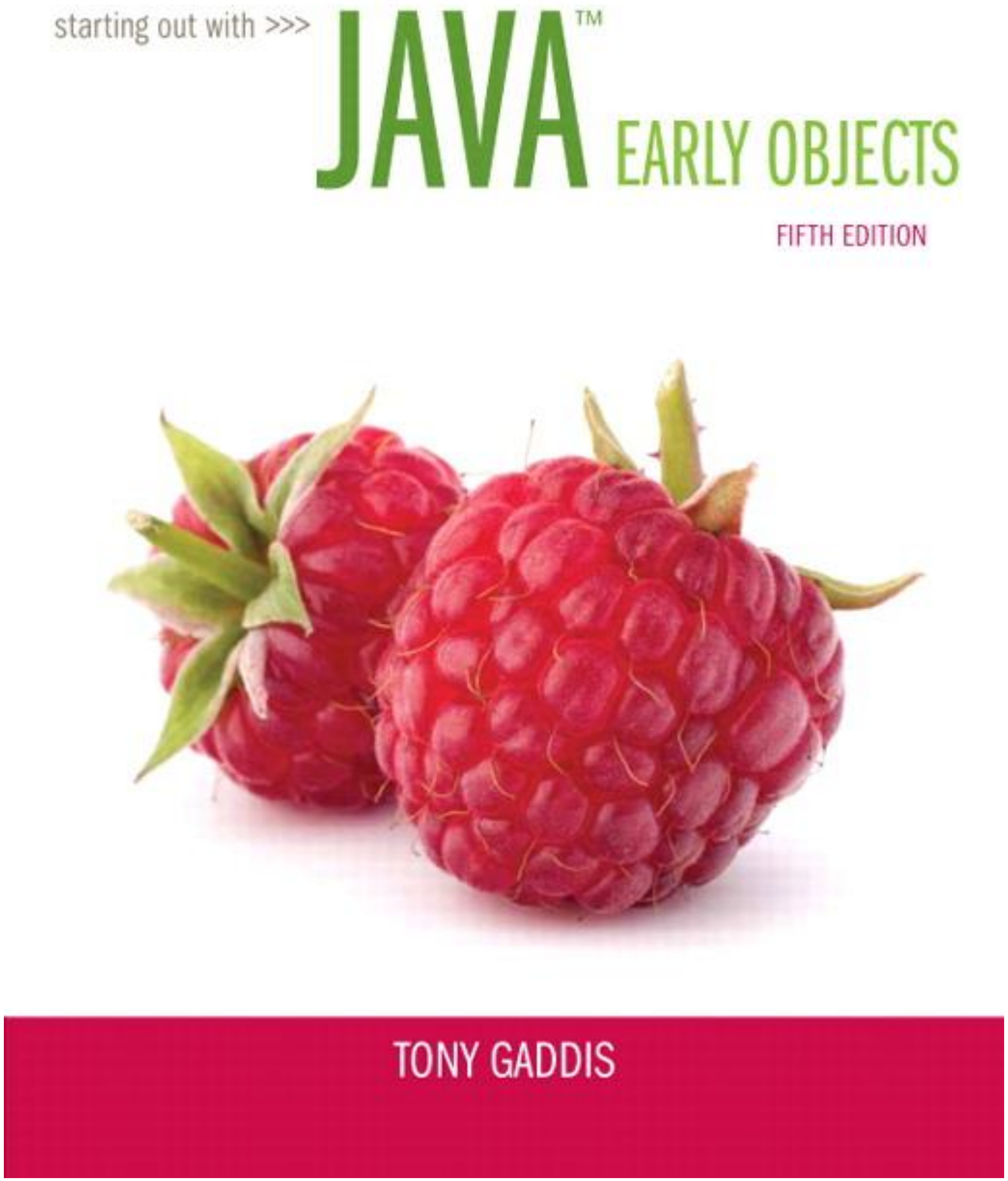


CHAPTER 2

Java Fundamentals

Addison-Wesley
is an imprint of
PEARSON

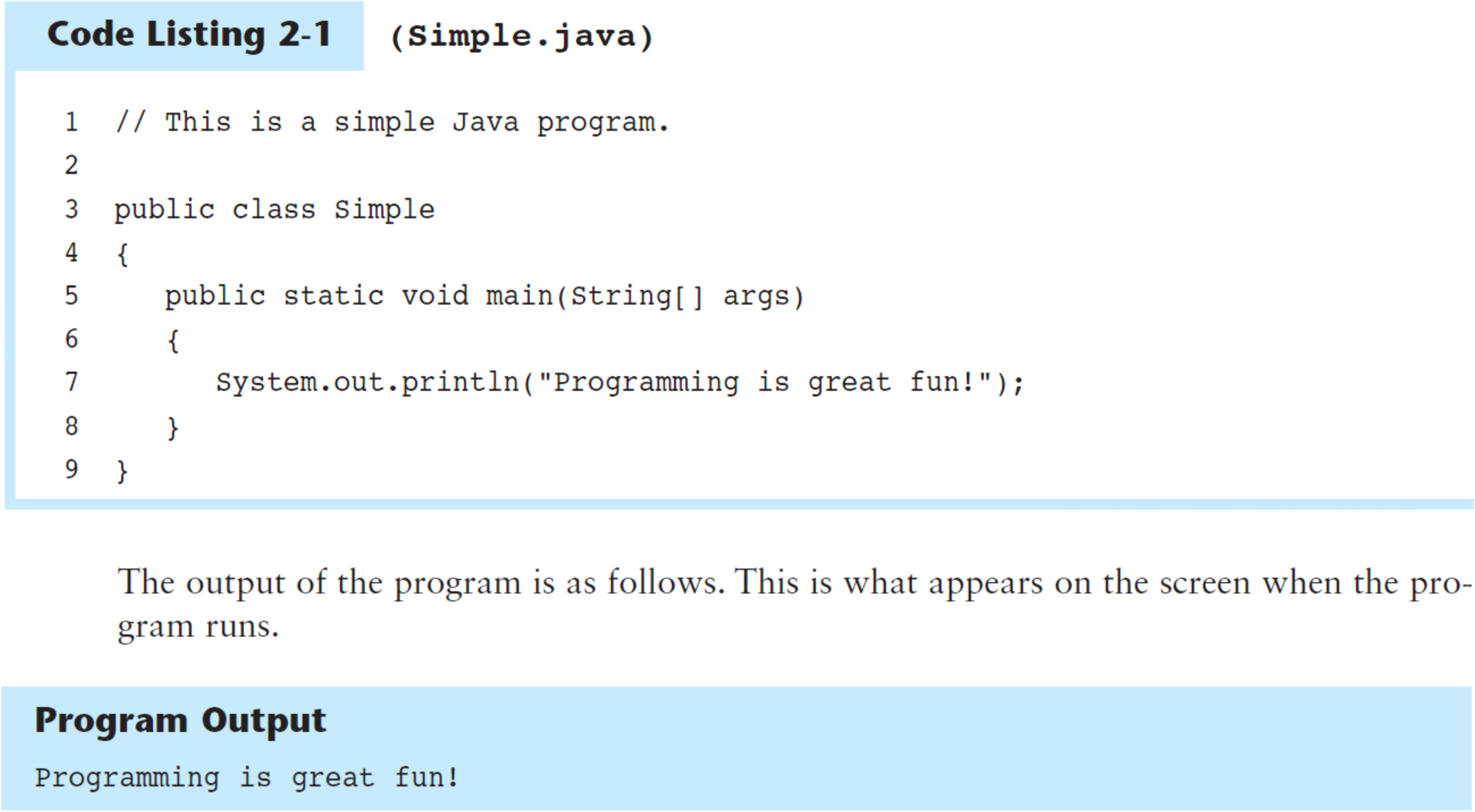
Topics

- The Parts of a Java Program
- The `System.out.print` and `System.out.println` Methods, and the Java API
- Variables and Literals
- Primitive Data Types
- Arithmetic Operators
- Combined Assignment Operators
- Conversion between Primitive Data Types
- Creating named constants with `final`

Topics (cont'd.)

- The `String` class
- Scope
- Comments
- Programming style
- Reading keyboard input
- Dialog boxes
- The `System.out.printf` method

The Parts of a Java Program



Code Listing 2-1 **(Simple.java)**

```
// This is a simple Java program.

public class Simple
{
   public static void main(String[] args)
   {
      System.out.println("Programming is great fun!");
   }
}
```

The output of the program is as follows. This is what appears on the screen when the program runs.

Program Output

```
Programming is great fun!
```

The Parts of a Java Program (cont'd.)

- **To compile the example:**

```
javac Simple.java
```

  - **Notice the `.java` file extension is needed.**
  - **This will result in a file named *`Simple.class`* being created.**

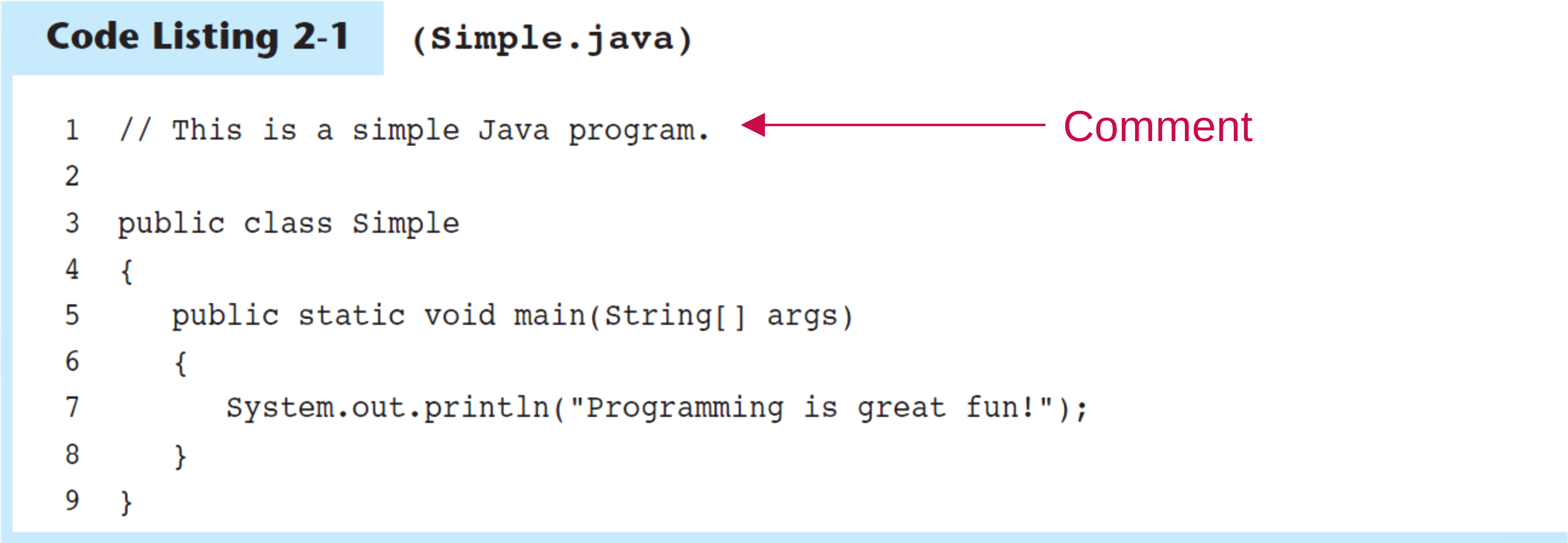
- **To run the example:**

```
java Simple
```

  - **Notice there is no file extension here.**
  - **The *java* command assumes the extension is `.class`.**

Addison-Wesley
is an imprint of
PEARSON

The Parts of a Java Program (cont’d.)



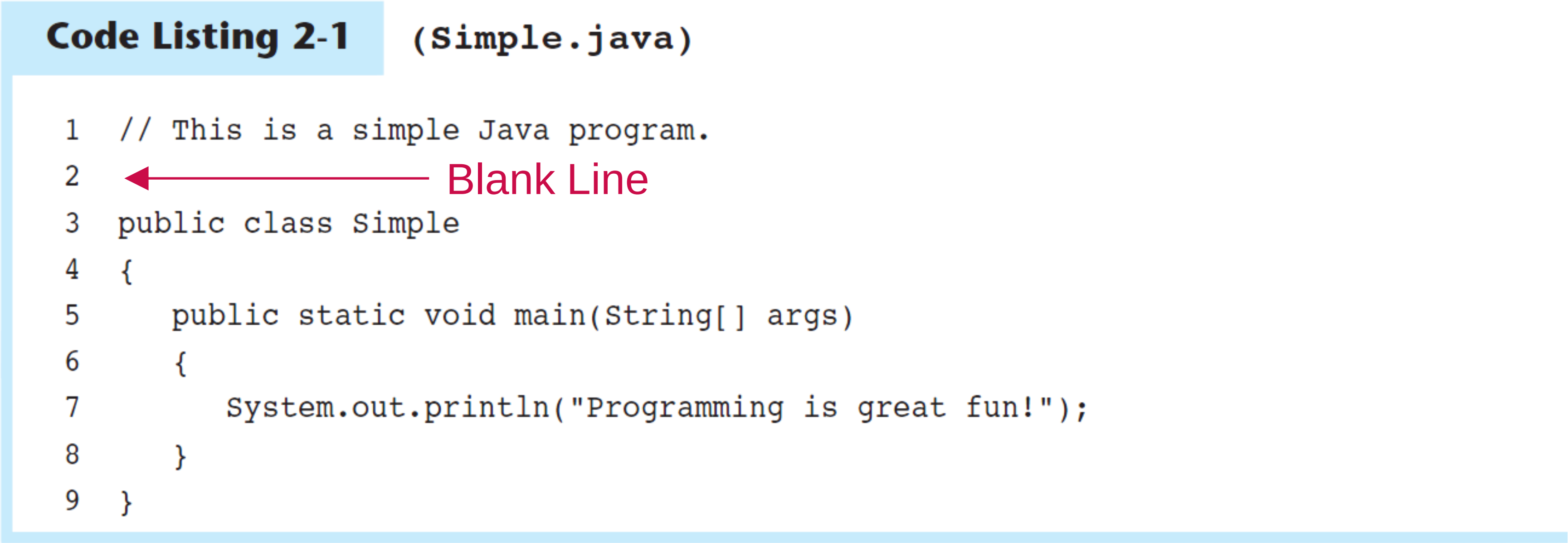
Code Listing 2-1 (**Simple.java**)

```
1  // This is a simple Java program.
2
3  public class Simple
4  {
5     public static void main(String[] args)
6     {
7        System.out.println("Programming is great fun!");
8     }
9  }
```

Comment

- The `//` in line 1 marks the beginning of a comment.
- The compiler ignores everything from the double slash to the end of the line.
- Comments are not required, but comments are very important because they help explain what is going on in the program.

The Parts of a Java Program (cont'd.)



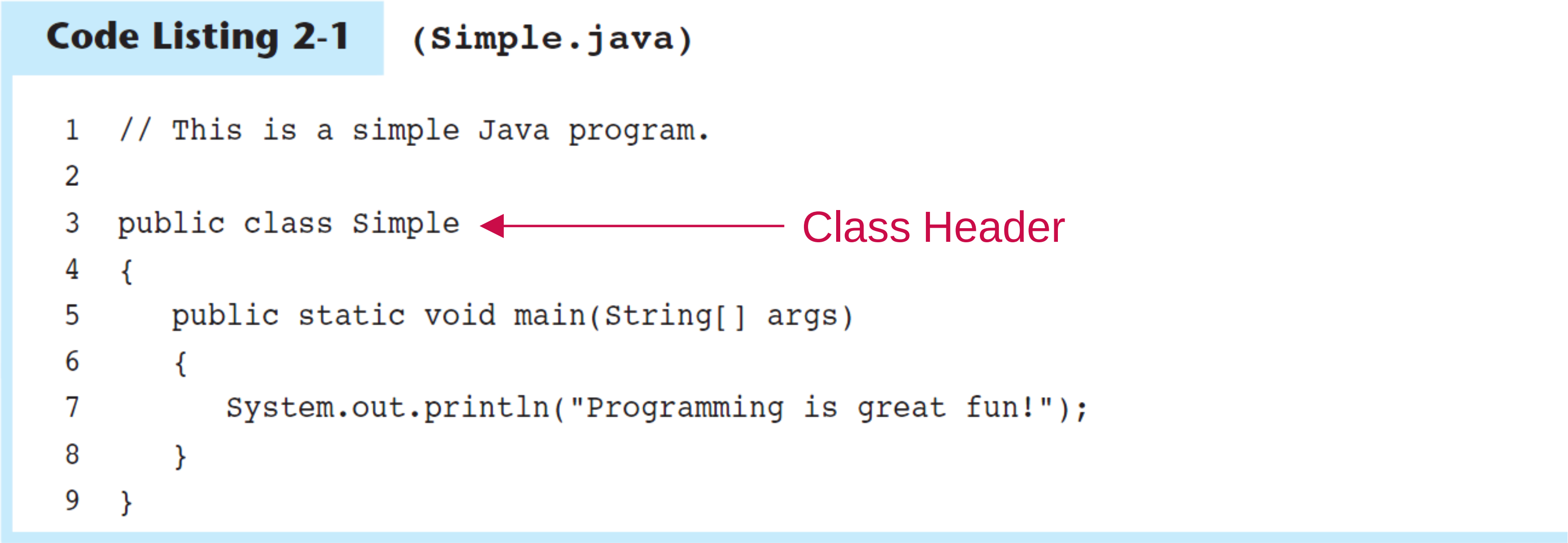
Code Listing 2-1 (Simple.java)

```
1  // This is a simple Java program.
2
3  public class Simple
4  {
5     public static void main(String[] args)
6     {
7        System.out.println("Programming is great fun!");
8     }
9  }
```

Blank Line

- **Line 2 is blank.**
- **Blank lines are often inserted by the programmer because they can make the program easier to read.**

The Parts of a Java Program (cont'd.)



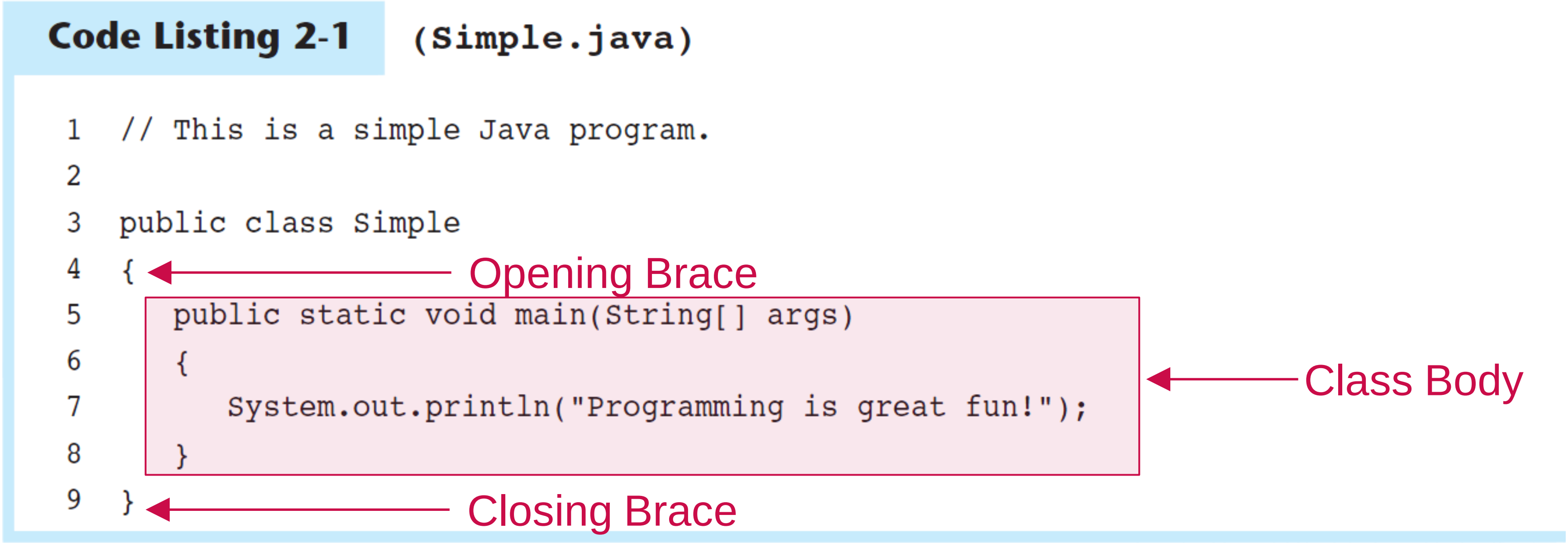
Code Listing 2-1 (Simple.java)

```
1  // This is a simple Java program.
2
3  public class Simple
4  {
5     public static void main(String[] args)
6     {
7        System.out.println("Programming is great fun!");
8     }
9  }
```

Class Header

- **Line 3 is known as a *class header*, and it marks the beginning of a *class definition*.**
- **This line of code tells the compiler that a publicly accessible class named `Simple` is being defined.**
- **A Java program must have at least one class definition.**

The Parts of a Java Program (cont’d.)



Code Listing 2-1 (**Simple.java**)

```
1  // This is a simple Java program.
2
3  public class Simple
4  {
5     public static void main(String[] args)
6     {
7        System.out.println("Programming is great fun!");
8     }
9  }
```

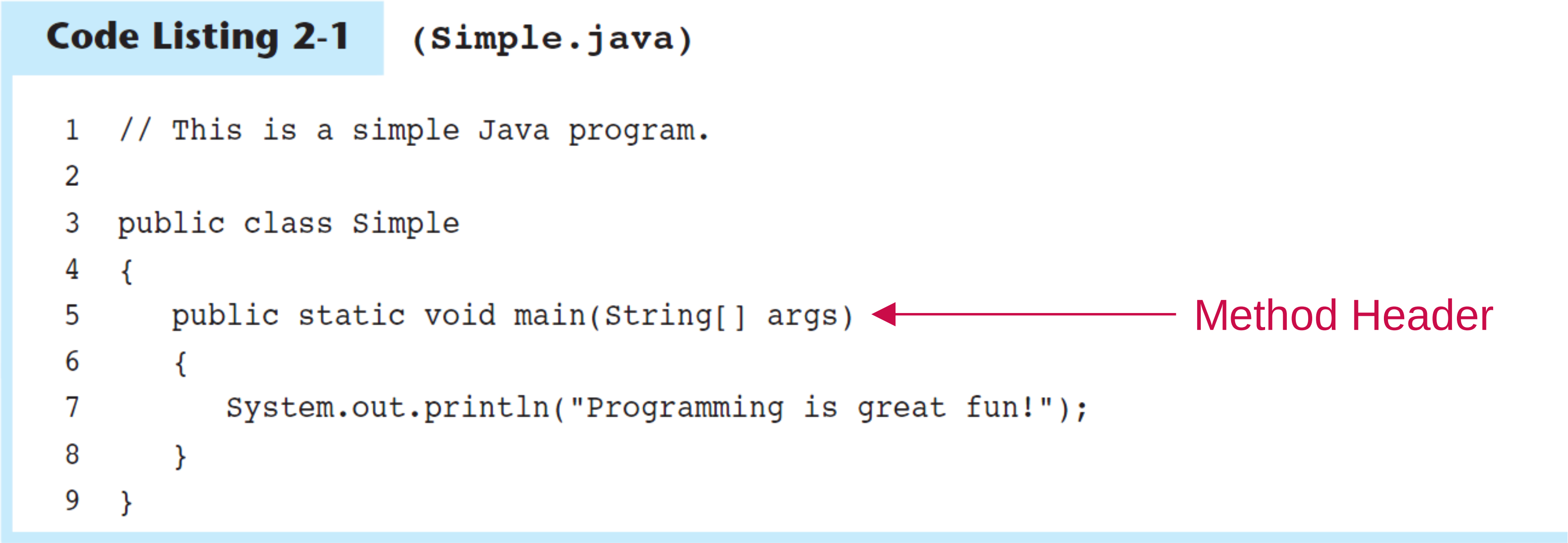
Opening Brace

Class Body

Closing Brace

- **Line 4 contains an opening brace, and it is associated with the beginning of the class definition.**
- **The last line in the program, line 9, contains the closing brace.**
- **Everything between the two braces is the *body* of the class named `Simple`.**

The Parts of a Java Program (cont’d.)



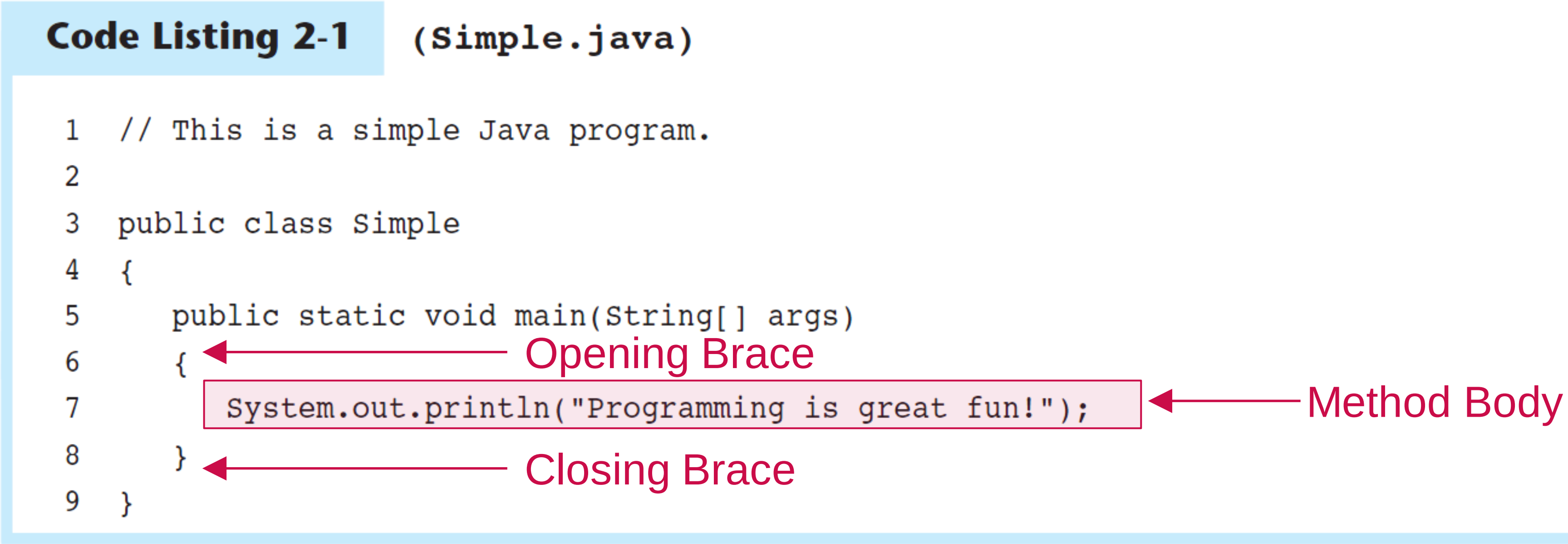
Code Listing 2-1 (Simple.java)

```
1  // This is a simple Java program.
2
3  public class Simple
4  {
5     public static void main(String[] args)
6     {
7        System.out.println("Programming is great fun!");
8     }
9  }
```

Method Header

- Line 5 is known as a *method header*, and it marks the beginning of a *method*.
- The name of the method is `main`, and the rest of the words are required for the method to be properly defined.
  - Every Java application must have a method named `main`.
  - The `main` method is the starting point of the application.

The Parts of a Java Program (cont'd.)



Code Listing 2-1 (Simple.java)

```
1  // This is a simple Java program.
2
3  public class Simple
4  {
5     public static void main(String[] args)
6     {
7        System.out.println("Programming is great fun!");
8     }
9  }
```

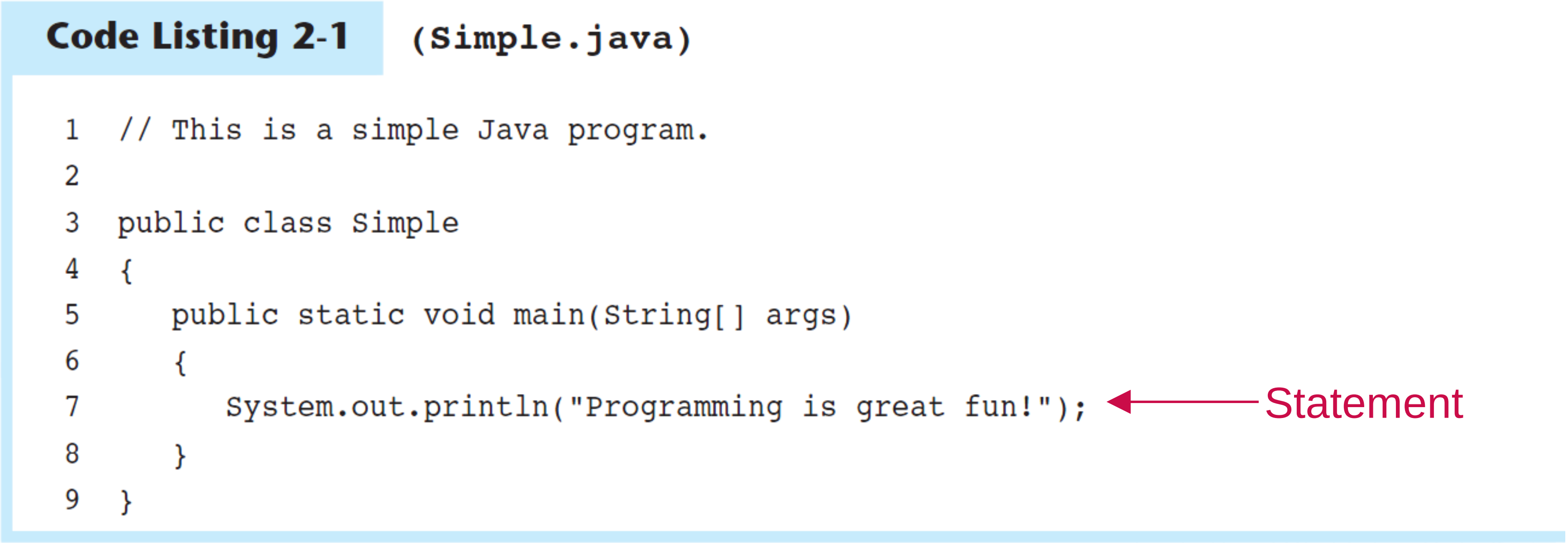
Opening Brace

Method Body

Closing Brace

- Line 6 contains an opening brace that belongs to the `main` method, and line 8 contains the closing brace.
- Everything between the two braces is the *body* of the `main` method.
- Make sure to have a closing brace for every opening brace in your program.

The Parts of a Java Program (cont’d.)



Code Listing 2-1 (Simple.java)

```
1  // This is a simple Java program.
2
3  public class Simple
4  {
5     public static void main(String[] args)
6     {
7        System.out.println("Programming is great fun!");
8     }
9  }
```

Statement

- **Line 7 contains a statement that displays a message on the screen.**
  - **The group of characters inside the quotation marks is called a *string literal*.**
  - **At the end of the line is a semicolon; it marks the end of a statement in Java.**
    - **Not every line of code ends with a semicolon, however**.

The Parts of a Java Program (cont'd.)

- Java is a case-sensitive language.
- All Java programs must be stored in a file with a `.java` file extension.
- Comments are ignored by the compiler.
- A `.java` file may contain many classes but may only have one public class.
- If a `.java` file has a public class, the class must have the same name as the file.
- Java applications must have a `main` method.
- For every left brace, or opening brace, there must be a corresponding right brace, or closing brace.
- Statements are terminated with semicolons, but comments, class headers, method headers, and braces are not.

Addison-Wesley
is an imprint of
PEARSON

The Parts of a Java Program (cont’d.)

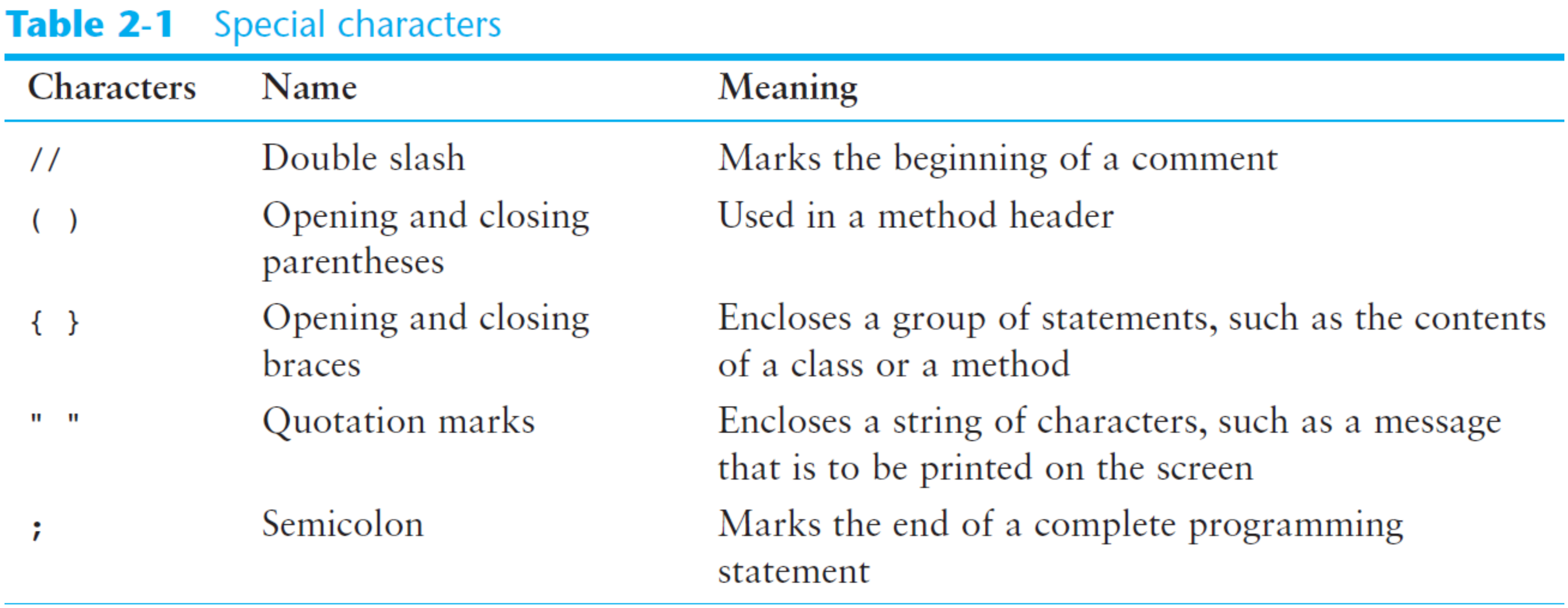


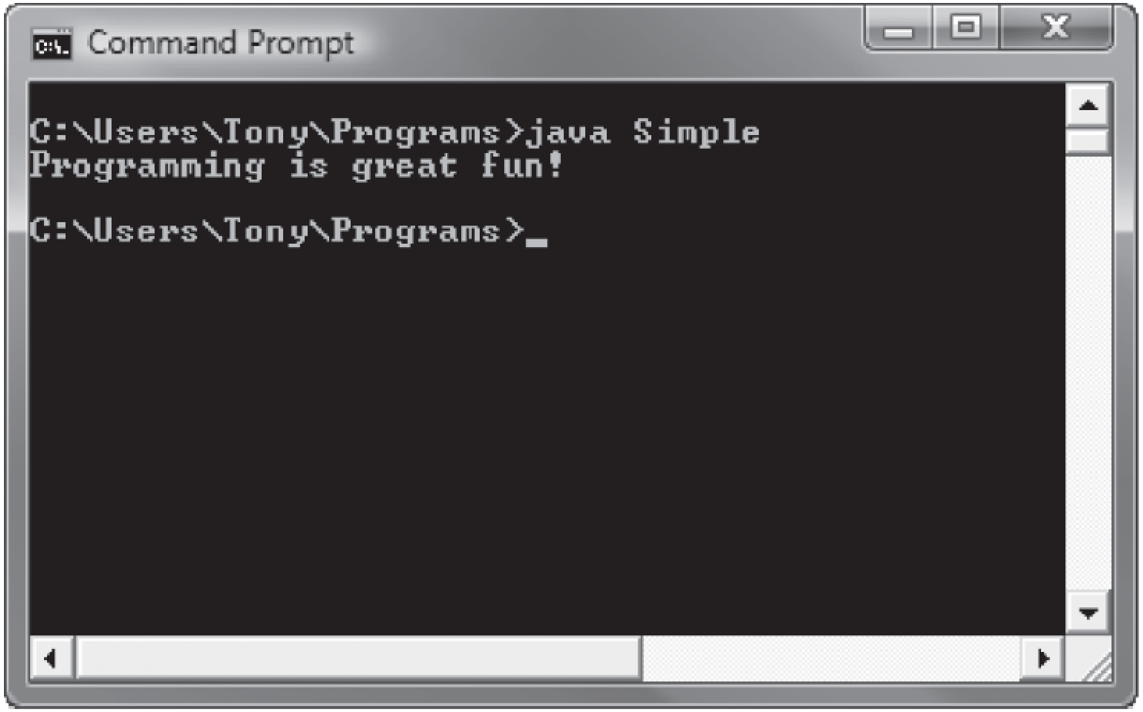
Table 2-1 Special characters

Characters	Name	Meaning
`//`	Double slash	Marks the beginning of a comment
`( )`	Opening and closing parentheses	Used in a method header
`{ }`	Opening and closing braces	Encloses a group of statements, such as the contents of a class or a method
`" "`	Quotation marks	Encloses a string of characters, such as a message that is to be printed on the screen
`;`	Semicolon	Marks the end of a complete programming statement

The System.out.print and System.out.println Methods, and the Java API

- Many of the programs that you will write will run in a console window.

Figure 2-2 A console window



```
C:\Users\Tony\Programs>java Simple
Programming is great fun!

C:\Users\Tony\Programs>_
```

Addison-Wesley is an imprint of PEARSON

The `System.out.print` and `System.out.println` Methods, and the Java API (cont’d.)

- The console window that starts a Java application is typically known as the *standard output* device.
- The *standard input* device is typically the keyboard.
- Java sends information to the standard output device by using a Java class stored in the standard Java library.

Addison-Wesley
is an imprint of
PEARSON

The `System.out.print` and `System.out.println` Methods, and the Java API (cont'd.)

- Java classes in the standard Java library are accessed using the Java Applications Programming Interface (API).
- The standard Java library is commonly referred to as the *Java API*.

The `System.out.print` and `System.out.println` Methods, and the Java API (cont’d.)

- The previous example uses the line:

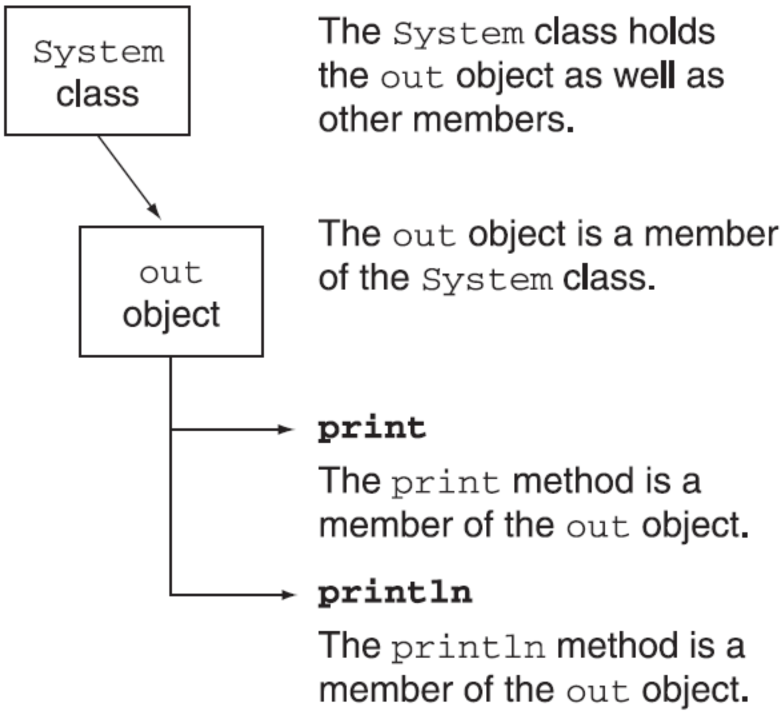
```
System.out.println("Programming is great fun!");
```

- This line uses the `System` class from the standard Java library.
- The `System` class contains methods and objects that perform system level tasks.
- The `out` object, a member of the `System` class, contains the methods `print` and `println`.

Addison-Wesley
is an imprint of
PEARSON

The `System.out.print` and `System.out.println` Methods, and the Java API (cont'd.)

Figure 2-3 Relationship among the `System` class, the `out` object, and the `print` and `println` methods



`System` class

The `System` class holds the `out` object as well as other members.

`out` object

The `out` object is a member of the `System` class.

`print`

The `print` method is a member of the `out` object.

`println`

The `println` method is a member of the `out` object.

The `System.out.print` and `System.out.println` Methods, and the Java API (cont'd.)

- **The `print` and `println` methods actually perform the task of sending characters to the output device.**
- **The line:**

```
System.out.println("Programming is great fun!");
```

  is pronounced: "*system dot out dot print line*"

- **The value inside the parenthesis, called an *argument*, will be sent to the output device (in this case, a string).**

Addison-Wesley
is an imprint of
PEARSON

The `System.out.print` and `System.out.println` Methods, and the Java API (cont'd.)

- **The `println` method places a newline character at the end of whatever is being printed out.**
  - The following lines:

```
System.out.println("This is being printed out");
System.out.println("on two separate lines.");
```

Would be printed out on separate lines since the first statement sends a newline command to the screen.

The `System.out.print` and `System.out.println` Methods, and the Java API (cont'd.)

- **The `print` statement works very similarly to the `println` statement.**
- **However, the `print` statement does not put a newline character at the end of the output.**
- **The lines:**

```
System.out.print("These lines will be");
System.out.print("printed on");
System.out.println("the same line.");
```

  - Produce the following output:

```
These lines will beprinted onthe same line.
```

  - Notice the odd spacing?
  - Why do some words run together?

The `System.out.print` and `System.out.println` Methods, and the Java API (cont'd.)

- For all of the previous examples, we have been printing out strings of characters.
- Later, we will see that much more can be printed.
- There are some special characters that can be put into the output.

```
System.out.print("This will have a newline.\n");
```

- The `\n` in the string is an escape sequence that represents the newline character.
- Escape sequences allow the programmer to print characters that otherwise would be unprintable.

The `System.out.print` and `System.out.println` Methods, and the Java API (cont’d.)

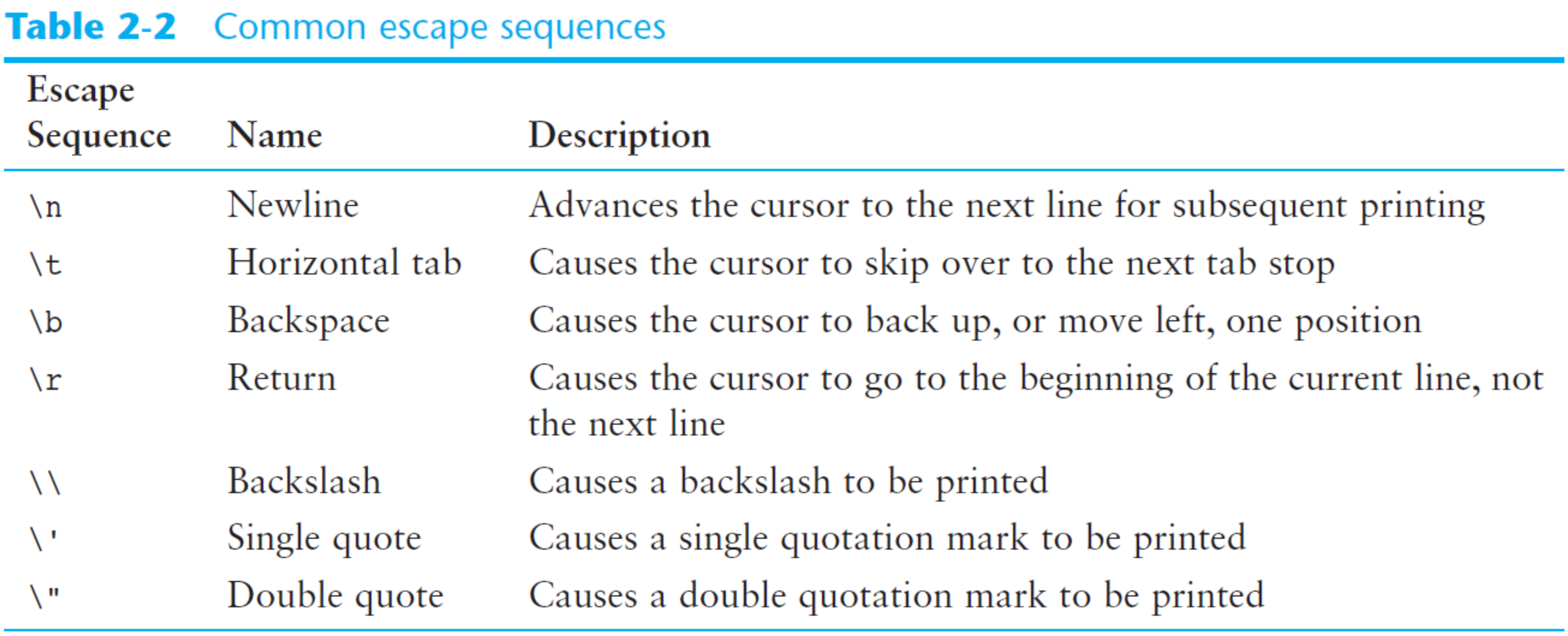


Table 2-2 Common escape sequences

Escape Sequence	Name	Description
`\n`	Newline	Advances the cursor to the next line for subsequent printing
`\t`	Horizontal tab	Causes the cursor to skip over to the next tab stop
`\b`	Backspace	Causes the cursor to back up, or move left, one position
`\r`	Return	Causes the cursor to go to the beginning of the current line, not the next line
`\\`	Backslash	Causes a backslash to be printed
`\'`	Single quote	Causes a single quotation mark to be printed
`\"`	Double quote	Causes a double quotation mark to be printed

The `System.out.print` and `System.out.println` Methods, and the Java API (cont'd.)

- Even though the escape sequences are comprised of two characters, they are treated by the compiler as a single character.

```
System.out.print("These are our top sellers:\n");
System.out.print("\tComputer games\n\tCoffee\n ");
System.out.println("\tAspirin");
```

- Would result in the following output:

```
These are our top sellers:
      Computer games
      Coffee
      Asprin
```

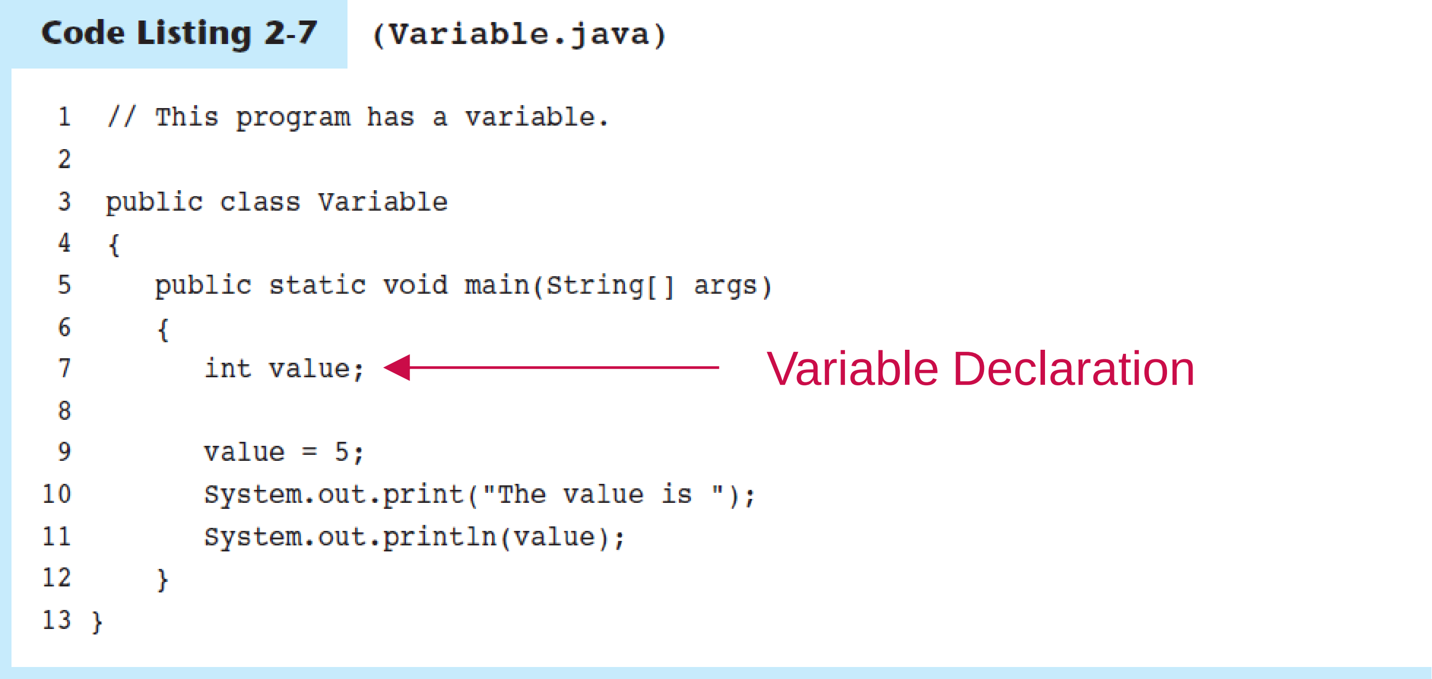
- With escape sequences, complex text output can be achieved.

Addison-Wesley
is an imprint of
PEARSON

Variables and Literals

- A *variable* is a named storage location in the computer's memory.
- A *literal* is a value that is written into the code of a program.
- Programmers determine the number and type of variables a program will need.

Variables and Literals (cont’d.)



Code Listing 2-7 (Variable.java)

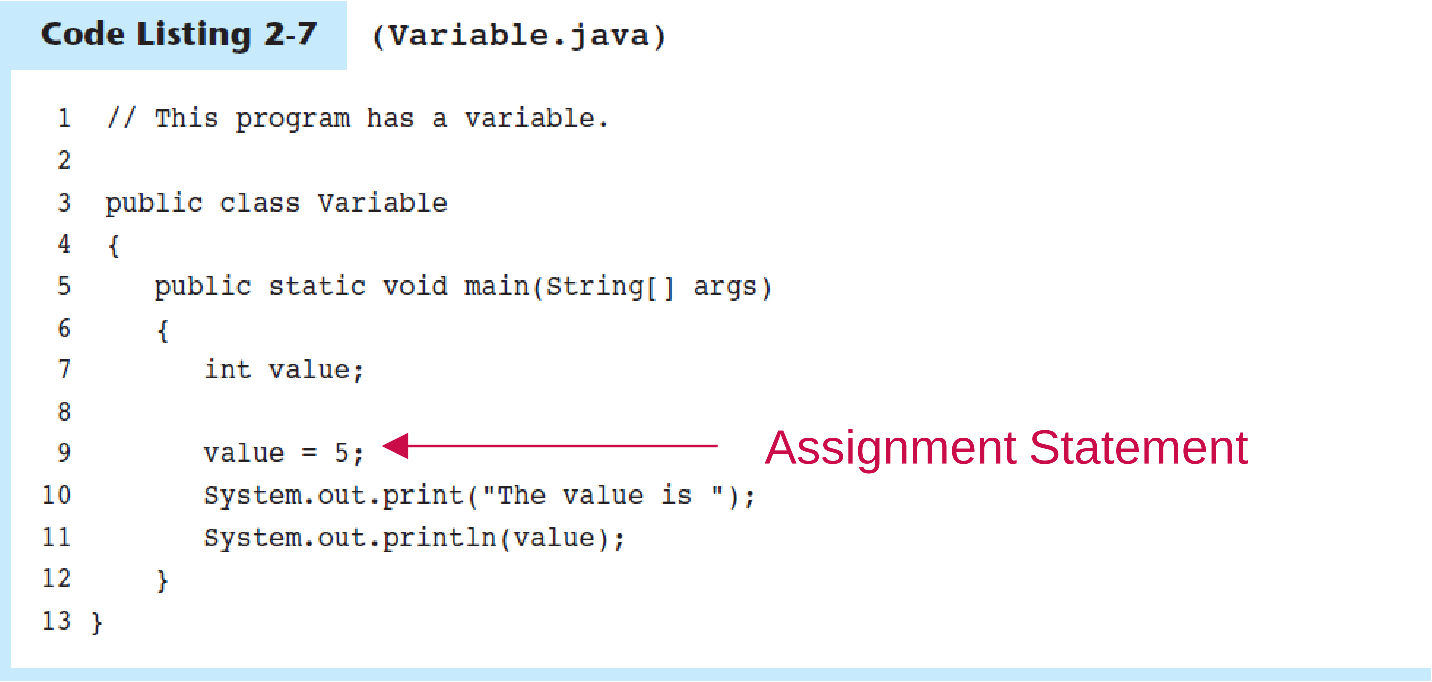
```
 1 // This program has a variable.
 2
 3 public class Variable
 4 {
 5    public static void main(String[] args)
 6    {
 7       int value;
 8
 9       value = 5;
10       System.out.print("The value is ");
11       System.out.println(value);
12    }
13 }
```

Variable Declaration

- **Line 7 contains a variable declaration.**
- **Variables must be declared before they are used.**
- **A variable declaration tells the compiler the variable’s name and the type of data it will hold.**
- **This variable’s name is `value`, and the word `int` means that it will hold an integer value.**

Notice that variable declarations end with a semicolon.

Variables and Literals (cont’d.)



Code Listing 2-7 (Variable.java)

```
 1  // This program has a variable.
 2
 3  public class Variable
 4  {
 5     public static void main(String[] args)
 6     {
 7        int value;
 8
 9        value = 5;
10        System.out.print("The value is ");
11        System.out.println(value);
12     }
13 }
```

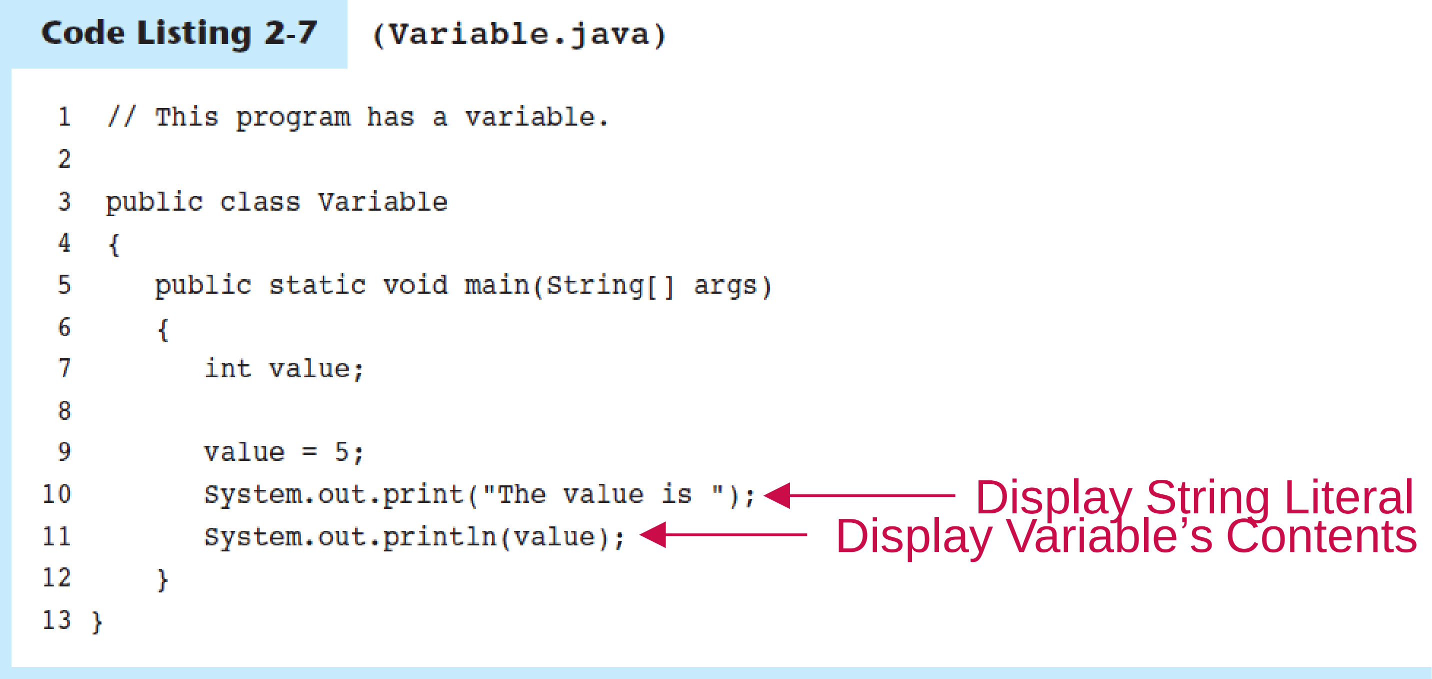
Assignment Statement

- **Line 9 contains an assignment statement.**
- **The equal sign is an operator that stores the value on its right (in this case 5) into the variable named on its left.**
- **After this line executes, the value variable will contain the value 5.**

Line 9 *doesn’t print anything. It runs silently behind the scenes.*

Addison-Wesley
is an imprint of
PEARSON

Variables and Literals (cont'd.)



Code Listing 2-7 (Variable.java)

```
 1  // This program has a variable.
 2
 3  public class Variable
 4  {
 5     public static void main(String[] args)
 6     {
 7        int value;
 8
 9        value = 5;
10        System.out.print("The value is ");
11        System.out.println(value);
12     }
13  }
```

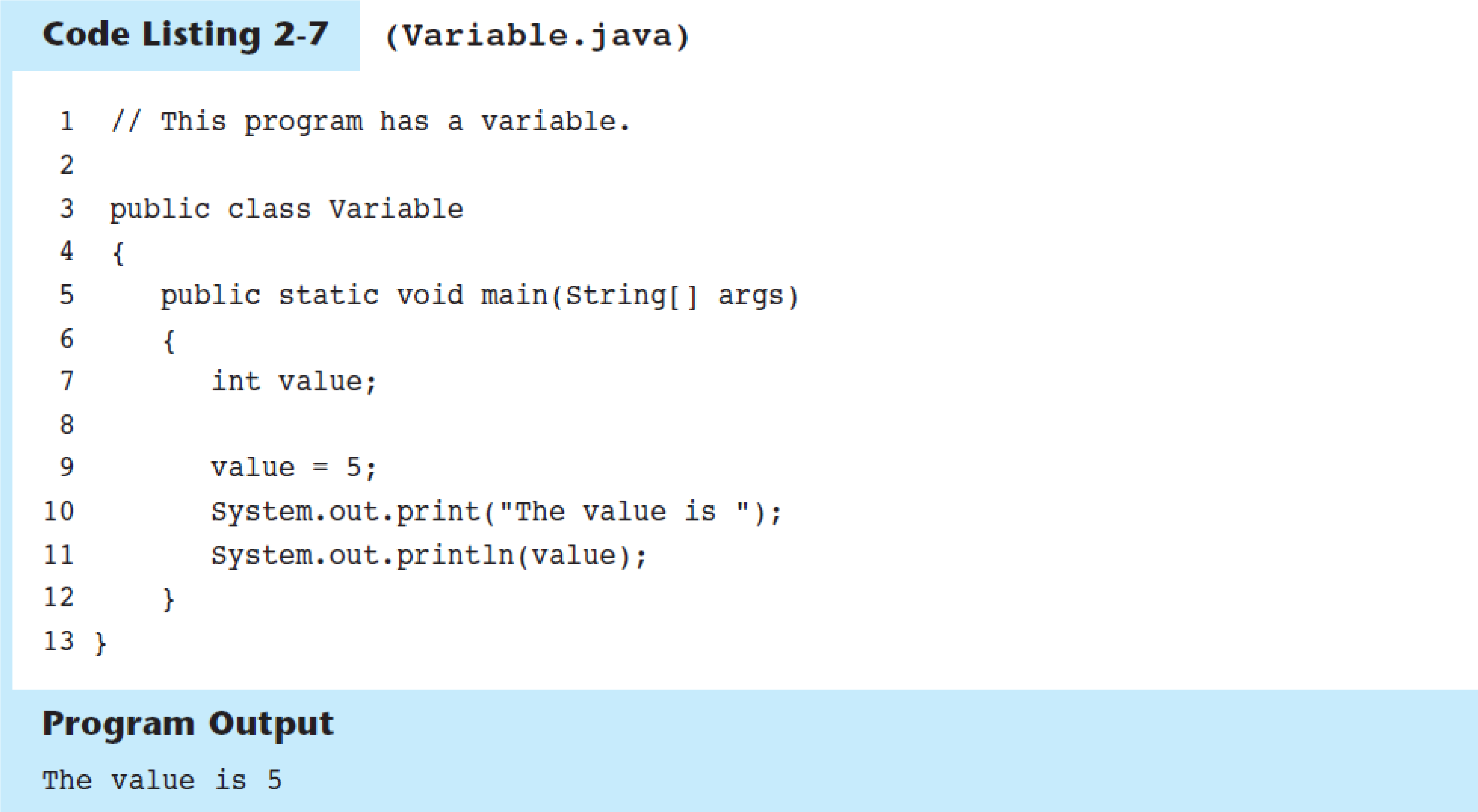
Display String Literal

Display Variable's Contents

- **Line 10 sends the string literal `"The value is "` to the `print` method.**
- **Line 11 send the name of the `value` variable to the `println` method.**
- **When you send a variable name to `print` or `println`, the variable's contents are displayed.**

Notice there are no quotation marks around the variable `value`.

Variables and Literals (cont’d.)



Code Listing 2-7 (Variable.java)

```
// This program has a variable.

public class Variable
{
   public static void main(String[] args)
   {
      int value;

      value = 5;
      System.out.print("The value is ");
      System.out.println(value);
   }
}
```

Program Output

```
The value is 5
```

Addison-Wesley
is an imprint of
PEARSON

Displaying Multiple Items with the + Operator

- **The + operator can be used in two ways.**
  - as a concatenation operator
  - as an addition operator
- **If either side of the + operator is a string, the result will be a string.**

```
System.out.println("Hello " + "World");
System.out.println("The value is: " + 5);
System.out.println("The value is: " + value);
System.out.println("The value is: " + '/n' + 5);
```

Addison-Wesley
is an imprint of
PEARSON

String Concatenation

- **Java commands that have string literals must be treated with care.**
- **A string literal value cannot span lines in a Java source code file.**

```
System.out.println("This line is too long and now it
has spanned more than one line, which will cause a
syntax error to be generated by the compiler. ");
```

String Concatenation (cont’d.)

- The String concatenation operator can be used to fix this problem.

```
System.out.println("These lines are " +
                   "now ok and will not " +
                   "cause the error as before.");
```

- String concatenation can join various data types.

```
System.out.println("We can join a string to " +
                   "a number like this: " + 5);
```

Addison-Wesley
is an imprint of

PEARSON

String Concatenation (cont’d.)

- The Concatenation operator can be used to format complex String objects.

```
System.out.println("The following will be printed " +
                   "in a tabbed format: " +
                   "\n\tFirst = " + 5 * 6 + ", " +
                   "\n\tSecond = " + (6 + 4) + "," +
                   "\n\tThird = " + 16.7 + ".");
```

- Notice that if an addition operation is also needed, it must be put in parenthesis.

Identifiers

- **Identifiers are programmer-defined names for:**
  - classes
  - variables
  - methods
- **Identifiers may not be any of the Java reserved key words.**

Identifiers (cont'd.)

- **Identifiers must follow certain rules:**
  - An identifier may only contain:
    - letters `a–z` or `A–Z`,
    - the digits `0–9`,
    - underscores (`_`), or
    - the dollar sign (`$`)
  - The first character may not be a digit.
  - Identifiers are case sensitive.
    - `itemsOrdered` is not the same as `itemsordered`.
  - Identifiers cannot include spaces.

Addison-Wesley
is an imprint of

PEARSON

Class Names

- **Variable names should begin with a lower case letter and then capitalize the first letter of each word thereafter:**

  Ex: `int caTaxRate`

- **Class names should begin with a capital letter and each word thereafter should be capitalized.**

  Ex: `public class BigLittle`

- **This helps differentiate the names of variables from the names of classes.**

Addison-Wesley
is an imprint of
PEARSON

Primitive Data Types

- **Primitive data types are built into the Java language and are not derived from classes.**
- **There are 8 Java primitive data types.**
  - `byte`
  - `short`
  - `int`
  - `long`
  - `float`
  - `double`
  - `boolean`
  - `char`

Numeric Data Types

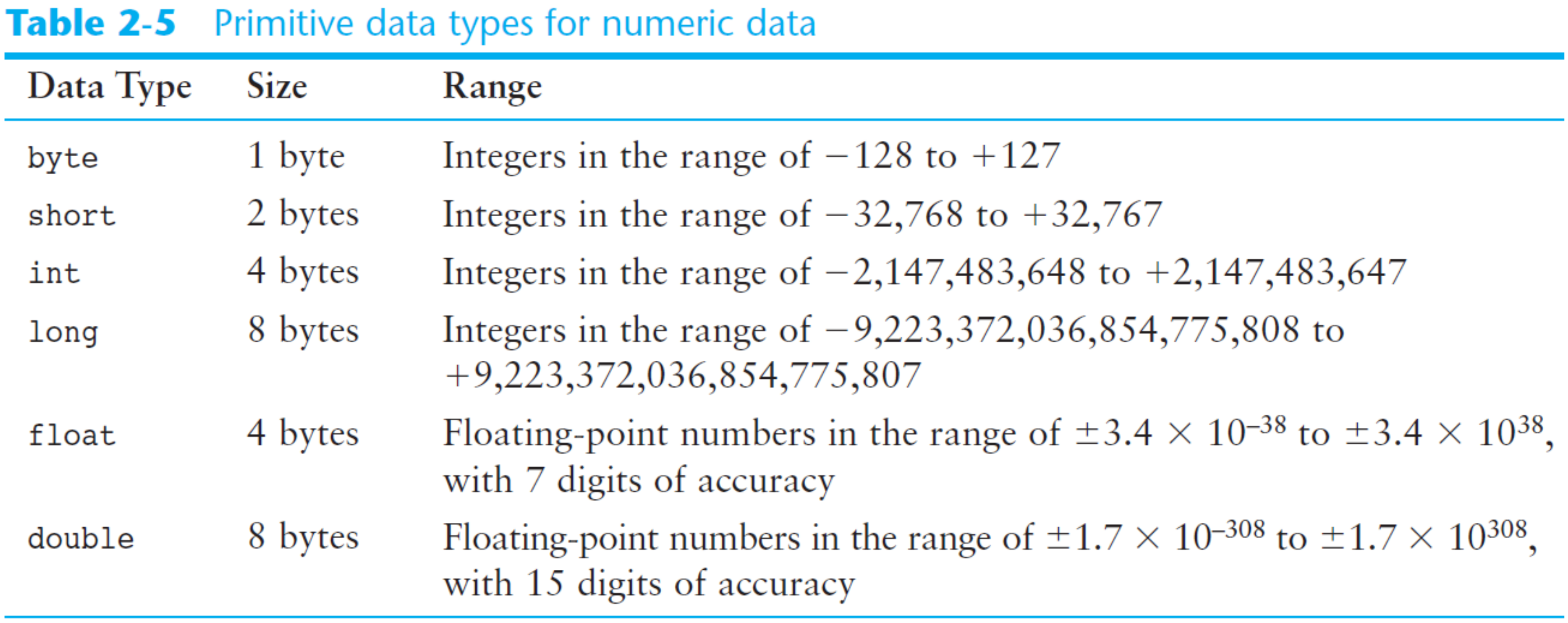


Table 2-5 Primitive data types for numeric data

Data Type	Size	Range
`byte`	1 byte	Integers in the range of −128 to +127
`short`	2 bytes	Integers in the range of −32,768 to +32,767
`int`	4 bytes	Integers in the range of −2,147,483,648 to +2,147,483,647
`long`	8 bytes	Integers in the range of −9,223,372,036,854,775,808 to +9,223,372,036,854,775,807
`float`	4 bytes	Floating-point numbers in the range of $\pm 3.4 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$, with 7 digits of accuracy
`double`	8 bytes	Floating-point numbers in the range of $\pm 1.7 \times 10^{-308}$ to $\pm 1.7 \times 10^{308}$, with 15 digits of accuracy

Addison-Wesley
is an imprint of
PEARSON

Variable Declarations

- **Variable Declarations take the following form:**
  - *DataType VariableName;*
    - `byte inches;`
    - `short month;`
    - `int speed;`
    - `long timeStamp;`
    - `float salesCommission;`
    - `double distance;`

Addison-Wesley
is an imprint of

PEARSON

Integer Data Types

- `byte`, `short`, `int`, and `long` are all integer data types.
- They can hold whole numbers such as 5, 10, 23, 89, etc.
- Integer data types cannot hold numbers that have a decimal point in them.
- Integers embedded into Java source code are called *integer literals*.

Floating-Point Data Types

- **Data types that allow fractional values are called *floating-point* numbers.**
  - 1.7 and -45.316 are floating-point numbers.
- **In Java there are two data types that can represent floating-point numbers.**
  - `float`- also called *single precision*
    - (7 decimal points)
  - `double` - also called *double precision*
    - (15 decimal points)

Floating-Point Literals

- **When floating-point numbers are embedded into Java source code they are called *floating-point literals*.**
- **The default data type for floating-point literals is `double`.**
  - 29.75, 1.76, and 31.51 are `double` data types.
- **Java is a *strongly-typed* language**

Addison-Wesley
is an imprint of
PEARSON

Floating-Point Literals (cont’d.)

- **Literals cannot contain embedded currency symbols or commas.**
  - `grossPay = $1,257.00; // ERROR!`
  - `grossPay = 1257.00;    // Correct.`
- **Floating-point literals can be represented in *scientific notation*.**
  - 47,281.97 == 4.728197 x 10^4.
- **Java uses *E notation* to represent values in scientific notation.**
  - 4.728197X10^4 == 4.728197E4.

Addison-Wesley
is an imprint of
PEARSON

Scientific and E Notation

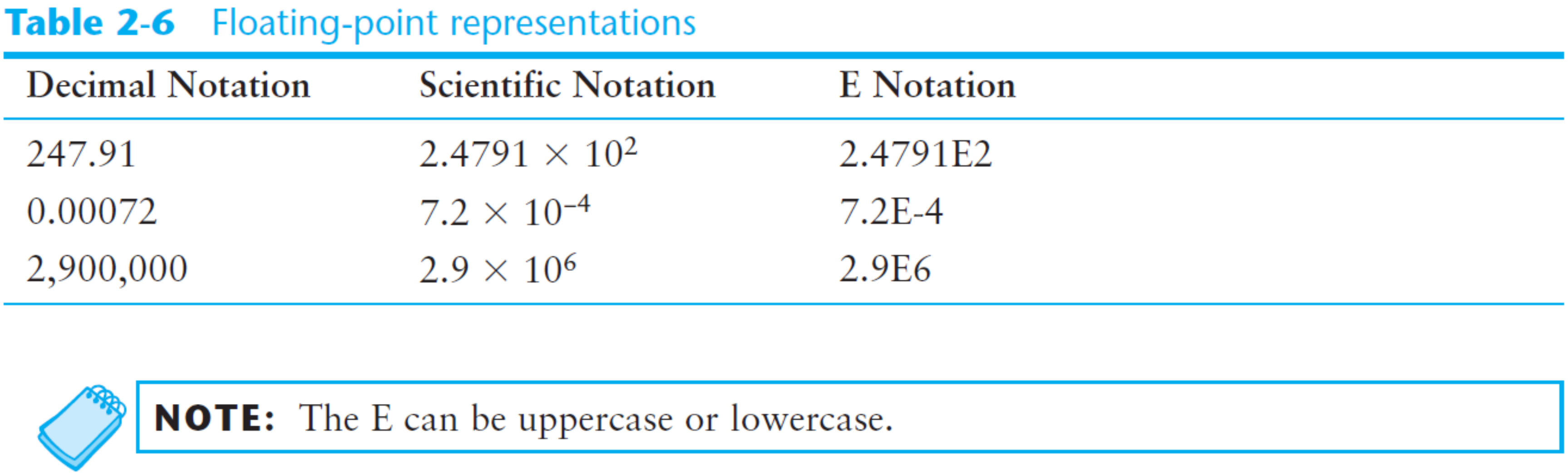


Table 2-6 Floating-point representations

Decimal Notation	Scientific Notation	E Notation
247.91	2.4791×10^{2}	2.4791E2
0.00072	7.2×10^{-4}	7.2E-4
2,900,000	2.9×10^{6}	2.9E6

NOTE: The E can be uppercase or lowercase.

The `boolean` Data Type

- **The Java `boolean` data type can have two possible values.**
  - `true`
  - `false`
- **The value of a `boolean` variable may only be copied into a `boolean` variable.**

Addison-Wesley
is an imprint of
PEARSON

The `char` Data Type

- **The Java `char` data type provides access to single characters.**
- **`char` literals are enclosed in single quote marks.**
  - `'a', 'Z', '\n', '1'`
- **Don’t confuse `char` literals with string literals.**
  - `char` literals are enclosed in single quotes.
  - String literals are enclosed in double quotes.

Unicode

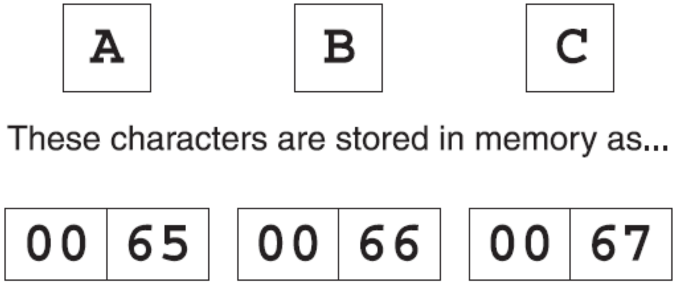
- Internally, characters are stored as numbers.
- Character data in Java is stored as Unicode characters.
- The Unicode character set can consist of 65536 (2^{16}) individual characters.
- This means that each character takes up 2 bytes in memory.
- The first 256 characters in the Unicode character set are compatible with the ASCII* character set.

*American Standard Code for Information Interchange

Addison-Wesley
is an imprint of
PEARSON

Unicode (cont'd.)

Figure 2-4 Characters and how they are stored in memory



These characters are stored in memory as...

Variable Assignment and Initialization

- **In order to store a value in a variable, an *assignment statement* must be used.**
- **The *assignment operator* is the equal (=) sign.**
- **The operand on the left side of the assignment operator must be a variable name.**
- **The operand on the right side must be either a literal or expression that evaluates to a type that is compatible with the type of the variable.**

Code Listing 2-16 (Initialize.java)

```
// This program shows variable initialization.

public class Initialize
{
   public static void main(String[] args)
   {
      int month = 2, days = 28;

      System.out.println("Month " + month + " has " +
                         days + " days.");
   }
}
```

Program Output

```
Month 2 has 28 days.
```

Addison-Wesley
is an imprint of
PEARSON

Variable Assignment and Initialization (cont'd.)

- Variables can only hold one value at a time.
- Local variables do not receive a default value.
- Local variables must have a valid type in order to be used.

Addison-Wesley
is an imprint of
PEARSON

Arithmetic Operators

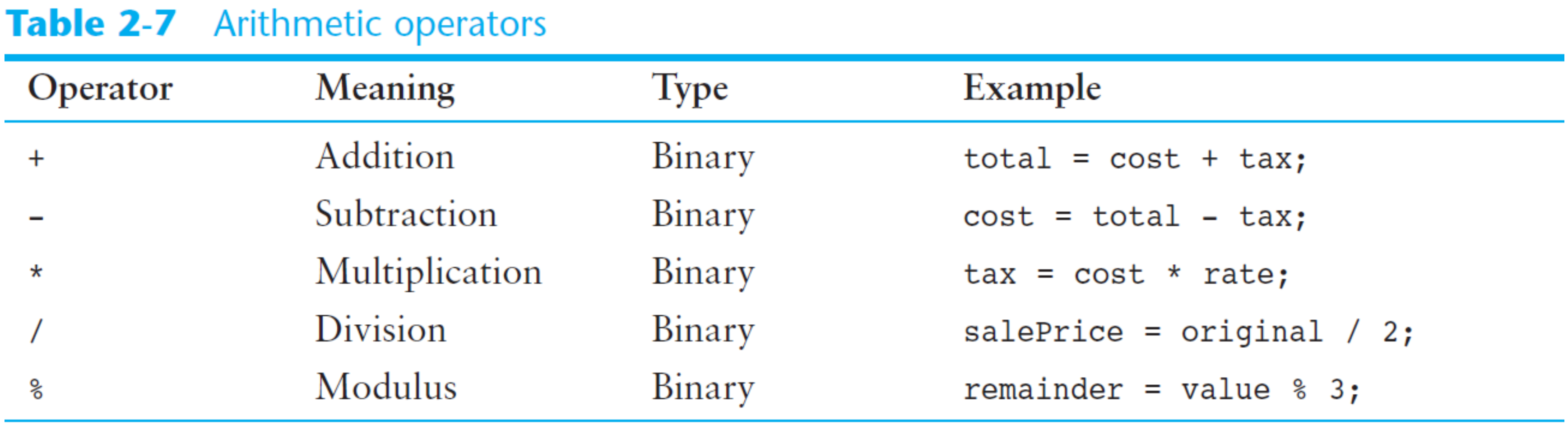


Table 2-7 Arithmetic operators

Operator	Meaning	Type	Example
`+`	Addition	Binary	`total = cost + tax;`
`-`	Subtraction	Binary	`cost = total - tax;`
`*`	Multiplication	Binary	`tax = cost * rate;`
`/`	Division	Binary	`salePrice = original / 2;`
`%`	Modulus	Binary	`remainder = value % 3;`

Arithmetic Operators (cont’d.)

- The operators are called binary operators because they must have two operands.
- Each operator must have a left and right operand.
- The arithmetic operators work as one would expect.
- It is an error to try to divide any number by zero.
- When working with two integer operands, the division operator requires special attention.

Addison-Wesley
is an imprint of
PEARSON

Integer Division

- **Division can be tricky.**

  In a Java program, what is the value of 1/2?

- **You might think the answer is 0.5…**
- **But, that’s wrong.**
- **The answer is simply 0.**
- **Integer division will truncate any decimal remainder.**

Addison-Wesley
is an imprint of
PEARSON

Operator Precedence

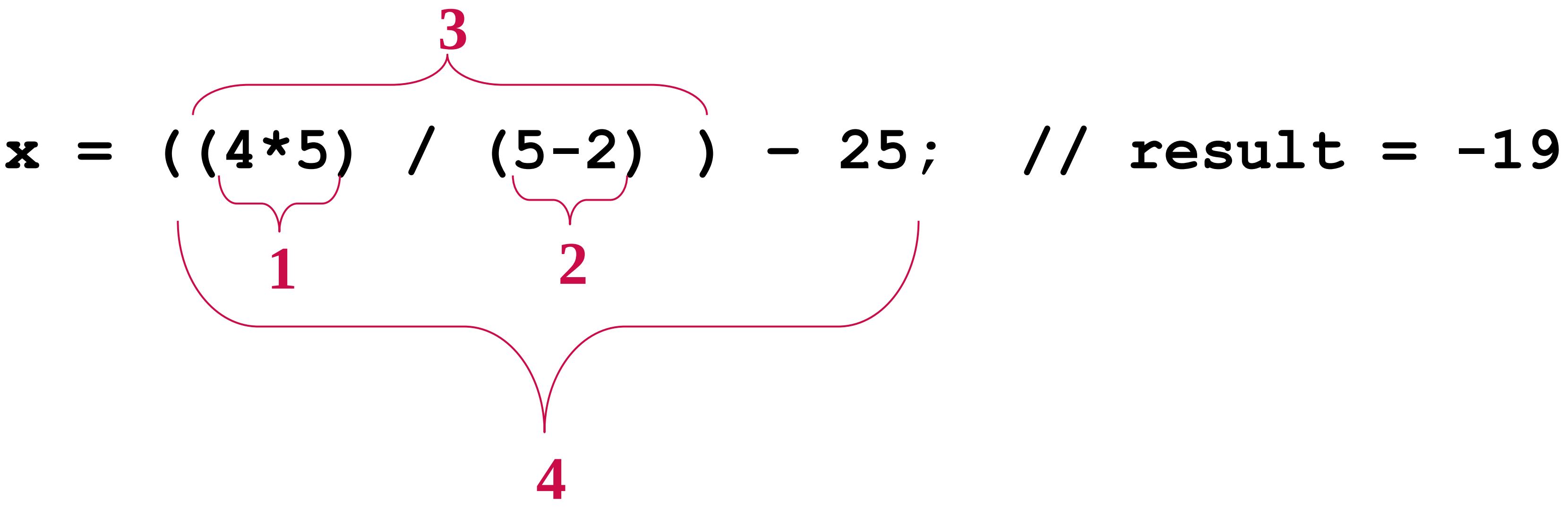
- Mathematical expressions can be very complex.
- There is a set order in which arithmetic operations will be carried out.

	Operator	Associativity	Example	Result
Higher Priority	– (unary negation)	right to left	`x = -4 + 3;`	-1
	`* / %`	left to right	`x = -4 + 4 % 3 * 13 + 2;`	11
Lower Priority	`+ -`	left to right	`x = 6 + 3 - 4 + 6 * 3;`	23

Addison-Wesley
is an imprint of
PEARSON

Grouping with Parenthesis

- When parenthesis are used in an expression, the inner most parenthesis are processed first.
- If two sets of parenthesis are at the same level, they are processed left to right.



```
x = ((4*5) / (5-2) ) - 25;  // result = -19
```

The `Math` Class

- **The Java API provides a class named `Math`, which contains several methods that are useful for performing complex mathematical operations.**
  - In Java, raising a number to a power requires the `Math.pow` method

```
double result = math.pow(4.0, 2.0);
```

  - The `Math.sqrt` method accepts a `double` value as its argument and returns the square root of the value

```
double result = math.sqrt(9.0);
```

Combined Assignment Operators

- Java has some combined assignment operators.
- These operators allow the programmer to perform an arithmetic operation and assignment with a single operator.
- Although not required, these operators are popular since they shorten simple equations.

Combined Assignment Operators (cont’d.)

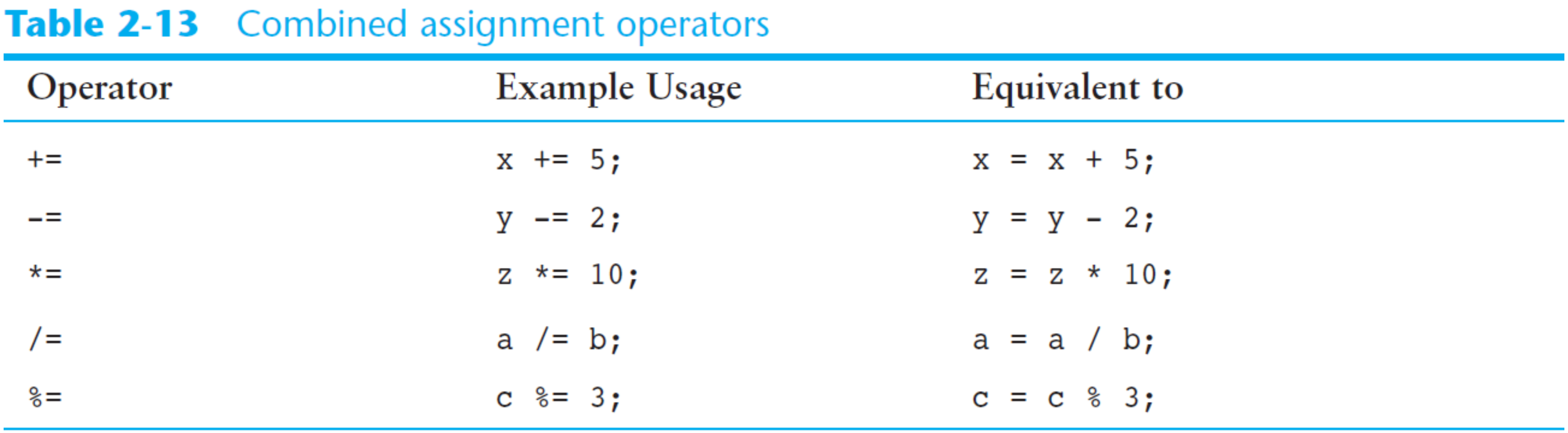


Table 2-13 Combined assignment operators

Operator	Example Usage	Equivalent to
`+=`	`x += 5;`	`x = x + 5;`
`-=`	`y -= 2;`	`y = y - 2;`
`*=`	`z *= 10;`	`z = z * 10;`
`/=`	`a /= b;`	`a = a / b;`
`%=`	`c %= 3;`	`c = c % 3;`

Conversion between Primitive Data Types

- **Java is a *strongly typed language.***
  - Before a value is assigned to a variable, Java checks the data types of the variable and the value being assigned to it to determine if they are compatible.
  - When you try to assign an incompatible value to a variable, an error occurs at compile-time.

Conversion between Primitive Data Types (cont'd.)

- **For example, look at the following statements:**

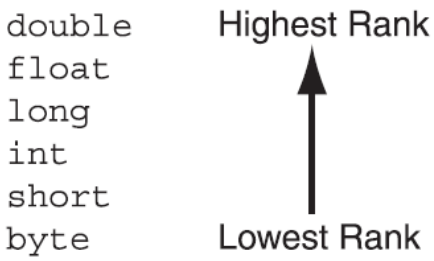
```
int x;
double y = 2.5;
x = y;
```

This statement will cause a compiler error because it is trying to assign a `double` value (2.5) in an `int` variable.

Conversion between Primitive Data Types (cont'd.)

- **The Java primitive data types are ranked, as shown here:**

Figure 2-6 Primitive data type ranking



double Highest Rank
float
long
int
short
byte Lowest Rank

Conversion between Primitive Data Types (cont’d.)

- **Widening conversions are allowed.**
  - This is when a value of a lower-ranked data type is assigned to a variable of a higher-ranked data type.
- **Example:**

```
double x;
int y = 10;
x = y;
```

Widening Conversion

Conversion between Primitive Data Types (cont'd.)

- **Narrowing conversions are *not* allowed.**
  - This is when a value of a higher-ranked data type is assigned to a variable of a lower-ranked data type.
- **Example:**

```
int x;
double y = 2.5;
x = y;
```

Narrowing Conversion

Conversion between Primitive Data Types (cont’d.)

- **Cast Operators**
  - Let you manually convert a value, even if it means that a narrowing conversion will take place.
- **Example:**

```
int x;
double y = 2.5;
x = (int)y;
```

Cast Operator

Conversion between Primitive Data Types (cont’d.)

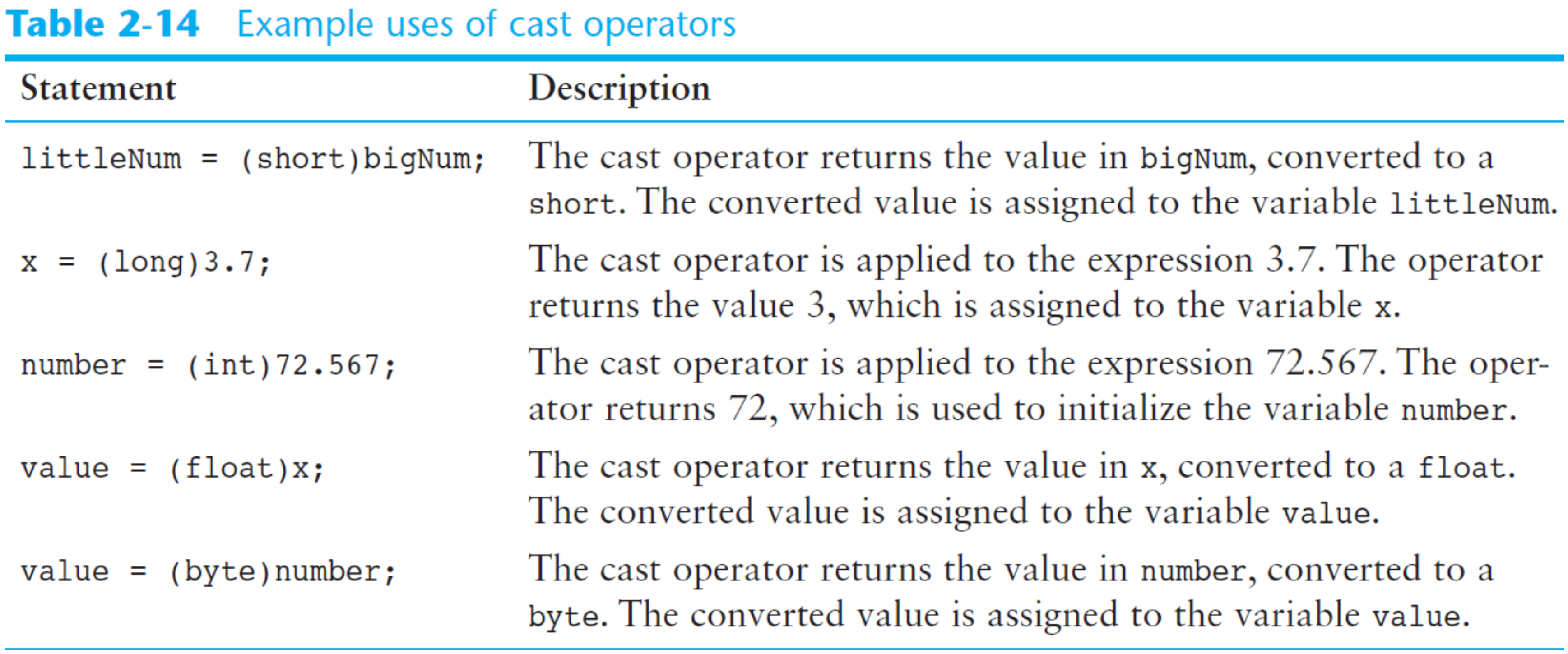


Table 2-14 Example uses of cast operators

Statement	Description
`littleNum = (short)bigNum;`	The cast operator returns the value in `bigNum`, converted to a `short`. The converted value is assigned to the variable `littleNum`.
`x = (long)3.7;`	The cast operator is applied to the expression 3.7. The operator returns the value 3, which is assigned to the variable `x`.
`number = (int)72.567;`	The cast operator is applied to the expression 72.567. The operator returns 72, which is used to initialize the variable `number`.
`value = (float)x;`	The cast operator returns the value in `x`, converted to a `float`. The converted value is assigned to the variable `value`.
`value = (byte)number;`	The cast operator returns the value in `number`, converted to a `byte`. The converted value is assigned to the variable `value`.

Conversion between Primitive Data Types (cont'd.)

- **Mixed Integer Operations**
  - When values of the `byte` or `short` data types are used in arithmetic expressions, they are temporarily converted to `int` values.
  - The result of an arithmetic operation using only a mixture of `byte`, `short`, or `int` values will always be an `int`.

Addison-Wesley
is an imprint of
PEARSON

Conversion between Primitive Data Types (cont’d.)

- **Mixed Integer Operations**
  - For example:

```
short a;
short b = 3;
short c = 7;
a = b + c;
```

This statement will cause an error because the result of `b + c` is an `int`. It cannot be assigned to a `short` variable.

```
a = (short)(b + c);
```

To fix the statement, rewrite the expression using a cast operator.

Conversion between Primitive Data Types (cont'd.)

- **Other Mixed Mathematical Expressions**
  - If one of an operator's operands is a `double`, the value of the other operand will be converted to a `double`.
  - The result of the expression will be a `double`.

  - If one of an operator's operands is a `float`, the value of the other operand will be converted to a `float`.
  - The result of the expression will be a `float`.

  - If one of an operator's operands is a `long`, the value of the other operand will be converted to a `long`.
  - The result of the expression will be a `long`.

Creating Named Constants with `final`

- Many programs have data that does not need to be changed.
- Littering programs with literal values can make the program hard do read and maintain.
- Replacing literal values with constants remedies this problem.
- Constants allow the programmer to use a name rather than a value throughout the program.
- Constants also give a singular point for changing those values when needed.

Creating Named Constants with `final` (cont’d.)

- Constants keep the program organized and easier to maintain.
- Constants are identifiers that can hold only a single value.
- Constants are declared using the keyword `final`.
- Constants need not be initialized when declared; however, they must be initialized before they are used or a compiler error will be generated.

Addison-Wesley
is an imprint of
PEARSON

Creating Named Constants with `final` (cont'd.)

- **Once initialized with a value, constants cannot be changed programmatically.**
- **By convention, constants are all upper case and words are separated by the underscore character.**
- **For example:**

```
final double CAL_SALES_TAX = 0.0725;
```

Addison-Wesley
is an imprint of
PEARSON

The `String` Class

- Java has no primitive data type that holds a series of characters.
- The `String` class from the Java standard library is used for this purpose.
- In order to be useful, the a variable must be created to reference a `String` object.

```
String number;
```

- Notice the `S` in `String` is upper case.
- By convention, class names should always begin with an upper case character.

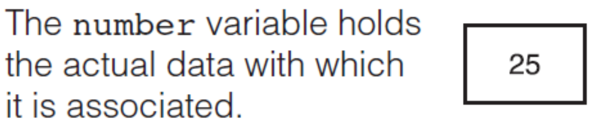
Primitive-Type Variables and Class-Type Variables

- **Primitive variables actually contain the value that they have been assigned.**

```
number = 25;
```

- **The value 25 will be stored in the memory location associated with the variable `number`.**

Figure 2-7 A primitive-type variable holds the data with which it is associated



The `number` variable holds the actual data with which it is associated.

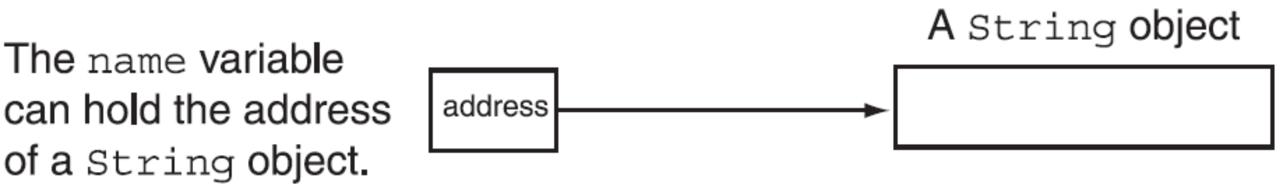
- **Objects are not stored in variables, however. Objects are *referenced* by variables.**

Primitive-Type Variables and Class-Type Variables (cont’d.)

- When a variable references an object, it contains the memory address of the object’s location.
- Then it is said that the variable *references* the object.

```
String name = "Joe Mahoney";
```

Figure 2-8 A `String` class variable can hold the address of a `String` object



Creating a `String` Object

- **A variable can be assigned a string literal.**

```
String value = "Hello";
```

- **`String` objects are the only objects that can be created in this way.**
- **A variable can be created using the *`new`* keyword.**

```
String value = new String("Hello");
```

- **This is the method that all other objects must use when they are created.**

Creating a `String` Object (cont’d.)

- Since `String` is a class, objects that are instances of it have methods.
- One of those methods is the `length` method.

```
stringSize = value.length();
```

- This statement calls the `length` method on the object pointed to by the `value` variable

Addison-Wesley
is an imprint of
PEARSON

Creating a `String` Object (cont’d.)

- **The `String` class contains many methods that help with the manipulation of `String` objects.**
- **`String` objects are *immutable*, meaning that they cannot be changed.**
- **Many of the methods of a `String` object can create new versions of the object.**

Addison-Wesley
is an imprint of
PEARSON

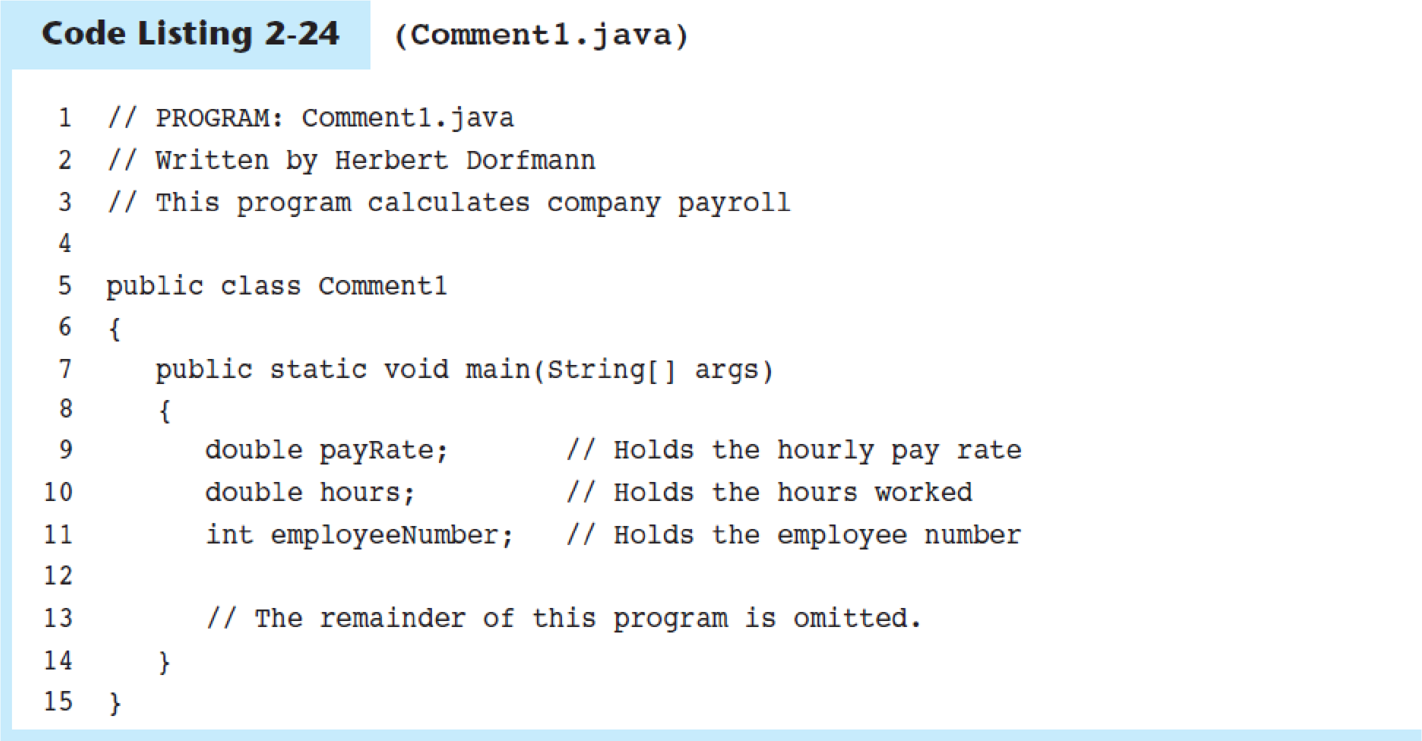
Scope

- *Scope* refers to the part of a program that has access to a variable's contents.
- Variables declared inside a method (like the `main` method) are called *local variables*.
- The scope of a local variable begins at the declaration of the variable and ends at the end of the method in which it was declared.

Comments

- **Comments are:**
  - notes of explanation that document lines or sections of a program.
  - part of the program, but the compiler ignores them.
  - intended for people who may be reading the source code.
- **In Java, there are three types of comments:**
  - Single-line comments
  - Multiline comments
  - Documentation comments

Single-Line Comments

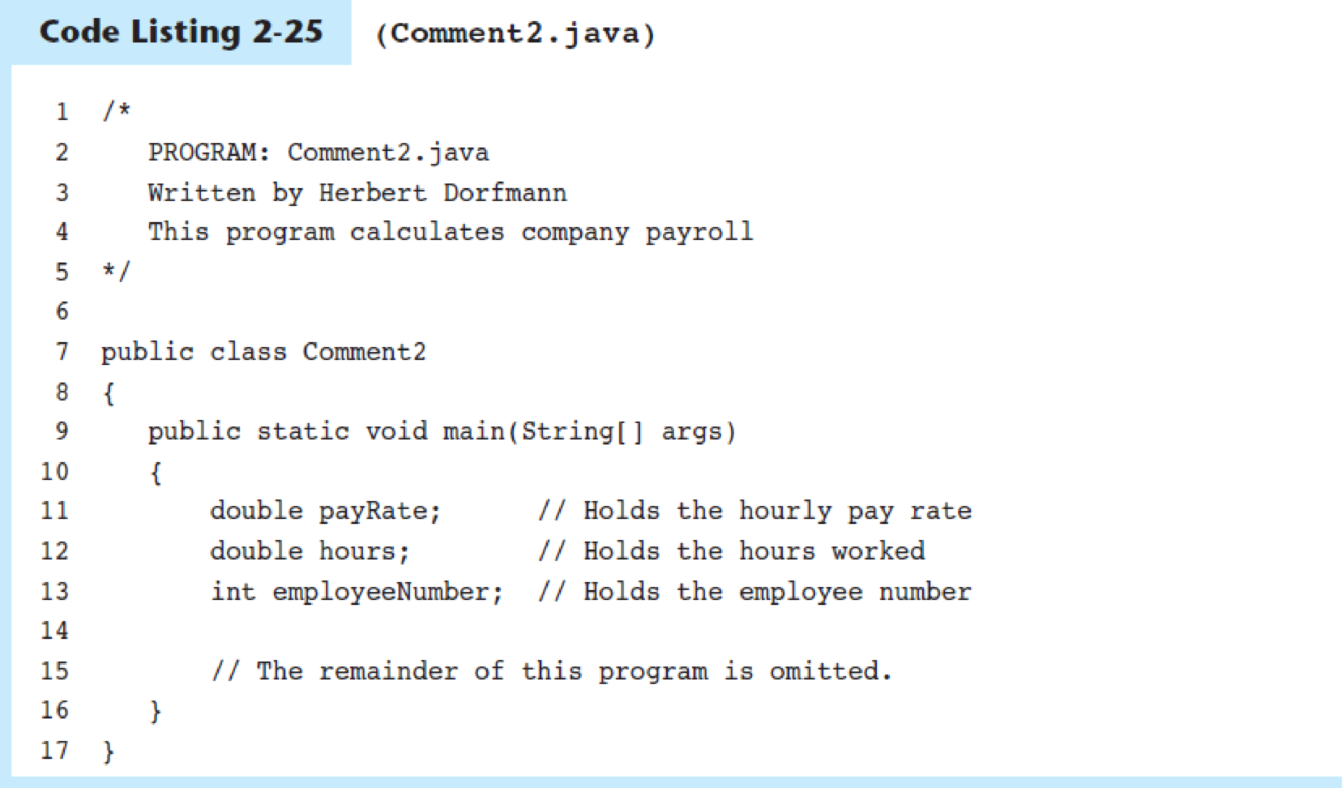


Code Listing 2-24 (Comment1.java)

```
// PROGRAM: Comment1.java
// Written by Herbert Dorfmann
// This program calculates company payroll

public class Comment1
{
   public static void main(String[] args)
   {
      double payRate;        // Holds the hourly pay rate
      double hours;          // Holds the hours worked
      int employeeNumber;    // Holds the employee number

      // The remainder of this program is omitted.
   }
}
```

- **Place two forward slashes (//) where you want the comment to begin.**
  - The compiler ignores everything from that point to the end of the line.

Multiline Comments



Code Listing 2-25 (Comment2.java)

```
/*
   PROGRAM: Comment2.java
   Written by Herbert Dorfmann
   This program calculates company payroll
*/

public class Comment2
{
   public static void main(String[] args)
   {
      double payRate;      // Holds the hourly pay rate
      double hours;        // Holds the hours worked
      int employeeNumber;  // Holds the employee number

      // The remainder of this program is omitted.
   }
}
```

- **Start with `/*` (a forward slash followed by an asterisk) and end with `*/` (an asterisk followed by a forward slash).**
  - **Everything between these markers is ignored.**
  - **Can span multiple lines**

Block Comments

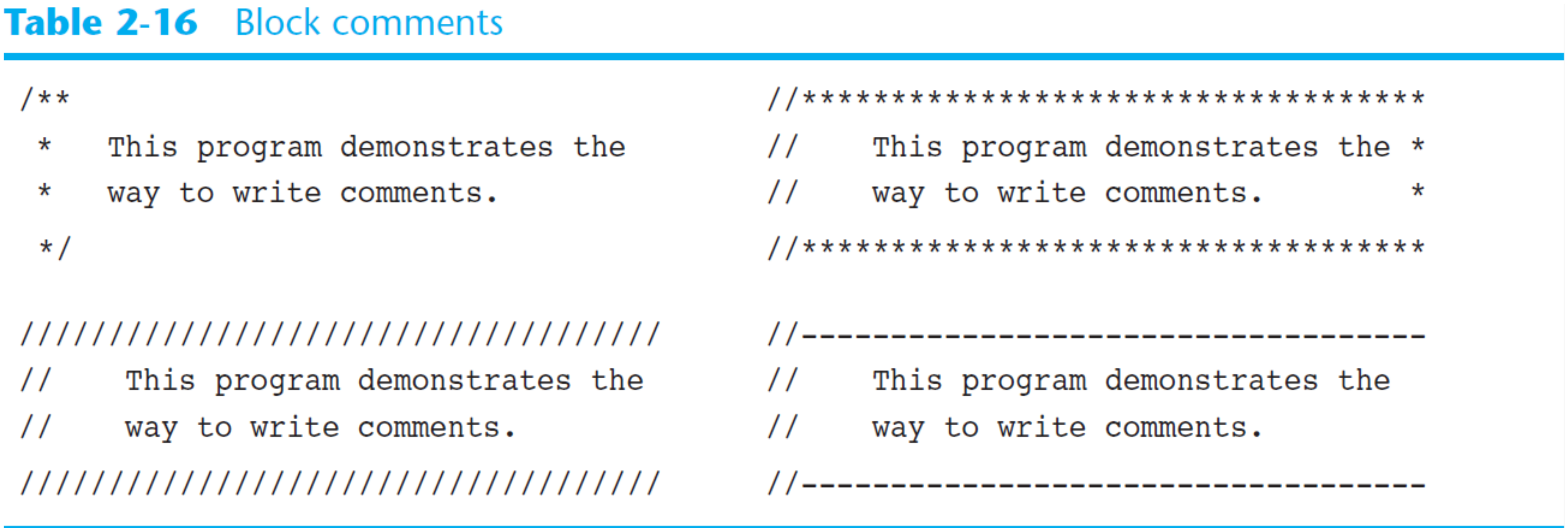


Table 2-16 Block comments

```
/**
 *   This program demonstrates the
 *   way to write comments.
 */
```

```
//************************************
//     This program demonstrates the *
//     way to write comments.        *
//************************************
```

```
////////////////////////////////////
//     This program demonstrates the
//     way to write comments.
////////////////////////////////////
```

```
//-----------------------------------
//     This program demonstrates the
//     way to write comments.
//-----------------------------------
```

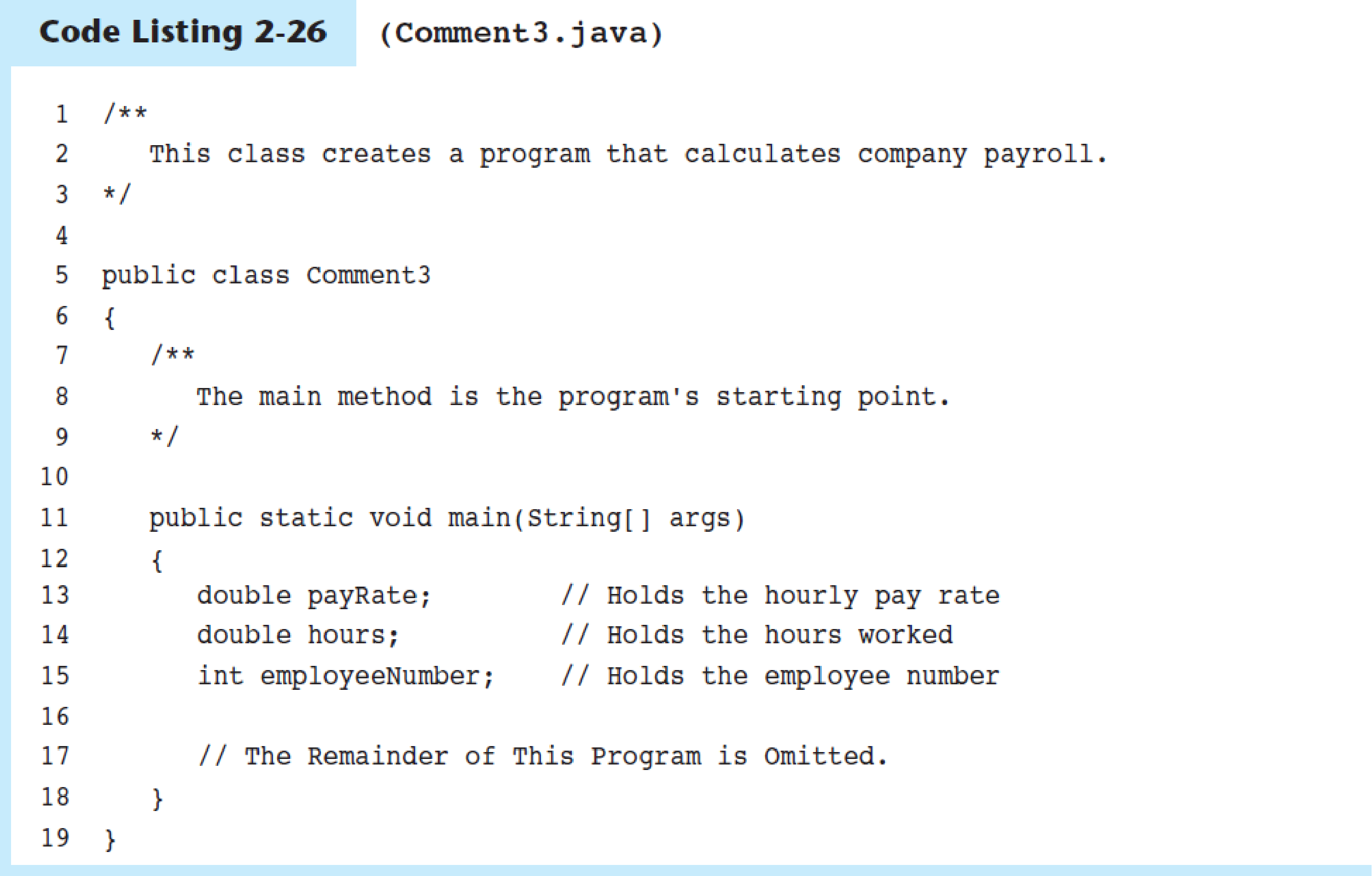
- Many programmers use asterisks or other characters to draw borders or boxes around their comments.
- This helps to visually separate the comments from surrounding code.

Documentation Comments

- Any comment that starts with `/**` and ends with `*/` is considered a documentation comment.
- You write a documentation comment just before:
  - a class header, giving a brief description of the class.
  - each method header, giving a brief description of the method.
- *Documentation comments* can be read and processed by a program named javadoc, which comes with the Sun JDK.

Addison-Wesley
is an imprint of
PEARSON

Documentation Comments (cont’d.)



Code Listing 2-26 (Comment3.java)

```
/**
   This class creates a program that calculates company payroll.
*/

public class Comment3
{
   /**
      The main method is the program's starting point.
   */

   public static void main(String[] args)
   {
      double payRate;        // Holds the hourly pay rate
      double hours;          // Holds the hours worked
      int employeeNumber;    // Holds the employee number

      // The Remainder of This Program is Omitted.
   }
}
```

Documentation Comments (cont'd.)

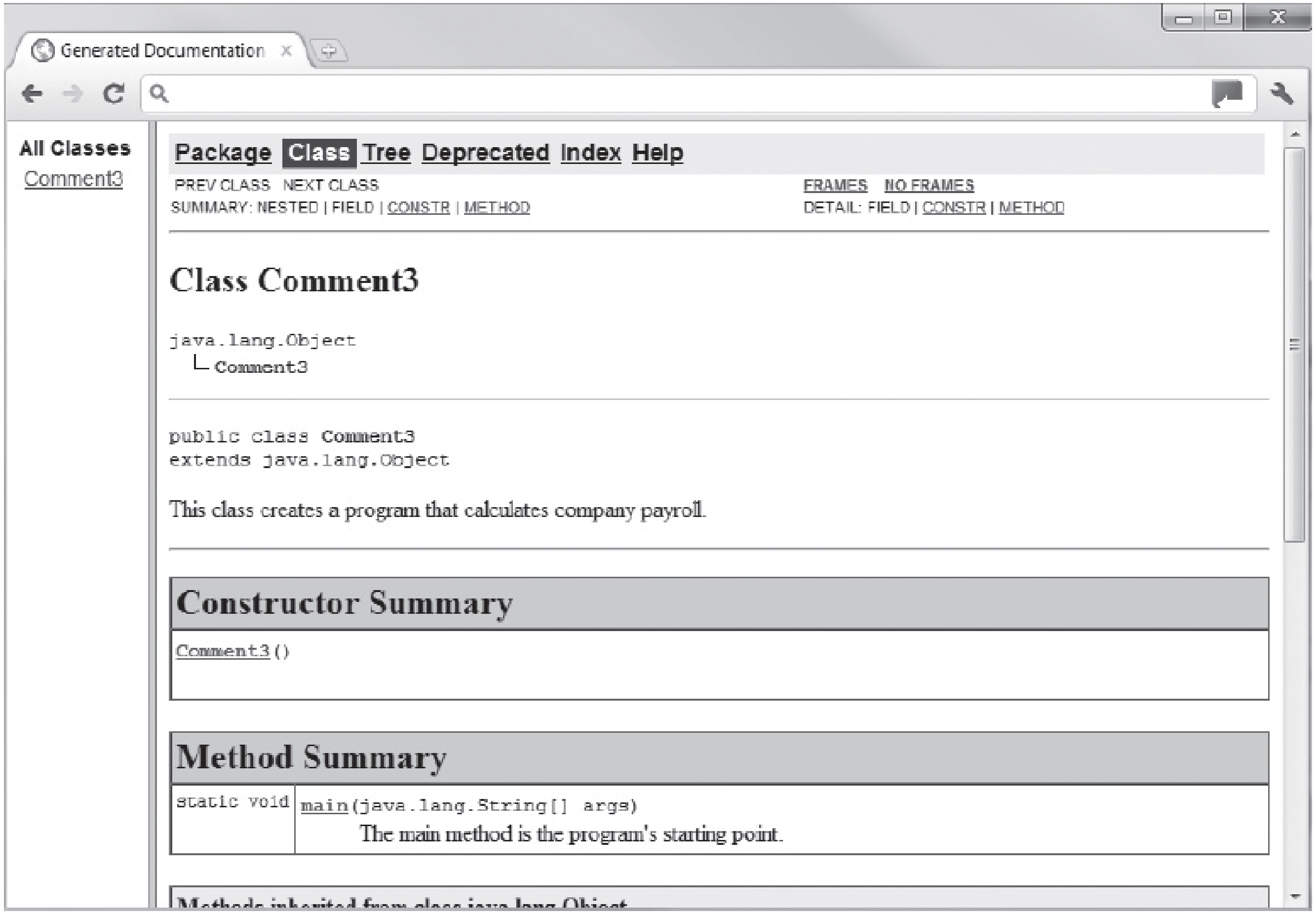
- The purpose of the javadoc program is to read Java source code files and generate attractively formatted HTML files that document the source code.
- To create the documentation, run the `javadoc` program with the source file as an argument.
  - For example:

```
javadoc Comment3.java
```

- The `javadoc` program will create `index.html` and several other documentation files in the same directory as the input file

Documentation Comments (cont’d.)

Figure 2-10 Documentation generated by `javadoc`


Generated Documentation
All Classes
Comment3
Package Class Tree Deprecated Index Help
PREV CLASS NEXT CLASS
FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD
DETAIL: FIELD | CONSTR | METHOD

Class Comment3

```
java.lang.Object
  └ Comment3
```

```
public class Comment3
extends java.lang.Object
```

This class creates a program that calculates company payroll.

Constructor Summary
`Comment3()`

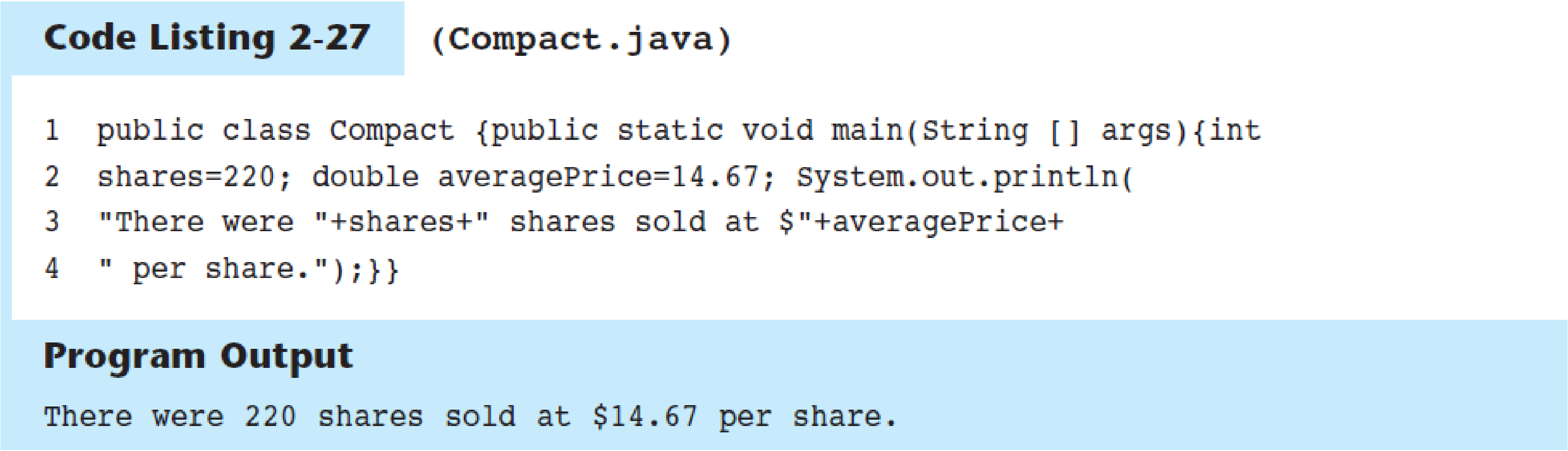
Method Summary	
`static void`	`main(java.lang.String[] args)` The main method is the program's starting point.

Methods inherited from class java.lang.Object

Programming Style

- **Programming style refers to the way a programmer visually arranges a program’s source code.**
- When the compiler reads a program it:
  - Processes it as one long stream of characters.
  - Doesn’t care that each statement is on a separate line, or that spaces separate operators from operands.
  - Humans, on the other hand, find it difficult to read programs that aren’t written in a visually pleasing manner.

Programming Style (cont’d.)



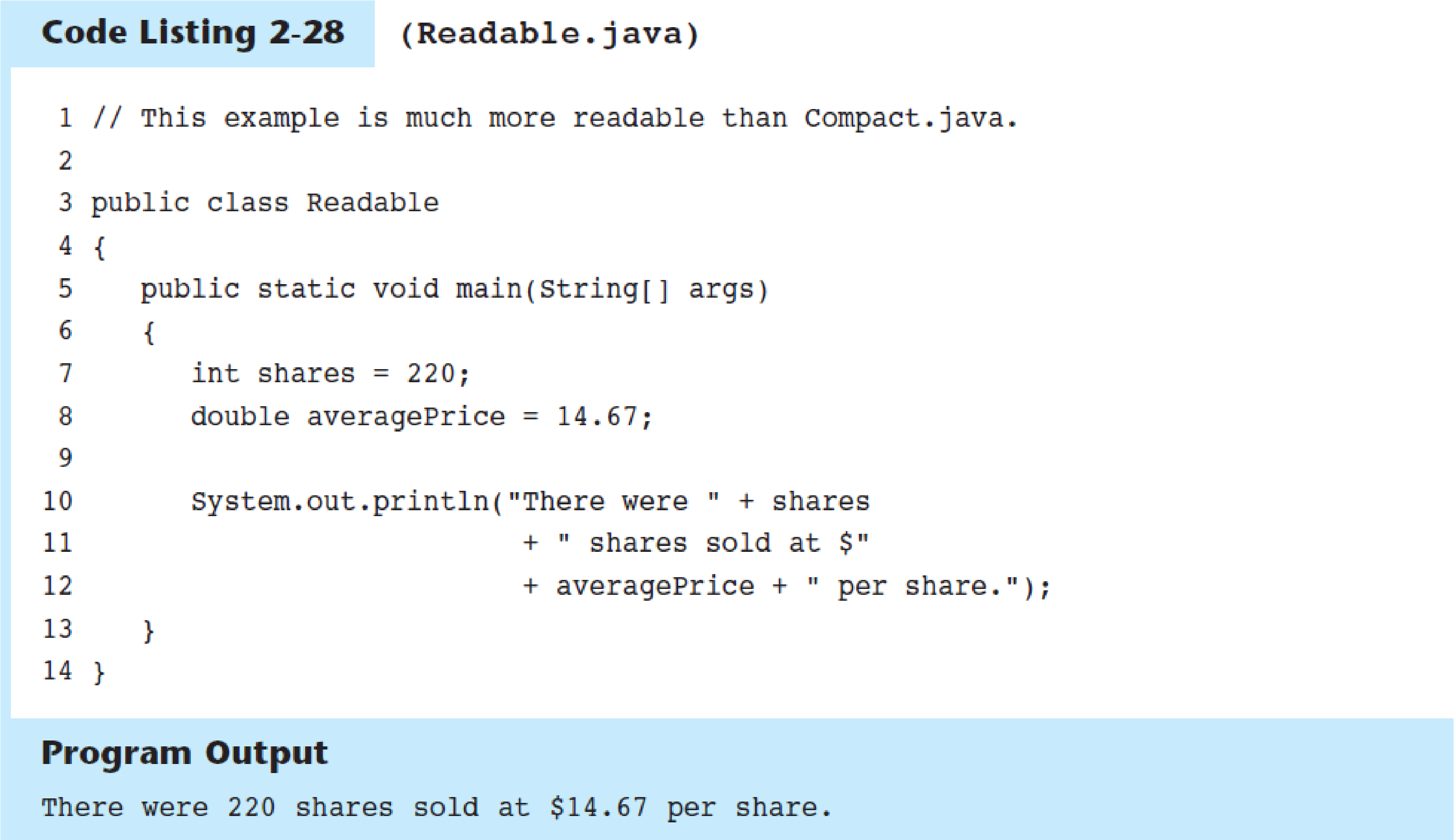
Code Listing 2-27 (Compact.java)

```java
public class Compact {public static void main(String [] args){int
shares=220; double averagePrice=14.67; System.out.println(
"There were "+shares+" shares sold at $"+averagePrice+
" per share.");}}
```

Program Output

```
There were 220 shares sold at $14.67 per share.
```

Programming Style (cont'd.)



Code Listing 2-28 (Readable.java)

```java
// This example is much more readable than Compact.java.

public class Readable
{
   public static void main(String[] args)
   {
      int shares = 220;
      double averagePrice = 14.67;

      System.out.println("There were " + shares
                           + " shares sold at $"
                           + averagePrice + " per share.");
   }
}
```

Program Output

```
There were 220 shares sold at $14.67 per share.
```

Reading Keyboard Input

- To read input from the keyboard we can use the `Scanner` class.
- The `Scanner` class is defined in `java.util`, so we will use the following statement at the top of our programs:

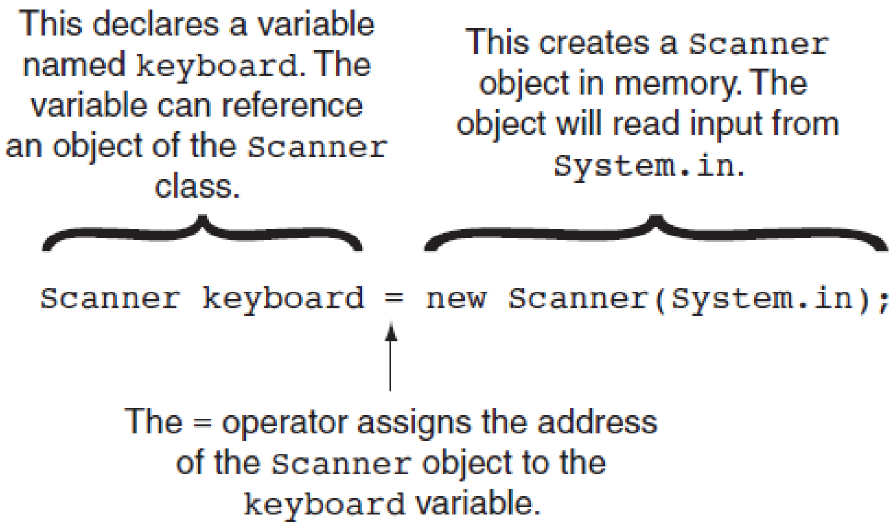
```
import java.util.Scanner;
```

Reading Keyboard Input (cont'd.)

- Scanner objects work with `System.in`
- To create a `Scanner` object and connect it to the `System.in` object:

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

Figure 2-12 The parts of the statement



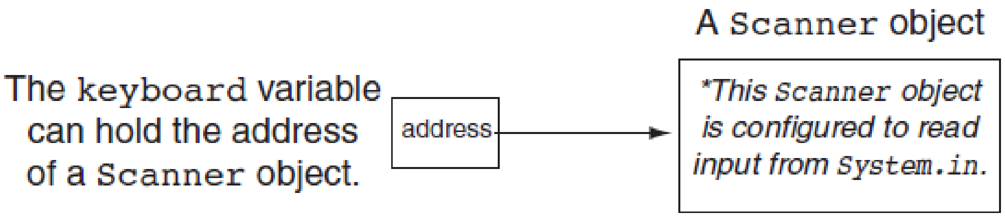
This declares a variable named keyboard. The variable can reference an object of the Scanner class.

This creates a Scanner object in memory. The object will read input from System.in.

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

The = operator assigns the address of the Scanner object to the keyboard variable.

Figure 2-13 The keyboard variable references a Scanner object



A Scanner object
The keyboard variable can hold the address of a Scanner object.
address
*This Scanner object is configured to read input from System.in.

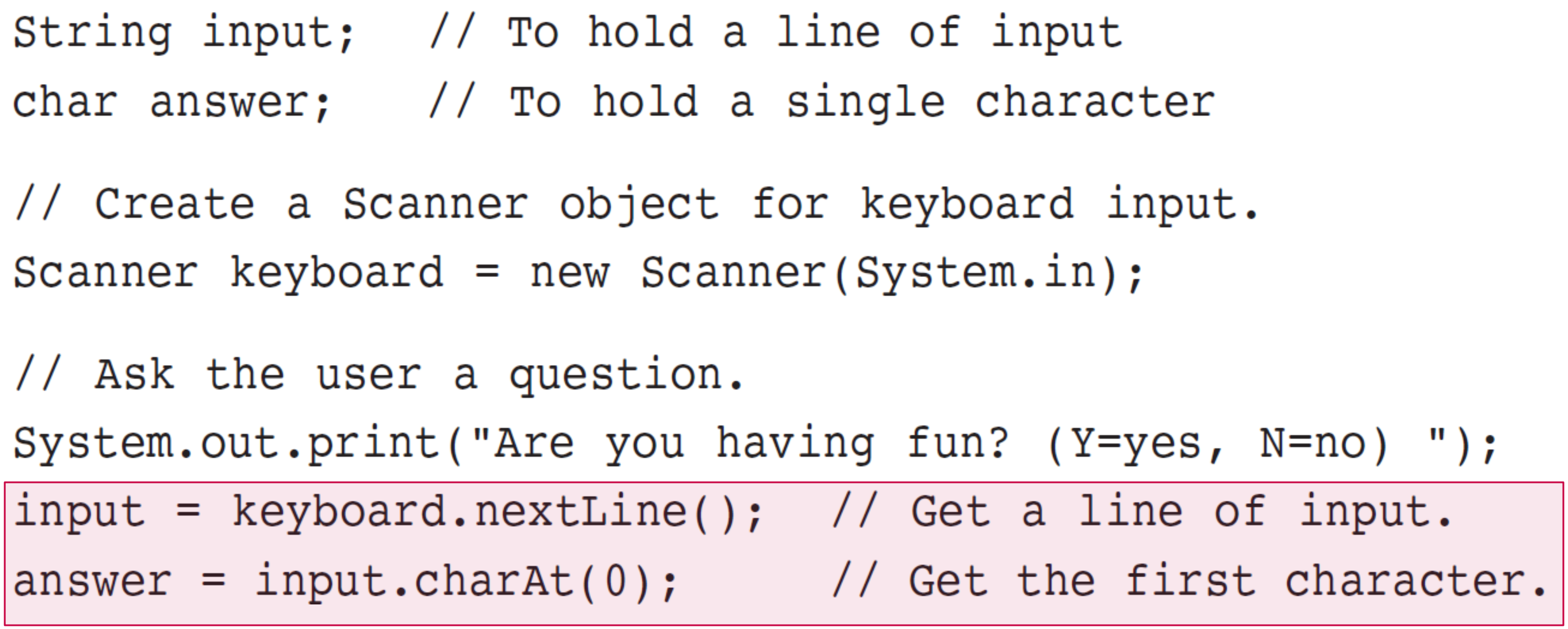
Reading Keyboard Input (cont'd.)

- The Scanner class has methods for reading:
  - strings using the `nextLine` method
  - `byte`s using the `nextByte` method
  - integers using the `nextInt` method
  - long integers using the `nextLong` method
  - short integers using the `nextShort` method
  - `float`s using the `nextFloat` method
  - `double`s using the `nextDouble` method

Reading a Character

- The `Scanner` class does not have a method for reading a single character.
  - Use the `Scanner` class’s `nextLine` method to read a string from the keyboard.
  - Then use the `String` class’s `charAt` method to extract the first character of the string.

Reading a Character (cont'd.)



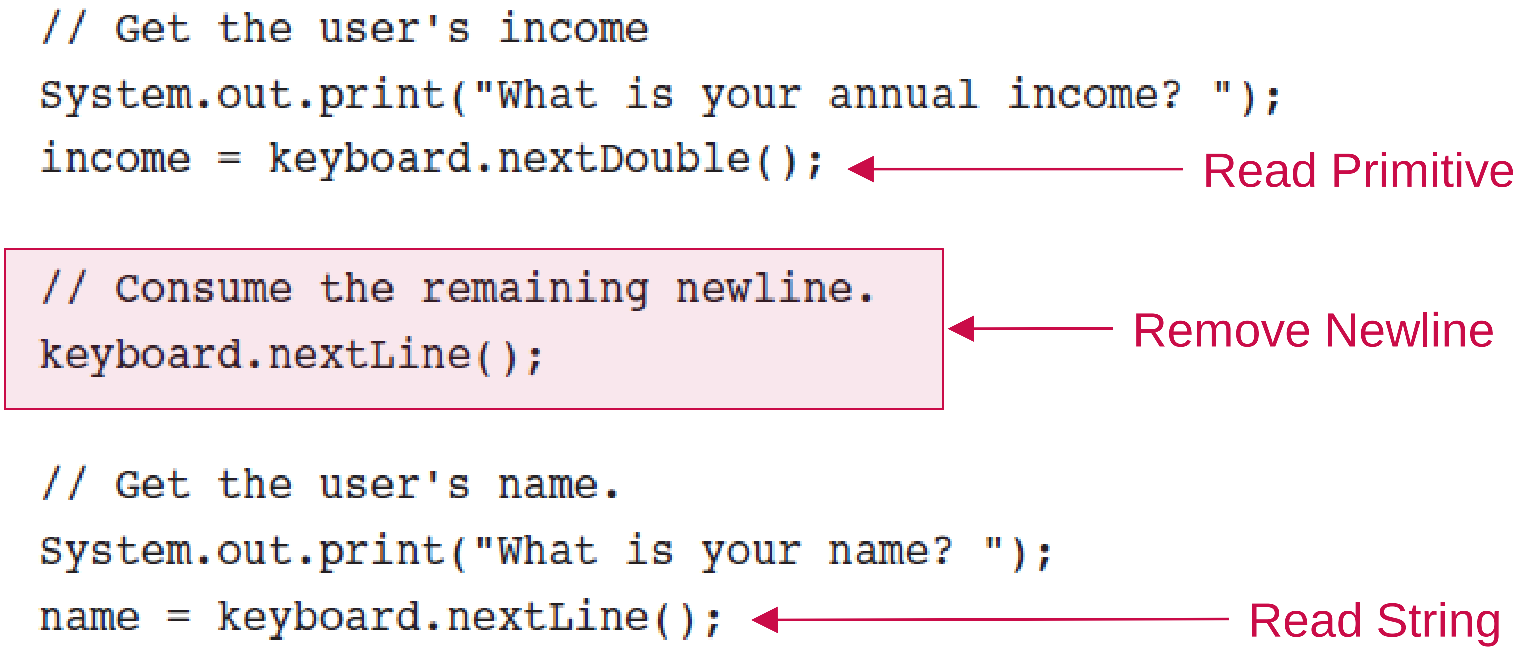
```
String input;   // To hold a line of input
char answer;    // To hold a single character

// Create a Scanner object for keyboard input.
Scanner keyboard = new Scanner(System.in);

// Ask the user a question.
System.out.print("Are you having fun? (Y=yes, N=no) ");
input = keyboard.nextLine();  // Get a line of input.
answer = input.charAt(0);     // Get the first character.
```

Mixing Calls to `nextLine` with Calls to Other `Scanner` Methods

- Keystrokes are stored in an area of memory that is sometimes called the *keyboard buffer*.
- Pressing the Enter key causes a newline character to be stored in the keyboard buffer.
- The `Scanner` methods that are designed to read primitive values, such as `nextInt` and `nextDouble`, will ignore the newline and return only the numeric value.
- The `Scanner` class's `nextLine` method will read the newline that is left over in the keyboard buffer, return it, and terminate without reading the intended input.

Mixing Calls to `nextLine` with Calls to Other `Scanner` Methods (cont'd.)

- **Remove the newline from the keyboard buffer by calling the `Scanner` class's `nextLine` method, ignoring the return value.**



```
// Get the user's income
System.out.print("What is your annual income? ");
income = keyboard.nextDouble();

// Consume the remaining newline.
keyboard.nextLine();

// Get the user's name.
System.out.print("What is your name? ");
name = keyboard.nextLine();
```

Read Primitive

Remove Newline

Read String

Dialog Boxes

- **A *dialog box* is a small graphical window that displays a message to the user or requests input.**
- **A variety of dialog boxes can be displayed using the `JOptionPane` class.**
- **Two of the dialog boxes are:**
  - Message Dialog - a dialog box that displays a message.
  - Input Dialog - a dialog box that prompts the user for input.

Dialog Boxes (cont'd.)

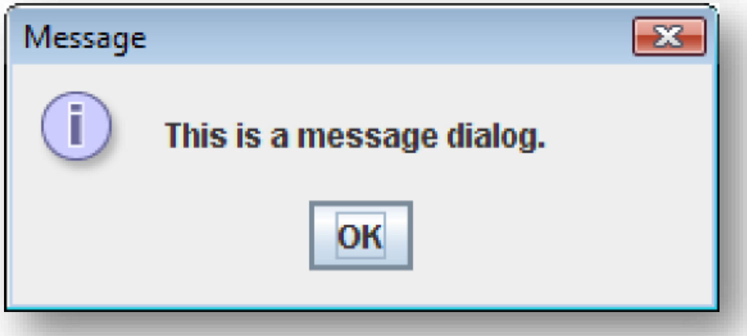
- The `JOptionPane` class is not automatically available to your Java programs.
- The following statement must appear before the program's class header:

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

- This statement tells the compiler where to find the `JOptionPane` class.

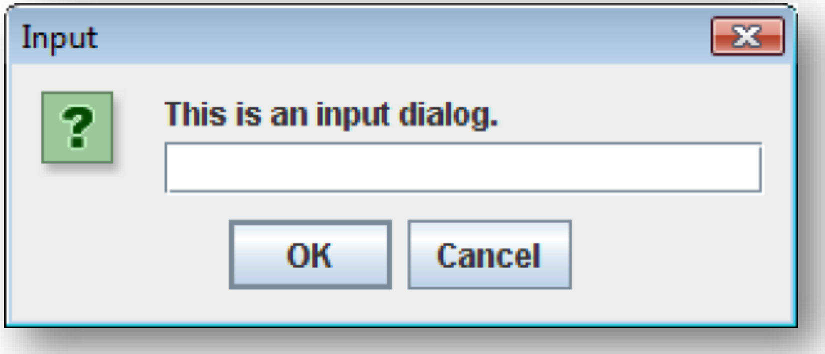
Dialog Boxes (cont’d.)

The `JOptionPane` class provides methods to display each type of dialog box.



Message

This is a message dialog.



Input

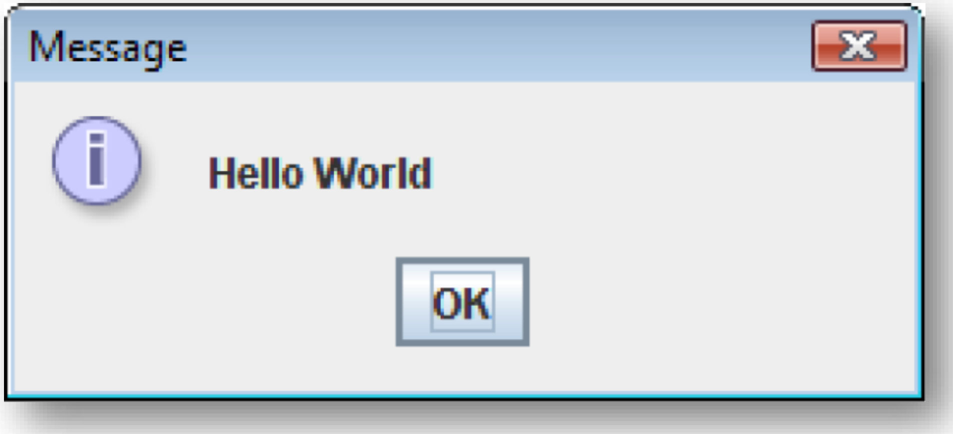
This is an input dialog.

Displaying Message Dialogs

- **`JOptionPane.showMessageDialog` method is used to display a message dialog.**

```
JOptionPane.showMessageDialog(null, "Hello World");
```

  - The first argument will be discussed in Chapter 7.
  - The second argument is the message that is to be displayed.



Message

Hello World

Addison-Wesley
is an imprint of
PEARSON

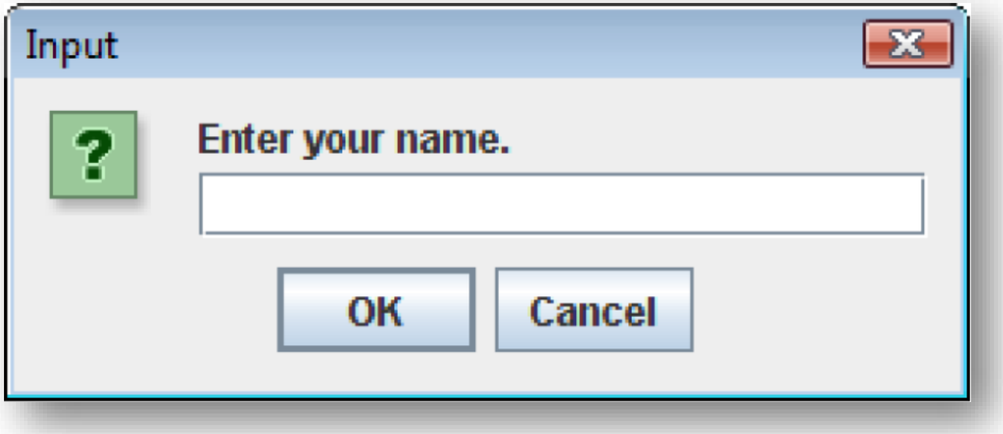
Displaying Input Dialogs

- **An input dialog is a quick and simple way to ask the user to enter data.**
- **The dialog displays a text field, an OK button and a Cancel button.**
- **If OK is pressed, the dialog returns the user’s input.**
- **If Cancel is pressed, the dialog returns `null`.**

Displaying Input Dialogs (cont'd.)

```
String name;
name = JOptionPane.showInputDialog("Enter your name.");
```

- The argument passed to the method is the message to display.
- If the user clicks on the OK button, `name` references the string entered by the user.
- If the user clicks on the Cancel button, `name` references `null`.



Enter your name.

Addison-Wesley
is an imprint of

PEARSON

Dialog Boxes (cont'd.)

- A program that uses `JOptionPane` does not automatically stop executing when the end of the `main` method is reached.
- Java generates a *thread*, which is a process running in the computer, when a `JOptionPane` is created.
- If the `System.exit` method is not called, this thread continues to execute.

Dialog Boxes (cont'd.)

- **The `System.exit` method requires an integer argument.**

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

- **This argument is an *exit code* that is passed back to the operating system.**
- **This code is usually ignored, however, it can be used outside the program:**
  - to indicate whether the program ended successfully or as the result of a failure.
  - The value `0` traditionally indicates that the program ended successfully.

Converting a String to a Number

- The `JOptionPane`’s `showInputDialog` method always returns the user's input as a `String`
- A `String` containing a number, such as `"127.89"`, can be converted to a numeric data type.

Addison-Wesley
is an imprint of
PEARSON

Converting a String to a Number (cont'd.)

- **Each of the numeric wrapper classes, (covered in Chapter 8) has a method that converts a string to a number.**
  - The `Integer` class has a method that converts a string to an `int`.
  - The `Double` class has a method that converts a string to a `double`.
  - etc.
- **These methods are known as *parse methods* because their names begin with the word "parse."**

Addison-Wesley
is an imprint of
PEARSON

Converting a String to a Number (cont'd.)

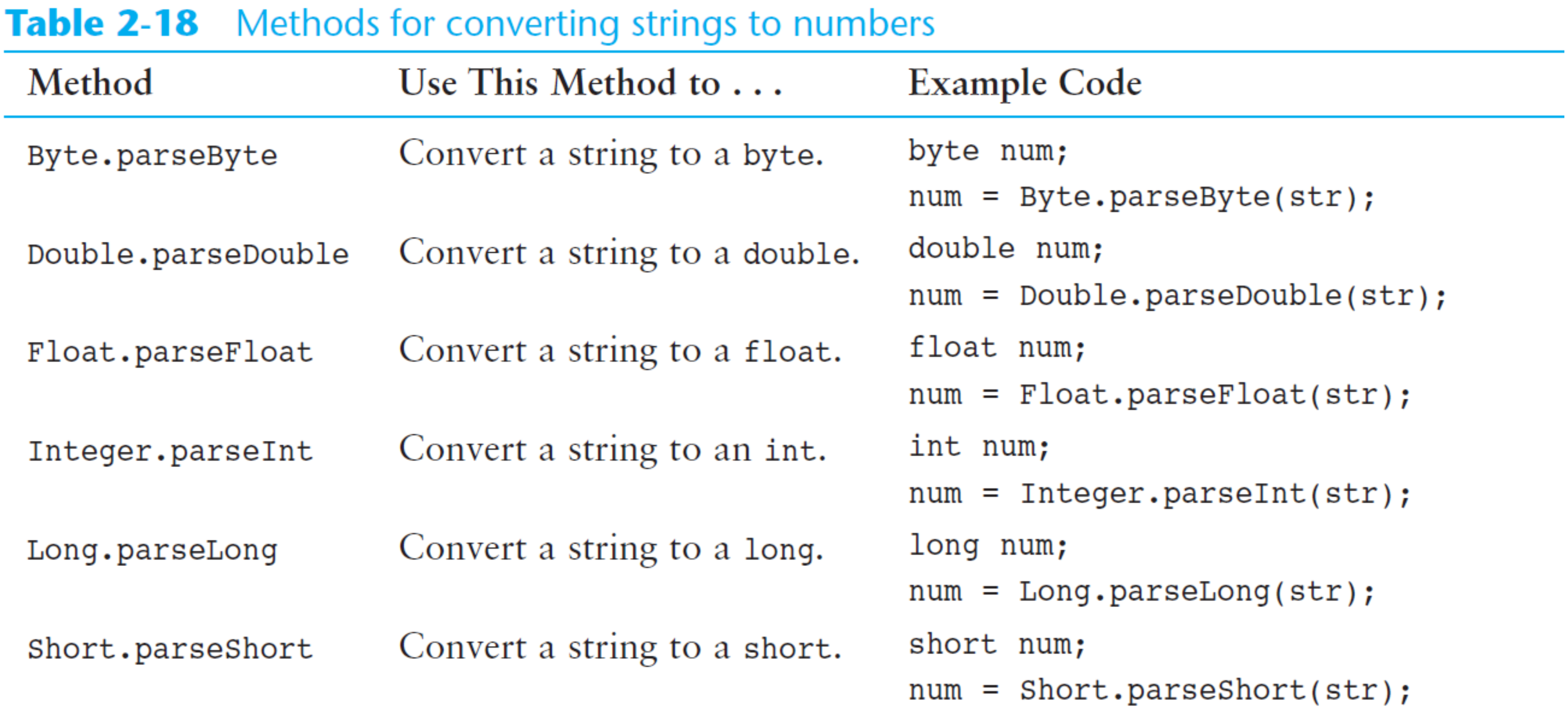


Table 2-18 Methods for converting strings to numbers

Method	Use This Method to . . .	Example Code

`Byte.parseByte`	Convert a string to a byte.	`byte num;`
		`num = Byte.parseByte(str);`

`Double.parseDouble`	Convert a string to a double.	`double num;`
		`num = Double.parseDouble(str);`

`Float.parseFloat`	Convert a string to a float.	`float num;`
		`num = Float.parseFloat(str);`

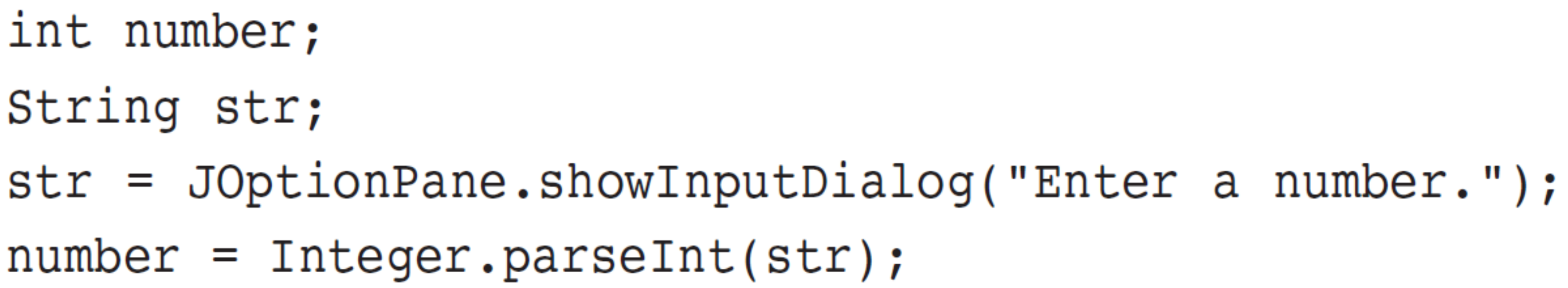
`Integer.parseInt`	Convert a string to an int.	`int num;`
		`num = Integer.parseInt(str);`

`Long.parseLong`	Convert a string to a long.	`long num;`
		`num = Long.parseLong(str);`

`Short.parseShort`	Convert a string to a short.	`short num;`
		`num = Short.parseShort(str);`

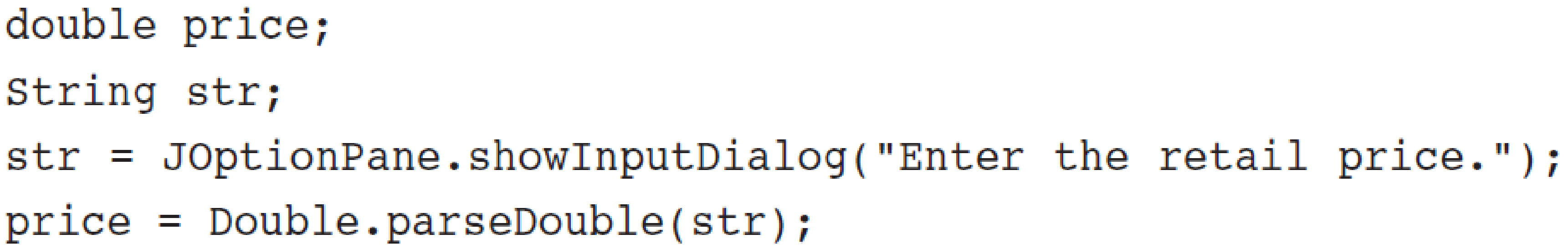
Converting a String to a Number (cont'd.)

- Example conversion from string to `int`:



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

- Example conversion from string to `double`:



```
double price;
String str;
str = JOptionPane.showInputDialog("Enter the retail price.");
price = Double.parseDouble(str);
```

The `System.out.printf` Method

- **You can perform formatted console output with the `System.out.printf` method.**
- **The method's general format is:**

```
System.out.printf(FormatString, ArgumentList)
```

  - *`FormatString`* is a string that contains text and/or special formatting specifiers
  - *`ArgumentList`* is a list of zero or more additional arguments, formatted according to the format specifiers listed in the *`FormatString`*.

Addison-Wesley
is an imprint of
PEARSON

Simple Output

- The simplest way you can use the `printf` method is with only a format string and no additional arguments.

```
System.out.printf("I love Java programming.\n");
```

- This method call simply prints the string

```
    I love Java programming.
```

- Using the method without any format specifiers is like using the `System.out.print` method.

Single Format Specifier and Argument

- Let’s look at an example that uses a format specifier and an additional argument:

```
int hours = 40;
System.out.printf("I worked %d hours this week.\n",hours);
```

- When this string is printed, the value of the `hours` argument will be printed in place of the `%d` format specifier.

```
I worked 40 hours this week.
```

- The `%d` format specifier was used because the `hours` variable is an `int`.
- An error will occur if you use `%d` with a non-integer value.

Addison-Wesley
is an imprint of
PEARSON

Multiple Format Specifiers and Arguments

- **Here's another example:**

```
int dogs = 2;
int cats = 4;
System.out.printf("We have %d dogs and %d cats.\n",dogs, cats);
```

- **First, notice that this example uses two `%d` format specifiers in the format string.**
- **Also notice that two arguments appear after the format string.**
  - The value of the first integer argument, **`dogs`**, is printed in place of the first `%d`.
  - The value of the second integer argument, **`cats`**, is printed in place of the second `%d`.

```
We have 2 dogs and 4 cats.
```

Addison-Wesley
is an imprint of
PEARSON

Multiple Format Specifiers and Arguments

- The following code shows another example:

```
int value1 = 3;
int value2 = 6;
int value3 = 9;
System.out.printf("%d %d %d\n", value1, value2, value3);
```

- In the `printf` method call, there are three format specifiers and three additional arguments after the format string.
- This code will produce the following output:

```
3 6 9
```

- These examples show the one-to-one correspondence between the format specifiers and the arguments that appear after the format string.

Setting the Field Width

- A format specifier may also include a field width. Here is an example:

```
int number = 9;
System.out.printf("The value is %6d\n", number);
```

- The format specifier `%6d` indicates that the argument number should be printed in a field that is 6 places wide. If the value in number is shorter than 6 places, it will be right justified. Here is the output of the code.

```
The value is      9
            123456
```

- If the value of the argument is wider than the specified field width, the field width will be expanded to accommodate the value.

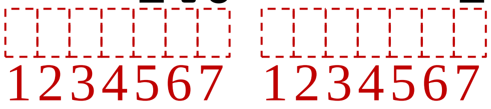
Using Field Widths to Print Columns

- Field widths can help when you need to print values aligned in columns. For example, look at the following code:

```
int num1 = 97654, num2 = 598;
int num3 = 86,    num4 = 56012;
int num5 = 246,   num6 = 2;
System.out.printf("%7d %7d\n", num1, num2);
System.out.printf("%7d %7d\n", num3, num4);
System.out.printf("%7d %7d\n", num5, num6);
```

- This code displays the values of the variables in a table with three rows and two columns. Each column has a width of seven spaces. Here is the output for the code:

```
  97654     598
     86   56012
    246       2
```



Printing Formatted Floating-Point Values

- If you wish to print a floating-point value, use the `%f` format specifier. Here is an example:

```
double number = 1278.92;
System.out.printf("The number is %f\n", number);
```

- This code produces the following output:

```
The number is 1278.920000
```

- You can also use a field width when printing floating-point values. For example the following code prints the value of number in a field that is 18 spaces wide:

```
System.out.printf("The number is %18f\n", number);
```

Addison-Wesley
is an imprint of
PEARSON

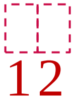
Printing Formatted Floating-Point Values

- In addition to the field width, you can also specify the number of digits that appear after the decimal point. Here is an example:

```
double grossPay = 874.12;
System.out.printf("Your pay is %.2f\n", grossPay);
```

- In this code, the `%.2f` specifier indicates that the value should appear with two digits after the decimal point. The output of the code is:

```
Your pay is 874.12
```


12

Printing Formatted Floating-Point Values

- When you specify the number of digits to appear after the decimal point, the number will be rounded. For example, look at the following code:

```
double number = 1278.92714;
System.out.printf("The number is %.2f\n", number);
```

- This code will produce the following output:

```
The number is 1278.93
```

Addison-Wesley
is an imprint of

PEARSON

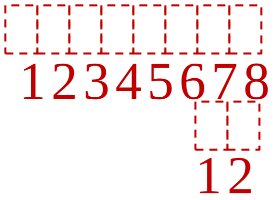
Printing Formatted Floating-Point Values

- You can specify both the field width and the number of decimal places together, as shown here:

```
double grossPay = 874.12;
System.out.printf("Your pay is %8.2f\n", grossPay);
```

- The output of the code is:

```
Your pay is   874.12
```



Printing Formatted Floating-Point Values

- You can also use commas to group digits in a number. To do this, place a comma after the % symbol in the format specifier. Here is an example:

```
double grossPay = 1253874.12;
System.out.printf("Your pay is %,.2f\n", grossPay);
```

- This code will produce the following output:

```
Your pay is 1,253,874.12
```

Addison-Wesley is an imprint of PEARSON

Printing Formatted String Values

- If you wish to print a string argument, use the `%s` format specifier. Here is an example:

```
String name = "Ringo";
System.out.printf("Your name is %s\n", name);
```

- This code produces the following output:

```
Your name is Ringo
```

Addison-Wesley
is an imprint of

PEARSON

Printing Formatted `String` Values

- You can also use a field width when printing strings. For example, look at the following code:

```
String name1 = "George",  name2 = "Franklin";
String name3 = "Jay",     name4 = "Ozzy";
String name5 = "Carmine", name6 = "Dee";
System.out.printf("%10s %10s\n", name1, name2);
System.out.printf("%10s %10s\n", name3, name4);
System.out.printf("%10s %10s\n", name5, name6);
```

- This code displays the values of the variables in a table with three rows and two columns. Each column has a width of ten spaces. Here is the output of the code:

```
    George   Franklin
       Jay       Ozzy
   Carmine        Dee
```