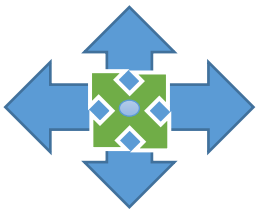
>>>>>>>>>>>>>Elementary Programming<<<<<<<<<<

Brought To You By



{My Life} Programming School

Learn Java By Reading First, Then Watch Someone Tackling Exercises Programmatically By Visiting The Link. https://www.mylifeprogrammingschool.com

2.1 Introduction

{Key Point} The focus of this chapter is on learning elementary programming techniques to solve problems. Through this chapter, you will learn elementary programming using primitive data types, variables, constants, operators, expressions, and input and output.

2.2 Writing a Simple Program

{Key Point} Writing a program involves designing a strategy for solving the problem and then using a programming language to implement that strategy. Let's first consider the simple problem of computing the area of a circle. How do we write a program for solving this problem?
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). 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.

You already know that every Java program begins with a class definition in which the keyword **class** is followed by the class name. 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
}
```

As you know, 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
    }
}
```

The program needs to read the radius entered by the user from the keyboard. This raises two important issues:

- Reading the radius.
- Storing the radius in the program.

Let's address the second issue first. In order to store the radius, 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. That is the kind of data stored in a variable, 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.*

Real numbers (i.e., numbers with a decimal point) are represented using a method known as *floating-point* in computers. So, the real numbers are also called *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
    }
}
```

The program declares **radius** and **area** as variables. The reserved word **double** indicates that **radius** and **area** are floating-point values stored in the computer.

The first step is to prompt the user to designate the circle's **radius**. You will soon learn how to prompt the user for information. For now, to learn how variables work, you can assign a fixed value to **radius** in the program as you write the code; later, you'll modify the program to prompt the user for this value.

The second step is to compute **area** by assigning the result of the expression **radius * radius * 3.14159** to **area**.

In the final step, the program will display the value of **area** on the console by using the **System.out.println** method.

The complete program on a file called MyFirstProgram.java (file name can be anything you want) looks as follows:

Listing 2.1

```
1 public class ComputeArea {
2     public static void main(String[] args) {
3         double radius; // Declare radius
4         double area; // Declare area
5
6         // Assign a radius
7         radius = 20; // radius is now 20
8
9         // Compute area
10        area = radius * radius * 3.14159;
11
12        // Display results
13        System.out.println("The area for the circle of radius "
          + radius + " is " + area);
15    }
16 }
```

Output: The area for the circle of radius 20 is 1256

Variables such as **radius** and **area** correspond to memory locations. Every variable has a name, a type, a size, and a value. Line 3 declares that **radius** can store a **double** value. The value is not defined until you assign a value. Line 7 assigns **20** into variable **radius**. Similarly, line 4 declares variable **area**, and line 10 assigns a value into **area**.

The plus sign (**+**) has two meanings: one for addition and the other for concatenating (combining) strings. The plus sign (**+**) in lines 13–14 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. Therefore, the plus signs (**+**) in lines 13–14 concatenate strings into a longer string, which is then displayed in the output. Strings and string concatenation will be discussed further in Chapter 4.

2.3 Reading Input from the Console

{Key Point} Reading input from the console enables the program to accept input from the user. In Listing 2.1, the radius is fixed in the source code. To use a different radius, you have to modify the source code and recompile it. Obviously, this is not convenient, so instead you can use the **Scanner** class for console input.

Java uses **System.out** to refer to the standard output device and **System.in** to the standard input device. By default, the output device is the display monitor and the input device is the keyboard. To perform console output, you simply use the **println** method to display a primitive value or a string to the console. 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 syntax **Scanner input** declares that **input** is a variable whose type is **Scanner**. The whole line **Scanner input = new Scanner(System.in)** creates a **Scanner** object and 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**. Listing 2.2 rewrites Listing 2.1 to prompt the user to enter a radius.

```
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 "
          + radius + " is " + area);
18    }
19 }
```

```
Enter a number for radius: 2.5
The area for the circle of radius 2.5 is 19.6349375

Enter a number for radius: 23
The area for the circle of radius 23.0 is 1661.90111
```

Line 9 displays a string **"Enter a number for radius: "** to the console. This is known as a *prompt*, because it directs the user to enter an input. Your program should always tell the user what to enter when expecting input from the keyboard. The **print** method in line 9 System.out.print(**"Enter a number for radius: "**); 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.

Line 6 creates a **Scanner** object. The statement in line 10 reads input from the keyboard.

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

After the user enters a number and presses the *Enter* key, the program reads the number and assigns it to **radius**. The **Scanner** class is in the **java.util** package. It is imported in line 1.

There are two types of **import** statements: *specific import* and *wildcard import*. The *specific import* specifies a single class in the import statement. For example, the following statement imports **Scanner** from the package **java.util**.

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

The *wildcard import* imports all the classes in a package by using the asterisk as the wildcard. For example, the following statement imports all the classes from the package **java.util**.

```
import java.uitl.*;
```

The information for the classes in an imported package is not read in at compile time or runtime unless the class is used in the program. The import statement simply tells the compiler

where to locate the classes. There is no performance difference between a specific import and a wildcard import declaration.

You can read multiple values from the keyboard just like below:

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

double number1 = input.nextDouble();
double number2 = input.nextDouble();
double number3 = input.nextDouble();
```

You can also read values who's type is one of other primitive datatypes or string like below:

```
int number1 = input.nextInt(); // Reads an integer
byte number2 = input.nextByte();// Reads a byte
long number3 = input.nextLong();// Reads a long
short number4 = input.nextShort();// Reads a short
String name = input.nextLine();// Reads a string
String surname = input.next();// Reads a string
boolean hasWon = input.nextBoolean();// Reads a Boolean
```

2.4 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. (See Appendix A for a list of reserved words.)
■ An identifier cannot be **true**, **false**, or **null**.
■ 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. Descriptive identifiers make programs easy to read. Avoid using abbreviations for identifiers. Using complete words is more descriptive.

2.5 Variables

{Key Point} Variables are used to represent values that may be changed in the program. You can assign any numerical value to **radius** and **area**, and the values of **radius** and **area** can be reassigned. For example, in the following code, **radius** is initially **1.0** (line 2) and then changed to **2.0** (line 7), and area is set to **3.14159** (line 3) and then reset to **12.56636**

```
1 // Compute the first area
2 radius = 1.0; radius: 1.0
3 area = radius * radius * 3.14159; area: 3.14159
4 System.out.println("The area is " + area + " for radius " + 5
radius);

6 // Compute the second area
7 radius = 2.0; radius: 2.0
8 area = radius * radius * 3.14159; area: 12.56636
9 System.out.println("The area is " + area + " for radius " + 10
radius);
```

Variables are for representing data of a certain type. To use a variable, you declare it by telling the compiler its name as well as what type of

data it can store. V*ariable declaration* tells the compiler to allocate appropriate memory space for the variable based on its data type. The syntax for declaring a variable is

```
int count; // Declare count to be an integer variable
double radius; // Declare radius to be a double variable
// Declare interestRate to be a double variable
double interestRate;
```

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

```
int i, j, k; // Declare i, j, and k as int variables
int x = 1, y = 2; // Declare & initialize two variables same time
```

You can declare a variable and initialize it in one step. Consider, for instance, the following code:

```
int count = 1;
//The above line is equivalent to the following two lines
int count;
count = 1;
```

2.6 Assignment Statements and Assignment Expressions

{Key Point} An assignment statement designates a value for a variable. An assignment statement can be used as an expression in Java. After a variable is declared, you can assign a value to it by using an *assignment statement.* In Java, the equal sign (**=**) is used as the *assignment operator.* The syntax for assignment statements is as follows:

variable = expression;

An *expression* represents a computation involving values, variables, and operators that, taking them together, evaluates to a value. For example, consider the following code:

```
int y = 1; // Assign 1 to variable y
double radius = 1.0; // Assign 1.0 to variable radius
int x = 5 * (3 / 2); // Assign the value of the expression to x
x = y + 1; // Assign the addition of y and 1 to x
double area = radius * radius * 3.14159; // Compute area
```

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; // This line adds 1 to the current value of x.
```

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 mathematics, **x = 2 * x + 1** denotes an equation. However, in Java, **x = 2 * x+ 1** is an assignment statement that evaluates the expression **2 * x + 1** and assigns the result to **x**.

In Java, an assignment statement is essentially an expression that evaluates to the value to be assigned to the variable on the left side of the assignment operator. For this reason, an assignment statement is also known as an *assignment expression.*

```
System.out.println(x = 1);
```

Is equivalent to

```
x = 1;
```

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

If a value is assigned to multiple variables, you can use this syntax:

```
i = j = k = 1;
```

Which is equivalent to

```
k = 1;
j = k;
i = j;
```

2.7 Named Constants

{Key Point} A named constant is an identifier that represents a permanent value. 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. Here is the syntax for declaring a constant:

```
final datatype CONSTANTNAME = value;
final double PI = 3.14159; // Declare a constant
```

A constant must be declared and initialized in the same statement. The word final is a Java keyword for declaring a constant.

2.8 Naming Conventions

Sticking with the Java naming conventions makes your programs easy to read and avoids errors. Make sure that you choose descriptive names with straightforward meanings for the variables, constants, classes, and methods in your program. As mentioned earlier, names are case sensitive. Listed below are the conventions for naming variables, methods, and classes.

■ *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**.*

It is important to follow the naming conventions to make your programs easy to read.

2.9 Numeric Data Types and Operations

{Key Point} Java has six numeric types for integers and floating-point numbers with operators **+**, **-**, *****, **/**, and **%**. Every data type has a range of values. The compiler allocates memory space for each variable or constant according to its data type. Java provides eight primitive data types for numeric values, characters, and Boolean values. This section introduces numeric data types and operators.

Name	Range	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 (i.e., -9223372036854775808 to 9223372036854775807)	64-bit signed
float	Negative range: -3.4028235E + 38 to -1.4E – 45 Positive range: 1.4E - 45 to 3.4028235E + 38	32-bit IEEE 754
double	Negative range: -1.7976931348623157E + 308 to -4.9E – 324	64-bit IEEE 754

	Positive range: 4.9E -324 to 1.7976931348623157E + 308	

Java uses four types for integers: **byte**, **short**, **int**, and **long**. Choose the type that is most appropriate for your variable. For example, if you know an integer stored in a variable is within a range of a byte, declare the variable as a **byte**.

Java uses two types for floating-point numbers: **float** and **double**. The **double** type is twice as big as **float**, so the **double** is known as *double precision* and **float** as *single precision*. Normally, you should use the **double** type, because it is more accurate than the **float** type.

Here are examples for reading values of various types from the keyboard:

```
1 Scanner input = new Scanner(System.in);
2 System.out.print("Enter a byte value: ");
3 byte byteValue = input.nextByte();
4
5 System.out.print("Enter a short value: ");
6 short shortValue = input.nextShort();
7
8 System.out.print("Enter an int value: ");
9 int intValue = input.nextInt();
10
11 System.out.print("Enter a long value: ");
12 long longValue = input.nextLong();
13
14 System.out.print("Enter a float value: ");
```

```
15 float floatValue = input.nextFloat();
```

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

+	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**, and **–5 / 2** yields **-2**, not **–2.5**.

The **%** operator, known as *remainder* or *modulo* operator, yields the remainder after division. The operand on the left is the dividend and the operand on the right is the divisor. Therefore, **7 % 3** yields **1**, **3 % 7** yields **3**, **12 % 4** yields **0**, **26 % 8** yields **2**, and **20 % 13** yields **7**.

```
1 import java.util.Scanner;
2
3 public class DisplayTime {
4     public static void main(String[] args) {
5         Scanner input = new Scanner(System.in);
6         // Prompt the user for input
7         System.out.print("Enter an integer for seconds: ");
8         int seconds = input.nextInt();
9
```

```
10          int minutes = seconds / 60; // Find minutes in seconds
11          int remainingSeconds = seconds % 60; /* Seconds
            remaining*/
12          System.out.println(seconds + " seconds is " + minutes +
13          " minutes and " + remainingSeconds + " seconds");
14    }
15 }
```

```
Enter an integer for seconds: 500
500 seconds is 8 minutes and 20 seconds
```

Exponents

The **Math.pow(a, b)** method can be used to compute *ab*. The **pow** method is defined in the **Math** class in the Java API. You invoke the method using the syntax **Math.pow(a, b)** (e.g., **Math.pow(2, 3)**), which returns the result of *ab* (2^3). Here, **a** and **b** are parameters for the **pow** method and the numbers **2** and **3** are actual values used to invoke the method. For example,

```
System.out.println(Math.pow(2, 3)); // Displays 8.0
System.out.println(Math.pow(4, 0.5)); // Displays 2.0
System.out.println(Math.pow(2.5, 2)); // Displays 6.25
System.out.println(Math.pow(2.5, -2)); // Displays 0.16
```

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.

2.10 Numeric Literals

{Key Point} 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;
```

2.10.1 Integer literal

An integer literal can be assigned to an integer variable as long as it can fit into the variable. A compile error will occur if the literal is too large for the variable to hold.

An integer literal is assumed to be of the **int** type, whose value is between -2^31 (-2147483648) and 2^31 - 1 (2147483647). To denote an integer literal of the **long** type, append the letter **L** or **l** to it. For example, to write integer **2147483648** in a Java program, you have to write it as **2147483648L** or **2147483648l**, because **2147483648** exceeds the range for the **int** value. **L** is preferred because **l** (lowercase **L**) can easily be confused with 1 (the digit one).

By default, an integer literal is a decimal integer number. To denote a binary integer literal, use a leading **0b** or **0B** (zero B), to denote an octal integer literal, use a leading **0** (zero), and to denote a hexadecimal integer literal, use a leading **0x** or **0X** (zero X). For example,

```
System.out.println(0B1111); // Byte literal displays 15
System.out.println(07777); // Octal literal displays 4095
System.out.println(0XFFFF); /* Hexadecimal literal displays
65535*/
```

2.10.2 Floating-Point Literals

Floating-point literals are written with a decimal point. By default, a floating-point literal is treated as a **double** type value. For example, **5.0** is considered a **double** value, not a **float** value. You can make a number a **float** by appending the letter **f** or **F**, and you can make a number a **double** by appending the letter **d** or **D**. For example, you can use **100.2f** or **100.2F** for a **float** number, and **100.2d** or **100.2D** for a **double** number.

The **double** type values are more accurate than the **float** type values. For example,

```
System.out.println("1.0 / 3.0 is " + 1.0 / 3.0);
```

displays **1.0 / 3.0 is 0.3333333333333333**

```
System.out.println("1.0F / 3.0F is " + 1.0F / 3.0F);
```

displays **1.0F / 3.0F is 0.33333334**

A float value has 7 to 8 number of significant digits and a double value has 15 to 17 number of significant digits.

Floating-point literals can be written in scientific notation in the form of *a* * 10^*b*. For example, the scientific notation for 123.456 is 1.23456 * 10^2 and for 0.0123456 is 1.23456 * 10^-2. A special syntax is used to write scientific notation numbers. For example, 1.23456 * 10^2 is written as **1.23456E2** or **1.23456E+2** and 1.23456 * 10^-2 as **1.23456E-2**. **E** (or **e**) represents an exponent and can be in either lowercase or uppercase.

2.10.3 Scientific Notation

The **float** and **double** types are used to represent numbers with a decimal point. Why are they called *floating-point numbers*? These numbers are stored in scientific notation internally. When a number such as **50.534** is converted into scientific notation, such as **5.0534E+1**, its decimal point is moved (i.e., floated) to a new position.

To improve readability, Java allows you to use underscores between two digits in a number literal. For example, the following literals are correct.

```
long ssn = 232_45_4519;
```

```
long creditCardNumber = 2324_4545_4519_3415L;
```

However, **45_** or **_45** is incorrect. The underscore must be placed between two digits.

2.11 Evaluating Expressions and Operator Precedence

{Key Point} Java expressions are evaluated in the same way as arithmetic expressions. Though Java has its own way to evaluate an expression behind the scene, the result of a Java expression and its corresponding arithmetic expression is the same. Therefore, you can safely apply the arithmetic rule for evaluating a Java expression. 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.

Listing 2.6 FahrenheitToCelsius.java

```
1 import java.util.Scanner;
2
3 public class FahrenheitToCelsius {
4     public static void main(String[] args) {
5         Scanner input = new Scanner(System.in);
6
7         System.out.print("Enter a degree in Fahrenheit: ");
```

```
8           double fahrenheit = input.nextDouble();
9
10          // Convert Fahrenheit to Celsius
11          double celsius = (5.0 / 9) * (fahrenheit - 32);
12          System.out.println("Fahrenheit " + fahrenheit + " is " +
13          celsius + " in Celsius");
14    }
15 }
```

```
Enter a degree in Fahrenheit: 100
Fahrenheit 100.0 is 37.77777777777778 in Celsius
```

2.12 Case Study: Displaying the Current Time

{Key Point} *You can invoke* **System.currentTimeMillis()** *to return the current time*. The problem is to develop a program that displays the current time in GMT (Greenwich Mean Time) in the format hour:minute:second, such as 13:19:8. The **currentTimeMillis** method in the **System** class returns the current time in milliseconds elapsed since midnight, January 1, 1970 GMT. This time is known as the *UNIX epoch.* The epoch is the point when time starts, and **1970** was the year when the UNIX operating system was formally introduced.

The **System.currentTimeMillis()** returns the number of milliseconds since the UNIX epoch.

```
1 public class ShowCurrentTime {
2     public static void main(String[] args) {
3         // Obtain the total milliseconds since midnight, Jan 1, 1970
4         long totalMilliseconds = System.currentTimeMillis();
5
6         // Obtain the total seconds since midnight, Jan 1, 1970
7         long totalSeconds = totalMilliseconds / 1000;
8
9         // Compute the current second in the minute in the hour
```

```
10          long currentSecond = totalSeconds % 60;
11
12          // Obtain the total minutes
13          long totalMinutes = totalSeconds / 60;
14
15          // Compute the current minute in the hour
16          long currentMinute = totalMinutes % 60;
17
18          // Obtain the total hours
19          long totalHours = totalMinutes / 60;
20
21          // Compute the current hour
22          long currentHour = totalHours % 24;
23
24          // Display results
25          System.out.println("Current time is " + currentHour +
            ":"
26          + currentMinute + ":" + currentSecond + " GMT");
27    }
28 }
```

Output: Current time is 17:31:8 GMT

Line 4 invokes **System.currentTimeMillis()** to obtain the current time in milliseconds as a **long** value. Thus, all the variables are declared as the long type in this program. The seconds, minutes, and hours are extracted from the current time using the **/** and **%** operators (lines 6–22).

2.13 Augmented Assignment Operators

{Key Point} *The operators* **+**, **-**, *****, **/**, *and* **%** *can be combined with the assignment operator to form augmented operators.*

+= Addition assignment **i += 8 equivalent to i = i + 8**

-= Subtraction assignment **i -= 8 equivalent to i = i – 8**

***=** Multiplication assignment **i *= 8 equivalent to i = i * 8**

/= Division assignment **i /= 8 equivalent to i = i / 8**

%= Remainder assignment **i %= 8 equivalent to i = i % 8**

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);
```

2.14 Increment and Decrement Operators

{The increment operator (++) and decrement operator (– –) are for incrementing and decrementing a variable by 1.

The **++** and —— are two shorthand operators for incrementing and decrementing a variable by **1**. These are handy because that's often how much the value needs to be changed in many programming tasks. For example, the following code increments **i** by **1** and decrements **j** by **1**.

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

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. These operators can also be placed before the variable. For example,

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

++i increments **i** by **1** and ——**j** decrements **j** by **1**. These operators are known as *prefix increment* (or preincrement) and *prefix decrement* (or predecrement). 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.

++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
```

2.15 Increment and Decrement Operators

{Key Point} Floating-point numbers can be converted into integers using explicit casting. Can you perform binary operations with two operands of different types? Yes. 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 ofvalues; 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.*

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. The following statement

```
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.

```
1 import java.util.Scanner;
2
3 public class SalesTax {
4     public static void main(String[] args) {
5         Scanner input = new Scanner(System.in);
6
7         System.out.print("Enter purchase amount: ");
8         double purchaseAmount = input.nextDouble();
9
10        double tax = purchaseAmount * 0.06;
11        System.out.println("Sales tax is $" + (int)(tax * 100) /
          100.0);
12    }
13 }
```

```
Enter purchase amount: 197.55
Sales tax is $11.85
```

==========End Of Chapter Two==========

Now You Can Checkout Chapter 2 Exercises And Their Corresponding Possible Solutions On https://www.mylifeprogrammingschool.com/java