Java Programming Tutorial

Introduction to Java Programming (for Novices & First-Time Programmers)

TABLE OF CONTENTS (HIDE)

1. Getting Started - Write your First Hello-world Java Program

You should have already installed Java Development Kit (JDK). Otherwise, Read "How to Install JDK and Get Started with Java Programming".

Let us begin by writing our first Java program that prints a message "Hello, world!" to the display console, as follows:

```
Hello, world!
```

Step 1: Write the Source Code: Enter the following source codes using a programming text editor (such as TextPad or NotePad++ for Windows or gedit for UNIX/Lunix/Mac) or an Interactive Development Environment (IDE) (such as Eclipse or Netbeans - Read the respective "How-To" article on how to install and get started with these IDEs).

Do not enter the line numbers (on the left panel), which were added to help in the explanation. Save the source file as "Hello.java". A Java source file should be saved with a file extension of ".java". The filename shall be the same as the classname - in this case "Hello".

```
1  /*
2   * First Java program, which says "Hello, world!"
3   */
4  public class Hello {   // Save as "Hello.java"
5     public static void main(String[] args) {
6        System.out.println("Hello, world!");   // print message
7     }
8  }
```

Step 2: Compile the Source Code: Compile the source code "Hello.java" into portable bytecode "Hello.class" using JDK compiler "javac". Start a CMD Shell (Windows) or Terminal (UNIX/Lunix/Mac) and issue this command:

```
prompt> javac Hello.java
```

where javac is the name of JDK compiler.

There is no need to explicitly compile the source code under IDEs (such as Eclipse or NetBeans), as they perform incremental compilation implicitly.

Step 3: Run the Program: Run the program using Java Runtime "java", by issuing this command:

```
prompt> java Hello
Hello, world!
```

On IDEs (such as Eclipse or NetBeans), right-click on the source file and choose "Run As..." ⇒ "Java Application".

Brief Explanation of the Program

```
/* ..... */
// ... until the end of the line
```

These are called *comments*. Comments are NOT executable and are ignored by the compiler. But they provide useful explanation and documentation to your readers (and to yourself three days later). There are two kinds of comments:

1. *Multi-line Comment*: begins with /* and ends with */, and may span more than one lines (as in Lines 1-3).
2. *End-of-line Comment*: begins with // and lasts until the end of the current line (as in Lines 4 and 6).

```
public class Hello { ..... }
```

The basic unit of a Java program is a *class*. A class called "Hello" is defined via the keyword "class" in Lines 4-8. The {...} is the body of the class. The keyword public will be discussed later.

In Java, the name of the source file must be the same as the name of the public class with a mandatory file extension of ".java". Hence, this file

MUST be saved as "`Hello.java`".

`public static void main(String[] args) { ..... }`

Lines 5-7 defines the so-called `main()` *method*, which is the starting point, or *entry point* for program execution. The `{...}` is the body of the method which contains programming statements.

`System.out.println("Hello, world!");`

In Line 6, the statement `System.out.println("Hello, world!")` is used to print the message string "Hello, world!" to the display console. A *string* is surrounded by a pair of double quotes and contain texts. The text will be printed as it is, without the double quotes.

2. Java Terminology and Syntax

Statement: A programming *statement* performs a piece of programming action. It must be terminated by a semi-colon (`;`) (just like an English sentence is ended with a period) as in Lines 6.

Block: A *block* is a group of programming statements enclosed by braces `{ }`. This group of statements is treated as one single unit. There are two blocks in this program. One contains the *body* of the class `Hello`. The other contains the *body* of the `main()` method. There is no need to put a semi-colon after the closing brace.

Comments: A multi-line comment begins with `/*` and ends with `*/`. An end-of-line comment begins with `//` and lasts till the end of the line. Comments are NOT executable statements and are ignored by the compiler. But they provide useful explanation and documentation. *Use comments liberally*.

Whitespaces: Blank, tab, and newline are collectively called *whitespace*. Extra whitespaces are ignored, i.e., only one whitespace is needed to separate the tokens. But they could help you and your readers better understand your program. *Use extra whitespaces liberally*.

Case Sensitivity: Java is *case sensitive* - a *ROSE* is NOT a *Rose*, and is NOT a *rose*. The *filename* is also case-sensitive.

3. Java Program Template

You can use the following *template* to write your Java programs. Choose a meaningful "*Classname*" that reflects the *purpose* of your program, and write your programming statements inside the body of the `main()` method. Don't worry about the other terms and keywords now. I will explain them in due course.

```
1  public class Classname {   // Choose a meaningful Classname. Save as "Classname.java"
2     public static void main(String[] args) {
3        // Your programming statements here!
4     }
5  }
```

4. Printing via `System.out.println()` and `System.out.print()`

You can use `System.out.println()` or `System.out.print()` to print message to the display console:

- `System.out.println(aString)` (Print-Line) prints the given *aString*, and brings the *cursor* to the beginning of the next line.
- `System.out.print(aString)` prints *aString* but places the cursor after the printed string.

Try the following program and explain the output produced:

```
1  /* Test System.out.println() and System.out.print() */
2  public class PrintTest {   // Save as "PrintTest.java"
3     public static void main(String[] args) {
4        System.out.println("Hello, world!"); // Advance the cursor to the beginning of next line after printing
5        System.out.println();               // Print a empty line
6        System.out.print("Hello, world!");   // Cursor stayed after the printed string
7        System.out.println("Hello,");
8        System.out.print(" ");              // Print a space
9        System.out.print("world!");
10       System.out.println("Hello, world!");
11    }
12 }
```

```
Hello, world!

Hello, world!Hello,
 world!Hello, world!
```

Exercises

1. Print each of the following patterns. Use one `System.out.println(...)` statement for each line of outputs.

```
* * * * *      * * * * *    * * * * *
 * * * * *     *       *     *     *
* * * * *      *       *      *   *
```

```
 * * * * *      *        *        * *
* * * * *       * * * * *          *
   (a)             (b)             (c)
```

5. Let's Write a Program to Add a Few Numbers

Let us write a program to add five integers as follows:

```
1  /*
2   * Sum five numbers and print the result
3   */
4  public class FiveNumberSum {   // Save as "FiveNumberSum.java"
5     public static void main(String[] args) {
6        int number1 = 11;  // Declare 5 int variables to hold 5 integers
7        int number2 = 22;
8        int number3 = 33;
9        int number4 = 44;
10       int number5 = 55;
11       int sum;           // Declare an int variable called sum to hold the sum
12       sum = number1 + number2 + number3 + number4 + number5;
13       System.out.print("The sum is ");    // Print a descriptive string
14       System.out.println(sum);           // Print the value stored in sum
15    }
16 }
```

```
The sum is 165
```

```
int number1 = 11;
int number2 = 22;
int number3 = 33;
int number4 = 44;
int number5 = 55;
```

declare five `int` (integer) *variables* called `number1`, `number2`, `number3`, `number4`, and `number5`; and *assign* values of 11, 22, 33, 44 and 55 to the variables, respectively, via the so-called *assignment operator* `'='`. You could also declare many variables in one statement, separating with commas, e.g.,

```
int number1 = 11, number2 = 22, number3 = 33, number4 = 44, number5 = 55;
```

```
int sum;
```

declares an `int` (integer) variable called `sum`, without assigning an initial value.

```
sum = number1 + number2 + number3 + number4 + number5;
```

computes the sum of `number1` to `number5` and assign the result to the variable `sum`. The symbol '+' denotes *arithmetic addition*, just like Mathematics.

```
System.out.print("The sum is ");
System.out.println(sum);
```

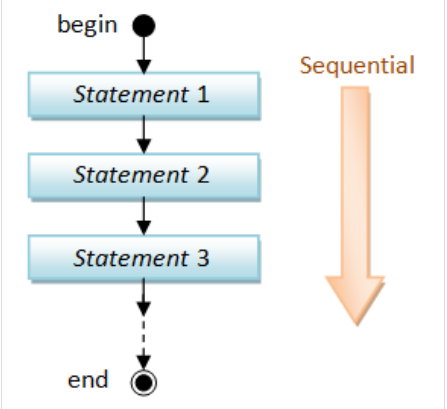
Line 13 prints a descriptive string. A `String` is surrounded by double quotes, and will be printed *as it is* (but without the double quotes). Line 14 prints the *value* stored in the variable `sum` (in this case, the sum of the five numbers). You should not surround a variable to be printed by double quotes; otherwise, the text will get printed instead of the value stored in the variable.

Exercise

1. Modify the above program to print the product of 5 integers. Use a variable called `product` to hold the product, and `*` for multiplication.

6. What is a Program?

A *program* is *a sequence of instructions* (called *programming statements*), executing one after another - usually in a *sequential* manner, as illustrated in the following flow chart.



Example

The following program prints the area and perimeter of a circle, given its radius. Take note that the programming statements are executed sequentially - one after another in the order that they are written.

```
1  /*
2   * Print the area and circumference of a circle, given its radius.
```

```
 */
public class CircleComputation {  // Saved as "CircleComputation.java"
   public static void main(String[] args) {
      // Declare variables
      double radius, area, circumference;
      final double PI = 3.14159265;

      // Assign a value to radius
      radius = 1.2;

      // Compute area and circumference
      area = radius * radius * PI;
      circumference = 2.0 * radius * PI;

      // Print results
      System.out.print("The radius is ");  // Print description
      System.out.println(radius);          // Print the value stored in the variable
      System.out.print("The area is ");
      System.out.println(area);
      System.out.print("The circumference is ");
      System.out.println(circumference);
   }
}
```

```
The radius is 1.2
The area is 4.523893416
The circumference is 7.5398223600000005
```

`double radius, area, circumference;`

declare three `double` variables `radius`, `area` and `circumference`. A `double` variable can hold real numbers (or floating-point numbers, with an optional fractional part).

`final double PI = 3.14159265;`

declare a `double` variables called `PI` and assign a value. `PI` is declared `final`, i.e., its value cannot be changed.

`radius = 1.2;`

assigns a value to the variable `radius`.

```
area = radius * radius * PI;
circumference = 2.0 * radius * PI;
```

compute the `area` and `circumference`, based on the `radius`.

```
System.out.print("The radius is ");
System.out.println(radius);
System.out.print("The area is ");
System.out.println(area);
System.out.print("The circumference is ");
System.out.println(circumference);
```

print the results with proper descriptions.

Take note that the programming statements inside the `main()` are executed one after another, in a *sequential* manner.

Exercises

1. Follow the above example, write a program (called `RectangleComputation`) to print the area and perimeter of a rectangle, given its length and width in `double`s.
2. Follow the above example, write a program (called `CylinderComputation`) to print the surface area and volume of a cylinder, given its radius and height in `double`s.

7. What is a Variable?

Computer programs manipulate (or process) data. A *variable* is used to store a piece of data for processing. It is called *variable* because you can change the value stored.

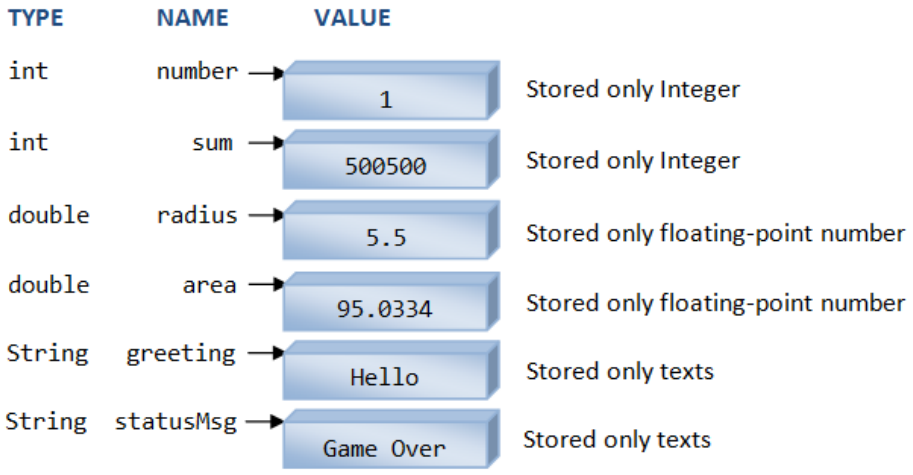
More precisely, a *variable* is a *named* storage location, that stores a *value* of a particular data *type*. In other words, a *variable* has a *name*, a *type* and stores a *value* of that type.

- A variable has a *name* (or *identifier*), e.g., `radius`, `area`, `age`, `height`. The name is needed to uniquely identify each variable, so as to assign a value to the variable (e.g., `radius=1.2`), and retrieve the value stored (e.g., `radius*radius*3.1416`).
- A variable has a *type*. Examples of *type* are:
  - `int`: for integers (whole numbers) such as `123` and `-456`;
  - `double`: for floating-point or real numbers, such as `3.1416`, `-55.66`, `7.8e9`, `-1.2e3.4` having an optional decimal point and fractional part, in fixed or scientific notations;

- `String`: for texts such as `"Hello"`, `"Good Morning!"`. Text strings are enclosed within a pair of double quotes.

- A variable can store a *value* of that particular *type*. It is important to take note that a variable in most programming languages is associated with a type, and can only store value of the particular type. For example, a `int` variable can store an integer value such as `123`, but NOT real number such as `12.34`, nor texts such as `"Hello"`. The concept of *type* was introduced into the early programming languages to simplify intrepretation of data.

The following diagram illustrates three types of variables: `int`, `double` and `String`. An `int` variable stores an integer (whole number). A `double` variable stores a real number. A `String` variable stores texts.



A *variable* has a ***name***, stores a ***value*** of the declared ***type***.

To use a variable, you need to first *declare* its *name* and *type*, in one of the following syntaxes:

```
varType varName;                 // Declare a variable of a type
varType varName1, varName2,...;  // Declare multiple variables of the same type
varType varName = initialValue;  // Declare a variable of a type, and assign an initial value
varType varName1 = initialValue1, varName2 = initialValue2,... ;  // Declare variables with initial values
```

Take note that:

- Each *declaration statement* is terminated with a semi-colon (`;`).
- In multiple-variable declaration, the names are separated by commas (`,`).
- The symbol `=`, known as the *assignment operator*, can be used to assign an initial value (of the declared type) to the variable.

For example,

```
int sum;              // Declare a variable named "sum" of the type "int" for storing an integer.
                      // Terminate the statement with a semi-colon.
int number1, number2; // Declare 2 "int" variables named "number1" and "number2", separated by a comma.
double average;       // Declare a variable named "average" of the type "double" for storing a real number.
int height = 20;      // Declare an "int" variable, and assign an initial value.
```

Once a variable is declared, you can *assign* and *re-assign* a value to the variable, via the so-called *assignment operator* `'='`. For example,

```
int number;           // Declare a variable named "number" of the type "int" (integer).
number = 99;          // Assign an integer value of 99 to the variable "number".
number = 88;          // Re-assign a value of 88 to "number".
number = number + 1;  // Evaluate "number + 1", and assign the result back to "number".
int sum = 0;          // Declare an int variable named "sum" and assign an initial value of 0.
sum = sum + number;   // Evaluate "sum + number", and assign the result back to "sum", i.e. add number into sum.
int num1 = 5, num2 = 6;  // Declare and initialize two int variables in one statement, separated by a comma.
double radius = 1.5;  // Declare a variable name "radius", and initialize to 1.5.
int number;           // ERROR: A variable named "number" has already been declared.
sum = 55.66;          // ERROR: The variable "sum" is an int. It cannot be assigned a floating-point number.
sum = "Hello";        // ERROR: The variable "sum" is an int. It cannot be assigned a text string.
```

Take note that:

- Each variable can only be declared once. (You cannot have two houses with the same address.)
- You can declare a variable anywhere inside the program, as long as it is declared before it is being used.
- Once the *type* of a variable is declared, it can only store a value of this particular *type*. For example, an `int` variable can hold only integer such as `123`, and NOT floating-point number such as `-2.17` or text string such as `"Hello"`.
- The *type* of a variable cannot be changed inside the program, once declared.

I have shown your two data types in the above example: `int` for integer and `double` for floating-point number (or real number). Take note that in programming, `int` and `double` are two *distinct* types and special caution must be taken when *mixing* them in an operation, which shall be explained later.

`x=x+1`?

Assignment (`=`) in programming is different from equality in Mathematics. e.g., "`x=x+1`" is invalid in Mathematics. However, in programming, it means

compute the value of `x` plus 1, and assign the result back to variable `x`.

"x+y=1" is valid in Mathematics, but is invalid in programming. In programming, the RHS of "=" has to be evaluted to a value; while the LHS shall be a variable. That is, evaluate the RHS first, then assign to LHS.

Some languages uses `:=` as the assignment operator to avoid confusion with equality.

8. Basic Arithmetic Operations

The basic *arithmetic operations* are:

Operator	Meaning	Example
`+`	Addition	`x + y`
`-`	Subtraction	`x - y`
`*`	Multiplication	`x * y`
`/`	Division	`x / y`
`%`	Modulus (Remainder)	`x % y`
`++`	Increment by 1 (Unary)	`++x` or `x++`
`--`	Decrement by 1 (Unary)	`--x` or `x--`

Addition, subtraction, multiplication, division and remainder are *binary operators* that take two operands (e.g., `x + y`); while negation (e.g., `-x`), increment and decrement (e.g., `x++`, `--x`) are *unary operators* that take only one operand.

Example

The following program illustrates these arithmetic operations:

```
/**
 * Test Arithmetic Operations
 */
public class ArithmeticTest {      // Save as "ArithmeticTest.java"
   public static void main(String[] args) {

      int number1 = 98;      // Declare an int variable number1 and initialize it to 98
      int number2 = 5;       // Declare an int variable number2 and initialize it to 5
      int sum, difference, product, quotient, remainder;  // Declare five int variables to hold results

      // Perform arithmetic Operations
      sum = number1 + number2;
      difference = number1 - number2;
      product = number1 * number2;
      quotient = number1 / number2;
      remainder = number1 % number2;

      // Print results
      System.out.print("The sum, difference, product, quotient and remainder of ");  // Print description
      System.out.print(number1);       // Print the value of the variable
      System.out.print(" and ");
      System.out.print(number2);
      System.out.print(" are ");
      System.out.print(sum);
      System.out.print(", ");
      System.out.print(difference);
      System.out.print(", ");
      System.out.print(product);
      System.out.print(", ");
      System.out.print(quotient);
      System.out.print(", and ");
      System.out.println(remainder);

      ++number1;  // Increment the value stored in the variable "number1" by 1
                  // Same as "number1 = number1 + 1"
      --number2;  // Decrement the value stored in the variable "number2" by 1
                  // Same as "number2 = number2 - 1"
      System.out.println("number1 after increment is " + number1);  // Print description and variable
      System.out.println("number2 after decrement is " + number2);
      quotient = number1 / number2;
      System.out.println("The new quotient of " + number1 + " and " + number2
            + " is " + quotient);
   }
}
```

```
The sum, difference, product, quotient and remainder of 98 and 5 are 103, 93, 490, 19, and 3
number1 after increment is 99
number2 after decrement is 4
The new quotient of 99 and 4 is 24
```

Disecting the Program

```
int number1 = 98;
int number2 = 5;
int sum, difference, product, quotient, remainder;
```

declare all the `int` (integer) variables `number1`, `number2`, `sum`, `difference`, `product`, `quotient`, and `remainder`, needed in this program.

```
sum = number1 + number2;
difference = number1 - number2;
product = number1 * number2;
quotient = number1 / number2;
remainder = number1 % number2;
```

carry out the arithmetic operations on `number1` and `number2`. Take note that division of two integers produces a *truncated* integer, e.g., `98/5 → 19`, `99/4 → 24`, and `1/2 → 0`.

```
System.out.print("The sum, difference, product, quotient and remainder of ");
......
```

prints the results of the arithmetic operations, with the appropriate string descriptions in between. Take note that text strings are enclosed within double-quotes, and will get printed as they are, including the white spaces (but without the double quotes). To print the *value* stored in a variable, no double quotes should be used. For example,

```
System.out.println("sum");    // Print text string "sum" - as it is
System.out.println(sum);      // Print the value stored in variable sum, e.g., 98
```

```
++number1;
--number2;
```

illustrate the increment and decrement operations. Unlike `'+'`, `'-'`, `'*'`, `'/'` and `'%'`, which work on two operands (*binary operators*), `'++'` and `'--'` operate on only one operand (*unary operators*). `++x` is equivalent to `x = x + 1`, i.e., increment x by 1. You may place the increment operator before or after the operand, i.e., `++x` (pre-increment) or `x++` (post-increment). In this example, the effects of pre-increment and post-increment are the same. I shall point out the differences in later section.

```
System.out.println("number1 after increment is " + number1);
System.out.println("number2 after decrement is " + number2);
```

print the new values stored after the increment/decrement operations. Take note that instead of using many `print()` statements as in Lines 17-30, we could simply place all the items (text strings and variables) into one `println()`, with the items separated by `'+'`. In this case, `'+'` does not perform *addition*. Instead, it *concatenates* or *joins* all the items together.

Exercises

1. Combining Lines 19-32 into one single `println()` statement, using `'+'` to concatenate all the items together.
2. Introduce one more `int` variable called `number3`, and assign it an integer value of `77`. Compute and print the *sum* and *product* of all the three numbers.
3. In Mathematics, we could omit the multiplication sign in an arithmetic expression, e.g., `x = 5a + 4b`. In programming, you need to explicitly provide all the operators, i.e., `x = 5*a + 4*b`. Try printing the sum of `31` times of `number1` and `17` times of `number2` and `87` time of `number3`.

9. What If Your Need To Add a Thousand Numbers? Use a Loop

Suppose that you want to add all the integers from `1` to `1000`. If you follow the previous example, you would require a thousand-line program! Instead, you could use a so-called *loop* in your program to perform a *repetitive* task, that is what the computer is good at.

Example

Try the following program, which sums all the integers from a lowerbound (=1) to an upperbound (=1000) using a so-called *while-loop*.

```
1  /*
2   * Sum from a lowerbound to an upperbound using a while-loop
3   */
4  public class RunningNumberSum {  // Save as "RunningNumberSum.java"
5     public static void main(String[] args) {
6        int lowerbound = 1;      // Store the lowerbound
7        int upperbound = 1000;   // Store the upperbound
8        int sum = 0;   // Declare an int variable "sum" to accumulate the numbers
9                       // Set the initial sum to 0
10       // Use a while-loop to repeatitively sum from the lowerbound to the upperbound
11       int number = lowerbound;
12       while (number <= upperbound) {
13          sum = sum + number;  // Accumulate number into sum
14          ++number;            // Next number
15       }
16       // Print the result
17       System.out.println("The sum from " + lowerbound + " to " + upperbound + " is " + sum);
```

```
    }
}
```

```
The sum from 1 to 1000 is 500500
```

Dissecting the Program

```
int lowerbound = 1;
int upperbound = 1000;
```

declare two `int` variables to hold the upperbound and lowerbound, respectively.

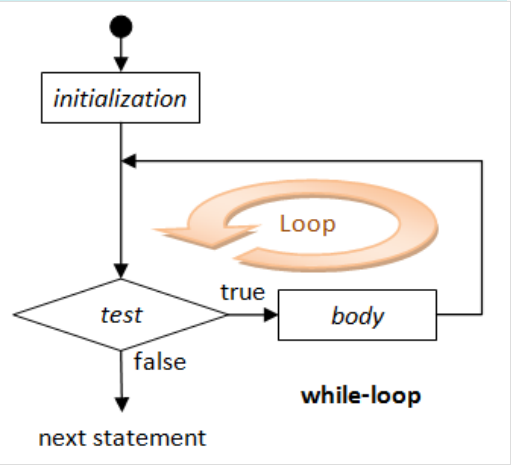
```
int sum = 0;
```

declares an `int` variable to hold the sum. This variable will be used to *accumulate* numbers over the steps in the repetitive loop, and thus initialized to 0.

```
int number = lowerbound;
while (number <= upperbound) {
   sum = sum + number;
   ++number;
}
```

This is the so-called *while-loop*. A *while-loop* takes the following syntax:

```
initialization-statement;
while (test) {
   loop-body;
}
next-statement;
```



As illustrated in the flow chart, the *initialization* statement is first executed. The *test* is then checked. If the *test* is true, the *body* is executed. The *test* is checked again and the process repeats until the *test* is false. When the *test* is false, the loop completes and program execution continues to the *next statement* after the loop.

In our program, the *initialization* statement declares an `int` variable named `number` and initializes it to `lowerbound`. The *test* checks if `number` is equal to or less than the `upperbound`. If it is true, the current value of `number` is added into the `sum`, and the statement `++number` increases the value of `number` by 1. The *test* is then checked again and the process repeats until the *test* is false (i.e., `number` increases to `upperbound+1`), which causes the loop to terminate. Execution then continues to the next statement (in Line 23).

In this example, the loop repeats `upperbound-lowerbound+1` times. After the loop is completed, Line 17 prints the result with a proper description.

```
System.out.println("The sum from " + lowerbound + " to " + upperbound + " is " + sum);
```

prints the results.

Exercises

1. Modify the above program to sum all the numbers from `9` to `888`. (*Ans*: `394680`.)
2. Modify the above program to sum all the *odd* numbers between `1` to `1000`. (*Hint*: Change the *post-processing* statement to "`number = number + 2`". *Ans*: `250000`)
3. Modify the above program to sum all the numbers between `1` to `1000` that are divisible by `7`. (*Hint*: Modify the initialization and post-processing statements. *Ans*: `71071`.)
4. Modify the above program to find the sum of the *square* of all the numbers from `1` to `100`, i.e. `1*1 + 2*2 + 3*3 +...` (*Ans*: `338350`.)
5. Modify the above program (called `RunningNumberProduct`) to compute the *product* of all the numbers from `1` to `10`. (*Hint*: Use a variable called `product` instead of `sum` and initialize `product` to 1. *Ans*: `3628800`.)

10. Conditional (or Decision)

What if you want to sum all the odd numbers and also all the even numbers between `1` and `1000`? There are many ways to do this. You could declare two variables: `sumOdd` and `sumEven`. You can then use a *conditional statement* to check whether the number is odd or even, and accumulate the number into the respective sums. The program is as follows:

```
/*
 * Sum the odd numbers and the even numbers from a lowerbound to an upperbound
 */
public class OddEvenSum {  // Save as "OddEvenSum.java"
   public static void main(String[] args) {
      int lowerbound = 1, upperbound = 1000;  // lowerbound and upperbound
      int sumOdd  = 0;    // For accumulating odd numbers, init to 0
      int sumEven = 0;    // For accumulating even numbers, init to 0
      int number = lowerbound;
      while (number <= upperbound) {
```

```
11            if (number % 2 == 0) {   // Even
12               sumEven += number;    // Same as sumEven = sumEven + number
13            } else {                 // Odd
14               sumOdd += number;     // Same as sumOdd = sumOdd + number
15            }
16            ++number;  // Next number
17         }
18         // Print the result
19         System.out.println("The sum of odd numbers from " + lowerbound + " to " + upperbound + " is " + sumOdd);
20         System.out.println("The sum of even numbers from " + lowerbound + " to " + upperbound + "  is " + sumEven);
21         System.out.println("The difference between the two sums is " + (sumOdd - sumEven));
23      }
24   }
```

```
The sum of odd numbers from 1 to 1000 is 250000
The sum of even numbers from 1 to 1000  is 250500
The difference between the two sums is -500
```

Dissecting the Program

`int lowerbound = 1, upperbound = 1000;`

declares the upperbound and lowerbound for the loop.

`int sumOdd = 0;`

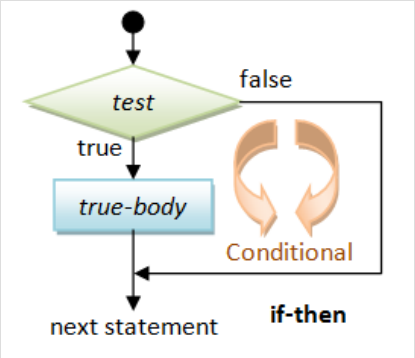
`int sumEven = 0;`

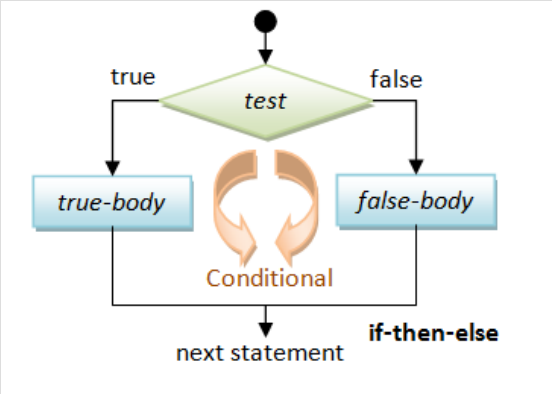
declare two `int` variables named `sumOdd` and `sumEven` and initialize them to 0, for accumulating the odd and even numbers, respectively.

```
if (number % 2 == 0) {
   sumEven = sumEven + number;
} else {
   sumOdd = sumOdd + number;
}
```

This is a *conditional statement*. The conditional statement can take one these forms: *if-then* or *if-then-else*.

```
// if-then
if ( test ) {
   true-body;
}
// if-then-else
if ( test ) {
   true-body;
} else {
   false-body;
}
```





For a *if-then* statement, the *true-body* is executed if the *test* is true. Otherwise, nothing is done and the execution continues to the next statement. For a *if-then-else* statement, the *true-body* is executed if the *test* is true; otherwise, the *false-body* is executed. Execution is then continued to the next statement.

In our program, we use the *remainder operator* `(%)` to compute the remainder of `number` divides by `2`. We then compare the remainder with `0` to test for even number.

Comparison Operators

There are six comparison (or relational) operators:

Operator	Meaning	Example
==	Equal to	x == y
!=	Not equal to	x != y
>	Greater than	x > y
>=	Greater than or equal to	x >= y
<	Less than	x < y
<=	Less than or equal to	x <= y

Take note that the comparison operator for equality is a double-equal sign `(==)`; whereas a single-equal sign `(=)` is the assignment operator.

Combining Simple Conditions

Suppose that you want to check whether a number `x` is between `1` and `100` (inclusive), i.e., `1 <= x <= 100`. There are two *simple conditions* here, `(x >= 1) AND (x <= 100)`. In programming, you cannot write `1 <= x <= 100`, but need to write `(x >= 1) && (x <= 100)`, where "`&&`" denotes the "`AND`" operator. Similarly, suppose that you want to check whether a number `x` is divisible by 2 OR by 3, you have to write `(x % 2 == 0) || (x`

`% 3 == 0)` where "`||`" denotes the "`OR`" operator.

There are three so-called *logical operators* that operate on the *boolean* conditions:

Operator	Meaning	Example
`&&`	Logical AND	`(x >= 1) && (x <= 100)`
`||`	Logical OR	`(x < 1) || (x > 100)`
`!`	Logical NOT	`!(x == 8)`

For examples:

```
// Return true if x is between 0 and 100 (inclusive)
(x >= 0) && (x <= 100)  // AND (&&)
// Incorrect to use 0 <= x <= 100

// Return true if x is outside 0 and 100 (inclusive)
(x < 0) || (x > 100)       // OR (||)
!((x >= 0) && (x <= 100))  // NOT (!), AND (&&)

// Return true if "year" is a leap year
// A year is a leap year if it is divisible by 4 but not by 100, or it is divisible by 400.
((year % 4 == 0) && (year % 100 != 0)) || (year % 400 == 0)
```

Exercises

1. Write a program to sum all the integers between 1 and 1000, that are divisible by 13, 15 or 17, but not by 30.
2. Write a program to print all the leap years between AD1 and AD2010, and also print the number of leap years. (Hints: use a variable called `count`, which is initialized to zero. Increment the `count` whenever a leap year is found.)

11. Type `double` and Floating-Point Numbers

Recall that a *variable* in Java has a *name* and a *type*, and can hold a *value* of only that particular *type*. We have so far used a type called `int`. A `int` variable holds an integer, such as `123` and `-456`; it cannot hold a real number, such as `12.34`.

In programming, real numbers such as `3.1416` and `-55.66` are called *floating-point numbers*, and belong to a type called `double`. You can express floating-point numbers in *fixed notation* (e.g., `1.23`, `-4.5`) or *scientific notation* (e.g., `1.2e3`, `-4E5.6`) where `e` or `E` denote the exponent of base 10.

Example

```
1  /*
2   * Convert temperature between Celsius and Fahrenheit
3   */
4  public class ConvertTemperature {  // Save as "ConvertTemperature.java"
5     public static void main(String[] args) {
6        double celsius, fahrenheit;
7
8        celsius = 37.5;
9        fahrenheit = celsius * 9.0 / 5.0 + 32.0;
10       System.out.println(celsius + " degree C is " + fahrenheit + " degree F.");
11
12       fahrenheit = 100.0;
13       celsius =  (fahrenheit - 32.0) * 5.0 / 9.0;
14       System.out.println(fahrenheit + " degree F is " + celsius + " degree C.");
15
16    }
17 }
```

```
37.5 degree C is 99.5 degree F.
100.0 degree F is 37.77777777777778 degree C.
```

12. Mixing `int` and `double`, and Type Casting

Although you can use a `double` to keep an integer value (e.g., `double count = 5.0`), you should use an `int` for integer, as `int` is far more efficient than `double` (e.g., in terms of running times, storage, among others).

At times, you may need both `int` and `double` in your program. For example, keeping the *sum* from 1 to 1000 as `int`, and their *average* as `double`. You need to be *extremely careful* when different types are mixed.

It is important to note that:

- Arithmetic operations (`'+'`, `'-'`, `'*'`, `'/'`) of two `int`'s produce an `int`; while arithmetic operations of two `double`'s produce a `double`. Hence, `1/2` → `0` and `1.0/2.0` → `0.5`.
- Arithmetic operations of an `int` and a `double` produce a `double`. Hence, `1.0/2` → `0.5` and `1/2.0` → `0.5`.

You can assign an integer value to a `double` variable. The integer value will be converted to a double value automatically, e.g., `3 → 3.0`. For example,

```
int i = 3;
double d;
d = i;    // 3 → 3.0, d = 3.0
d = 88;   // 88 → 88.0, d = 88.0
double nought = 0;  // 0 → 0.0; there is a subtle difference between int 0 and double 0.0
```

However, you CANNOT assign a `double` value directly to an `int` variable. This is because the *fractional* part could be lost, and the compiler signals an error in case that you were not aware. For example,

```
double d = 5.5;
int i;
i = d;     // error: possible loss of precision
i = 6.6;   // error: possible loss of precision
```

Type Casting Operators

To assign an `double` value to an `int` variable, you need to explicitly invoke a *type-casting operation* to *truncate the fractional part*, as follows:

```
(new-type)expression;
```

For example,

```
double d = 5.5;
int i;
i = (int) d;         // Type-cast the value of double d, which returns an int value,
                     // assign the resultant int value to int i.
                     // The value stored in d is not affected.
i = (int) 3.1416;    // i = 3
```

Take note that type-casting operator, in the form of `(int)` or `(double)`, applies to one operand immediately after the operator (i.e., unary operator).

Type-casting is an operation, like increment or addition, which operates on a operand and return a value (in the specified type), e.g., `(int)3.1416` takes a `double` value of `3.1416` and returns `3` (of type `int`); `(double)5` takes an `int` value of `5` and returns `5.0` (of type `double`).

Example

Try the following program and explain the outputs produced:

```
1  /*
2   * Find the sum and average from a lowerbound to an upperbound
3   */
4  public class TypeCastingTest {   // Save as "TypeCastingTest.java"
5     public static void main(String[] args) {
6        int lowerbound = 1, upperbound = 1000;
7        int sum = 0;      // sum is "int"
8        double average;   // average is "double"
9
10       // Compute the sum (in "int")
11       int number = lowerbound;
12       while (number <= upperbound) {
13          sum = sum + number;
14          ++number;
15       }
16       System.out.println("The sum from " + lowerbound + " to " + upperbound + " is " + sum);
17
18       // Compute the average (in "double")
19       average = sum / (upperbound - lowerbound + 1);
20       System.out.println("Average 1 is " + average);
21       average = (double)sum / (upperbound - lowerbound + 1);
22       System.out.println("Average 2 is " + average);
23       average = sum / 1000;
24       System.out.println("Average 3 is " + average);
25       average = sum / 1000.0;
26       System.out.println("Average 4 is " + average);
27       average = (double)(sum / 1000);
28       System.out.println("Average 5 is " + average);
29    }
30 }
```

```
The sum from 1 to 1000 is 500500
Average 1 is 500.0  <== incorrect
Average 2 is 500.5
Average 3 is 500.0  <== incorrect
Average 4 is 500.5
Average 5 is 500.0  <== incorrect
```

The first average is incorrect, as `int/int` produces an `int` (of `500`), which is converted to `double` (of `500.0`) to be stored in `average` (of `double`).

For the second average, the value of `sum` (of `int`) is first converted to `double`. Subsequently, `double/int` produces a `double`.

For the fifth average, `int/int` produces an `int` (of `500`), which is casted to `double` (of `500.0`) and assigned to `average` (of `double`).

Exercises

1. Write a program called `HarmonicSeriesSum` to compute the sum of a harmonic series `1 + 1/2 + 1/3 + 1/4 + .... + 1/n`, where `n = 1000`. Keep the sum in a `double` variable, and take note that `1/2` gives `0` but `1.0/2` gives `0.5`.
   Try computing the sum for `n=1000`, `5000`, `10000`, `50000`, `100000`.
   Hints:

```
public class HarmonicSeriesSum {   // Saved as "HarmonicSeriesSum.java"
   public static void main (String[] args) {
      int numTerms = 1000;
      double sum = 0.0;      // For accumulating sum in double
      int denominator = 1;
      while (denominator <= numTerms) {
          // Beware that int/int gives int
          ......
          ++denominator;  // next
      }
      // Print the sum
      ......
   }
}
```

2. Modify the above program (called `GeometricSeriesSum`) to compute the sum of this series: `1 + 1/2 + 1/4 + 1/8 + ....` (for `1000` terms). (Hints: Use post-processing statement of `denominator = denominator * 2`.)

13. Summary

I have presented the basics for you to get start in programming. To learn programming, you need to understand the syntaxes and features involved in the programming language that you chosen, and you have to practice, practice and practice, on as many problems as you could.

Link TO "Java References & Resources"

Latest version tested: JDK 1.7.0_21
Last modified: June, 2013