Chapter 2

Elementary Programming

Objectives

- To obtain input from the console using the **Scanner** class
- To use identifiers to name variables, constants, methods, and classes and their naming conventions
- To use variables to store data and constants to store permanent data
- To program with assignment statements and assignment expressions
- To explore Java numeric primitive data types: **byte**, **short**, **int**, **long**, **float**, and **double**
- To perform operations using operators **+**, **-**, *****, **/**, and **%**

Objectives

- To write integer literals, floating-point literals
- To write and evaluate numeric expressions
- To use augmented assignment operators
- To distinguish between postincrement and preincrement and between postdecrement and predecrement
- To cast the value of one type to another type
- To describe the software development process

Algorithm

- Writing a program involves designing algorithms and translating algorithms into programming instructions, or code.
- An *algorithm* describes how a problem is solved by listing the actions that need to be taken and the order of their execution.
- Algorithms can help the programmer plan a program before writing it in a programming language.
- Algorithms can be described in natural languages or in *pseudocode* (natural language mixed with some programming code).

Writing a program

- Let's consider the problem of computing the area of a circle. How do we write a program for solving this problem?
- The algorithm for calculating the area of a circle can be described as follows:
  1. Read in the circle's radius.
  2. Compute the area using the following formula:
     area = *radius* * *radius* * p
  3. Display the result.

Writing a program (cont'd)

- Assume that you have chosen **ComputeArea** as the class name. The outline of the program would look like this:

```
public class ComputeArea {
  // Details to be given later
}
```

- Every Java program must have a **main** method where program execution begins. The program is then expanded as follows:

```
public class ComputeArea {
  public static void main(String[] args) {
    // Step 1: Read in radius

    // Step 2: Compute area

    // Step 3: Display the area
  }
}
```

Signature of the main method

Variables

- To read in the radius and store it, the program needs to declare a symbol called a *variable*.
- A variable represents a value stored in the computer's memory.
- Rather than using **x** and **y** as variable names, choose descriptive names: in this case, **radius** for radius, and **area** for area.
- To let the compiler know what **radius** and **area** are, specify their data types, whether integer, real number, or something else. This is known as *declaring variables*.
- Java provides simple data types for representing integers, real numbers, characters, and Boolean types. These types are known as *primitive data types* or *fundamental types* (*e.g.* **int**, **double**, **byte**, **short**, **long**, **float**, **char**, and **boolean**.)

Variables (cont'd)

- If variables are of the same type, they can be declared together, separated by commas, as follows:

  datatype variable1, variable2, ..., variablen;
- Variables often have initial values. You can declare a variable and initialize it in one step: **int** count = **1**;
- You can also use a shorthand form to declare and initialize variables of the same type together. For example, **int** i = **1**, j = **2**;

Example on declaring variables

- Real numbers (i.e., numbers with a decimal point) are represented as *floating-point numbers*. In Java, you can use the keyword **double** to declare a floating-point variable.
- Declare **radius** and **area** as **double**. The program can be expanded as follows:

```
public class ComputeArea {
  public static void main(String[] args) {
    double radius;
    double area;

    // Step 1: Read in radius

    // Step 2: Compute area

    // Step 3: Display the area
  }
}
```


Reading input from the console

- *Reading input from the console enables the program to accept input from the user.*
- Java uses **System.in** to the standard input device. Console input is not directly supported in Java, but you can use the **Scanner** class to create an object to read input from **System.in**, as follows:

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

- The syntax **new Scanner(System.in)** creates an object of the **Scanner** type. The whole line of statement then assigns its reference to the variable **input**.
- An object may invoke its methods. To invoke a method on an object is to ask the object to perform a task. You can invoke the **nextDouble()** method to read a **double** value as follows:

```
double radius = input.nextDouble();
```

This statement reads a number from the keyboard and assigns the number to **radius**.

- The **Scanner** class is in the **java.util** package. It has to be imported before you can use it.

Example: ComputeAreaWithConsoleInput

LISTING 2.2 ComputeAreaWithConsoleInput.java

```
1  import java.util.Scanner; // Scanner is in the java.util package
2
3  public class ComputeAreaWithConsoleInput {
4    public static void main(String[] args) {
5      // Create a Scanner object
6      Scanner input = new Scanner(System.in);
7
8      // Prompt the user to enter a radius
9      System.out.print("Enter a number for radius: ");
10     double radius = input.nextDouble();
11
12     // Compute area
13     double area = radius * radius * 3.14159;
14
15     // Display results
16     System.out.println("The area for the circle of radius " +
17       radius + " is " + area);
18   }
19 }
```

The **print** method in line 9 is identical to the **println** method except that **println** moves to the beginning of the next line after displaying the string, but **print** does not advance to the next line when completed.

Variable declaration and initialization

The plus sign (**+**) here is called a *string concatenation operator*. It combines two strings into one. If a string is combined with a number, the number is converted into a string and concatenated with the other string. Strings and string concatenation will be discussed further in Chapter 4.

Brief introduction on Java Classes and Objects

- Java is an object-oriented programming language. Everything in Java is associated with classes and objects, along with its attributes and methods.
- A Class is like an object constructor, or a "blueprint" for creating objects (e.g. the Scanner class we have just seen).
- An object is created from a class. To create an object of a class, specify the class name, followed by the object name, and use the keyword new, such as **new** Scanner(System.in) as discussed in the previous slide.
- The object created can then invoke the methods of the classes.
- For example, a car is an object. The car has **attributes**, such as weight and color, and **methods**, such as drive and brake.

Identifiers

- *Identifiers are the names that identify the elements such as classes, methods, and variables in a program.* All identifiers must obey the following rules:
  - An identifier is a sequence of characters that consists of letters, digits, underscores (_), and dollar signs (**$**).
  - An identifier must start with a letter, an underscore (_), or a dollar sign (**$**). It cannot start with a digit.
  - An identifier cannot be a reserved word.
  - An identifier cannot be **true**, **false**, or **null**, which are reserved words for literal values
  - An identifier can be of any length.
- For example, **$2**, **ComputeArea**, **area**, **radius**, and **print** are legal identifiers, whereas **2A** and **d+4** are not because they do not follow the rules.
- The Java compiler detects illegal identifiers and reports syntax errors.
- Since Java is case sensitive, **area**, **Area**, and **AREA** are all different identifiers.

Named constants

- *A named constant is an identifier that represents a permanent value.*
- The value of a variable may change during the execution of a program, but a *named constant,* or simply *constant*, represents permanent data that never changes.
- In our **ComputeArea** program, p is a constant. If you use it frequently, you don't want to keep typing **3.14159**; instead, you can declare a constant for p.

  final datatype CONSTANTNAME = value;
- The word **final** is a Java keyword for declaring a constant.
- What do you think are the advantages of using constants?

Naming conventions

- *Sticking with the Java naming conventions makes your programs easy to read and avoids errors.*
- Choose descriptive names with straightforward meanings.
- Remember that names are case sensitive in Java.
- Use lowercase for variables and methods. If a name consists of several words, concatenate them into one, making the first word lowercase and capitalizing the first letter of each subsequent word—for example, the variables **radius** and **area** and the method **print**.
- Capitalize the first letter of each word in a class name—for example, the class names **ComputeArea** and **System**.
- Capitalize every letter in a constant, and use underscores between words—for example, the constants **PI** and **MAX_VALUE**.

Assignment statements and assignment expressions

- *An assignment statement designates a value for a variable.*
- In Java, the equal sign (**=**) is used as the *assignment operator.* The syntax for assignment statements is as follows: variable = expression;
- You can use a variable in an expression. A variable can also be used in both sides of the **=** operator. For example, x = x + **1**;
- To assign a value to a variable, you must place the variable name to the left of the assignment operator. Thus, the following statement is wrong: **1** = x; // Wrong
- In an assignment statement, the data type of the variable on the left must be compatible with the data type of the value on the right. For example, **int x = 1.0** would be illegal if the data type of **x** is **int**.

Numeric data types and operations

- Java provides eight primitive data types for numeric values, characters, and Boolean values. Table 2.1 lists the six numeric data types, their ranges, and their storage sizes.

TABLE 2.1 Numeric Data Types

Name	*Range*	*Storage Size*
`byte`	-2^7 to $2^7 - 1$ (−128 to 127)	8-bit signed
`short`	-2^{15} to $2^{15} - 1$ (−32768 to 32767)	16-bit signed
`int`	-2^{31} to $2^{31} - 1$ (−2147483648 to 2147483647)	32-bit signed
`long`	-2^{63} to $2^{63} - 1$	64-bit signed
	(i.e., −9223372036854775808 to 9223372036854775807)	
`float`	Negative range: −3.4028235E + 38 to −1.4E − 45	32-bit IEEE 754
	Positive range: 1.4E − 45 to 3.4028235E + 38	
`double`	Negative range: −1.7976931348623157E + 308 to −4.9E − 324	64-bit IEEE 754
	Positive range: 4.9E − 324 to 1.7976931348623157E + 308	

17

Reading Numbers from the keyboard

- You know how to use the **nextDouble()** method in the **Scanner** class to read a double value from the keyboard. You can also use the methods listed in Table 2.2 to read a number of the **byte**, **short**, **int**, **long**, and **float** type.

TABLE 2.2 Methods for `Scanner` Objects

Method	*Description*
`nextByte()`	reads an integer of the `byte` type.
`nextShort()`	reads an integer of the `short` type.
`nextInt()`	reads an integer of the `int` type.
`nextLong()`	reads an integer of the `long` type.
`nextFloat()`	reads a number of the `float` type.
`nextDouble()`	reads a number of the `double` type.

Numeric operators

- The operators for numeric data types include the standard arithmetic operators: addition (**+**), subtraction (**–**), multiplication (*****), division (**/**), and remainder (**%**), as shown in Table 2.3. The *operands* are the values operated by an operator.

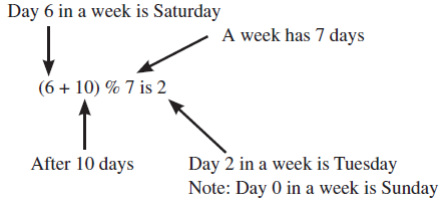
TABLE 2.3 Numeric Operators

Name	*Meaning*	*Example*	*Result*
+	Addition	34 + 1	35
-	Subtraction	34.0 – 0.1	33.9
*	Multiplication	300 * 30	9000
/	Division	1.0 / 2.0	0.5
%	Remainder	20 % 3	2

When both operands of a division are integers, the result of the division is the quotient and the fractional part is truncated. For example, **5 / 2** yields **2**, not **2.5**. To perform a float-point division, one of the operands must be a floating-point number. For example, **5.0 / 2** yields **2.5**.

Some applications on *remainder* operator

- Remainder is very useful in programming. For example, an even number **% 2** is always **0** and an odd number **% 2** is always **1**. Thus, you can use this property to determine whether a number is even or odd.
- If today is Saturday, it will be Saturday again in 7 days. Suppose you and your friends are going to meet in 10 days. What day is in 10 days? You can find that the day is Tuesday using the following expression:



Exponent Operations - Math.pow(a, b)

- The **Math.pow(a, b)** method can be used to compute a^b. The **pow** method is defined in the **Math** class in the Java API.
- Chapter 5 introduces more details on methods. For now, all you need to know is how to invoke the **pow** method to perform the exponent operation.

Numeric literals

- *A* literal *is a constant value that appears directly in a program.*
- For example, **34** and **0.305** are literals in the following statements:

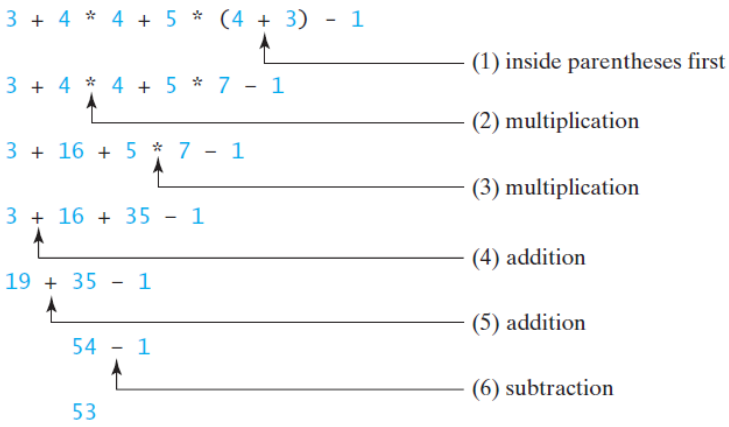
```
int numberOfYears = 34;
double weight = 0.305;
```

Evaluating expressions and operator precedence

- Operators contained within pairs of parentheses are evaluated first.
- Parentheses can be nested, in which case the expression in the inner parentheses is evaluated first.
- When more than one operator is used in an expression, the following operator precedence rule is used to determine the order of evaluation.
  - Multiplication, division, and remainder operators are applied first. If an expression contains several multiplication, division, and remainder operators, they are applied from left to right.
  - Addition and subtraction operators are applied last. If an expression contains several addition and subtraction operators, they are applied from left to right.

Example on operator precedence

Here is an example of how an expression is evaluated:



Augmented Assignment Operators

- *The operators **+**, **-**, *****, **/**, and **%** can be combined with the assignment operator to form augmented operators.*
- Very often the current value of a variable is used, modified, and then reassigned back to the same variable. For example, the following statement increases the variable **count** by **1**: count += **1**;
- The **+=** is called the *addition assignment operator.* Table 2.4 shows other augmented assignment operators.

TABLE 2.4 Augmented Assignment Operators

Operator	*Name*	*Example*	*Equivalent*
+=	Addition assignment	`i += 8`	`i = i + 8`
-=	Subtraction assignment	`i -= 8`	`i = i - 8`
*=	Multiplication assignment	`i *= 8`	`i = i * 8`
/=	Division assignment	`i /= 8`	`i = i / 8`
%=	Remainder assignment	`i %= 8`	`i = i % 8`

Augmented Assignment Operators (cont'd)

- The augmented assignment operator is performed last after all the other operators in the expression are evaluated. For example,
  x /= **4** + **5.5** * **1.5**;

  is same as

  x = x / (**4** + **5.5** * **1.5**);
- Note that there are no spaces in the augmented assignment operators. For example, **+ =** should be **+=**.
- Like the assignment operator (**=**), the operators (**+=**, **-=**, ***=**, **/=**, **%=**) can be used to form an assignment statement as well as an expression. For example, in the following code, **x += 2** is a statement in the first line and an expression in the second line.

```
x += 2; // Statement
System.out.println(x += 2); // Expression
```

Increment and Decrement operators

- *The increment operator (++) and decrement operator (– –) are for incrementing and decrementing a variable by 1.*
- **i++** is pronounced as **i** plus plus and **i——** as **i** minus minus. These operators are known as *postfix increment* (or postincrement) and *postfix decrement* (or postdecrement), because the operators **++** and **——** are placed after the variable.

```
int i = 3, j = 3;
i++; // i becomes 4
j--; // j becomes 2
```

- These operators can also be placed before the variable. These operators are known as *prefix increment* (or preincrement) and *prefix decrement* (or predecrement).

```
int i = 3, j = 3;
++i;  // i becomes 4
--j; // j becomes 2
```


Postfix vs Prefix

- As you see, the effect of **i++** and **++i** or **i——** and **——i** are the same in the preceding examples.
- However, their effects are different when they are used in statements that do more than just increment and decrement. Table 2.5 describes their differences and gives examples.

TABLE 2.5 Increment and Decrement Operators

Operator	*Name*	*Description*	*Example (assume i = 1)*
`++var`	preincrement	Increment `var` by `1`, and use the new `var` value in the statement	`int j = ++i;` `// j is 2, i is 2`
`var++`	postincrement	Increment `var` by `1`, but use the original `var` value in the statement	`int j = i++;` `// j is 1, i is 2`
`--var`	predecrement	Decrement `var` by `1`, and use the new `var` value in the statement	`int j = --i;` `// j is 0, i is 0`
`var--`	postdecrement	Decrement `var` by `1`, and use the original `var` value in the statement	`int j = i--;` `// j is 1, i is 0`

28

Numeric Type Conversions

- *Floating-point numbers can be converted into integers using explicit casting.*
- If an integer and a floating-point number are involved in a binary operation, Java automatically converts the integer to a floating-point value. So, **3 * 4.5** is same as **3.0 * 4.5**.
- You can always assign a value to a numeric variable whose type supports a larger range of values; thus, for instance, you can assign a **long** value to a **float** variable.
- You cannot, however, assign a value to a variable of a type with a smaller range unless you use *type casting.*
- *Casting* is an operation that converts a value of one data type into a value of another data type.
- Casting a type with a small range to a type with a larger range is known as *widening a type.*
- Casting a type with a large range to a type with a smaller range is known as *narrowing a type.*
- Java will automatically widen a type, but you must narrow a type explicitly.

Syntax for casting a type

- The syntax for casting a type is to specify the target type in parentheses, followed by the variable's name or the value to be cast.
- For example, the following statement System.out.println((**int**)**1.7**); displays **1**. When a **double** value is cast into an **int** value, the fractional part is truncated.
- System.out.println((**double**)**1 / 2**); displays **0.5**, because **1** is cast to **1.0** first, then **1.0** is divided by **2**.
- However, the statement System.out.println(**1 / 2**); displays **0**, because **1** and **2** are both integers and the resulting value should also be an integer.

Software development process

- *The software development life cycle is a multistage process that includes requirements specification, analysis, design, implementation, testing, deployment, and maintenance.*

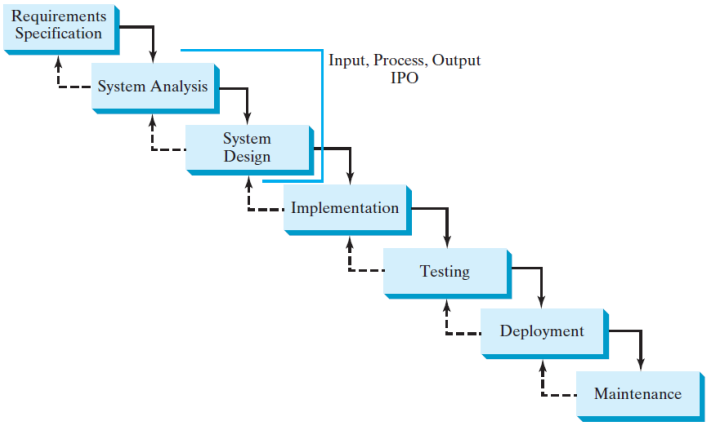


FIGURE 2.3 At any stage of the software development life cycle, it may be necessary to go back to a previous stage to correct errors or deal with other issues that might prevent the software from functioning as expected.

Software development process in action – Stage 1: Requirements Specification

- To see the software development process in action, we will now create a program that computes loan payments.
- The loan can be a car loan, a student loan, or a home mortgage loan.
- For an introductory programming course, we focus on requirements specification, analysis, design, implementation, and testing.

Stage 1: Requirements Specification

The program must satisfy the following requirements:

- It must let the user enter the interest rate, the loan amount, and the number of years for which payments will be made.
- It must compute and display the monthly payment and total payment amounts.

Software development process in action – Stage 2: System Analysis

Stage 2: System Analysis

- The output is the monthly payment and total payment, which can be obtained using the following formulas:

$$monthlyPayment = \frac{loanAmount \times monthlyInterestRate}{1 - \frac{1}{(1 + monthlyInterestRate)^{numberOfYears \times 12}}}$$

$$totalPayment = monthlyPayment \times numberOfYears \times 12$$

- So, the input needed for the program is the monthly interest rate, the length of the loan in years, and the loan amount.
- The requirements specification says that the user must enter the annual interest rate, the loan amount, and the number of years for which payments will be made. During analysis, however, it is possible that you may discover that input is not sufficient or that some values are unnecessary for the output. If this happens, you can go back and modify the requirements specification.

Software development process in action – Stage 3: System Design

Stage 3: System Design (identify the steps in the program)

1. Prompt the user to enter the annual interest rate, the number of years, and the loan amount.
2. The input for the annual interest rate is a number in percent format, such as 4.5%. The program needs to convert it into a decimal by dividing it by **100**. To obtain the monthly interest rate from the annual interest rate, divide it by **12**. So, to obtain the monthly interest rate in decimal format, you need to divide the annual interest rate in percentage by **1200**.
3. Compute the monthly payment using the preceding formula.
4. Compute the total payment, which is the monthly payment multiplied by **12** and multiplied by the number of years.
5. Display the monthly payment and total payment.

Software development process in action – Stage 4: Implementation

Stage 4: Implementation (also known as *coding* (writing the code).)

- Line 10 reads the annual interest rate, which is converted into the monthly interest rate in line 13.
- Choose the most appropriate data type for the variable. For example, **numberOfYears** is best declared as an **int** (line 18), although it could be declared as a **long**, **float**, or **double**.
- Note that **byte** might be the most appropriate for **numberOfYears**. However, the examples in this book will use **int** for integer and **double** for floating-point values.
- Casting is used in lines 31 and 33 to obtain a new **monthlyPayment** and **totalPayment** with two digits after the decimal points.
- The program uses the **Scanner** class, imported in line 1.
- The program also uses the **Math** class, and since all classes in th **java.lang** package are implicitly imported. Therefore, you don't need to explicitly import the **Math** class.

LISTING 2.9 ComputeLoan.java

```
1  import java.util.Scanner;
2
3  public class ComputeLoan {
4    public static void main(String[] args) {
5      // Create a Scanner
6      Scanner input = new Scanner(System.in);
7
8      // Enter annual interest rate in percentage, e.g., 7.25%
9      System.out.print("Enter annual interest rate, e.g., 7.25%: ");
10     double annualInterestRate = input.nextDouble();
11
12     // Obtain monthly interest rate
13     double monthlyInterestRate = annualInterestRate / 1200;
14
15     // Enter number of years
16     System.out.print(
17       "Enter number of years as an integer, e.g., 5: ");
18     int numberOfYears = input.nextInt();
19
20     // Enter loan amount
21     System.out.print("Enter loan amount, e.g., 120000.95: ");
22     double loanAmount = input.nextDouble();
23
24     // Calculate payment
25     double monthlyPayment = loanAmount * monthlyInterestRate / (1
26       - 1 / Math.pow(1 + monthlyInterestRate, numberOfYears * 12));
27     double totalPayment = monthlyPayment * numberOfYears * 12;
28
29     // Display results
30     System.out.println("The monthly payment is $" +
31       (int)(monthlyPayment * 100) / 100.0);
32     System.out.println("The total payment is $" +
33       (int)(totalPayment * 100) / 100.0);
34   }
35 }
```


Software development process in action – Stage 5: Testing

- After the program is implemented, test it with some sample input data and verify whether the output is correct.

Read 2.17 Case Study on Counting Monetary Units

Chapter Summary

- *Identifiers* are names for naming elements such as variables, constants, methods, classes, packages in a program.
- *Variables* are used to store data in a program. To declare a variable is to tell the compiler what type of data a variable can hold.
- A *named constant* (or simply a *constant*) represents permanent data that never changes, declared by using the keyword **final.**
- Java provides the augmented assignment operators **+=** (addition assignment), **–=** (subtraction assignment), ***=** (multiplication assignment), **/=** (division assignment), and **%=** (remainder assignment).
- You can explicitly convert a value from one type to another using the **(type)value** notation.

Chapter Summary

- *Casting* a variable of a type with a small range to a variable of a type with a larger range is known as *widening a type.* Widening a type can be performed automatically without explicit casting.
- Casting a variable of a type with a large range to a variable of a type with a smaller range is known as *narrowing a type.* Narrowing a type must be performed explicitly.

Ideas for further practice

- Write a program that outputs minutes and remaining seconds from an amount of time in seconds.
- (*Convert pounds into kilograms*) Write a program that converts pounds into kilograms. The program prompts the user to enter a number in pounds, converts it to kilograms, and displays the result. One pound is **0.454** kilograms. Here is a sample run:

```
Enter a number in pounds: 55.5 ↵Enter
55.5 pounds is 25.197 kilograms
```