

Name: _____ () Date: _____

Chapter 2: Programming in Java

Lesson Objectives

After completing the lesson, the student will be able to

- *write Java programs to perform simple computations*
- *use identifiers to name variables, constants, methods, and classes*
- *use variables to store data*
- *program with assignment statements and assignment expressions*
- *use constants to store permanent data*
- *explore Java numeric primitive data types: byte, short, int, long, float, and double*
- *explore character and Boolean type*
- *perform operations using arithmetic operators +, -, *, /, and %*
- *write literals in scientific notation*
- *write and evaluate numeric expressions*
- *use augmented assignment operators*
- *distinguish between post-increment and pre-increment and between post-decrement and pre-decrement*
- *cast the value of one type to another type*
- *understand streams in Java*
- *obtain input from the console using the Scanner class with console input (System.in)*
- *format console output using format specifiers in System.out.printf()*
- *get an overview of exceptions and exception handling*
- *write a try-catch block to handle exceptions*

2.1 Writing A Simple Program

Writing a program involves designing a strategy (algorithm) for solving the problem and then using a programming language to implement that strategy. 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).

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 * pi
3. Display the result

This raises two important issues:

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

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, you have to 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. Declaring `radius` and `area` as `double`, the area of circle program can be written as shown in PROGRAM 2-1.

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

PROGRAM 2-1

PROGRAM OUTPUT

The area for the circle of radius 20.0 is 1256.636

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 6 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 10–11 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 10–11 concatenate strings into a longer string, which is then displayed in the output using standard output (covered in Chapter 1)

2.2 Identifier

Identifiers are the names that identify the elements such as classes, methods, and variables in a program. In earlier program examples, `ComputeAverage`, `main`, `area`, `radius`, and so on are the names of things that appear in the program. In programming terminology, such names are called **identifiers**. **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 B-1 for a list of reserved words.)
- An identifier cannot be `true`, `false`, or `null`.
- An identifier can be of any length

For example, the following are all valid identifiers:

```
g    g1    g_1    _xyz    EFG    _123k7    total    RATE    count    data2    bigDiscount
```

All the above names are legal and would be accepted by the compiler, but the first five are poor choices for identifiers because they are not descriptive of the identifier's use. None of the following are legal identifiers, and all would be rejected by the compiler:

```
145    3Z    @change    data-1    myfirst.JAVA    PROG.java
```

The first three are not allowed because they do not start with a letter or an underscore. The remaining three are not identifiers because they contain symbols other than letters, digits, and the underscore symbol. Although it is legal to start an identifier with an underscore, you should avoid doing so, because identifiers starting with an underscore are informally reserved for system identifiers and standard libraries. **Java is a case-sensitive** language; that is, it distinguishes between uppercase and lowercase letters in the spelling of identifiers. Hence, the following are three distinct identifiers and could be used to name three distinct variables:

```
total    TOTAL    Total
```

However, it is not a good idea to use two such variants in the same program, since that might be confusing. Although it is not required by Java, variables are usually spelled with their first letter in lowercase. The convention that is now becoming universal in object-oriented programming is to spell variable names with a mix of upper- and lowercase letters (and digits) known as **camel case**. You always start a variable name with a lowercase letter and to indicate “word” boundaries with an uppercase letter, as illustrated by the following variable names:

```
topSpeed    bankRate1    bankRate2    timeOfArrival
```

The Java compiler detects illegal identifiers and reports syntax errors. 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.** 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 `InterestRate`.
- Capitalize every letter in a constant, and use underscores between words—for example, the constants `PI` and `MAX_VALUE`.

2.3 Variables

Variables are used to represent values that may be changed in the program. They are called variables because their values can be changed. 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. **The variable declaration tells the compiler to allocate appropriate memory space for the variable based on its data type.**

The syntax for declaring a variable is `datatype variableName;` Here are some examples of variable declarations:

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

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

```
datatype variable1, variable2, ..., variablen;
```

The variables are separated by commas. For example,

```
int i, j, k; // Declare i, j, and k as int variables
```

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

```
int count = 1;
```

This is equivalent to the next two statements:

```
int count; // uninitialized yet
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;
```

A variable must be declared before it can be assigned a value. A variable declared in a method (function) must be assigned a value before it can be used. Whenever possible, declare a variable and assign its initial value in one step. This will make the program easy to read and avoid programming errors. **Every variable has a scope. The scope of a variable is the part of the program where the variable can be referenced.**

2.4 Assignment Statements and Assignment Expression

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;
```

In this assignment statement, the result of $x + 1$ is assigned to x . If x is 1 before the statement is executed, then it becomes 2 after the statement is executed. 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**. For example, the following statement is correct:

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

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, because the data type of `x` is `int`. You cannot assign a double value, `1.0`, to an `int` variable without using **type casting**.

2.5 Name 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 PROGRAM 2-1 program, `3.14159` is a literal constant. If you use it frequently, you do not want to keep typing `3.14159`; instead, you can declare a constant to represent `3.14159`. Here is the syntax for declaring a constant:

```
final datatype CONSTANTNAME = value;
```

A constant must be declared and initialized in the same statement. The word `final` is a Java keyword for declaring a constant. For example, you can declare `PI` as a constant and rewrite PROGRAM 2-1 as in PROGRAM 2-2.

<pre> 1 public class ComputeAreaWithConstant { 2 public static void main(String[] args) { 3 final double PI = 3.14159; // Declare a constant 4 double radius; // Declare radius 5 double area; // Declare area 6 radius = 20; // Assign a radius and radius is now 20 7 8 area = radius * radius * PI; // Compute area 9 // Display results 10 System.out.println("The area for the circle of radius " + 11 radius + " is " + area); 12 } 13 }</pre>	PROGRAM 2-2
---	--------------------

PROGRAM OUTPUT

```
The area for the circle of radius 20.0 is 1256.636
```

There are three benefits of using constants: **(1)** you do not 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.

2.6 Primitive Data Types

Name	Range	Storage
boolean	true or false	System dependent
char	(0 to 65535) '\u0000' to '\uFFFF'	16-bit Unicode
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
double	Negative range: $-1.7976931348623157E + 308$ to $-4.9E - 324$ Positive range: $4.9E - 324$ to $1.7976931348623157E + 308$	64-bit

Table 2.1 Primitive Data Types

The eight data types in Table 2.1 are called **primitive** because they are simple and uncomplicated. Primitive types also serve as the building blocks for more complex data types, called **reference types**, which hold memory addresses. Reference type will be further discussed in Chapter 3.

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.**

2.7 Scientific Notation

Floating-point numbers 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.

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.

2.8 Character Type

In Java single characters are represented by using the data type `char`. **The char data type holds two 8-bit bytes and is meant to represent characters.** It is stored as an unsigned 16-bit integer with a minimum value of 0 and a maximum value of 65,535. However, you should never use a `char` to store a number, because that can lead to confusion.

Literal values of type char are enclosed in single quotes. For example, 'A' is a character constant with value 65. It is different from "A", a string containing a single character. Values of type `char` can also be expressed as hexadecimal values that run from `\u0000` to `\uFFFF`. For example, `\u2122` is the trademark symbol (™).

Besides the `\u` escape sequences, there are several escape sequences for special characters, as shown in Table 2.2 You can use these escape sequences inside quoted character literals and strings, such as `'\u2122'` or `"Hello\n"`. The `\u` escape sequence (but none of the other escape sequences) can even be used outside quoted character constants and strings. For example,

```
public static void main(String\u005B\u005D args)
```

is perfectly legal —`\u005B` and `\u005D` are the encodings for `[` and `]`.

Escape Sequence	Name	Unicode
<code>\b</code>	Backspace	<code>\u0008</code>
<code>\t</code>	Tab	<code>\u0009</code>
<code>\n</code>	Linefeed	<code>\u000a</code>
<code>\r</code>	Carriage return	<code>\u000d</code>
<code>\"</code>	Double quote	<code>\u0022</code>
<code>\'</code>	Single quote	<code>\u0027</code>
<code>\\</code>	Backslash	<code>\u005c</code>

Table 2.2 Escape Sequence For Special Characters

Characters are declared and used in a manner similar to data of other types as shown in PROGRAM 2-3. The declaration `char1` is initialized to 'B'. We can display the ASCII code of a character by converting it to an integer using type casting. For example, we can execute

```
System.out.println( "ASCII code of character B is " + (int)char1) ;
```

Conversely, we can see a character by converting its ASCII code to the `char` data type, for example,

```
System.out.println("Character with ASCII code 88 is " + (char)88);
```

1	<code>public class CharacterDemo {</code>	PROGRAM 2-3
2	<code> public static void main(String[] args) {</code>	
3	<code> char char1 = 'B';</code>	
4	<code> System.out.println("Character with ASCII code 88 is " + (char)88));</code>	
5	<code> System.out.println("ASCII code of character B is " + (int) char1);</code>	
6	<code> System.out.println('\u2122');</code>	
7	<code> System.out.println('\u005B' + " " + '\u005D');</code>	
8	<code> }</code>	
9	<code>}</code>	

PROGRAM OUTPUT

```
Character with ASCII code 88 is X
```

```
ASCII code of character B is 66
```

```
™
```

```
[ ]
```

Because the characters have numerical ASCII values, we can compare characters just as we compare integers and real numbers. For example, the comparison `'A' < 'c'` returns `true` because the ASCII value of `'A'` is 65 while that of `'c'` is 99.

2.9 Boolean Type

The **boolean type** has two literal values, `false` and `true`. It is used for evaluating logical conditions. You cannot convert between integers and boolean values. (Unlike some other programming languages, the Java values `true` and `false` are not integers and will not be automatically converted to integers.). For example, a variable of `boolean` type can be declared as follows:

```
boolean raining = true; // variable raining is assigned the value true
if(raining == true){
    System.out.println("Bring umbrella");
}
```

2.10 Evaluating Expressions and Arithmetic Operator's Precedence

Java expressions are evaluated in the same way as arithmetic expressions. Most programs perform arithmetic calculations. The arithmetic operators are summarized in Table 2.2. Note the use of various special symbols not used in algebra. The asterisk (*) indicates multiplication, and the percent sign (%) is the remainder operator. The arithmetic operators in Table 2.2 are **binary operators**, because each operates on **two operands**. For example, the expression `f + 7` contains the binary operator `+` and the two operands `f` and `7`.

Operator	Name	Algebraic Expression	Java Expression
+	Addition	$f + 7$	<code>f + 7</code>
-	Subtraction	$p - c$	<code>p - c</code>
*	Multiplication	bm	<code>b * m</code>
/	Division	x / y , $\frac{x}{y}$ or $x \div y$	<code>x / y</code>
%	Remainder	$r \bmod s$	<code>r % s</code>

Table 2.2 Arithmetic Operators

Integer division yields an integer quotient. For example, the expression `7 / 4` evaluates to 1, and the expression `17 / 5` evaluates to 3. **Any fractional part in integer division is simply truncated (i.e., discarded)—no rounding occurs. Be careful when applying division**, as shown in PROGRAM 2-4. Division of two integers yields an integer in Java. 5 divided by 9 has to be written as `5.0 / 9` instead of `5 / 9` line 8, because `5 / 9` yields 0 in Java.

1	<code>public class FahrenheitToCelsius {</code>	PROGRAM 2-4
2	<code> public static void main(String[] args) {</code>	
3	<code> double fahrenheit = 99;</code>	
4	<code> double celsius = (5.0 / 9) * (fahrenheit - 32);</code>	
5	<code> System.out.println("Fahrenheit " + fahrenheit + " is " +</code>	
6	<code> celsius + " in Celsius");</code>	
7	<code> }</code>	
8	<code>}</code>	

PROGRAM OUTPUT

```
Enter a degree in Fahrenheit: 99
Fahrenheit 99.0 is 37.22222222222222 in Celsius
```

Java provides the remainder operator, %, which yields the remainder after division. The expression `x % y` yields the remainder after `x` is divided by `y`. Thus, `7 % 4` yields 3, and `17 % 5` yields 2. This operator is most commonly used with integer operands but it can also be used with other arithmetic types.

For example, the arithmetic expression

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

can be translated into a Java expression as:

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

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.

These rules enable Java to apply operators in the correct order. When we say that operators are applied from left to right, we are referring to their **associativity**. Some operators associate from right to left. Refer to Appendix B-2 for detailed operator precedence.

2.11 Augmented Assignment Operators (Compound Assignment)

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 = count + 1;
```

Java allows you to combine assignment and addition operators using an augmented (or compound) assignment operator. For example, the preceding statement can be written as

```
count += 1;
```

The += is called the addition assignment operator. Table 2.3 shows other 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

Table 2.3 Augmented Assignment Operators

The augmented assignment operator is performed last after all the other operators in the expression are evaluated. There are no spaces in the augmented assignment operators. For example, `x /= 4 + 5.5 * 1.5;` is same as `x = x / (4 + 5.5 * 1.5);`

2.12 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. 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 post-increment) and **postfix decrement** (or post-decrement), as the operators ++ and -- are placed after the variable. Similarly, they can 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 pre-increment) and **prefix decrement** (or pre-decrement). The effect of ++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**, as seen below:

Operator	Name	Description	Example (Let initial value of i be 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, but 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 0, i is -1

Table 2.5 Increment and Decrement Operator

Here are additional examples to illustrate the differences between the prefix form of ++ (or --) and the postfix form of ++ (or --). Consider the following code:

```
int i = 10;
int newNum = 10 * i++;
System.out.print("i is " + i
    + ", newNum is " + newNum);
```

Same effect as →

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

i is 11, newNum is 100

i is incremented by 1, then the old value of i is used in the multiplication. So newNum becomes 100. If i++ is replaced by ++i as follows,

```
int i = 10;
int newNum = 10 * (++i);
System.out.print("i is " + i
    + ", newNum is " + newNum);
```

Same effect as →

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

i is 11, newNum is 110

i is incremented by 1, and the new value is used in the multiplication, i.e newNum becomes 110.

Using increment and decrement operators makes expressions short, but it also makes them complex and difficult to read. Avoid using these operators in expressions that modify multiple variables or the same variable multiple times, such as this one: `int k = ++i + i.`

2.13 Type Casting

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.

The syntax for casting is to specify the target type in parentheses, followed by the variable's name or value to be cast. For example, `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. **Casting is necessary if you are assigning a value to a variable of a smaller type range, 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. However, be careful when using casting, as loss of information might lead to inaccurate results. **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
```

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
sum = (int)(sum + 4.5);
```

To assign a variable of the `int` type to a variable of the short or byte type, explicit casting must be used. For example, the following statements have a compile error:

```
int i = 1;
byte b = i; // Error because explicit casting is required
```

However, so long as the integer literal is within the permissible range of the target variable, explicit casting is not needed to assign an integer literal to a variable of the short or byte type.

```
byte c = 100; // within the range of a byte
byte d = 1000; // not ok, out of permissible range
```

2.14 Modularizing Code with Methods

Methods in Java functions in a similar way to Python. **Methods can be used to define reusable code and organize and simplify coding. A method is a collection of statements grouped together to perform an operation.**

Back to the example on computing the area of a circle (See PROGRAM 2-1).

We can modularize the computation of area using methods as follows:

<pre> 1 public class CircleComputation { 2 3 public static void main(String[] args) { 4 double radius = 20; 5 double area = computeArea(radius); // calls method computeArea 6 System.out.println("The area for the circle of radius " + 7 radius + " is " + area); 8 } 9 10 public static double computeArea(double r) { 11 double a = r * r * 3.14159; 12 return a; 13 } 14 }</pre>	PROGRAM 2-5
---	--------------------

In line 5, a method named `computeArea()` is invoked (or called).

In lines 10 to 14, a method named `computeArea()` is defined.

The method takes in an input variable named `r`, which is of double data type.

The method computes the area in line 11, and return the value computed (variable `a`) in line 12.

The variable returned, `a`, is a double. Hence, the return type is double in line 10.

A method definition consists of its method name, parameters, return value type, and body. The syntax for defining a method is as follows:

```

modifier returnType methodName(list of parameters) {
}

public static double computeArea(double r) {
    double a = r * r * 3.14159;
    return a;
}
```

Similarly, PROGRAM 2-4 may be modularized as follows:

<pre> 1 public class FahrenheitToCelsius { 2 public static void main(String[] args) { 3 double fahrenheit = 99; 4 double celsius = convertFtoC(fahrenheit); 5 System.out.println("Fahrenheit " + fahrenheit + " is " + 6 celsius + " in Celsius"); 7 } 8 9 public static double convertFtoC(double f){ 10 double c = (5.0 / 9) * (f - 32); 11 return c; 12 } 13 }</pre>	PROGRAM 2-6
--	--------------------

2.15 Console Output: System.out.printf

In Java, a source of input data is called an **input stream** and the output data, such as `System.out.println` and `System.out.print`, covered in Chapter 1, is called an **output stream**.

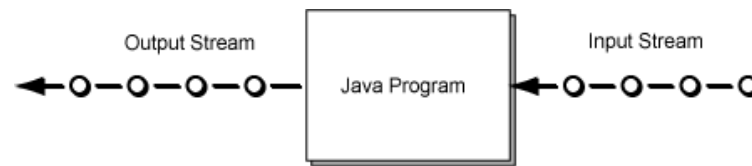


Figure 2.1 Input and Output Streams

In the picture, each "O" represents data waiting to be input or output. The input stream is a pipeline which the program inputs one at a time, in order. The output stream is another pipeline displaying program output. Often a program reads and combines data to produce one output value. For example, the input data might be a list of numbers, the output data might be their sum.

There are three standard I/O streams:

- `System.in` — the input stream (connected to keyboard and contain character data)
- `System.out` — the output stream (connected to monitor) for normal results.
- `System.err` — the output stream (connected to monitor) for error messages.

In this section, we will cover another variation of console output. If you have a variable of type `double` that stores some amount of money, and you want to output the amount in a nice format. If you just use `System.out.println`, you are likely to get output that looks like the following:

```
Your cost, including tax, is $19.98327634144
```

If you would like the output to look like this:

```
Your cost, including tax, is $19.98
```

To obtain this nicer form of output, you need some formatting tools. **Java includes a method named `System.out.printf` that can be used to give output in a specific format.** This method is used the same way as the method `print` but allows you to add formatting instructions that specify such things as the number of digits to include after a decimal point.

1 2 3 4 5 6 7	<pre>public class ConsoleOutput { public static void main(String[] args) { double price = 19.8; System.out.print("\$"); System.out.printf("%6.2f", price); System.out.println(" each"); } }</pre>	PROGRAM 2-6
---------------------------------	---	--------------------

PROGRAM OUTPUT

```
$ 19.80 each
```

PROGRAM 2-6 at line 5 outputs the string " 19.80" (one blank followed by 19.80), which is the value of the variable `price` written in the format `%6.2f`. The first argument to `printf` is a string known as the **format specifier**, and the second argument is the intended output in that format. The format specifier `%6.2f` outputs a floating-point number in a field (number of spaces) of width 6 (room for six characters) and a precision of two digits after the decimal point. So, 19.8 is expressed as "19.80" in a field of width 6. Because "19.80" has only five characters, a blank character is added to obtain the six-character string " 19.80". Any extra blank space is added to the front (left-hand end) of the value output. The `f` means the output is a floating-point number, that is, a number with a decimal point.

The first argument to `printf` can include text as well as a format specifier. Thus, Line 4 to 6 can be combined into a single statement as follows:

```
System.out.printf("$%6.2f each", price);
```

The text before and after the format specifier `%6.2f` is output along with the formatted number. The character `%` signals the end of text to output and the start of the format specifier. The end of a format specifier is indicated by a conversion character (`f` in our example). Table 2.6 shows other types of format specifiers.

Conversion Character	Type of Output	Example
d	Decimal (ordinary) integer.	<code>%5d</code> , <code>%d</code>
f	Fixed-point (everyday notation) floating-point	<code>%6.2f</code> , <code>%f</code>
e	E-notation floating point	<code>%8.3e</code> , <code>%e</code>
g	General floating point. (Java decides whether to use E-notation or not.)	<code>%8.3g</code> , <code>%g</code>
s	String	<code>%12s</code> , <code>%s</code>
c	Character	<code>%2c</code> , <code>%c</code>
n	Denotes a line break. This does not correspond to an output argument. It is approximately equivalent to <code>\n</code> .	<code>%n</code>

Table 2.6 Types of Format Specifiers

In summary, the format specifier in the form **N.M**, specifies a field width of **N spaces** with **M digits** after the decimal point. **If only N is given, it specifies a field width**; if there is a decimal point in the output, then the number of digits after the decimal point is determined by Java. When the value output does not fill the field width specified, then blanks are added in front of the value output. The output is then said to be **right justified**. **If you add a hyphen (-) after the %, then any extra blank space is placed after the value output and the output is said to be left justified**. For example, `%8.2f` is right justified and `%-8.2f` is left justified.

`System.out.printf` can have any number of arguments. The first argument is always a format string for the remaining arguments. All the arguments except the first are values to be output and these values are output in the specified formats. The format string can contain text as well as format specifiers, and this text is output along with the values, as seen below:

1	<code>public class ConsoleOutputMultipleArguments {</code>	PROGRAM 2-7
2	<code> public static void main(String[] args) {</code>	
3	<code> int age = 15;</code>	
4	<code> double amount = 253.48;</code>	
5	<code> String name = "Jack";</code>	
6	<code> char letter = 'Z';</code>	
7	<code> System.out.printf("%s is %d years old. He has \$%6.2f in savings "</code>	
8	<code> + "and he likes the letter %c.", name, age, amount, letter);</code>	
9	<code> }</code>	
10	<code>}</code>	

PROGRAM OUTPUT

Jack is 15 years old. He has \$253.48 in savings and he likes the letter Z.

```

1 public class ConsoleOutputFormatting{
2     public static void main(String[] args) {
3         String aString = "abc";
4         System.out.println("String output:");
5         System.out.println("*1234567890");
6         System.out.printf("%s* %n", aString);
7         System.out.printf("%4s* %n", aString);
8         System.out.printf("%2s* %n", aString);
9         System.out.println();
10        char oneCharacter = 'Z';
11        System.out.println("Character output:");
12        System.out.println("*1234567890");
13        System.out.printf("%c* %n", oneCharacter);
14        System.out.printf("%4c* %n", oneCharacter);
15        System.out.println();
16        int number = 256;
17        System.out.println("Integer output:");
18        System.out.printf("%d* %n", number);
19        System.out.printf("%2d* %n", number);
20        System.out.printf("%6d* %n", number);
21        System.out.printf("%-6d* %n", number);
22        System.out.println();
23        double d = 12345.123456789;
24        System.out.println("Floating-point output:");
25        System.out.println("*1234567890");
26        System.out.printf("%f* %n", d);
27        System.out.printf("%.4f* %n", d);
28        System.out.printf("%.2f* %n", d);
29        System.out.printf("%12.4f* %n", d);
30        System.out.printf("%e* %n", d);
31        System.out.printf("%12.5e* %n", d);
32    }
33 }

```

PROGRAM 2-8**PROGRAM OUTPUT**

String output:

*1234567890

abc

* abc*

abc

Character output:

*1234567890

Z

* Z*

Integer output:

256

256

* 256*

*256 *

Floating-point output:

*1234567890

12345.123457

12345.1235

12345.12

* 12345.1235*

1.234512e+04

* 1.23451e+04*

PROGRAM 2-8 illustrates the formatting capabilities of `printf` using format specifiers. The value is always output. If the specified field width is too small, extra space is taken as shown in line 8.

2.16 Console Input

You can make your programs more flexible if you ask the program user for inputs rather than using fixed values. Reading input from the console enables the program to accept input from the user. In PROGRAM 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, which is neither convenient nor practical.

2.16.1 Scanner Class

The Scanner class is located in the `java.util` package. A package is simply a library of classes. You can use the Scanner class to create an object to read input from `System.in` (keyboard). **Creating a Scanner object is like setting up a pipeline for input to flow into your program.** To make the Scanner class available, you need to use the `import` declaration for the compiler locate the class. **Import declarations must appear before the first class declaration in the file.** Placing an import declaration inside or after a class declaration is a syntax error.

The following statement creates an object which handles the reading of input.

```
Scanner input = new Scanner(System.in); // create an input object
```

The syntax `Scanner input` declares that `input` is a variable whose type is `Scanner`. The syntax `new Scanner(System.in)` creates an object of the `Scanner` type. The whole statement creates a Scanner object and **assigns its reference (address)** 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.** PROGRAM 2-9 shows how the program reads a number from the keyboard and assigns the number to `radius`.

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

PROGRAM OUTPUT

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

Line 7 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 prompt the user when expecting input from the keyboard.** Line 8 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 assign it to `radius` and perform the calculation.

You have seen 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 4.2 to read other types of input such as the `byte`, `short`, `int`, `long`, and `float` type or text string.

Method	Description
<code>next()</code>	Returns the String value consisting of the next keyboard characters up to, but not including, the first delimiter character. The default delimiters are whitespace characters.
<code>nextLine()</code>	Reads the rest of the current keyboard input line and returns the characters read as a value of type String. Note that the line terminator '\n' is read and discarded; it is not included in the string returned
<code>nextBoolean()</code>	Returns the next value of type boolean that is typed on the keyboard. The values of true and false are entered as the strings "true" and "false". Any combination of upper- and/or lowercase letters is allowed in spelling "true" and "false"
<code>nextByte()</code>	Returns the next value of type byte that is typed on the keyboard
<code>nextShort()</code>	Returns the next value of type short that is typed on the keyboard.
<code>nextInt()</code>	Returns the next value of type int that is typed on the keyboard.
<code>nextLong()</code>	Returns the next value of type long that is typed on the keyboard
<code>nextFloat()</code>	Returns the next value of type float that is typed on the keyboard.
<code>nextDouble()</code>	Returns the next value of type double that is typed on the keyboard

Table 2.7 Methods for Scanner objects

1	<code>import java.util.Scanner;</code>	PROGRAM 2-10
2	<code>public class ScannerInput {</code>	
3	<code> public static void main(String[] args) {</code>	
4	<code> byte n1;</code>	
5	<code> int n2;</code>	
6	<code> Scanner scannerObject = new Scanner(System.in);</code>	
7	<code> System.out.println("Enter two whole numbers");</code>	
8	<code> System.out.print(" separated by one or more spaces:");</code>	
9	<code> n1 = scannerObject.nextByte();</code>	
10	<code> n2 = scannerObject.nextInt();</code>	
11	<code> System.out.println("You entered " + n1 + " and " + n2);</code>	
12	<code> System.out.println("Next enter two words:");</code>	
13	<code> String word1 = scannerObject.next();</code>	
14	<code> String word2 = scannerObject.next();</code>	
15	<code> System.out.println("You entered \"" +</code>	
16	<code>word1 + "\" and \"" + word2 + "\"");</code>	
17	<code> String junk = scannerObject.nextLine(); //To get rid of ' \n'</code>	
18	<code> System.out.println("Next enter a line of text:");</code>	
19	<code> String line = scannerObject.nextLine();</code>	
20	<code> System.out.println("You entered: \"" + line + "\"");</code>	
21	<code>}</code>	

PROGRAM OUTPUT

```

Enter two whole numbers separated by one or more spaces: 42 43
You entered 42 and 43
Next enter two words:
jelly beans
You entered "jelly" and "beans"
Next enter a line of text:
Java flavored jelly beans are my favorite.
You entered "Java flavored jelly beans are my favorite."

```

PROGRAM 2-10 also illustrates other Scanner methods for reading values from the keyboard. The method `nextByte()` and `nextInt()` works in exactly the same way as `nextDouble()`, except that it reads a value of type `byte` and `int` respectively.

Line 13 and 14, the method `next()` reads in a word.

```

String word1 = scannerObject.next();
String word2 = scannerObject.next();

```

If the input line is `jelly beans` then this will assign `w1` the string `"jelly"` and `w2` the string `"beans"`. A word is any string of non-whitespace characters delimited by whitespace characters such as blanks or the beginning or ending of a line.

If you want to read in an entire line, you would use the method `nextLine()`. For example,

```
String line = scannerObject.nextLine();
```

reads in one line of input and places the string that is read into the variable `line`. **The end of an input line is indicated by the escape sequence `'\n'`.** This `'\n'` character is what you input when you press the Enter (Return) key on the keyboard. On the screen, it is indicated by the ending of one line and the beginning of the next line. When `nextLine()` reads a line of text, it reads this `'\n'` character, so the next reading of input begins on the next line. However, **the `'\n'` does not become part of the string value returned.** So, in the previous code, the string named by the variable `line` does not end with the `'\n'` character.

2.17 Exception Handling

Exception handling enables a program to deal with exceptional situations and continue its normal execution or gracefully exit. **Runtime errors occur while a program is running if the JVM detects an operation that is impossible to carry out.** If you enter a double value when your program expects an integer, you will get a runtime error with an `InputMismatchException`.

In Java, runtime errors are triggered (thrown) as exceptions. An exception is an object that represents an error or a condition that prevents execution from proceeding normally. If the exception is not handled, the program will terminate abnormally. When an exception is thrown, it can be caught and handled in a `try-catch` block, as follows:

```
try {
    statements; // Statements that may throw exceptions
}
catch (Exception1 exVar1) {
    handler for exception1;
}
catch (Exception2 exVar2) {
    handler for exception2;
}
...
catch (ExceptionN exVarN) {
    handler for exceptionN;
}
```

PROGRAM 2-11 illustrates how you can use exception handling to inform a user of the errors encountered by the program and thereafter allow it to terminate gracefully.

<pre> 1 import java.util.*; 2 public class ScannerInput { 3 public static void main(String[] args) { 4 Scanner input = new Scanner(System.in); 5 int result = 0; 6 try{ 7 System.out.print("Enter 2 numbers:"); 8 byte num1 = input.nextByte(); 9 int num2 = input.nextInt(); 10 if (num2 == 0) 11 throw new ArithmeticException("Divisor cannot be zero"); 12 result = num1/num2; 13 } 14 catch(InputMismatchException ex){ // handle type out of range 15 </pre>	PROGRAM 2-11
---	---------------------

```

16      System.out.println("The number enter is out of range of a byte"
17  +
18      "type");
19      System.out.println(ex);
20      System.out.println("Program Exited!");
21      System.exit(1);
22  }
23  catch(ArithmeticException ex){ // handle division by zero
24      System.out.println("Cannot divide by zero");
25      System.out.println(ex);
26      System.out.println("Program exited!");
27      System.exit(1);
28  }
29  System.out.println("The result of the division is " + result);
    }
}

```

PROGRAM OUTPUT 1

Enter 2 numbers: 120 4
The result of the division is 30

PROGRAM OUTPUT 2

Enter 2 numbers: 135
The number enter is out of range of a byte type
java.util.InputMismatchException: Value out of range. Value:"135" Radix:10
Program Exited!

PROGRAM OUTPUT 3

Enter 2 numbers: 100 0
Cannot divide by zero
java.lang.ArithmeticException: Divisor cannot be zero
Program exited!

In PROGRAM 2-11, there are two possible runtime errors that may occur; (1) Input out of range and (2) Division by zero. When a value that is not within the range of a byte is entered, it will trigger an `InputMismatchException`. This exception will be handled by the `catch` block in line 14 because an exception handler is defined to handle the error.

If a zero is entered for the second number, the program will trigger line 11 to throw an `ArithmeticException`. The flow of execution will jump to line 21, the `catch` block that was defined to handle the division by zero error.

If no exceptions arise during the execution of the `try` block, the `catch` blocks are skipped. If one of the statements inside the `try` block throws an exception, Java skips the remaining statements in the `try` block and starts the process of finding the code to handle the exception. The process of finding a handler is called catching an exception.

The code that handles the exception is called the exception handler. Each `catch` block is examined in turn, from first to last, to see whether the type of the exception object is an instance of the exception class in the `catch` block. If so, the exception object is assigned to the variable declared, and the code in the `catch` block is executed. If no handler is found, the program terminates and prints an error message on the console.

Note: Even if Line 10-11 is omitted, Java is still able to detect the type of exception and handle it accordingly. Other types of exceptions will be introduced in later chapters when appropriate.

2.18 Common Errors in Java Programming (Self-Reading)

Common Error 1: 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. Consider the following code:

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

This code is wrong, because `interestRate` is assigned a value 0.05; but `interestrate` has not been declared and initialized. Java is case sensitive, so it considers `interestRate` and `interestrate` to be two different variables. If a variable is declared but not used in the program, it might be a potential programming error. Hence, remove the unused variable from your program. 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 use an IDE such as IntelliJ or Eclipse, you will receive a warning on unused variables.

Common Error 2: Integer Overflow

Numbers are stored with a limited number 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, 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.

```
int value = 2147483647 + 1; // value will actually be -2147483648
```

Likewise, executing the following causes overflow, because the smallest value that can be stored in a variable of the `int` type is -2147483648. The magnitude of -2147483649 is too large to be stored in an `int` variable.

```
int value = -2147483648 - 1; // value will actually be 2147483647
```

Java does not report warnings or errors on overflow, so be careful when working with numbers close to the maximum or minimum range of a given type. When a floating-point number is too small (i.e., too close to zero) it causes **underflow**. Java approximates it to zero, so you do not need to be concerned about underflow.

Common Error 3: Round-off Errors

A round-off error, also called a **rounding error**, is the difference between the calculated approximation of a number and its exact mathematical value. For example, $1/3$ is approximately 0.333 if you keep three decimal places. Since the number of digits that can be stored in a variable is limited, round-off errors are inevitable. Calculations involving floating-point numbers are approximated because these numbers are not stored with complete accuracy.

For example,

```
System.out.println(1.0 - 0.1 - 0.1 - 0.1 - 0.1 - 0.1);
```

displays 0.5000000000000001, not 0.5, and

```
System.out.println(1.0 - 0.9);
```

displays 0.09999999999999998, not 0.1. Integers are stored precisely. Therefore, calculations with integers yield a precise integer result.

Common Error 4: 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. For example, the code in (a) displays that average is 1 and the code in (b) displays that average is 1.5.

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

(a)

```
int number1 = 1;
int number2 = 2;
double average = (number1 + number2) / 2.0;
System.out.println(average);
```

(b)

Common Error 5: Dealing with the Line Terminator `\n`

The method `nextLine()` of the class `Scanner` reads the remainder of a line of text starting wherever the last keyboard reading left off. For example, consider the following code:

```
Scanner keyboard = new Scanner(System.in);
int n = keyboard.nextInt();
String s1 = keyboard.nextLine();
String s2 = keyboard.nextLine();
```

Now, assume that the input typed on the keyboard is the following:

```
2 heads are
better than
1 head.
```

This sets the value of the variable `n` to 2, the variable `s1` to "heads are", and the variable `s2` to "better than". So far there are no problems, but suppose the inputs were instead as follows:

```
2
heads are better than
1 head.
```

You might expect the value of `n` to be set to 2, the value of the variable `s1` to "heads are better than", and the variable `s2` to "1 head". However, this does not happen. What happens is that the value of the variable `n` is set to 2, the variable `s1` is set to the empty string, and the variable `s2` to "heads are better than". The method `nextInt()` reads the 2 but does not read the end-of-line character `\n`. So the first `nextLine()` invocation reads the rest of the line that contains 2.

There is nothing more on that line (except for `\n`), so `nextLine()` returns the empty string. The second invocation of `nextLine()` begins on the next line and reads "heads are better than". When combining different methods for reading from the keyboard, you sometimes have to include an extra invocation of `nextLine()` to get rid of the end of a line (to get rid of a `\n`).

[Reference]

- [1] Introduction to Java Programming Comprehensive Version 10th Ed, Daniel Liang, 2016.
- [2] Java How To Program 10th Ed, Paul Deitel, Harvey Deitel, 2016.
- [3] Introduction to JShell <https://docs.oracle.com/javase/9/jshell/introduction-jshell.htm>

[Self-Review Question]

1. Examine the following program, which reads in a temperature in Fahrenheit and converts it to Celsius. Fill in blanks.

```
import _____;  
  
class FahrenheitToCelsius {  
  
    public static void main(String[] args) {  
  
        Scanner in = _____;  
  
        System.out.println("Enter temperature in Fahrenheit");  
  
        double temp = _____;  
  
        temp = _____;  
  
        System.out.println("Temperature in Celsius = " _____);  
  
    }  
}
```

Appendix: Java Keywords, Operator Precedence Chart and ASCII Chart

1 Java Keywords

The following fifty keywords are reserved for use by the Java language:

<code>abstract</code>	<code>double</code>	<code>int</code>	<code>super</code>
<code>assert</code>	<code>else</code>	<code>interface</code>	<code>switch</code>
<code>boolean</code>	<code>enum</code>	<code>long</code>	<code>synchronized</code>
<code>break</code>	<code>extends</code>	<code>native</code>	<code>this</code>
<code>byte</code>	<code>final</code>	<code>new</code>	<code>throw</code>
<code>case</code>	<code>finally</code>	<code>package</code>	<code>throws</code>
<code>catch</code>	<code>float</code>	<code>private</code>	<code>transient</code>
<code>char</code>	<code>for</code>	<code>protected</code>	<code>try</code>
<code>class</code>	<code>goto</code>	<code>public</code>	<code>void</code>
<code>const</code>	<code>if</code>	<code>return</code>	<code>volatile</code>
<code>continue</code>	<code>implements</code>	<code>short</code>	<code>while</code>
<code>default</code>	<code>import</code>	<code>static</code>	
<code>do</code>	<code>instanceof</code>	<code>strictfp*</code>	

2 Operator Precedence Chart

The operators are shown in decreasing order of precedence from top to bottom. Operators in the same group have the same precedence, and their associativity is shown in the table.

<i>Operator</i>	<i>Name</i>	<i>Associativity</i>
<code>()</code>	Parentheses	Left to right
<code>()</code>	Function call	Left to right
<code>[]</code>	Array subscript	Left to right
<code>.</code>	Object member access	Left to right
<code>++</code>	Postincrement	Left to right
<code>--</code>	Postdecrement	Left to right
<code>++</code>	Preincrement	Right to left
<code>--</code>	Predecrement	Right to left
<code>+</code>	Unary plus	Right to left
<code>-</code>	Unary minus	Right to left
<code>!</code>	Unary logical negation	Right to left
<code>(type)</code>	Unary casting	Right to left
<code>new</code>	Creating object	Right to left
<code>*</code>	Multiplication	Left to right
<code>/</code>	Division	Left to right
<code>%</code>	Remainder	Left to right
<code>+</code>	Addition	Left to right
<code>-</code>	Subtraction	Left to right
<code><<</code>	Left shift	Left to right
<code>>></code>	Right shift with sign extension	Left to right
<code>>>></code>	Right shift with zero extension	Left to right
<code><</code>	Less than	Left to right
<code><=</code>	Less than or equal to	Left to right
<code>></code>	Greater than	Left to right
<code>>=</code>	Greater than or equal to	Left to right
<code>instanceof</code>	Checking object type	Left to right

Operator	Name	Associativity
==	Equal comparison	Left to right
!=	Not equal	Left to right
&	(Unconditional AND)	Left to right
^	(Exclusive OR)	Left to right
	(Unconditional OR)	Left to right
&&	Conditional AND	Left to right
	Conditional OR	Left to right
?:	Ternary condition	Right to left
=	Assignment	Right to left
+=	Addition assignment	Right to left
-=	Subtraction assignment	Right to left
*=	Multiplication assignment	Right to left
/=	Division assignment	Right to left
%=	Remainder assignment	Right to left

3 ASCII Table - Standard and Extended ASCII Table

The ASCII character set is a subset of the Unicode character set used by Java to represent characters from most of the world's languages.

Regular ASCII Chart (character codes 0 - 127)

000	(nul)	016	► (dle)	032	sp	048	0	064	Ø	080	P	096	`	112	p
001	☉ (soh)	017	◄ (dc1)	033	!	049	1	065	A	081	Q	097	a	113	q
002	⊙ (stx)	018	↑ (dc2)	034	"	050	2	066	B	082	R	098	b	114	r
003	♥ (etx)	019	!! (dc3)	035	#	051	3	067	C	083	S	099	c	115	s
004	✚ (eot)	020	¶ (dc4)	036	\$	052	4	068	D	084	T	100	d	116	t
005	♣ (enq)	021	§ (nak)	037	%	053	5	069	E	085	U	101	e	117	u
006	♠ (ack)	022	— (syn)	038	&	054	6	070	F	086	V	102	f	118	v
007	▪ (bel)	023	‡ (etb)	039	'	055	7	071	G	087	W	103	g	119	w
008	▣ (bs)	024	† (can)	040	(056	8	072	H	088	X	104	h	120	x
009	(tab)	025	‡ (em)	041)	057	9	073	I	089	Y	105	i	121	y
010	(lf)	026	(eof)	042	*	058	:	074	J	090	Z	106	j	122	z
011	♂ (vt)	027	← (esc)	043	+	059	;	075	K	091	[107	k	123	{
012	♀ (np)	028	L (fs)	044	,	060	<	076	L	092	\	108	l	124	
013	(cr)	029	↔ (gs)	045	-	061	=	077	M	093]	109	m	125	}
014	♠ (so)	030	▲ (rs)	046	.	062	>	078	N	094	^	110	n	126	~
015	♠ (si)	031	▼ (us)	047	/	063	?	079	O	095	_	111	o	127	ó

Extended ASCII Chart (character codes 128 - 255)

128	Ç	143	À	158	Ê	172	¼	186	¶	200	ℓ	214	ƒ	228	Σ	242	≥
129	ù	144	Á	159	ë	173	½	187	¶	201	ℓ	215	ƒ	229	σ	243	≤
130	é	145	â	160	á	174	«	188	¶	202	ℓ	216	ƒ	230	μ	244	∫
131	â	146	Æ	161	í	175	»	189	¶	203	ℓ	217	ƒ	231	τ	245	∫
132	ä	147	ô	162	ó	176	⋮	190	¶	204	ℓ	218	ƒ	232	Φ	246	÷
133	à	148	ö	163	ú	177	⋮	191	¶	205	=	219	█	233	©	247	≈
134	ã	149	ò	164	ñ	178	⋮	192	¶	206	¶	220	█	234	Ω	248	°
135	ç	150	û	165	Ñ	179	⋮	193	¶	207	¶	221	█	235	δ	249	•
136	ê	151	ù	166	²	180	¶	194	¶	208	¶	222	█	236	∞	250	•
137	ë	152	ÿ	167	°	181	¶	195	¶	209	¶	223	█	237	φ	251	√
138	è	153	Û	168	¿	182	¶	196	¶	210	¶	224	α	238	ε	252	π
139	í	154	Ü	169	¬	183	¶	197	¶	211	¶	225	ß	239	∩	253	²
140	î	155	ô	170	¬	184	¶	198	¶	212	¶	226	Γ	240	≡	254	■
141	ï	156	£	171	½	185	¶	199	¶	213	¶	227	π	241	±	255	
142	Ä	157	¥														

4 Java Quick Reference Guide

Class Declaration

```
public class CashRegister
{
    private int itemCount;
    private double totalPrice;
    public void addItem(double price)
    {
        itemCount++;
        totalPrice = totalPrice + price;
    }
    ...
}
```

Instance variables

Method

Selected Operators and Their Precedence

(See Appendix B for the complete list.)

[]	Array element access
++ -- !	Increment, decrement, Boolean <i>not</i>
* / %	Multiplication, division, remainder
+ -	Addition, subtraction
< <= > >=	Comparisons
== !=	Equal, not equal
&&	Boolean <i>and</i>
	Boolean <i>or</i>
=	Assignment

Conditional Statement

```
if (floor >= 13)
{
    actualFloor = floor - 1;
}
else if (floor >= 0)
{
    actualFloor = floor;
}
else
{
    System.out.println("Floor negative");
}
```

Condition

Executed when condition is true

Second condition (optional)

Executed when all conditions are false (optional)

Loop Statements

```
while (balance < TARGET)
{
    year++;
    balance = balance * (1 + rate / 100);
}

for (int i = 0; i < 10; i++)
{
    System.out.println(i);
}
```

Condition

Executed while condition is true

Initialization Condition Update

Variable and Constant Declarations

Type	Name	Initial value
int	cansPerPack	= 6;
final double	CAN_VOLUME	= 0.335;

Method Declaration

```
public static double cubeVolume(double sideLength)
{
    double volume = sideLength * sideLength * sideLength;
    return volume;
}
```

Modifiers

Return type

Parameter type and name

Exits method and returns result.

Mathematical Operations

Math.pow(x, y)	Raising to a power x^y
Math.sqrt(x)	Square root \sqrt{x}
Math.log10(x)	Decimal log $\log_{10}(x)$
Math.abs(x)	Absolute value $ x $
Math.sin(x)	Sine, cosine, tangent of x (x in radians)
Math.cos(x)	
Math.tan(x)	

String Operations

```
String s = "Hello";
int n = s.length(); // 5
char ch = s.charAt(1); // 'e'
String t = s.substring(1, 4); // "ell"
String u = s.toUpperCase(); // "HELLO"
if (u.equals("HELLO")) ... // Use equals, not ==
for (int i = 0; i < s.length(); i++)
{
    char ch = s.charAt(i);
    Process ch
}
```

```
do
{
    System.out.print("Enter a positive integer: ");
    input = in.nextInt();
}
while (input <= 0);

for (double value : values)
{
    sum = sum + value;
}
```

Loop body executed at least once

Set to a new element in each iteration

An array or collection

Executed for each element

Input

```
Scanner in = new Scanner(System.in);
// Can also use new Scanner(new File("input.txt"));

int n = in.nextInt();
double x = in.nextDouble();
String word = in.next();
String line = in.nextLine();

while (in.hasNextDouble())
{
    double x = in.nextDouble();
    Process x
}
```

Output

Does not advance to new line.

```
System.out.print("Enter a value: ");
```

Use + to concatenate values.

```
System.out.println("Volume: " + volume);
```

Field width Precision

```
System.out.printf("%-10s %10d %10.2f", name, qty, price);
```

Left-justified string Integer Floating-point number

```
try (PrintWriter out = new PrintWriter("output.txt"))
{
    Write to out
}
// Use the print/println/printf methods.
```

The output is closed at the end of the try-with-resources statement.

Arrays

Element type Element type Length All elements are zero.

```
int[] numbers = new int[5];
int[] squares = { 0, 1, 4, 9, 16 };
int[][] magicSquare =
{
    { 16, 3, 2, 13},
    { 5, 10, 11, 8},
    { 9, 6, 7, 12},
    { 4, 15, 14, 1}
};

for (int i = 0; i < numbers.length; i++)
{
    numbers[i] = i * i;
}

for (int element : numbers)
{
    Process element
}

System.out.println(Arrays.toString(numbers));
// Prints [0, 1, 4, 9, 16]
```

Array Lists

Use wrapper type, Integer, Double, etc., for primitive types. Initially empty Element type (optional)

```
ArrayList<String> names = new ArrayList<String>();

names.add("Ann");
names.add("Cindy"); // [Ann, Cindy], names.size() is now 2

names.add(1, "Bob"); // [Ann, Bob, Cindy]
names.remove(2); // [Ann, Bob]
names.set(1, "Bill"); // [Ann, Bill]

String name = names.get(0); // Gets "Ann"
System.out.println(names); // Prints [Ann, Bill]
```

Linked Lists, Sets, and Iterators

```
LinkedList<String> names = new LinkedList<>();
names.add("Bob"); // Adds at end

ListIterator<String> iter = names.listIterator();
iter.add("Ann"); // Adds before current position

String name = iter.next(); // Returns "Ann"
iter.remove(); // Removes "Ann"

Set<String> names = new HashSet<>();
names.add("Ann"); // Adds to set if not present
names.remove("Bob"); // Removes if present

Iterator<String> iter = names.iterator();
while (iter.hasNext())
{
    Process iter.next()
}
```

Maps

Key type Value type Returns null if key not present

```
Map<String, Integer> scores = new HashMap<>();

scores.put("Bob", 10);
Integer score = scores.get("Bob");

for (String key : scores.keySet())
{
    Process key and scores.get(key)
}
```