
2 Data and Expressions

Chapter Objectives

- Discuss the use of character strings, concatenation, and escape sequences.
- Explore the declaration and use of variables.
- Describe the Java primitive data types.
- Discuss the syntax and processing of expressions.
- Define the types of data conversions and the mechanisms for accomplishing them.
- Introduce the `Scanner` class to create interactive programs.

This chapter explores some of the basic types of data used in a Java program and the use of expressions to perform calculations. It discusses the conversion of data from one type to another and how to read input interactively from the user running a program.

2.1 Character Strings

In [Chapter 1](#), we discussed the basic structure of a Java program, including the use of comments, identifiers, and white space, using the `Lincoln` program as an example. [Chapter 1](#) also included an overview of the various concepts involved in object-oriented programming, such as objects, classes, and methods. Take a moment to review these ideas if necessary.

A character string is an object in Java, defined by the class `String`. Because strings are so fundamental to computer programming, Java provides the ability to use a *string literal*, delimited by double quotation characters, as we've seen in previous examples. We explore the `String` class and its methods in more detail in [Chapter 3](#). For now, let's explore the use of string literals in more detail.

The following are all examples of valid string literals:

```
"The quick brown fox jumped over the lazy dog."
```


```
"602 Greenbriar Court, Chalfont PA 18914"
```

```
"x"
```

```
" "
```

A string literal can contain any valid characters, including numeric digits, punctuation, and other special characters. The last example in the list above contains no characters at all.

The `print` and `println` Methods

In the `Lincoln` program in [Chapter 1](#) , we invoked the `println` method as follows:

```
System.out.println("Whatever you are, be a good one.");
```

This statement demonstrates the use of objects. The `System.out` object represents an output device or file, which by default is the monitor screen. To be more precise, the object's name is `out` and it is stored in the `System` class. We explore that relationship in more detail at the appropriate point in the text.

The `println` method is a service that the `System.out` object performs for us. Whenever we request it, the object will print a character string to the screen. We can say that we send the `println` message to the `System.out` object to request that some text be printed.

Each piece of data that we send to a method is called a **parameter** ⓘ. In this case, the `println` method takes only one parameter: the string of characters to be printed.

The `System.out` object also provides another service we can use: the `print` method. The difference between `print` and `println` is small but important. The `println` method prints the information sent to it, then moves to the beginning of the next line. The `print` method is similar to `println` but does not advance to the next line when completed.

Key Concept

The `print` and `println` methods represent two services provided by the `System.out` object.

The program shown in **Listing 2.1** ⓘ is called `Countdown`, and it invokes both the `print` and `println` methods.

Listing 2.1

```
//*****  
  
// Countdown.java      Author: Lewis/Loftus
```

```
//
// Demonstrates the difference between print and println.
//*****

public class Countdown
{
    //-----
    -----
    // Prints two lines of output representing a rocket
    countdown.
    //-----
    -----

    public static void main(String[] args)
    {
        System.out.print("Three... ");
        System.out.print("Two... ");
        System.out.print("One... ");
        System.out.print("Zero... ");
        System.out.println("Liftoff!"); // appears on first
        output line
        System.out.println("Houston, we have a problem.");
    }
}
```

Output

```
Three... Two... One... Zero... Liftoff!
```

```
Houston, we have a problem.
```

Carefully compare the output of the `Countdown` program, shown at the bottom of the program listing, to the program code. Note that the word `Liftoff` is printed on the same line as the first few words, even though it is printed using the `println` method. Remember that the `println` method moves to the beginning of the next line *after* the information passed to it has been printed.

String Concatenation

A string literal cannot span multiple lines in a program. The following program statement is improper syntax and would produce an error when attempting to compile:

```
// The following statement will not compile  
System.out.println("The only stupid question is  
the one that is not asked.");
```

When we want to print a string that is too long to fit on one line in a program, we can rely on *string concatenation* to append one string to the end of another. The string concatenation operator is the plus sign (+). The following expression concatenates one character string to another, producing one long string:

```
"The only stupid question is " + "the one that is not asked."
```

The program called `Facts` shown in [Listing 2.2](#) contains several `println` statements. The first one prints a sentence that is somewhat long and will not fit on one line of the program. Since a character literal cannot span two lines in a program, we split the string into two and use string concatenation to append them. Therefore, the string concatenation operation in the first `println` statement results in one large string that is passed to the method to be printed.

Listing 2.2

```
//*****

//  Facts.java      Author: Lewis/Loftus
//
//  Demonstrates the use of the string concatenation operator
//  and the
//  automatic conversion of an integer to a string.
//*****

public class Facts
{
    //-----
    -----
}
```



```
// Prints various facts.

//-----

-----

public static void main(String[] args)
{
    // Strings can be concatenated into one long string
    System.out.println("We present the following facts for
your "
                        + "extracurricular edification:");

    System.out.println();

    // A string can contain numeric digits
    System.out.println("Letters in the Hawaiian alphabet:
12");

    // A numeric value can be concatenated to a string
    System.out.println("Dialing code for Antarctica: " +
672);

    System.out.println("Year in which Leonardo da Vinci
invented "
                        + "the parachute: " + 1515);

    System.out.println("Speed of ketchup: " + 40 + " km per
year");
}
}
```

Output

```
We present the following facts for your
extracurricular edification:
Letters in the Hawaiian alphabet: 12
Dialing code for Antarctica: 672
Year in which Leonardo da Vinci invented the parachute:
1515
Speed of ketchup: 40 km per year
```

Note that we don't have to pass any information to the `println` method, as shown in the second line of the `Facts` program. This call does not print any visible characters, but it does move to the next line of output. So in this case, calling `println` with no parameters has the effect of printing a blank line.


The last three calls to `println` in the `Facts` program demonstrate another interesting thing about string concatenation: Strings can be concatenated with numbers. Note that the numbers in those lines are not enclosed in double quotes and are therefore not character strings. In these cases, the number is automatically converted to a string, and then the two strings are concatenated.

Because we are printing particular values, we simply could have included the numeric value as part of the string literal, such as

```
"Speed of ketchup: 40 km per year"
```

Digits are characters and can be included in strings as needed. We separate them in the `Facts` program to demonstrate the ability to concatenate a string and a number. This technique will be useful in upcoming examples.

As you can imagine, the `+` operator is also used for arithmetic addition. Therefore, what the `+` operator does depends on the types of data on which it operates. If either or both of the operands of the `+` operator are strings, then string concatenation is performed.

The `Addition` program shown in [Listing 2.3](#)  demonstrates the distinction between string concatenation and arithmetic addition. The `Addition` program uses the `+` operator four times. In the first call to `println`, both `+` operations perform string concatenation because the operators are executed left to right. The first operator concatenates the string with the first number (`24`), creating a larger string. Then that string is concatenated with the second number (`45`), creating an even larger string, which gets printed.

Listing 2.3

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

```
// Addition.java      Author: Lewis/Loftus
//
// Demonstrates the difference between the addition and
string
// concatenation operators.
//*****

public class Addition
{
    //-----
    -----
    // Concatenates and adds two numbers and prints the
results.
    //-----
    -----
    public static void main(String[] args)
    {
        System.out.println("24 and 45 concatenated: " + 24 +
45);

        System.out.println("24 and 45 added: " + (24 + 45));
    }
}
```

Output

```
24 and 45 concatenated: 2445  
24 and 45 added: 69
```

In the second call to `println`, we use parentheses to group the `+` operation with the two numeric operands. This forces that operation to happen first. Because both operands are numbers, the numbers are added in the arithmetic sense, producing the result `69`. That number is then concatenated with the string, producing a larger string that gets printed.

We revisit this type of situation later in this chapter when we formalize the precedence rules that define the order in which operators get evaluated.

Escape Sequences

Because the double quotation character (`"`) is used in the Java language to indicate the beginning and end of a string, we must use a special technique to print the quotation character. If we simply put it in a string (`""`), the compiler gets confused because it thinks the second quotation character is the end of the string and doesn't know what to do with the third one. This results in a compile-time error.



Example using strings and escape sequences.


To overcome this problem, Java defines several **escape sequences** ⓘ to represent special characters. An escape sequence begins with the backslash character (\), which indicates that the character or characters that follow should be interpreted in a special way. **Figure 2.1** ⓘ lists the Java escape sequences.

Escape Sequence	Meaning
\b	backspace
\t	tab
\n	newline
\r	carriage return
\"	double quote
\'	single quote
\\	backslash

Figure 2.1 Java escape sequences

Key Concept

An escape sequence can be used to represent a character that would otherwise cause compilation problems.

The program in [Listing 2.4](#) , called `Roses`, prints some text resembling a poem. It uses only one `println` statement to do so, despite the fact that the poem is several lines long. Note the escape sequences used throughout the string. The `\n` escape sequence forces the output to a new line, and the `\t` escape sequence represents a tab character. The `\"` escape sequence ensures that the quote character is treated as part of the string, not the termination of it, which enables it to be printed as part of the output.

Listing 2.4

```
//*****  
  
//  Roses.java      Author: Lewis/Loftus  
//  
//  Demonstrates the use of escape sequences.  
//*****  
  
public class Roses  
{
```

```
//-----  
-----  
  
// Prints a poem (of sorts) on multiple lines.  
//-----  
-----  
  
public static void main(String[] args)  
{  
  
    System.out.println("Roses are red,\n\tViolets are  
blue,\n" +  
        "Sugar is sweet,\n\tBut I have \"commitment  
issues\",\n\t" +  
        "So I'd rather just be friends\n\tAt this point in  
our " +  
        "relationship.");  
}  
}
```

Output

```
Roses are red,  
    Violets are blue,  
Sugar is sweet,  
    But I have "commitment issues",  
    So I'd rather just be friends  
    At this point in our relationship.
```


Self-Review Questions

(see answers in [Appendix L](#) )

SR 2.1 What is a string literal?

SR 2.2 What is the difference between the `print` and `println` methods?

SR 2.3 What is a parameter?

SR 2.4 What output is produced by the following code fragment?

```
System.out.println("One ");  
System.out.print("Two ");  
System.out.println("Three ");
```

SR 2.5 What output is produced by the following code fragment?

```
System.out.print("Ready ");  
System.out.println();  
System.out.println("Set ");  
System.out.println();  
System.out.println("Go ");
```

SR 2.6 What output is produced by the following statement?
What is produced if the inner parentheses are removed?

```
System.out.println("It is good to be " + (5 + 5));
```

SR 2.7 What is an escape sequence? Give some examples.

SR 2.8 Write a single `println` statement that will output the following exactly as shown (including line breaks and quotation marks).

"I made this letter longer than usual because I lack the time to make it short."

Blaise Pascal

2.2 Variables and Assignment

Most of the information we manage in a program is represented by variables. Let's examine how we declare and use them in a program.

Variables

A **variable** ⓘ is a name for a location in memory used to hold a data value. A variable declaration instructs the compiler to reserve a portion of main memory space large enough to hold a particular type of value and indicates the name by which we refer to that location.

Key Concept

A variable is a name for a memory location used to hold a value of a particular data type.

Consider the program `PianoKeys`, shown in **Listing 2.5** 📄. The first line of the `main` method is the declaration of a variable named `keys` that holds an integer (`int`) value. The declaration also gives `keys` an initial value of 88. If an initial value is not specified for a variable, the

value is undefined. Most Java compilers give errors or warnings if you attempt to use a variable before you've explicitly given it a value.

Listing 2.5

```
//*****

//  PianoKeys.java      Author: Lewis/Loftus
//
//  Demonstrates the declaration, initialization, and use of
//  an
//  integer variable.
//*****

public class PianoKeys
{
    //-----
    -----
    //  Prints the number of keys on a piano.
    //-----
    -----
    public static void main(String[] args)
    {
        int keys = 88;

        System.out.println("A piano has " + keys + " keys.");
    }
}
```

Output

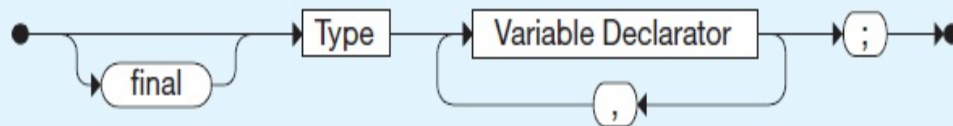
```
A piano has 88 keys.
```

The `keys` variable, with its value, could be pictured as follows:

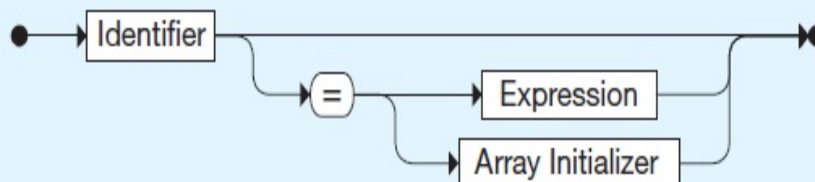
keys

88

Local Variable Declaration



Variable Declarator



A variable declaration consists of a `Type` followed by a list of variables. Each variable can be initialized in the declaration to the value of the specified `Expression`. If the `final` modifier precedes the declaration, the identifiers

are declared as named constants whose values cannot be changed once set.

Examples:


```
int total;  
double num1, num2 = 4.356, num3;  
char letter = 'A', digit = '7';  
final int MAX = 45;
```

In the `PianoKeys` program, two pieces of information are used in the call to the `println` method. The first is a string and the second is the variable `keys`. When a variable is referenced, the value currently stored in it is used. Therefore, when the call to `println` is executed, the value of `keys`, which is 88, is obtained. Because that value is an integer, it is automatically converted to a string and concatenated with the initial string. The concatenated string is passed to `println` and printed.

A variable declaration can have multiple variables of the same type declared on one line. Each variable on the line can be declared with or without an initializing value. For example:

```
int count, minimum = 0, result;
```

The Assignment Statement

Let's examine a program that changes the value of a variable. **Listing 2.6**  shows a program called `Geometry`. This program first declares an integer variable called `sides` and initializes it to `7`. It then prints out the current value of `sides`.

Listing 2.6

```
//*****

//  Geometry.java      Author: Lewis/Loftus
//
//  Demonstrates the use of an assignment statement to change
//  the
//  value stored in a variable.
//*****

public class Geometry
{
    //-----
```

```
-----  
    // Prints the number of sides of several geometric shapes.  
    //-----  
-----  
  
    public static void main(String[] args)  
    {  
        int sides = 7;    // declaration with initialization  
        System.out.println("A heptagon has " + sides + "  
sides.");  
  
        sides = 10;    // assignment statement  
        System.out.println("A decagon has " + sides + "  
sides.");  
  
        sides = 12;  
        System.out.println("A dodecagon has " + sides + "  
sides.");  
    }  
}
```

Output

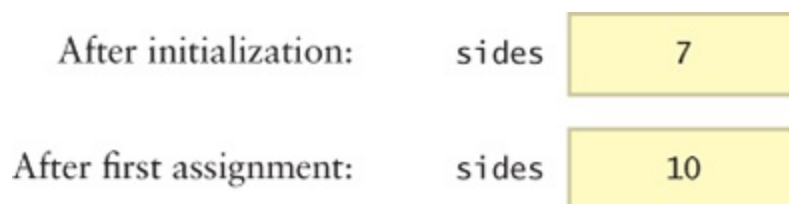
```
A heptagon has 7 sides.  
A decagon has 10 sides.  
A dodecagon has 12 sides.
```


The next line in `main` changes the value stored in the variable `sides`:

```
sides = 10;
```

This is called an *assignment statement* because it assigns a value to a variable. When executed, the expression on the right-hand side of the assignment operator (=) is evaluated, and the result is stored in the memory location indicated by the variable on the left-hand side. In this example, the expression is simply a number, `10`. We discuss expressions that are more involved than this in the next section.

A variable can store only one value of its declared type. A new value overwrites the old one. In this case, when the value `10` is assigned to `sides`, the original value `7` is overwritten and lost forever, as follows:



Key Concept

Accessing data leaves it intact in memory, but an assignment statement overwrites the old data.

Basic Assignment



The basic assignment statement uses the assignment operator (=) to store the result of the Expression into the specified Identifier, usually a variable.

Examples:

```
total = 57;  
count = count + 1;  
value = (min / 2) * lastValue;
```

When a reference is made to a variable, such as when it is printed, the value of the variable is not changed. This is the nature of computer memory: Accessing (reading) data leaves the values in memory intact, but writing data replaces the old data with the new.

The Java language is *strongly typed*, meaning that we are not allowed to assign a value to a variable that is inconsistent with its declared type. Trying to combine incompatible types will generate an error

when you attempt to compile the program. Therefore, the expression on the right-hand side of an assignment statement must evaluate to a value compatible with the type of the variable on the left-hand side.

Key Concept

We cannot assign a value of one type to a variable of an incompatible type.

Constants

Sometimes we use data that is constant throughout a program. For instance, we might write a program that deals with a theater that can hold no more than 427 people. It is often helpful to give a constant value a name, such as `MAX_OCCUPANCY`, instead of using a literal value, such as `427`, throughout the code. The purpose and meaning of literal values such as `427` is often confusing to someone reading the code. By giving the value a name, you help explain its role in the program.

Constants are identifiers and are similar to variables except that they hold a particular value for the duration of their existence. Constants are, to use the English meaning of the words, not variable. Their value doesn't change.

Key Concept

Constants hold a particular value for the duration of their existence.

In Java, if you precede a declaration with the reserved word `final`, the identifier is made a constant. By convention, uppercase letters are used when naming constants to distinguish them from regular variables, and individual words are separated using the underscore character. For example, the constant describing the maximum occupancy of a theater could be declared as follows:

```
final int MAX_OCCUPANCY = 427;
```

The compiler will produce an error message if you attempt to change the value of a constant once it has been given its initial value. This is another good reason to use constants. Constants prevent inadvertent coding errors because the only valid place to change their value is in the initial assignment.

There is a third good reason to use constants. If a constant is used throughout a program and its value needs to be modified, then you have to change it in only one place. For example, if the capacity of the theater changes (because of a renovation) from 427 to 535, then you have to change only one declaration, and all uses of `MAX_OCCUPANCY`

automatically reflect the change. If the literal `427` had been used throughout the code, each use would have to be found and changed. If you were to miss any uses of the literal value, problems would surely arise.

Self-Review Questions

(see answers in [Appendix L](#) )

SR 2.9 What is a variable declaration?

SR 2.10 Given the following variable declarations, answer each question.

```
int count = 0, value, total;
```

```
final int MAX_VALUE = 100;
```

```
int myValue = 50;
```

- How many variables are declared?
- What is the type of these declared variables?
- Which of the variables are given an initial value?
- Based on the above declarations, is the following assignment statement valid? Explain.

```
myValue = 100;
```

- Based on the above declarations, is the following assignment statement valid? Explain.

```
MAX_VALUE = 50;
```

SR 2.11 Your program needs a variable of type `int` to hold the number of CDs in a music collection. The initial value should be zero. Write a declaration statement for the variable.


SR 2.12 Your program needs a variable of type `int` to hold the number of feet in a mile (5280). Write a declaration statement for the variable.

SR 2.13 Briefly describe three reasons for using a constant in a program instead of a literal value.

2.3 Primitive Data Types

There are eight *primitive data types* in Java: four types of integers, two types of floating point numbers, a character data type, and a boolean data type. Everything else is represented using objects. Let's examine these eight primitive data types in detail.

Integers and Floating Points

Java has two basic kinds of numeric values: integers, which have no fractional part, and floating points, which do. There are four integer data types (`byte`, `short`, `int`, and `long`) and two floating point data types (`float` and `double`). All of the numeric types differ by the amount of memory space used to store a value of that type, which determines the range of values that can be represented. The size of each data type is the same for all hardware platforms. All numeric types are *signed*, meaning that both positive and negative values can be stored in them. **Figure 2.2**  summarizes the numeric primitive types.

Type	Storage	Min Value	Max Value
byte	8 bits	−128	127
short	16 bits	−32,768	32,767
int	32 bits	−2,147,483,648	2,147,483,647
long	64 bits	−9,223,372,036,854,775,808	9,223,372,036,854,775,807
float	32 bits	Approximately $-3.4\text{E}+38$ with 7 significant digits	Approximately $3.4\text{E}+38$ with 7 significant digits
double	64 bits	Approximately $-1.7\text{E}+308$ with 15 significant digits	Approximately $1.7\text{E}+308$ with 15 significant digits

Figure 2.2 The Java numeric primitive types

Key Concept

Java has two kinds of numeric values: integer and floating point. There are four integer data types and two floating point data types.

Recall from our discussion in [Chapter 1](#) that a bit can be either a 1 or a 0. Because each bit can represent two different states, a string of N bits can be used to represent 2^N different values. [Appendix B](#) describes number systems and these kinds of relationships in more detail.

When designing programs, we occasionally need to be careful about picking variables of appropriate size so that memory space is not

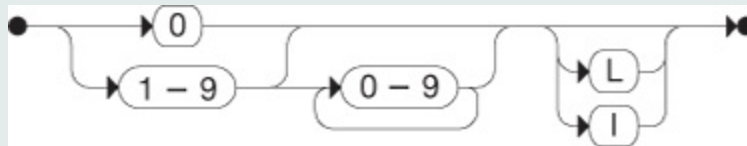
wasted. For example, if the value of a particular variable will not vary outside of a range of 1–1000, then a two-byte integer (`short`) is large enough to accommodate it. On the other hand, when it's not clear what the range of a particular variable will be, we should provide a reasonable, even generous, amount of space. In most situations, memory space is not a serious restriction, and we can usually afford generous assumptions.

Note that even though a `float` value supports very large (and very small) numbers, it has only seven significant digits. Therefore, if it is important to accurately maintain a value such as 50341.2077, we need to use a `double`.

As we've already discussed, a **literal** ⓘ is an explicit data value used in a program. The various numbers used in programs such as `Facts` and `Addition` and `PianoKeys` are all *integer literals*. Java assumes all integer literals are of type `int`, unless an `L` or `l` is appended to the end of the value to indicate that it should be considered a literal of type `long`, such as `45L`.

Likewise, Java assumes that all *floating point literals* are of type `double`. If we need to treat a floating point literal as a `float`, we append an `F` or `f` to the end of the value, as in `2.718F` or `123.45f`. Numeric literals of type `double` can be followed by a `D` or `d` if desired.

Decimal Integer Literal



An integer literal is composed of a series of digits followed by an optional suffix to indicate that it should be considered an integer. Negation of a literal is considered a separate operation.

Examples:

```
5
```

```
2594
```

```
4920328L
```

The following are examples of numeric variable declarations in Java:

```
int answer = 42;
```

```
byte smallNumber1, smallNumber2;
```

```
long countedStars = 86827263927L;
```

```
float ratio = 0.2363F;
```

```
double delta = 453.523311903;
```

Characters

Characters are another fundamental type of data used and managed on a computer. Individual characters can be treated as separate data items, and, as we've seen in several examples, they can be combined to form character strings.

A *character literal* is expressed in a Java program with single quotes, such as `'b'` or `'J'` or `';'`. You will recall that *string literals* are delineated using double quotation marks, and that the `String` type is not a primitive data type in Java; it is a class name. We discuss the `String` class in detail in Chapter 3.

Note the difference between a digit as a character (or part of a string) and a digit as a number (or part of a larger number). The number `602` is a numeric value that can be used in an arithmetic calculation. But in the string `"602 Greenbriar Court"` the `6`, `0`, and `2` are characters, just like the rest of the characters that make up the string.

The characters we can manage are defined by a **character set** ⓘ, which is simply a list of characters in a particular order. Each programming language supports a particular character set that defines the valid values for a character variable in that language. Several character sets have been proposed, but only a few have been used

regularly over the years. The *ASCII character set* is a popular choice. ASCII stands for the American Standard Code for Information Interchange. The basic ASCII set uses seven bits per character, providing room to support 128 different characters, including:


- uppercase letters, such as 'A', 'B', and 'C'
- lowercase letters, such as 'a', 'b', and 'c'
- punctuation, such as the period ('.'), semicolon(';'), and comma(',')
- the digits '0' through '9'
- the space character, ' '
- special symbols, such as the ampersand('&'), vertical bar('|'), and backslash('\')
- control characters, such as the carriage return, null, and end-of-text marks

The **control characters** ⓘ are sometimes called nonprinting or invisible characters because they do not have a specific symbol that represents them. Yet they are as valid as any other character and can be stored and used in the same ways. Many control characters have special meaning to certain software applications.

As computing became a worldwide endeavor, users demanded a more flexible character set containing other language alphabets. ASCII was extended to use eight bits per character, and the number of characters in the set doubled to 256. The extended ASCII contains many accented and diacritical characters used in languages other than English.

Key Concept

Java uses the 16-bit Unicode character set to represent character data.

However, even with 256 characters, the ASCII character set cannot represent the world's alphabets, especially given the various Asian alphabets and their many thousands of ideograms. Therefore, the developers of the Java programming language chose the *Unicode character set*, which uses 16 bits per character, supporting 65,536 unique characters (and techniques that allow even more characters to be represented using multiple bytes). The characters and symbols from many languages are included in the Unicode definition. ASCII is a subset of the Unicode character set, comprising the first 256 characters. [Appendix C](#)  discusses the Unicode character set in more detail.

A character set assigns a particular number to each character, so by definition the characters are in a particular order. This is referred to as lexicographic order. In the ASCII and Unicode ordering, the digit characters `'0'` through `'9'` are continuous (no other characters intervene) and in order. Similarly, the lowercase alphabetic characters `'a'` through `'z'` are continuous and in order, as are the uppercase

alphabetic characters `'A'` through `'Z'`. These characteristics make it relatively easy to keep things in alphabetical order.

In Java, the data type `char` represents a single character. The following are some examples of character variable declarations in Java:

```
char topGrade = 'A';  
char symbol1, symbol2, symbol3;  
char terminator = ';', separator = ' ';
```

Booleans

A boolean value, defined in Java using the reserved word `boolean`, has only two valid values: `true` and `false`. A boolean variable is usually used to indicate whether a particular condition is true, but it can also be used to represent any situation that has two states, such as a light bulb being on or off.

A boolean value cannot be converted to any other data type, nor can any other data type be converted to a boolean value. The words `true` and `false` are reserved in Java as *boolean literals* and cannot be used outside of this context.

The following are some examples of boolean variable declarations in Java:

```
boolean flag = true;  
boolean tooHigh, tooSmall, tooRough;  
boolean done = false;
```

Self-Review Questions

(see answers in [Appendix L](#) )

SR 2.14 What is primitive data? How are primitive data types different from objects?

SR 2.15 How many values can be stored in an integer variable?

SR 2.16 What are the four integer data types in Java? How are they different?

SR 2.17 What type does Java automatically assign to an integer literal? How can you indicate that an integer literal should be considered a different type?

SR 2.18 What type does Java automatically assign to a floating point literal? How can you indicate that a floating point literal should be considered a different type?

SR 2.19 What is a character set?

SR 2.20 How many characters are supported by the ASCII character set, the extended ASCII character set, and the

Unicode character set?

2.4 Expressions

An **expression** ⓘ is a combination of one or more operators and operands that usually perform a calculation. The value calculated does not have to be a number, but it often is. The operands used in the operations might be literals, constants, variables, or other sources of data. The manner in which expressions are evaluated and used is fundamental to programming. For now, we will focus on arithmetic expressions that use numeric operands and produce numeric results.

Key Concept

Expressions are combinations of operators and operands used to perform a calculation.

Arithmetic Operators

The usual arithmetic operations are defined for both integer and floating point numeric types, including addition (+), subtraction (−), multiplication (*), and division (/). Java also has another arithmetic operation: The *remainder operator* (%) returns the remainder after dividing the second operand into the first. The remainder operator is

sometimes called the modulus operator. The sign of the result of a remainder operation is the sign of the numerator. Therefore,

Operation	Operation
$17 \% 4$	1
$-20 \% 3$	-2
$10 \% -5$	0
$3 \% 8$	3




Review of primitive data and expressions.

As you might expect, if either or both operands to any numeric operator are floating point values, the result is a floating point value. However, the division operator produces results that are less intuitive, depending on the types of the operands. If both operands are integers, the `/` operator performs *integer division*, meaning that any fractional part of the result is discarded. If one or the other or both operands are floating point values, the `/` operator performs *floating*

point division, and the fractional part of the result is kept. For example, the result of `10/4` is 2, but the results of `10.0/4` and `10/4.0` and `10.0/4.0` are all 2.5.

A *unary operator* has only one operand, while a *binary operator* has two. The + and - arithmetic operators can be either unary or binary. The binary versions accomplish addition and subtraction, and the unary versions represent positive and negative numbers. For example, -1 is an example of using the unary negation operator to make the value negative. The unary + operator is rarely used.

Java does not have a built-in operator for raising a value to an exponent. However, the `Math` class provides methods that perform exponentiation and many other mathematical functions. The `Math` class is discussed in [Chapter 3](#) .

Operator Precedence

Operators can be combined to create more complex expressions. For example, consider the following assignment statement:

```
result = 14 + 8 / 2;
```

The entire right-hand side of the assignment is evaluated, and then the result is stored in the variable. But what is the result? If the

addition is performed first, the result is 11; if the division operation is performed first, the result is 18. The order of operator evaluation makes a big difference. In this case, the division is performed before the addition, yielding a result of 18.

Key Concept

Java follows a well-defined set of precedence rules that governs the order in which operators will be evaluated in an expression.

Note that in this and subsequent examples, we use literal values rather than variables to simplify the expression. The order of operator evaluation is the same if the operands are variables or any other source of data.

All expressions are evaluated according to an *operator precedence hierarchy* that establishes the rules that govern the order in which operations are evaluated. The arithmetic operators generally follow the same rules you learned in algebra. Multiplication, division, and the remainder operator all have equal precedence and are performed before (have higher precedence than) addition and subtraction. Addition and subtraction have equal precedence.

Any arithmetic operators at the same level of precedence are performed left to right. Therefore, we say the arithmetic operators have a *left-to-right association*.

Precedence, however, can be forced in an expression by using parentheses. For instance, if we really wanted the addition to be performed first in the previous example, we could write the expression as follows:

```
result = (14 + 8) / 2;
```

Any expression in parentheses is evaluated first. In complicated expressions, it is good practice to use parentheses, even when it is not strictly necessary, to make it clear how the expression is evaluated.

Parentheses can be nested, and the innermost nested expressions are evaluated first. Consider the following expression:

```
result = 3 * ((18 - 4) / 2);
```

In this example, the result is 21. First, the subtraction is performed, forced by the inner parentheses. Then, even though multiplication and division are at the same level of precedence and usually would be

evaluated left to right, the division is performed first because of the outer parentheses. Finally, the multiplication is performed.

After the arithmetic operations are complete, the computed result is stored in the variable on the left-hand side of the assignment operator (`=`). In other words, the assignment operator has a lower precedence than any of the arithmetic operators.

The evaluation of a particular expression can be shown using an *expression tree*, such as the one in [Figure 2.3](#). The operators are executed from the bottom up, creating values that are used in the rest of the expression. Therefore, the operations lower in the tree have a higher precedence than those above, or they are forced to be executed earlier using parentheses.

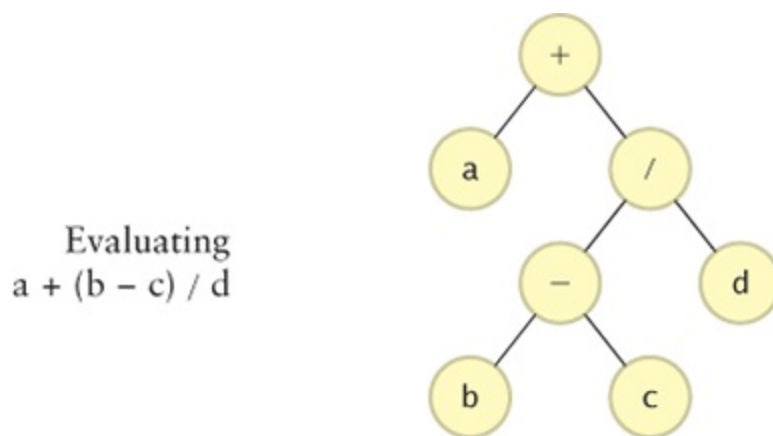


Figure 2.3 An expression tree

The parentheses used in expressions are actually operators themselves. Parentheses have a higher precedence than almost any other operator. [Figure 2.4](#) shows a precedence table with the relationships between the arithmetic operators, parentheses, and the

assignment operator. [Appendix D](#)  includes a full precedence table showing all Java operators.

Precedence Level	Operator	Operation	Associates
1	+	unary plus	R to L
	-	unary minus	
2	*	multiplication	L to R
	/	division	
	%	remainder	
3	+	addition	L to R
	-	subtraction	
	+	string concatenation	
4	=	assignment	R to L

Figure 2.4 Precedence among some of the Java operators

For an expression to be syntactically correct, the number of left parentheses must match the number of right parentheses and they must be properly nested. The following examples are *not* valid expressions:

```
result = ((19 + 8) % 3) - 4);    // not valid
result = (19 (+ 8 %) 3 - 4);    // not valid
```

Keep in mind that when a variable is referenced in an expression, its current value is used to perform the calculation. In the following

assignment statement, the current value of the variable `count` is added to the current value of the variable `total`, and the result is stored in the variable `sum`:


```
sum = count + total;
```

The original value contained in `sum` before this assignment is overwritten by the calculated value. The values stored in `count` and `total` are not changed.

The same variable can appear on both the left-hand side and the right-hand side of an assignment statement. Suppose the current value of a variable called `count` is 15 when the following assignment statement is executed:

```
count = count + 1;
```

Because the right-hand expression is evaluated first, the original value of `count` is obtained and the value 1 is added to it, producing the result 16. That result is then stored in the variable `count`, overwriting the original value of 15 with the new value of 16. Therefore, this assignment statement *increments*, or adds 1 to, the variable `count`.

Let's look at another example of expression processing. The program in **Listing 2.7** , called `TempConverter`, converts a particular Celsius temperature value to its equivalent Fahrenheit value using an expression that computes the following formula:

Fahrenheit=95 Celsius+32

Listing 2.7

```
//*****

//  TempConverter.java      Author: Lewis/Loftus
//
//  Demonstrates the use of primitive data types and
//  arithmetic
//  expressions.
//*****

public class TempConverter
{
    //-----

    //  Computes the Fahrenheit equivalent of a specific
    Celsius
    //  value using the formula F = (9/5)C + 32.
    //-----
    -----
}
```

```
public static void main(String[] args)
{
    final int BASE = 32;
    final double CONVERSION_FACTOR = 9.0 / 5.0;

    double fahrenheitTemp;
    int celsiusTemp = 24;    // value to convert

    fahrenheitTemp = celsiusTemp * CONVERSION_FACTOR +
BASE;

    System.out.println("Celsius Temperature: " +
celsiusTemp);
    System.out.println("Fahrenheit Equivalent: " +
fahrenheitTemp);
}
}
```

Output

```
Celsius Temperature: 24
Fahrenheit Equivalent: 75.2
```

Note that in the temperature conversion program, the operands to the division operation are floating point literals to ensure that the fractional part of the number is kept. The precedence rules dictate that the

multiplication happens before the addition in the final conversion computation.

The `TempConverter` program is not very useful because it converts only one data value that we included in the program as a constant (24 degrees Celsius). Every time the program is run it produces the same result. A far more useful version of the program would obtain the value to be converted from the user each time the program is executed. Interactive programs that read user input are discussed later in this chapter.

Increment and Decrement Operators

There are two other useful arithmetic operators. The *increment operator* (`++`) adds 1 to any integer or floating point value. The two plus signs that make up the operator cannot be separated by white space. The *decrement operator* (`--`) is similar except that it subtracts 1 from the value. They are both unary operators because they operate on only one operand. The following statement causes the value of `count` to be incremented:

```
count++;
```

The result is stored back into the variable `count`. Therefore, it is functionally equivalent to the following statement, which we discussed in the previous section:

```
count = count + 1;
```

The increment and decrement operators can be applied after the variable (such as `count++` or `count--`), creating what is called the *postfix form* of the operator. They can also be applied before the variable (such as `++count` or `--count`), in what is called the *prefix form*. When used alone in a statement, the prefix and postfix forms are functionally equivalent. That is, it doesn't matter if you write

```
count++;
```

or

```
++count;
```

However, when such a form is written as a statement by itself, it is usually written in its postfix form.

When the increment or decrement operator is used in a larger expression, it can yield different results depending on the form used.

For example, if the variable `count` currently contains the value `15`, the following statement assigns the value `15` to `total` and the value `16` to `count`:

```
total = count++;
```

However, the following statement assigns the value `16` to both `total` and `count`:

```
total = ++count;
```

The value of `count` is incremented in both situations, but the value used in the larger expression depends on whether a prefix or postfix form of the increment operator is used.

Because of the subtle differences between the prefix and postfix forms of the increment and decrement operators, they should be used with care. As always, favor the side of readability.

Assignment Operators

As a convenience, several *assignment operators* have been defined in Java that combine a basic operation with assignment. For example,

the `+=` operator can be used as follows:

```
total += 5;
```

This performs the same operation as the following statement:

```
total = total + 5;
```

The right-hand side of the assignment operator can be a full expression. The expression on the right-hand side of the operator is evaluated, then that result is added to the current value of the variable on the left-hand side, and that value is stored in the variable.

Therefore, the following statement:

```
total += (sum - 12) / count;
```

is equivalent to:

```
total = total + ((sum - 12) / count);
```

Many similar assignment operators are defined in Java, including those that perform subtraction (`-=`), multiplication (`*=`), division (`/=`),

and remainder (`%=`). The entire set of Java operators is discussed in [Appendix D](#).

All of the assignment operators evaluate the entire expression on the right-hand side first, then use the result as the right operand of the other operation. Therefore, the following statement:

```
result *= count1 + count2;
```

is equivalent to:

```
result = result * (count1 + count2);
```

Likewise, the following statement:

```
result %= (highest - 40) / 2;
```

is equivalent to:

```
result = result % ((highest - 40) / 2);
```

Some assignment operators perform particular functions depending on the types of the operands, just as their corresponding regular operators do. For example, if the operands to the `+=` operator are strings, then the assignment operator performs string concatenation.

Self-Review Questions

(see answers in [Appendix L](#) )

SR 2.21 What is the result of `19%5` when evaluated in a Java expression? Explain.

SR 2.22 What is the result of `13/4` when evaluated in a Java expression? Explain.

SR 2.23 If an integer variable `diameter` currently holds the value 5, what is its value after the following statement is executed? Explain.

```
diameter = diameter * 4;
```

SR 2.24 What is operator precedence?

SR 2.25 What is the value of each of the following expressions?

- a. `15 + 7 * 3`
- b. `(15 + 7) * 3`
- c. `3 * 6 + 10 / 5 + 5`
- d. `27 % 5 + 7 % 3`
- e. `100 / 2 / 2 / 2`

f. `100 / (2 / 2) / 2`

SR 2.26 For each of the following expressions state whether they are valid or invalid. If invalid, explain why.

a. `result = (5 + 2);`

b. `result = (5 + 2 * (15 - 3));`

c. `result = (5 + 2 (;`

d. `result = (5 + 2 (4));`

SR 2.27 What value is contained in the integer variable `result` after the following sequence of statements is executed?

```
result = 27;
```

```
result = result + 3;
```

```
result = result / 7;
```

```
result = result * 2;
```

SR 2.28 What value is contained in the integer variable `result` after the following sequence of statements is executed?

```
int base;
```

```
int result;
```

```
base = 5;
```

```
result = base + 3;
```

```
base = 7;
```

SR 2.29 What is an assignment operator?

SR 2.30 If an integer variable `weight` currently holds the value 100, what is its value after the following statement is executed? Explain.

```
weight -= 17;
```

2.5 Data Conversion

Because Java is a strongly typed language, each data value is associated with a particular type. It is sometimes helpful or necessary to convert a data value of one type to another type, but we must be careful that we don't lose important information in the process. For example, suppose a `short` variable that holds the number 1000 is converted to a `byte` value. Because a `byte` does not have enough bits to represent the value 1000, some bits would be lost in the conversion, and the number represented in the `byte` would not keep its original value.

A conversion between one primitive type and another falls into one of two categories: widening conversions and narrowing conversions.

Widening conversions ⓘ are the safest because they usually do not lose information. They are called widening conversions because they go from one data type to another type that uses an equal or greater amount of space to store the value. **Figure 2.5** ☐ lists the Java widening conversions.

From	To
byte	short, int, long, float, or double
short	int, long, float, or double
char	int, long, float, or double
int	long, float, or double
long	float or double
float	double

Figure 2.5 Java widening conversions

For example, it is safe to convert from a `byte` to a `short` because a `byte` is stored in 8 bits and a `short` is stored in 16 bits. There is no loss of information. All widening conversions that go from an integer type to another integer type, or from a floating point type to another floating point type, preserve the numeric value exactly.

Although widening conversions do not lose any information about the magnitude of a value, the widening conversions that result in a floating point value can lose precision. When converting from an `int` or a `long` to a `float`, or from a `long` to a `double`, some of the least significant digits may be lost. In this case, the resulting floating point value will be a rounded version of the integer value, following the rounding techniques defined in the IEEE 754 floating point standard.

Key Concept

Narrowing conversions should be avoided because they can lose information.

Narrowing conversions ⓘ are more likely to lose information than widening conversions are. They often go from one type to a type that uses less space to store a value, and therefore some of the information may be compromised. Narrowing conversions can lose both numeric magnitude and precision. Therefore, in general, they should be avoided. **Figure 2.6** ☐ lists the Java narrowing conversions.

From	To
byte	char
short	byte or char
char	byte or short
int	byte, short, or char
long	byte, short, char, or int
float	byte, short, char, int, or long
double	byte, short, char, int, long, or float

Figure 2.6 Java narrowing conversions

An exception to the space-shrinking situation in narrowing conversions is when we convert a `byte` (8 bits) or `short` (16 bits) to a `char` (16 bits). These are still considered narrowing conversions because the sign bit is incorporated into the new character value. Since a character value is unsigned, a negative integer will be converted into a character

that has no particular relationship to the numeric value of the original integer.

Note that `boolean` values are not mentioned in either widening or narrowing conversions. A `boolean` value cannot be converted to any other primitive type and vice versa.

Conversion Techniques

In Java, conversions can occur in three ways:

- assignment conversion
- promotion
- casting

Assignment conversion ⓘ occurs when a value of one type is assigned to a variable of another type during which the value is converted to the new type. Only widening conversions can be accomplished through assignment. For example, if `money` is a `float` variable and `dollars` is an `int` variable, then the following assignment statement automatically converts the value in `dollars` to a `float`:

```
money = dollars;
```

Therefore, if `dollars` contains the value `25`, after the assignment, `money` contains the value `25.0`. However, if we attempt to assign `money` to `dollars`, the compiler will issue an error message alerting us to the fact that we are attempting a narrowing conversion that could lose information. If we really want to do this assignment, we have to make the conversion explicit by using a cast.

Conversion via *promotion* occurs automatically when certain operators need to modify their operands in order to perform the operation. For example, when a floating point value called `sum` is divided by an integer value called `count`, the value of `count` is promoted to a floating point value automatically, before the division takes place, producing a floating point result:

```
result = sum / count;
```

A similar conversion is taking place when a number is concatenated with a string. The number is first converted (promoted) to a string, then the two strings are concatenated.

Casting is the most general form of conversion in Java. If a conversion can be accomplished at all in a Java program, it can be accomplished using a cast. A cast is a Java operator that is specified by a type name in parentheses. It is placed in front of the value to be converted. For example, to convert `money` to an integer value, we could put a cast in front of it:

```
dollars = (int) money;
```

The cast returns the value in `money`, truncating any fractional part. If `money` contained the value `84.69`, then after the assignment, `dollars` would contain the value `84`. Note, however, that the cast does not change the value in `money`. After the assignment operation is complete, `money` still contains the value `84.69`.

Casts are helpful in many situations where we need to treat a value temporarily as another type. For example, if we want to divide the integer value `total` by the integer value `count` and get a floating point result, we could do it as follows:

```
result = (float) total / count;
```

First, the cast operator returns a floating point version of the value in `total`. This operation does not change the value in `total`. Then, `count` is treated as a floating point value via arithmetic promotion. Now, the division operator will perform floating point division and produce the intended result. If the cast had not been included, the operation would have performed integer division and truncated the answer before assigning it to `result`. Also note that because the cast

operator has a higher precedence than the division operator, the cast operates on the value of `total`, not on the result of the division.

Self-Review Questions

(see answers in [Appendix L](#) )

SR 2.31 Why are widening conversions safer than narrowing conversions?

SR 2.32 Identify each of the following conversions as either a widening conversion or a narrowing conversion.

- a. `int` to `long`
- b. `int` to `byte`
- c. `byte` to `short`
- d. `byte` to `char`
- e. `short` to `double`

SR 2.33 Assuming `result` is a `float` variable and `value` is an `int` variable, what type of variable will `value` be after the following assignment statement is executed? Explain.

```
result = value;
```

SR 2.34 Assuming `result` is a `float` variable that contains the value 27.32 and `value` is an `int` variable that contains the

value 15, what are the values of each of the variables after the following assignment statement is executed? Explain.

```
value = (int) result;
```

SR 2.35 Given the following declarations, what result is stored by each of the following assignment statements?

```
int iResult, num1 = 17, num2 = 5;
```


```
double fResult, val1 = 12.0, val2 = 2.34;
```

- a. `iResult = num1 / num2;`
- b. `fResult = num1 / num2;`
- c. `fResult = val1 / num2;`
- d. `fResult = (double) num1 / num2;`
- e. `iResult = (int) val1 / num2;`

2.6 Interactive Programs

It is often useful to design a program to read data from the user interactively during execution. That way, new results can be computed each time the program is run, depending on the data that is entered.

The `Scanner` Class

The `Scanner` class, which is part of the Java API, provides convenient methods for reading input values of various types. The input could come from various sources, including data typed interactively by the user or data stored in a file. The `Scanner` class can also be used to parse a character string into separate pieces. **Figure 2.7**  lists some of the methods provided by the `Scanner` class.

Scanner(InputStream source)

Scanner(File source)

Scanner(String source)

Constructors: sets up the new scanner to scan values from the specified source.

String next()

Returns the next input token as a character string.

String nextLine()

Returns all input remaining on the current line as a character string.

boolean nextBoolean()

byte nextByte()

double nextDouble()

float nextFloat()

int nextInt()

long nextLong()

short nextShort()

Returns the next input token as the indicated type. Throws
InputMismatchException if the next token is inconsistent with the type.

boolean hasNext()

Returns true if the scanner has another token in its input.

Scanner useDelimiter(String pattern)

Scanner useDelimiter(Pattern pattern)

Sets the scanner's delimiting pattern.

Pattern delimiter()

Returns the pattern the scanner is currently using to match delimiters.

String findInLine(String pattern)

String findInLine(Pattern pattern)

Attempts to find the next occurrence of the specified pattern, ignoring delimiters.

Figure 2.7 Some methods of the `Scanner` class

Key Concept

The `Scanner` class provides methods for reading input of various types from various sources.

We must first create a `Scanner` object in order to invoke its methods. Objects in Java are created using the `new` operator. The following declaration creates a `Scanner` object that reads input from the keyboard:

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

This declaration creates a variable called `scan` that represents a `Scanner` object. The object itself is created by the `new` operator and a call to a special method called a **constructor** ⓘ to set up the object. The `Scanner` constructor accepts a parameter that indicates the source of the input. The `System.in` object represents the *standard input stream*, which by default is the keyboard. Creating objects using the `new` operator is discussed further in **Chapter 3** 📄.

Unless specified otherwise, a `Scanner` object assumes that white space characters (space characters, tabs, and new lines) are used to separate the elements of the input, called *tokens*, from each other. These characters are called the input **delimiters** ⓘ. The set of delimiters can be changed if the input tokens are separated by characters other than white space.

The `next` method of the `Scanner` class reads the next input token as a string and returns it. Therefore, if the input consisted of a series of words separated by spaces, each call to `next` would return the next word. The `nextLine` method reads all of the input until the end of the line is found, and returns it as one string.

The program `Echo`, shown in **Listing 2.8** □, simply reads a line of text typed by the user, stores it in a variable that holds a character string, then echoes it back to the screen.

Listing 2.8

```
//*****  
  
//  Echo.java      Author: Lewis/Loftus  
//  
//  Demonstrates the use of the nextLine method of the Scanner  
class  
//  to read a string from the user.  
//*****
```

```
import java.util.Scanner;

public class Echo
{
    //-----
    // Reads a character string from the user and prints it.
    //-----

    public static void main(String[] args)
    {
        String message;

        Scanner scan = new Scanner(System.in);

        System.out.println("Enter a line of text:");

        message = scan.nextLine();

        System.out.println("You entered: \"" + message + "\"");
    }
}
```

Output

```
Enter a line of text:
```

```
Set your laser printer on stun!  
You entered: "Set your laser printer on stun!"
```

The `import` statement above the definition of the `Echo` class tells the program that we will be using the `Scanner` class in this program. The `Scanner` class is part of the `java.util` class library. The use of the `import` statement is discussed further in [Chapter 3](#).

Various `Scanner` methods such as `nextInt` and `nextDouble` are provided to read data of particular types. The `GasMileage` program, shown in [Listing 2.9](#), reads the number of miles traveled as an integer, and the number of gallons of fuel consumed as a double, then computes the gas mileage.

Listing 2.9

```
/** *****  
  
// GasMileage.java      Author: Lewis/Loftus  
//  
// Demonstrates the use of the Scanner class to read numeric  
data.  
// *****  
  
import java.util.Scanner;
```



```
public class GasMileage
{
    //-----

    // Calculates fuel efficiency based on values entered by
    the
    // user.
    //-----

    public static void main(String[] args)
    {
        int miles;
        double gallons, mpg;

        Scanner scan = new Scanner(System.in);

        System.out.print("Enter the number of miles: ");
        miles = scan.nextInt();

        System.out.print("Enter the gallons of fuel used: ");
        gallons = scan.nextDouble();

        mpg = miles / gallons;

        System.out.println("Miles Per Gallon: " + mpg);
    }
}
```

Output

```
Enter the number of miles: 328
Enter the gallons of fuel used: 11.2
Miles Per Gallon: 29.28571428571429
```

As you can see by the output of the `GasMileage` program, the calculation produces a floating point result that is accurate to several decimal places. In Chapter 3, we discuss classes that help us format our output in various ways, including rounding a floating point value to a particular number of decimal places.

A `Scanner` object processes the input one token at a time, based on the methods used to read the data and the delimiters used to separate the input values. Therefore, multiple values can be put on the same line of input or can be separated over multiple lines, as appropriate for the situation.



Example using the Scanner class.

In [Chapter 5](#), [☐](#) we use the `Scanner` class to read input from a data file and modify the delimiters it uses to parse the data. [Appendix H](#) [☐](#) explores how to use the `Scanner` class to analyze its input using patterns called *regular expressions*.

Self-Review Questions

(see answers in [Appendix L](#) [☐](#))

SR 2.36 Identify which line of the `GasMileage` program does each of the following.

- Tells the program that we will be using the `Scanner` class.
- Creates a `Scanner` object.
- Sets up the `Scanner` object `scan` to read from the standard input stream.
- Reads an integer from the standard input stream.

SR 2.37 Assume you already have instantiated a `Scanner` object named `myScanner` and an `int` variable named `value` as follows in your program:

```
Scanner myScanner = new Scanner(System.in);  
int value = 0;
```

Write program statements that will ask the user to enter their age, and store their response in `value`.

Summary of Key Concepts

- The `print` and `println` methods represent two services provided by the `System.out` object.
- An escape sequence can be used to represent a character that would otherwise cause compilation problems.
- A variable is a name for a memory location used to hold a value of a particular data type.
- Accessing data leaves it intact in memory, but an assignment statement overwrites the old data.
- We cannot assign a value of one type to a variable of an incompatible type.
- Constants hold a particular value for the duration of their existence.
- Java has two kinds of numeric values: integer and floating point. There are four integer data types and two floating point data types.
- Java uses the 16-bit Unicode character set to represent character data.
- Expressions are combinations of operators and operands used to perform a calculation.
- Java follows a well-defined set of precedence rules that governs the order in which operators will be evaluated in an expression.
- Narrowing conversions should be avoided because they can lose information.
- The `Scanner` class provides methods for reading input of various types from various sources.

Exercises

EX 2.1 What is the difference between the literals 4, 4.0, '4', and "4"?

EX 2.2 Explain the following programming statement in terms of objects and the services they provide:

```
System.out.println("I gotta be me!");
```

EX 2.3 What output is produced by the following code fragment? Explain.

```
System.out.print("Here we go!");  
System.out.println("12345");  
System.out.print("Test this if you are not sure.");  
System.out.print("Another.");  
System.out.println();  
System.out.println("All done.");
```

EX 2.4 What is wrong with the following program statement? How can it be fixed?

```
System.out.println("To be or not to be, that  
is the question.");
```

EX 2.5 What output is produced by the following statement?
Explain.

```
System.out.println("50 plus 25 is " + 50 + 25);
```

EX 2.6 What is the output produced by the following statement?
Explain.

```
System.out.println("He thrusts his fists\n\tagainst" +  
"the post\nand still insists\n\tthe sees the \"ghost\"");
```

EX 2.7 What value is contained in the integer variable `size` after the following statements are executed?

```
size = 18;  
size = size + 12;  
size = size * 2;  
size = size / 4;
```

EX 2.8 What value is contained in the floating point variable `depth` after the following statements are executed?

```
depth = 2.4;  
depth = 20 - depth * 4;  
depth = depth / 5;
```

EX 2.9 What value is contained in the integer variable `length` after the following statements are executed?

```
length = 5;  
length *= 2;  
length *= length;  
length /= 100;
```

EX 2.10 Write four different program statements that increment the value of an integer variable `total`.

EX 2.11 Given the following declarations, what result is stored in each of the listed assignment statements?

```
int iResult, num1 = 25, num2 = 40, num3 = 17, num4 = 5;  
double fResult, val1 = 17.0, val2 = 12.78;
```

- a. `iResult = num1 / num4;`
- b. `fResult = num1 / num4;`
- c. `iResult = num3 / num4;`
- d. `fResult = num3 / num4;`
- e. `fResult = val1 / num4;`
- f. `fResult = val1 / val2;`
- g. `iResult = num1 / num2;`
- h. `fResult = (double) num1 / num2;`
- i. `fResult = num1 / (double) num2;`
- j. `fResult = (double) (num1 / num2);`

- k. `iResult = (int) (val1 / num4);`
- l. `fResult = (int) (val1 / num4);`
- m. `fResult = (int) ((double) num1 / num2);`
- n. `iResult = num3 % num4;`
- o. `iResult = num2 % num3;`
- p. `iResult = num3 % num2;`
- q. `iResult = num2 % num4;`

EX 2.12 For each of the following expressions, indicate the order in which the operators will be evaluated by writing a number beneath each operator.

- a. `a - b - c - d`
- b. `a - b + c - d`
- c. `a + b / c / d`
- d. `a + b / c * d`
- e. `a / b * c * d`
- f. `a % b / c * d`
- g. `a % b % c % d`
- h. `a - (b - c) - d`
- i. `(a - (b - c)) - d`
- j. `a - ((b - c) - d)`
- k. `a % (b % c) * d * e`
- l. `a + (b - c) * d - e`
- m. `(a + b) * c + d * e`
- n. `(a + b) * (c / d) % e`