.println("wxyz");`
    b. `System.out.println("w" + "xyz");`
    c. `System.out.println("w\nxyz");`
    d. `System.out.println("w\nx\ny\nz");`

11. The remainder operator ________.
    a. is represented by a forward slash
    b. must follow a division operation
    c. provides the quotient of integer division
    d. is none of the above

12. According to the rules of operator precedence, when division occurs in the same arithmetic statement as ________, the division operation always takes place first.
    a. multiplication
    b. remainder
    c. subtraction
    d. Answers a and b are correct.

13. The "equal to" relational operator is ________.
    a. =
    b. ==
    c. !=
    d. !!

14. When you perform arithmetic with values of diverse types, Java ________.
    a. issues an error message
    b. implicitly converts the values to a unifying type
    c. requires you to explicitly convert the values to a unifying type
    d. implicitly converts the values to the type of the first operand

15. If you attempt to add a `float`, an `int`, and a `byte`, the result will be a(n) ________.
    a. `float`
    b. `int`
    c. `byte`
    d. error message

16. You use a ________ to explicitly override an implicit type.
    a. mistake
    b. type cast
    c. format
    d. type set

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

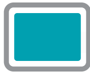
17. In Java, what is the value of 3 + 7 * 4 + 2?
    a. 21
    b. 33
    c. 42
    d. 48

18. Which assignment is correct in Java?
    a. `int value = (float) 4.5;`
    b. `float value = 4 (double);`
    c. `double value = 2.12;`
    d. `char value = 5c;`

19. Which assignment is correct in Java?
    a. `double money = 12;`
    b. `double money = 12.0;`
    c. `double money = 12.0d;`
    d. All of the above are correct.

20. Which assignment is correct in Java?
    a. `char aChar = 5.5;`
    b. `char aChar = "W";`
    c. `char aChar = '*';`
    d. Two of the preceding answers are correct.

Exercises



Programming Exercises

1. What is the numeric value of each of the following expressions as evaluated by Java?
    a. 3 + 7 * 2
    b. 18 / 3 + 4
    c. 9 / 3 + 12 / 4
    d. 15 / 2
    e. 14 / 3
    f. 29 / 10
    g. 14 % 2
    h. 15 % 2
    i. 31 % 7
    j. 6 % 4 + 1
    k. (5 + 6) * 3
    l. 25 / (3 + 2)
    m. 13 % 15

2. What is the value of each of the following Boolean expressions?
    a. 5 < 8
    b. 4 <= 9
    c. 3 == 4
    d. 12 >= 12
    e. 3 + 4 == 8
    f. 7 < 9 – 2
    g. 5 != 5
    h. 15 != 3 * 5
    i. 9 != –9
    j. 3 + 5 * 2 == 16

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

3. Choose the best data type for each of the following so that any reasonable value is accommodated but no memory storage is wasted. Give an example of a typical value that would be held by the variable, and explain why you chose the type you did.

   a. the number of siblings you have
   b. your final grade in this class
   c. the population of Earth
   d. the population of a U.S. county
   e. the number of passengers on a bus
   f. one player's score in a Scrabble game
   g. one team's score in a Major League Baseball game
   h. the year an historical event occurred
   i. the number of legs on an animal

4. a. Write a Java class that declares a named constant that holds the number of feet in a mile: 5,280. Also declare a variable to represent the distance in miles between your house and your uncle's house. Assign an appropriate value to the variable—for example, 8.5. Compute and display the distance to your uncle's house in both miles and feet. Display explanatory text with the values—for example, `The distance to my uncle's house is 8.5 miles or 44880.0 feet`. Save the class as **MilesToFeet.java**.

   b. Convert the `MilesToFeet` class to an interactive application. Instead of assigning a value to the distance, accept the value from the user as input. Save the revised class as **MilesToFeetInteractive.java**.

5. a. Write a Java class that declares a named constant that represents next year's anticipated 10 percent increase in sales for each division of a company. (You can represent 10 percent as 0.10.) Also declare variables to represent this year's sales total in dollars for the North and South divisions. Assign appropriate values to the variables—for example, 4000 and 5500. Compute and display, with explanatory text, next year's projected sales for each division. Save the class as **ProjectedSales.java**.

   b. Convert the `ProjectedSales` class to an interactive application. Instead of assigning values to the North and South current year sales variables, accept them from the user as input. Save the revised class as **ProjectedSalesInteractive.java**.

6. a. Write a class that declares a variable named `inches` that holds a length in inches, and assign a value. Display the value in feet and inches; for example, 86 inches becomes 7 feet and 2 inches. Be sure to use a named constant where appropriate. Save the class as **InchesToFeet.java**.

   b. Write an interactive version of the `InchesToFeet` class that accepts the inches value from a user. Save the class as **InchesToFeetInteractive.java**.

7. Write a class that declares variables to hold your three initials. Display the three initials with a period following each one, as in J.M.F. Save the class as **Initials.java**.

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

8. Meadowdale Dairy Farm sells organic brown eggs to local customers. They charge $3.25 for a dozen eggs, or 45 cents for individual eggs that are not part of a dozen. Write a class that prompts a user for the number of eggs in the order and then display the amount owed with a full explanation. For example, typical output might be, "You ordered 27 eggs. That's 2 dozen at $3.25 per dozen and 3 loose eggs at 45.0 cents each for a total of $7.85." Save the class as **Eggs.java**.
9. The Happy Yappy Kennel boards dogs at a cost of 50 cents per pound per day. Write a class that accepts a dog's weight and number of days to be boarded and displays the total price for boarding. Save the class as **DogBoarding.java**.
10. Write a class that calculates and displays the conversion of an entered number of dollars into currency denominations—*20*s, *10*s, *5*s, and *1*s. Save the class as **Dollars.java**.
11. Write a program that accepts a temperature in Fahrenheit from a user and converts it to Celsius by subtracting 32 from the Fahrenheit value and multiplying the result by 5/9. Display both values. Save the class as **FahrenheitToCelsius.java**.
12. Travel Tickets Company sells tickets for airlines, tours, and other travel-related services. Because ticket agents frequently mistype long ticket numbers, Travel Tickets has asked you to write an application that indicates invalid ticket number entries. The class prompts a ticket agent to enter a six-digit ticket number. Ticket numbers are designed so that if you drop the last digit of the number, then divide the number by 7, the remainder of the division will be identical to the last dropped digit. This process is illustrated in the following example:

Step 1	Enter the ticket number; for example, 123454.
Step 2	Remove the last digit, leaving 12345.
Step 3	Determine the remainder when the ticket number is divided by 7. In this case, 12345 divided by 7 leaves a remainder of 4.
Step 4	Assign the Boolean value of the comparison between the remainder and the digit dropped from the ticket number.
Step 5	Display the result—`true` or `false`—in a message box.

Accept the ticket number from the agent and verify whether it is a valid number. Test the application with the following ticket numbers:

- 123454; the comparison should evaluate to `true`
- 147103; the comparison should evaluate to `true`
- 154123; the comparison should evaluate to `false`

Save the program as **TicketNumber.java**.

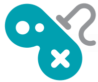
Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Debugging Exercises

1. Each of the following files in the Chapter02 folder of your downloadable student files has syntax and/or logic errors. In each case, determine the problem and fix the application. After you correct the errors, save each file using the same filename preceded with *Fix*. For example, DebugTwo1.java will become FixDebugTwo1.java.

   a. DebugTwo1.java
   b. DebugTwo2.java
   c. DebugTwo3.java
   d. DebugTwo4.java

When you change a filename, remember to change every instance of the class name within the file so that it matches the new filename. In Java, the filename and class name must always match.



Game Zone

1. *Mad Libs* is a children's game in which the players provide a few words that are then incorporated into a silly story. The game helps children understand different parts of speech because they are asked to provide specific types of words. For example, you might ask a child for a noun, another noun, an adjective, and a past-tense verb. The child might reply with such answers as *table*, *book*, *silly*, and *studied*. The newly created Mad Lib might be:

   Mary had a little *table*
   Its *book* was *silly* as snow
   And everywhere that Mary *studied*
   The *table* was sure to go.

   Create a Mad Libs program that asks the user to provide at least four or five words, and then create and display a short story or nursery rhyme that uses them. Save the file as **MadLib.java**.

2. In the "Game Zone" section in Chapter 1, you learned how to obtain a random number. For example, the following statement generates a random number between the constants `MIN` and `MAX` inclusive and assigns it to a variable named `random`:

   ```
   random = 1 + (int)(Math.random() * MAX);
   ```

   Write a program that selects a random number between 1 and 5 and asks the user to guess the number. Display a message that indicates the difference between the random number and the user's guess. Display another message that displays the random number and the Boolean value `true` or `false` depending on whether the user's guess equals the random number. Save the file as **RandomGuessMatch.java**.

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Case Problems

1. Carly's Catering provides meals for parties and special events. Write a program that prompts the user for the number of guests attending an event and then computes the total price, which is $35 per person. Display the company motto with the border that you created in the `CarlysMotto2` class in Chapter 1, and then display the number of guests, price per guest, and total price. Also display a message that indicates `true` or `false` depending on whether the job is classified as a large event—an event with 50 or more guests. Save the file as **CarlysEventPrice.java**.

2. Sammy's Seashore Supplies rents beach equipment such as kayaks, canoes, beach chairs, and umbrellas to tourists. Write a program that prompts the user for the number of minutes he rented a piece of sports equipment. Compute the rental cost as $40 per hour plus $1 per additional minute. (You might have surmised already that this rate has a logical flaw, but for now, calculate rates as described here. You can fix the problem after you read the chapter on decision making.) Display Sammy's motto with the border that you created in the `SammysMotto2` class in Chapter 1. Then display the hours, minutes, and total price. Save the file as **SammysRentalPrice.java**.

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.