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

Algorithm

- Writing a program involves designing algorithms and translating algorithms into programming instructions, or code.
- An *algorithm* is a plan that describes how a problem is solved by listing the actions that need to be taken and the order of their execution.

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.

Declaring Variables

- *Declaring variables* is to specify the data types, whether integer, real number, or something else.
- Java provides 8 primitive data types for numeric values, characters, and Boolean values. *(e.g.* **int**, **double**, **byte**, **short**, **long**, **float**, **char**, and **boolean**.)

```
int x;          // Declare x to be an integer variable;
double radius;  // Declare radius to be a double variable;
char gender;    // Declare gender to be a character variable;
```

- In a program, you only need to declare a variable **ONCE**.
- If variables are of the same type, they can be declared together, separated by commas, as follows:
  datatype variable1, variable2, ..., variableN;

Numeric data types

Primitive number types are divided into two groups:

- **Integer types** stores whole numbers, positive or negative (such as 123 or -456), without decimals. Valid types are byte, short, int and long.
- **Floating point types** represents numbers containing one or more decimals. There are two types: float and double.

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	

- 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.

https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html

Assignment statements

- A variable must be declared before it can be assigned a value.

```
x = 1;              // Assign 1 to x;
radius = 1.0;       // Assign 1.0 to radius;
gender = 'M';         // Assign 'M' to gender;
```

- 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;
- Place the variable name to the left of the assignment operator. Thus, the following statement is wrong: **1** = x; // Wrong
- 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**; // the value of x becomes 2

Declaring and Assignment in One Step

- You can declare a variable and initialize it in one step: **int** count = **1**;
- This is equivalent to the next two statements:

```
int count;
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**;
- 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**.

Common error: **Undeclared/Uninitialized Variables and Unused Variables**

- A variable must be declared with a type and assigned a value before using it. A common error is not declaring a variable or initializing a variable.
- Remove unused variables because they might be potential programming error. For example, in the following code, **taxRate** is never used. It should be removed from the code.

```
double interestRate = 0.05;
double taxRate = 0.05;
double interest = interestRate * 45;
System.out.println("Interest is " + interest);
```

If you mistype interestrate here, you will have an undeclared variable because Java is case sensitive.

- If you use an IDE such as Eclipse, you will receive a warning on unused variables.

Common error: *integer overflow*

- Numbers are stored with a limited numbers of digits. When a variable is assigned a value that is too large (*in size*) to be stored, it causes *overflow*.
- For example, executing the following statement causes overflow,
  int value = **2147483647** + **1**; // value will actually be -2147483648
  because the largest value that can be stored in a variable of the **int** type is **2147483647**. **2147483648** will be too large for an **int** value.
  Likewise for the following statement:
  int value = **-2147483648** - **1**; // value will actually be 2147483647
- Java does not report warnings or errors on the above overflow cases. Be careful when working with numbers close to the maximum or minimum range of a given type.
- Don't think of overflow as "a value exceeding its maximum size". The bits simply wrap around and stay within that data type's limits.

Common error: *integer overflow*

- However, **int** value = **2147483648** will result in error being reported upon compilation.
- Floating-point operations will not cause overflow.

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
  }
}
```


Scanner class: Reading input from console

- Java uses **System.in** to the standard input device. 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.

Reading Numbers from the keyboard

- We have discussed 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.

Token-based input

- nextByte(), nextShort(), nextInt(), nextLong(), nextFloat(), nextDouble() are called token-based input methods.
- They read input separated by delimiters (whitespace characters by default, or the Enter key).
- A token-based input first skips any delimiters then reads a token ending at a delimiter.
- It does not read the delimiter after the token.

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.

Notes regarding ComputeAreaWithConsoleInput.java

- The **Scanner** class is in the **java.util** package. It is imported in line 1.
  - *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;
  - *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.util.*;
  - 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.
- A string cannot cross lines in the source code. Thus, the following statement would result in a compile error:

  System.out.println("**Introduction to Java Programming,**
  by Y. Daniel Liang");
  - To fix the error, break the string into separate substrings, and use the concatenation operator (**+**) to combine them:

    System.out.println("**Introduction to Java Programming, " +**
    "by Y. Daniel Liang");

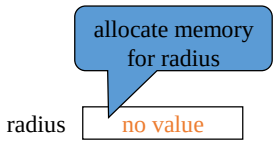
ComputeAreaWithConsoleInput.java: Eclipse issues a warning in Line 6

- Eclipse issues a warning that you have a "Resource leak: 'input' is never closed" for the line Scanner input = **new Scanner(System.*in);***
- You might be tempted to get rid of this warning by calling the Scanner.close() method when you're done using the Scanner.
- Since the System.in object is opened by the JVM, you should leave it to the JVM to close it.
- If you close it and later on try to use System.in, then you will find that you no longer can. Therefore, when it comes to closing a Scanner that is tied to System.in, **DON'T**.
- Just ignore this warning for the time being. Leave the closing of System.in to the JVM.

animation

Trace a Program Execution

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

    // Assign a radius
    radius = 20;

    // Compute area
    area = radius * radius * 3.14159;

    // Display results
    System.out.println("The area for the circle of radius " +
      radius + " is " + area);
  }
}
```

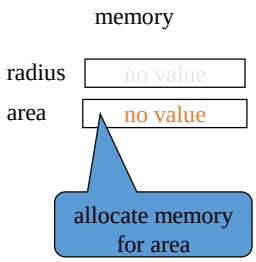


24

animation

Trace a Program Execution

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

  // Assign a radius
  radius = 20;

  // Compute area
  area = radius * radius * 3.14159;

  // Display results
  System.out.println("The area for the circle of radius " +
    radius + " is " + area);
 }
}
```

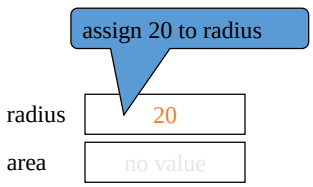


25

animation

Trace a Program Execution

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

  // Assign a radius
  radius = 20;

  // Compute area
  area = radius * radius * 3.14159;

  // Display results
  System.out.println("The area for the circle of radius " +
    radius + " is " + area);
 }
}
```



26

animation

Trace a Program Execution

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

  // Assign a radius
  radius = 20;

  // Compute area
  area = radius * radius * 3.14159;

  // Display results
  System.out.println("The area for the circle of radius " +
    radius + " is " + area);
 }
}
```

memory

radius 20

area 1256.636

compute area and assign it to variable area

animation

Trace a Program Execution

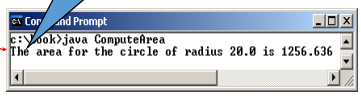
```
public class ComputeArea {
 /** Main method */
 public static void main(String[] args) {
  double radius;
  double area;

  // Assign a radius
  radius = 20;

  // Compute area
  area = radius * radius * 3.14159;

  // Display results
  System.out.println("The area for the circle of radius " +
   radius + " is " + area);
 }
}
```

memory

radius 20

area 1256.636

print a message to the console



Listing 2.3 ComputeAverage.java

```
import java.util.Scanner;
public class ComputeAverage {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    // Prompt the user to enter three numbers; without this prompt, users do not know what is expected to do.
    System.out.print("Enter three numbers: ");
    double number1 = input.nextDouble();
    double number2 = input.nextDouble();
    double number3 = input.nextDouble();

    // Compute average
    double average = (number1 + number2 + number3) / 3;

    // Display result
    System.out.println("The average of " + number1 + " " + number2
      + " " + number3 + " is " + average);
  }
}
```

Note that you only need to create a Scanner object (here is input) **ONCE** and then use it multiple times.

Most of the programs in the early chapters of this book perform three steps— input, process, and output—called *IPO*.

- Input is receiving input from the user;
- process is producing results using the input;
- output is displaying the results.

Brief introduction on Java Classes and Objects

- 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, but NOT 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.

Compile-time error

- The final keyword makes a variable unchangeable. This means that you can initialize it and you will never be able to change it.

```
final double PI = 3.14159;
PI = 2; // You are trying to change the value of a constant,
        // this will result in a compile-time error
```

- To change a variable's value, just re-assign a value to that variable:

```
double pi = 3.14159;
pi = 2; //pi is not a constant, so its value can be changed
```


Advantages of using constants

There are three benefits of using constants:

(1) You don't have to repeatedly type the same value if it is used multiple times;

(2) If you have to change the constant value (e.g., from **3.14** to **3.14159** for **PI**), you need to change it only in a single location in the source code; and

(3) A descriptive name for a constant makes the program easy to read.

Naming conventions

- Use lowercase for variables and methods - for example, the variables **radius** and **area** and the method **print**.
- 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, e.g. finalMark.
- 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**.

Literals for Primitive Types

- A *literal* is a *specific constant value* used in the program source, such as 123, -456, 3.1416, 4.5e6, 'a', "Hello".
- It can be assigned directly to a variable; or used as part of an expression.
- We call it *literal* to distinguish it from a *variable*.

Integer (int) literals

- A whole number literal, whose value is between -2^{31} (-2147483648) to 2^{31}–1 (2147483647), is treated as an int by default, such as 123 and -456.
- An int literal may precede with a plus (+) or minus (-) sign, followed by digits. No commas or special symbols (e.g., $, %, or space) is allowed.
- For example,

```
int number = -456;
int intRate = 6%; // error: no percent sign
int pay = $1234; // error: no dollar sign
int product = 1,234,567; // error: no grouping commas
```


Integer (long, short, byte) literals

- A long literal outside the int range requires a suffix 'L' or 'l' (avoid lowercase 'l', which could be confused with the number one '1'), e.g., 123456789012L, -9876543210l. For example,

```
long sum = 123; // Within the "int" range, no need for suffix 'L'
long bigSum = 1234567890123L; // Outside "int" range, suffix 'L' needed
```

- No suffix is needed for byte and short literals. But you can only use values in the permitted range. For example,

```
byte smallNumber1 = 123; // This is within the range of byte [-128, 127]
byte smallNumber2 = -1234; // error: this value is out of range
short midSizeNumber1 = -12345; // This is within the range of short [-32768, 32767]
short midSizeNumber2 = 123456; // error: this value is out of range
```

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 make a number a double by appending the letter d or D. For example,

```
float average = 55.66; // syntax error: 55.66 is a double. Need suffix 'f' for float.
float average = 55.66F; // float literal needs suffix 'f' or 'F'
```

- For example, you can use 100.2f or 100.2F for a float number, and 100.2d or 100.2D for a double number

Scientific Notation

- Floating-point literals can also be a scientific number with an "e" to indicate the power of 10.
- For example, 1.23456e+2, same as 1.23456e2, is equivalent to 123.456, and 1.23456e-2 is equivalent to 0.0123456. E (or e) represents an exponent and it can be either in lowercase or uppercase.

Arithmetic Expressions

$$\frac{3+4x}{5} - \frac{10(y-5)(a+b+c)}{x} + 9\left(\frac{4}{x} + \frac{9+x}{y}\right)$$

is translated to

```
(3+4*x)/5 – 10*(y-5)*(a+b+c)/x + 9*(4/x + (9+x)/y)
```


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.

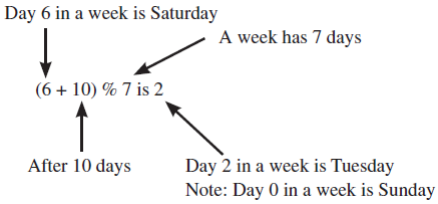
TABLE 2.3 Numeric Operators

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.

```
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
```

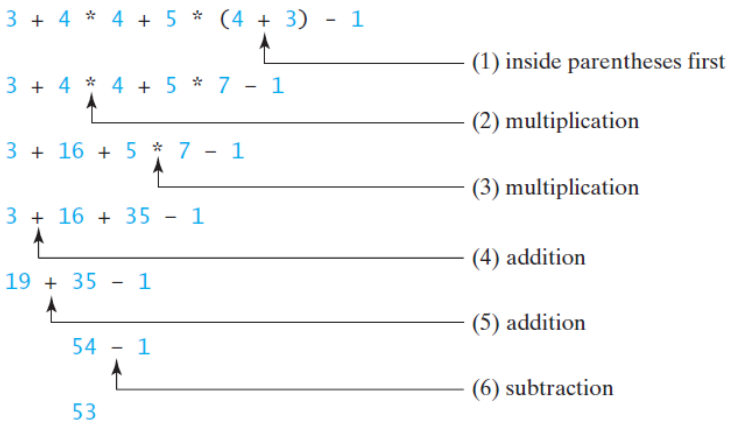
- Chapter 4 introduces more details on methods. For now, all you need to know is how to invoke the **pow** method to perform the exponent operation.
- Since Math is in the java.lang package, it does not need to be imported. java.lang is the "default package" and everything in it is already implicitly imported for you.

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`

In short, it is a matter of doing which of the two tasks first, namely:
(A) Increment / decrement;
(B) assignment

Postfix vs Prefix (cont'd)

```
int i = 10;
int newNum = 10 * i++;
```

Same effect as

```
int newNum = 10 * i;
i = i + 1;
```

```
int i = 10;
int newNum = 10 * (++i);
```

Same effect as

```
i = i + 1;
int newNum = 10 * i;
```


Numeric Type Conversion Rules

- 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**.
- When performing a binary operation involving two operands of different types, Java automatically converts the operand based on the following rules:
  1. If one of the operands is double, the other is converted into double.
  2. Otherwise, if one of the operands is float, the other is converted into float.
  3. Otherwise, if one of the operands is long, the other is converted into long.
  4. Otherwise, both operands are converted into int.

```
range increases
------------------------------>
byte, short, int, long, float, double
```

Implicit Casting

- You can always assign a value to a numeric variable whose type supports a larger range of values; thus, for instance, you can assign an **int** value to a **double** variable.

  double x = 4; // *widening a type*
- *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.* Java will automatically widen a type.
- You cannot, however, assign a value to a variable of a type with a smaller range unless you use *type casting,* such as assigning a **double** value to an **int** variable. A compile error will occur if casting is not used in situations of this kind.

  int x = 4.5; // syntax error

Explicit Casting

- Floating-point numbers can be converted into integers using explicit casting.

  int x = (int) 4.5; // type narrowing, fraction part is truncated
- Casting a type with a large range to a type with a smaller range is known as *narrowing a type.* However, be careful when using casting, as loss of information might lead to inaccurate results.
- Java will automatically widen a type, but you must narrow a type explicitly.
- What is wrong?

  int x = 5 / 2.0;

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.

Examples of casting

- Casting does not change the variable being cast. For example, **d** is not changed after casting in the following code:

```
double d = 4.5;
int i = (int)d;    // i becomes 4, but d is still 4.5
```

  The code above narrows a type from double to int, which has to be done explicitly.
  There will be syntax error if the explicit casting is missing.
- In Java, an augmented expression of the form **x1 op= x2** is implemented as **x1 =(T)(x1 op x2)**, where **T** is the type for **x1**. Therefore, the following code is correct.
  - **int** sum = **0**;
  - sum += **4.5**; // sum becomes 4 after this statement

  Note that sum += 4.5 is equivalent to **sum = (int)(sum + 4.5)**.
- To display a floating-point (e.g. tax) with two digits after the decimal point:
  - System.out.println("**Sales tax is $**" + (**int**)(tax * **100**) / **100.0**);

Common Error: Unintended Integer Division

- Java uses the same divide operator, namely **/**, to perform both integer and floating-point division.
- When two operands are integers, the **/** operator performs an integer division. The result of the operation is an integer. The fractional part is truncated.
- To force two integers to perform a floating-point division, make one of the integers into a floating-point number.

Common Pitfall 1: Redundant Input Objects

```
Scanner input = new Scanner(System.in);
System.out.print("Enter an integer: ");
int v1 = input.nextInt();

Scanner input1 = new Scanner(System.in);
System.out.print("Enter a double value: ");
double v2 = input1.nextDouble();
```

BAD CODE

```
Scanner input = new Scanner(System.in);
System.out.print("Enter an integer: ");
int v1 = input.nextInt();
System.out.print("Enter a double value: ");
double v2 = input.nextDouble();
```

GOOD CODE

Note that you only need to create a Scanner object (here is input) **ONCE** and then use it multiple times.

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.

Exercises

1. Which of these data types requires the most amount of memory?
   long, int, short, byte
2. Identify and fix the error(s) in the following code.

```
double interestRate = 0.05;
double interest = interestrate * 45;
```

3. Write the code to declare a constant named PI with value equals 3.14159.
4. What is the output of the following code?

```
int number1 = 1;
int number2 = 2;
double average = (number1 + number2) / 2;
System.out.println(average);
```

5. According to Java naming convention, which of the following can be variable names?
   FindArea, findArea, totalLength, TOTAL_LENGTH, class
6. Which of the following are Java keywords?
   class, **public**, **int**, **x**, **radius**