TABLE OF CONTENTS (HIDE)

Java Programming Tutorial

Introduction to Java Programmin for First-Time Programmers

1. Getting Started - Write your First Hel Program

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

```
Hello, world!
```

You could write Java programs using a *programming text editor* (such as TextPad or NotePad++) or an *integrated development tool* (such as Eclipse or NetBeans). Depending on your choice, read:

- Writing your first Java program using JDK and a Programming Text Editor (try this if you are not sure).
- Writing your first Java program using Eclipse.
- Writing your first Java program using NetBeans.

Hello.java

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

Dissecting Hello.java

The statements in green are called *comments*. Comments are not executable, but provide useful explanation to you and your readers. 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).

The basic unit of a Java program is a *class*. A class called "Hello" is defined via the keyword "class" in Lines 4-8, as follows:

```
public class Hello { ...... }    // Use keyword "class" to define a class.
                                 // { ...... } 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".

Lines 5-7 defines the so-called main() *method*, which is the starting point, or *entry point*, of program execution:

```
public static void main(String[] args) { ...... }  // main() method is the entry point of program execution.
                                                   // { ...... } is the "body" of the method,
                                                   //   which contains your programming statements.
                                                   // Other keywords will be discussed later.
```

In Line 6, the *method* 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.

A programming *statement* performs a piece of programmming action, which must be terminated by a semi-colon (;), as in Line 6.

A *block* is a group of programming statements enclosed by braces {...}. 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.

Extra white-spaces, tabs, and lines are ignored, but they could help you and your readers to better understand your program. Use them *liberally*.

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

2. 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.

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

3. Printing via System.out.println() and System.out.print()

`System.out.println(`*aString*`)` prints the given *aString* to the *display console*, 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:

```
public class PrintTest {   // Save as "PrintTest.java"
   public static void main(String[] args) {
      System.out.println("Hello, world!"); // Advance the cursor to the beginning of next line after printing
      System.out.println();                // Print a empty line
      System.out.print("Hello, world!");   // Cursor stayed after the printed string
      System.out.println("Hello,");
      System.out.print(" ");               // Print a space
      System.out.print("world!");
      System.out.println("Hello, world!");
   }
}
```

```
Hello, world!

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

4. Let's Write a Program to Add a Few Num

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
```

Lines 6-10 *declare* five `int` (integer) *variables* called `number1`, `number2`, `number3`, `number4`, and `number5`; and *assign* values of 11, 22, 33, 44, and 55, respectively, via the so-called *assignment operator* `'='`.

Line 11 declares a `int` (integer) variable called `sum`, without assigning an initial value.

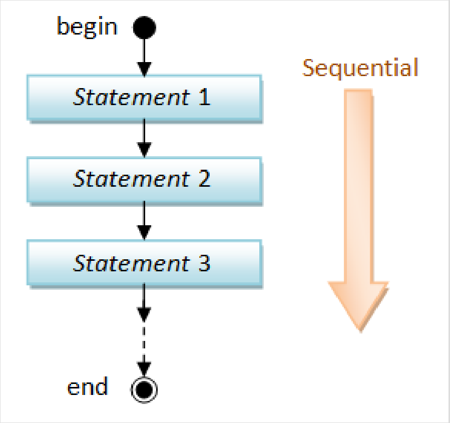
Line 12 computes the sum of `number1` to `number5` and assign the result to the variable `sum`. The symbol '+' denotes *arithmetic addition*, just like Mathematics.

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.

5. 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.

```java
 1  /*
 2   * Print the area and perimeter of a circle, given its radius.
 3   */
 4  public class CircleComputation {  // Saved as "CircleComputation.java"
 5     public static void main(String[] args) {
 6        // Declare variables
 7        double radius;
 8        double area;
 9        double perimeter;
10  
11        // Assign a value to radius
12        radius = 1.2;
13  
14        // Compute area and perimeter
15        area = radius * radius * 3.1416;
16        perimeter = 2.0 * radius * 3.1416;
17  
18        // Print results
19        System.out.print("The radius is ");  // Print description
20        System.out.println(radius);          // Print the value stored in the variable
21        System.out.print("The area is ");
22        System.out.println(area);
23        System.out.print("The perimeter is ");
24        System.out.println(perimeter);
25     }
26  }
```

```
The radius is 1.2
The area is 4.523904
The perimeter is 7.53984
```

Lines 7-9 declare three `double` variables, which can hold real numbers (or floating-point numbers, with an optional fractional part). Line 12 assigns a value to the variable `radius`. Lines 15-16 compute the `area` and `perimeter`, based on the `radius`. Lines 19-24 print the results.

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

6. What is a Variable?

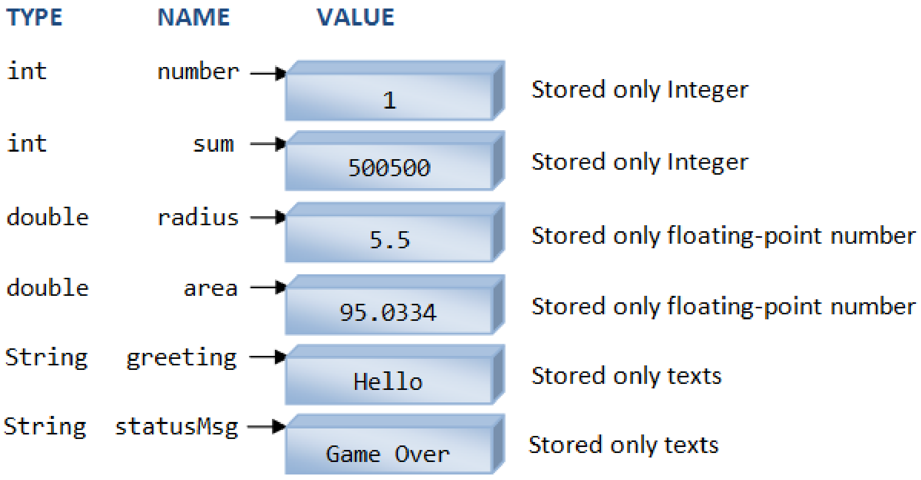
Computer programs manipulate (or process) data. A *variable* is used to store a piece of data for processing. It is called *variable* because you can change the value stored.

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*.

- 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`, having an optional decimal point and fractional part;
  - `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 intrepetation of data made up of 0s and 1s.

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 alphanumeric 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.

7. Basic Arithmetic Operations

The basic *arithmetic operations* are:

- addition (`+`)
- subtraction (`-`)
- multiplication (`*`)
- division (`/`)
- remainder or modulo (`%`)
- increment (by 1) (`++`)
- decrement (by 1) (`--`)

Addition, subtraction, multiplication, division and remainder take two operands, e.g., `number1+number2`. They are called *binary* operators. The increment and decrement take only one operand, e.g., `number++`. They are called *unary* operators.

The following program illustrates these arithmetic operations.

```
 1  /**
 2   * Test Arithmetic Operations
 3   */
 4  public class ArithmeticTest {     // Save as "ArithmeticTest.java"
 5     public static void main(String[] args) {
 6  
 7        int number1 = 98;     // Declare an int variable number1 and initialize it to 98
 8        int number2 = 5;      // Declare an int variable number2 and initialize it to 5
 9        int sum, difference, product, quotient, remainder;  // Declare five int variables to hold results
10  
11        // Perform arithmetic Operations
12        sum = number1 + number2;
13        difference = number1 - number2;
14        product = number1 * number2;
15        quotient = number1 / number2;
16        remainder = number1 % number2;
17        System.out.print("The sum, difference, product, quotient and remainder of ");  // Print description
18        System.out.print(number1);      // Print the value of the variable
19        System.out.print(" and ");
20        System.out.print(number2);
21        System.out.print(" are ");
22        System.out.print(sum);
23        System.out.print(", ");
24        System.out.print(difference);
25        System.out.print(", ");
26        System.out.print(product);
27        System.out.print(", ");
28        System.out.print(quotient);
29        System.out.print(", and ");
30        System.out.println(remainder);
31  
32        number1++;  // Increment the value stored in the variable "number1" by 1
33                    // Same as "number1 = number1 + 1"
34        number2--;  // Decrement the value stored in the variable "number2" by 1
35                    // Same as "number2 = number2 - 1"
36        System.out.println("number1 after increment is " + number1);  // Print description and variable
37        System.out.println("number2 after decrement is " + number2);
38        quotient = number1 / number2;
39        System.out.println("The new quotient of " + number1 + " and " + number2
40              + " is " + quotient);
41     }
42  }
```

```
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
```

Lines 7-8 declare and initialize two `int` (integer) variables: `number1` and `number2`. Line 9 declares five `int` variables: `sum`, `difference`, `product`, `quotient`, and `remainder` to hold the results of operations, in one statement (with items separated by commas), without assigning initial values.

Lines 12-16 carry out the arithmetic operations on variables `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`.

Lines 17-30 print the results of the arithmetic operations, with appropriate descriptions in between. Take note that text string are enclosed within double-quotes, and will get printed as it is, 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
```

Lines 32 and 34 illustrate the increment and decrement operations. Unlike `'+'`, `'-'`, `'*'`, `'/'` and `'%'`, which work on two operands (*binary operators*), `'++'` and `'--'` operate on only one operand (*unary operators*).

Lines 36-37 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. Line 36 provides another example.

TRY:

1. Combining Lines 17-30 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`.

8. What If Your Need To Add Many Numbers

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.

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

```
 1  /*
 2   * Sum from a lowerbound to an upperbound using a for-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 for-loop to repeatitively sum from the lowerbound to the upperbound
11        for (int number = 1; number <= upperbound; number++) {
12           sum = sum + number;   // Accumulate number into sum
13        }
14        // Print the result
15        System.out.println("The sum from " + lowerbound + " to " + upperbound + " is " + sum);
16     }
17  }
```

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

Let us dissect this program:

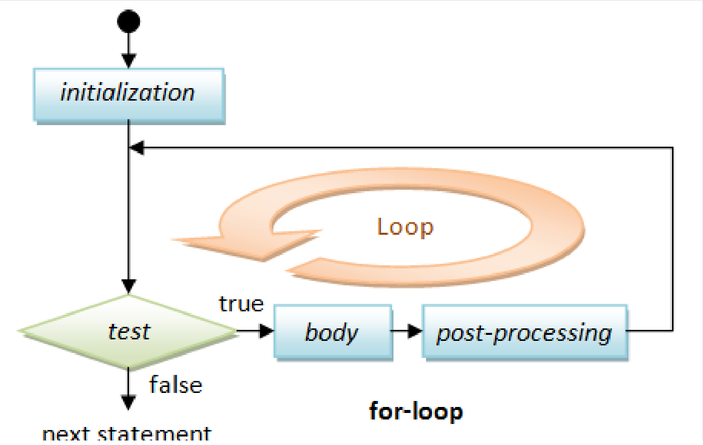
Lines 6 and 7 declare two `int` variables to store the lowerbound and upperbound respectively.

Line 8 declares an `int` variable named `sum` and initializes it to `0`. This variable will be used to *accumulate* numbers over the steps in the repetitive loop.

Lines 11-13 contain a so-called *for-loop*. A *for-loop* takes the following syntax:

```
// Syntax
for ( initialization ; test ; post-processing ) {
   body ;
}
```

```
// Example
int sum = 0;
for (int number = 1; number <= 1000; number++) {
   sum = sum + number;
}
```



There are four parts in a *for-loop*. Three of them, *initialization*, *test condition* and *post-processing*, are enclosed in brackets `(...; ...; ...)`, and separated by two semi-colons `';'`. The *body* contains the *repetitive task* to be performed. As illustrated in the above flow chart, the *initialization* statement is first executed. The *test* is then evaluated. If the *test* returns true, the body is executed; followed by the *post-processing* statement. The *test* is checked again and the process repeats until the *test* returns false. When the *test* is false, the for-loop completes and program execution continues to the next statement after the for-loop.

In our program, the *initialization* statement declares an `int` variable named `number` and initializes it to `lowerbound` (=1). The *test* checks if `number` is equal to or less than `upperbound` (=`1000`). If it is true, the

current value of number is added into the sum, and the post-processing 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 for-loop to terminate. Execution then continues to the next statement (in Line 15).

In this example, the loop repeats 1000 times (number having value of 1 to 1000). After the loop is completed, Line 15 prints the result with a proper description.

TRY:

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.)

9. 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:

```
 1  /*
 2   * Sum the odd numbers and the even numbers from a lowerbound to an upperbound
 3   */
 4  public class OddEvenSum {  // Save as "OddEvenSum.java"
 5     public static void main(String[] args) {
 6        int lowerbound = 1;
 7        int upperbound = 1000;
 8        int sumOdd  = 0;    // For accumulating odd numbers, init to 0
 9        int sumEven = 0;    // For accumulating even numbers, init to 0
10        for (int number = lowerbound; number <= upperbound; number++) {
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        }
17        // Print the result
18        System.out.println("The sum of odd numbers from " + lowerbound + " to " + upperbound + " is " + sumOdd);
19        System.out.println("The sum of even numbers from " + lowerbound + " to " + upperbound + "  is " + sumEven);
20        System.out.println("The difference between the two sums is " + (sumOdd - sumEven));
21     }
23  }
```

```
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
```

Lines 8 and 9 declare two int variables named sumOdd and sumEven and initialize them to 0, for accumulating the odd and even numbers respectively.

Lines 11-15 contain a *conditional statement*. The conditional statement can take one the following forms: *if-then*, *if-then-else*.

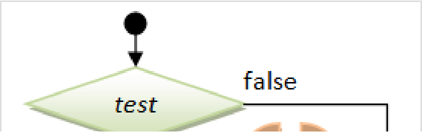
```
// if-then syntax
if ( condition ) {
   true-body ;
}
```

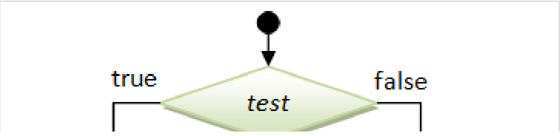
```
// Example
if (mark >= 50) {
   System.out.println("Congratulation!");
}
```

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

```
// Example
if (mark >= 50) {
   System.out.println("Congratulation!");
} else {
   System.out.println("Try Harder!");
}
```

For a *if-then* statement, the *true-body* is executed if the *test condition* is true. Otherwise, nothing is done and the execution continues to the next

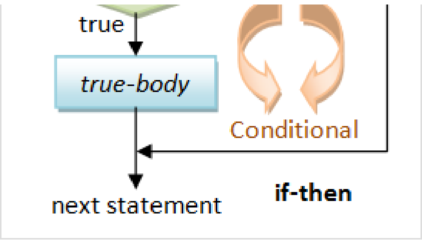


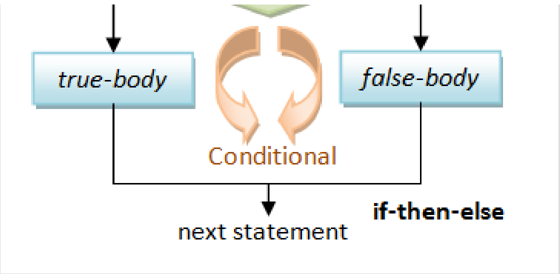


statement.

For a *if-then-else* statement, the *true-body* is executed if the *condition* is `true`; otherwise, the *false-body* is executed. Execution is then continued to the next statement.

The following flow chart illustrates the *if-then* and *if-then-else* statements.





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

There are six comparison operators:

- equal to (`==`)
- not equal to (`!=`)
- greater than (`>`)
- less than (`<`)
- greater than or equal to (`>=`)
- less than or equal to (`<=`)

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

10. 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:

- `AND (&&)`
- `OR (||)`
- `NOT (!)`

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)
```

TRY:

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.

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`, `-55.66` are called *floating-point numbers*, and belong to a type called `double`. For example,

```
public class CircleOperation {  // Saved as "CircleOperation.java"
   public static void main(String[] args) {
      double radius = 1.2;     // Type double for floating-point numbers
      double pi = 3.1416;
      double area;
      area = radius * radius * pi;
      System.out.println("The radius is " + radius);
      System.out.println("The area is " + area);
      System.out.println("The perimeter is " + (2.0 * pi * radius));
   }
}
```

```
The radius is 1.2
The area is 4.523904
The perimeter is 7.53984
```

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;     // Compilation Error
i = 6.6;   // Compilation Error
```

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:

```
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`).

Try the following program and explain the outputs produced.

```
/*
 * Find the sum and average from a lowerbound to an upperbound
 */
public class TypeCastingTest {   // Save as "TypeCastingTest.java"
   public static void main(String[] args) {
      int lowerbound = 1;
      int upperbound = 1000;
      int sum = 0;      // sum is "int"
      double average;   // average is "double"
      // Compute the sum (in "int")
      for (int number = lowerbound; number <= upperbound; number++) {
         sum = sum + number;
      }
      System.out.println("The sum from " + lowerbound + " to " + upperbound + " is " + sum);
      // Compute the average (in "double")
      average = sum/1000;
      System.out.println("Average 1 is " + average);
      average = (double)sum/1000;
      System.out.println("Average 2 is " + average);
      average = sum/1000.0;
      System.out.println("Average 3 is " + average);
      average = (double)(sum/1000);
      System.out.println("Average 4 is " + average);
   }
}
```

```
The sum is 500500
Average 1 is 500.0  <== incorrect
Average 2 is 500.5
Average 3 is 500.5
Average 4 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 third average, `int/double` produces `double`.

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

TRY:

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
         for (int denominator = 1; denominator <= numTerms ; denominator++) {
             // Beware that int/int gives int
             ......
         }
         // Print the sum
         ......
      }
   }
   ```

   ```
   The sum is 7.484470860550343
   ```

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_03
Last modified: May, 2012

Feedback, comments, corrections, and errata can be sent to Chua Hock-Chuan (ehchua@ntu.edu.sg) | HOME