

Name: \_\_\_\_\_ ( ) Date: \_\_\_\_\_

### Chapter 2: Elementary 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*

#### 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 simple 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:  
     $\text{area} = \text{radius} * \text{radius} * \pi$
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.

<pre>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 }</pre>	<b>PROGRAM 2-1</b>
---	--------------------

#### 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 (`System.out.println`). **Standard output will be further discussed in Chapter 3.**

## 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.

1 2 3 4 5 6 7 8 9 10 11 12 13 14	<pre>public class ComputeAreaWithConstant {     public static void main(String[] args) {         final double PI = 3.14159; // Declare a constant         double radius; // Declare radius         double area; // Declare area         // Assign a radius         radius = 20; // radius is now 20         // Compute area         area = radius * radius * PI;         // Display results         System.out.println("The area for the circle of radius " +             radius + " is " + area);     } }</pre>	<b>PROGRAM 2-2</b>
---	--	--------------------

### 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: $-3.4028235 \times 10^{+38}$ to $-1.4 \times 10^{-45}$ Positive range: $1.4 \times 10^{-45}$ to $3.4028235 \times 10^{+38}$	32-bit
double	Negative range: $-1.7976931348623157 \times 10^{+308}$ to $-4.9 \times 10^{-324}$ Positive range: $4.9 \times 10^{-324}$ to $1.7976931348623157 \times 10^{+308}$	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  $a \times 10^b$ .** For example, the scientific notation for 123.456 is  $1.23456 \times 10^2$  and for 0.0123456 is  $1.23456 \times 10^{-2}$ . A special syntax is used to write scientific notation numbers. For example,  $1.23456 \times 10^2$  is written as 1.23456E2 or 1.23456E+2 and  $1.23456 \times 10^{-2}$  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  $5.0534 \times 10^1$ , 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>	<b>PROGRAM 2-3</b>
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 " +</code>	
5	<code>(char)88));</code>	
6	<code>        System.out.println("ASCII code of character B is " + (int) char1);</code>	
7	<code>        System.out.println('\u2122');</code>	
8	<code>        System.out.println('\u005B' + " " + '\u005D' );</code>	
9	<code>    }</code>	
	<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.

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

### 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. These are handy because that's often how much the value needs to be changed in many programming tasks. For example, the following code increments *i* by 1 and decrements *j* by 1.

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

*i++* is pronounced as *i* plus plus and *i--* as *i* minus minus. These operators are known as **postfix increment** (or post-increment) and **postfix decrement** (or post-decrement), because the operators ++ and -- are placed after the variable. These operators can also be placed before the variable. For example,

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

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

Operator	Name	Description	Example (Let initial value of <i>i</i> be 1)
<b>++var</b>	preincrement	Increment var by 1, and use the new var value in the statement	<code>int j = ++i;</code> <code>// j is 2, i is 2</code>
<b>var++</b>	postincrement	Increment var by 1, but use the original var value in the statement	<code>int j = i++;</code> <code>// j is 1, i is 2</code>
<b>--var</b>	predecrement	Decrement var by 1, but use the new var value in the statement	<code>int j = --i;</code> <code>// j is 0, i is 0</code>
<b>var--</b>	postdecrement	Decrement var by 1, and use the original var value in the statement	<code>int j = i--;</code> <code>// j is 0, i is -1</code>

**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

In this case, *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 of i is used in the multiplication. Thus 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 a type is to specify the target type in parentheses, followed by the variable's name or the value to be cast.

For example, the following statement `System.out.println((int) 1.7);` displays 1. **When a double value is cast into an int value, the fractional part is truncated.** The following statement

```
System.out.println((double) 1 / 2);
```

displays 0.5, because 1 is cast to 1.0 first, then 1.0 is divided by 2. However, the statement

```
System.out.println(1 / 2);
```

displays 0, because 1 and 2 are both integers and the resulting value should also be an integer. **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 += 4.5` is equivalent to `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 is much similar to functions in 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>	<b>PROGRAM 2-5</b>
---	--------------------

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 modularize as follows:

1	public class FahrenheitToCelsius {	<b>PROGRAM 2-6</b>
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	}	

How would you write a similar method to convert Celsius to Fahrenheit?

## 2.15 Common Errors (Self-Reading)

Common elementary programming errors often involve undeclared variables, uninitialized variables, integer overflow, unintended integer division, and round-off errors.

### 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. So, you should 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 Eclipse and NetBeans, you will receive a warning on unused variables.

### Common Error 2: 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, 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 statement causes overflow, because the smallest value that can be stored in a variable of the `int` type is -2147483648. -2147483649 is too large in size 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) to be stored, it causes **underflow**. Java approximates it to zero, so normally 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, and is 0.3333333 if you keep seven 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)

## 2.16 JShell (Self-Reading)

JShell is a new feature in Java from JDK 9 onwards. The Java Shell tool (JShell) is an interactive tool for learning the Java programming language and prototyping Java code (it works much like IDLE shell in Python). JShell is a Read-Evaluate-Print Loop (REPL), which evaluates declarations, statements, and expressions as they are entered and immediately shows the results. The tool is run from the command line.

Using JShell, you can enter program elements one at a time, immediately see the result, and make adjustments as needed.

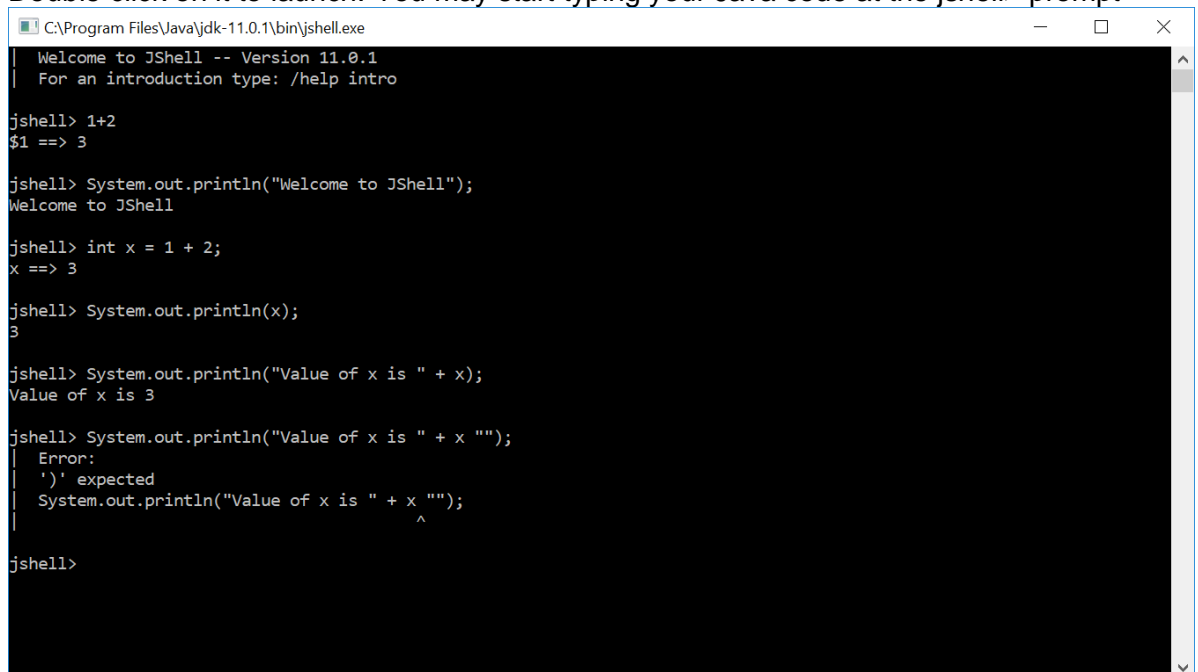
Java program development typically involves the following process:

- Write a complete program.
- Compile it and fix any errors.
- Run the program.
- Figure out what is wrong with it.
- Edit it.
- Repeat the process.

JShell helps you try out code and easily explore options as you develop your program. You can test individual statements, try out different variations of a method, and experiment with unfamiliar APIs within the JShell session. JShell doesn't replace an IDE. As you develop your program, paste code into JShell to try it out, and then paste working code from JShell into your program editor or IDE.

### Running JShell in command prompt

1. To start your JShell in command prompt, go to the bin directory of your JDK 11.  
For example: C:\Program Files\Java\jdk-11.0.1\bin
2. Locate the jshell Application.
3. Double click on it to launch. You may start typing your Java code at the jshell> prompt



```
C:\Program Files\Java\jdk-11.0.1\bin\jshell.exe
Welcome to JShell -- Version 11.0.1
For an introduction type: /help intro

jshell> 1+2
$1 ==> 3

jshell> System.out.println("Welcome to JShell");
Welcome to JShell

jshell> int x = 1 + 2;
x ==> 3

jshell> System.out.println(x);
3

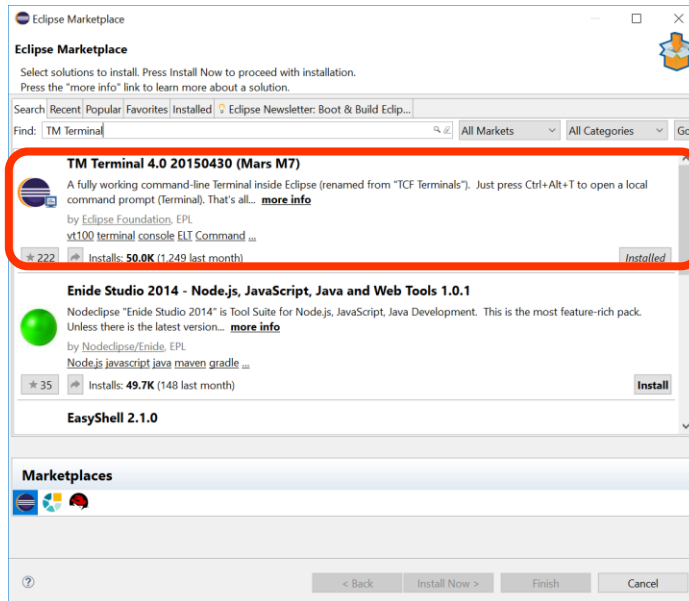
jshell> System.out.println("Value of x is " + x);
Value of x is 3

jshell> System.out.println("Value of x is " + x "");
Error:
  ')' expected
  System.out.println("Value of x is " + x "");
                        ^
jshell>
```

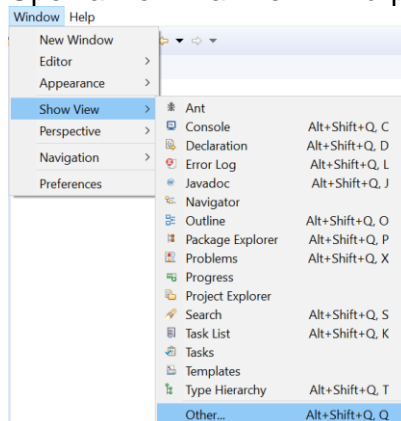
## Running JShell from Eclipse

You can use the TM Terminal to run JShell in Eclipse.

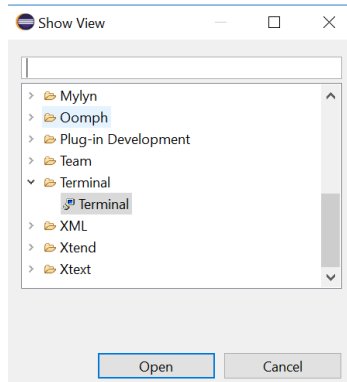
1. Go to Help > Eclipse Marketplace.
2. Search for TM Terminal and install.



3. Relaunch Eclipse after installation.
4. Open a 'Terminal' view in Eclipse. Go to Window > Show View > Other...

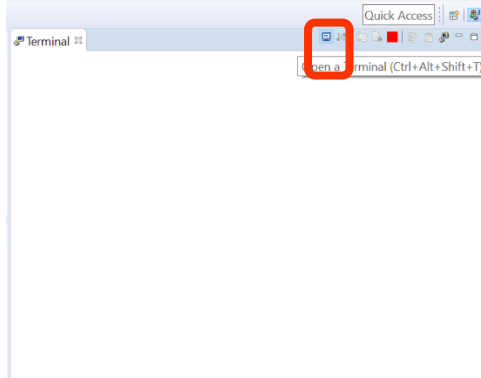


5. Select Terminal > Terminal





## 6. Launch a new Local Terminal



## 7. Run JShell, e. g. on Windows type "C:\Program Files\Java\jdk-11.0.1\bin\jshell" followed by Enter. Once you see the prompt jshell&gt; , you have successfully started jshell. You may proceed to type your Java code.

A screenshot of a Windows terminal window. The title bar shows 'Help', 'Welcome', and 'Terminal'. The terminal content shows the following commands and output:

```
NHSNSMP1DPNFP
Microsoft Windows [Version 10.0.17134.407]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\nhscsp>cd ..

C:\Users>cd ..

C:\>cd C:\Program Files\Java\jdk-11.0.1\bin

C:\Program Files\Java\jdk-11.0.1\bin>jshell
| Welcome to JShell -- Version 11.0.1
| For an introduction type: /help intro

jshell> 1+3
$1 ==> 4

jshell> int x = 1+3;
x ==> 4

jshell> System.out.println(x);
4

jshell> 
```

**[Reference]**

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