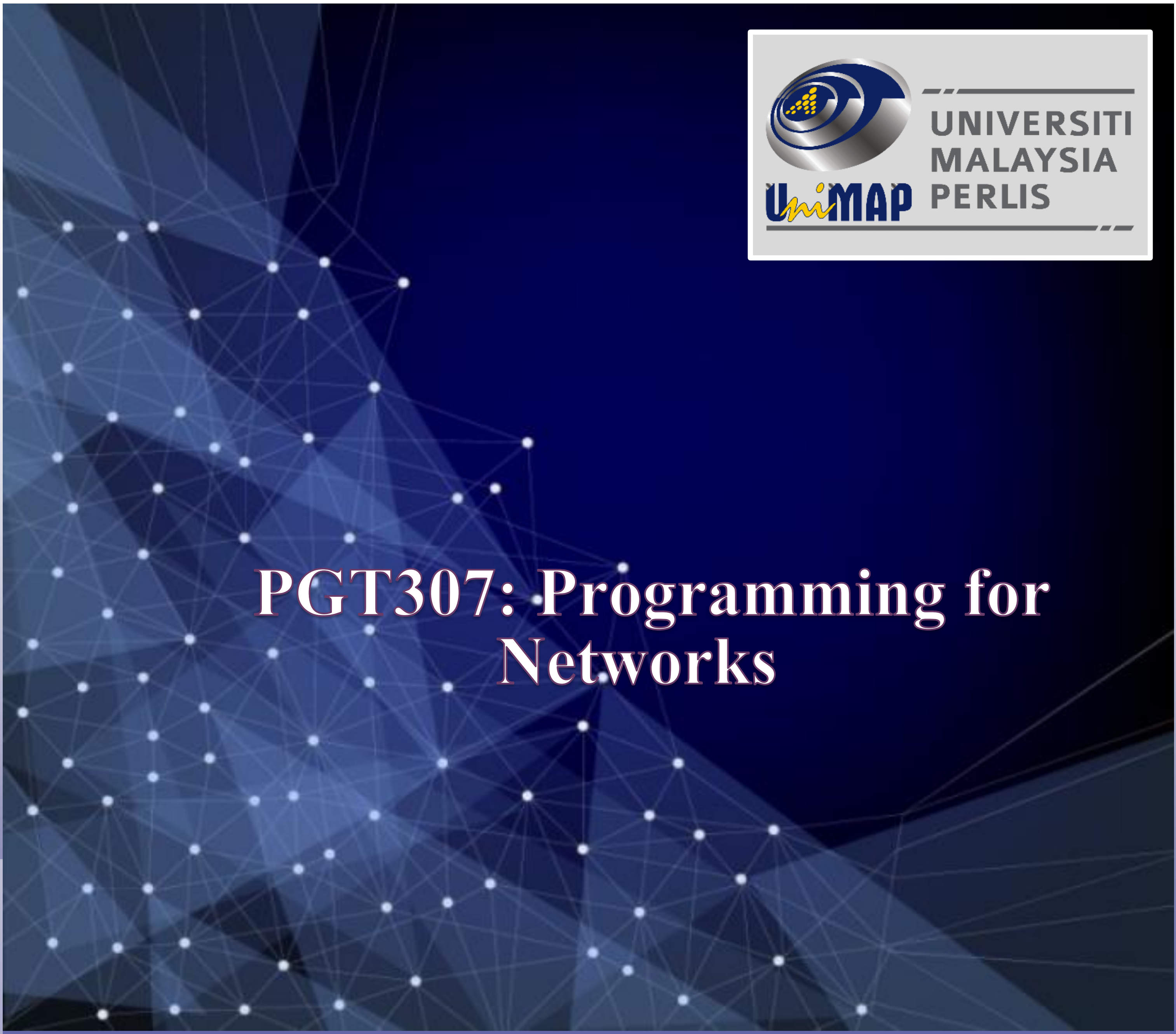


UNIVERSITI MALAYSIA PERLIS

UniMAP

PGT307: Programming for Networks

Lab01:

Introduction to JAVA

Mohamed Elshaikh

Faculty of Electronics Engineering Technology – UniMAP (FTKEN-UniMAP)

Objectives

- Introduction to JAVA programming
- JAVA programming structure
- The IDE

Introduction to JAVA Programming

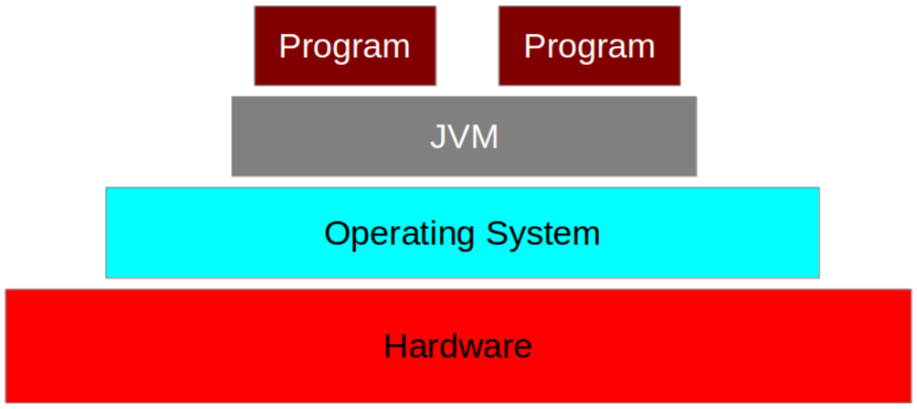
Java is a programming language created by James Gosling from Sun Microsystems (Sun) in 1991. The target of Java is to write a program once and then run this program on multiple operating systems. The first publicly available version of Java (Java 1.0) was released in 1995. Sun Microsystems was acquired by the Oracle Corporation in 2010. Oracle has now the steermanship for Java. In 2006 Sun started to make Java available under the GNU General Public License (GPL). Oracle continues this project called OpenJDK. Over time new enhanced versions of Java have been released. The current version of Java is Java 1.8 which is also known as Java 8.

i. Java is defined by a specification and consists of a programming language, a compiler, core libraries and a runtime (Java virtual machine) The Java runtime allows software developers to write program code in other languages than the Java programming language which still runs on the Java virtual machine. The Java platform is usually associated with the Java virtual machine and the Java core libraries. The Java language was designed with the following properties:
ii. Platform independent: Java programs use the Java virtual machine as abstraction and do not access the operating system directly. This makes Java programs highly portable. A Java program (which is standard-compliant and follows certain rules) can run unmodified on all supported platforms, e.g., Windows or Linux.
iii. Object-orientated programming language: Except the primitive data types, all elements in Java are objects.
iv. Strongly-typed programming language: Java is strongly-typed, e.g., the types of the used variables must be pre-defined and conversion to other objects is relatively strict, e.g., must be done in most cases by the programmer.
v. Interpreted and compiled language: Java source code is transferred into the bytecode format which does not depend on the target platform. These bytecode instructions will be interpreted by the Java Virtual machine (JVM). The JVM contains a so-called Hotspot-Compiler which translates performance critical bytecode instructions into native code instructions.
vi. Automatic memory management: Java manages the memory allocation and de-allocation for creating new objects. The program does not have direct access to the memory. The so-called garbage collector automatically deletes objects to which no active pointer exists.

The Java syntax is similar to C++. Java is case-sensitive, e.g., variables called myValue and myvalue are treated as different variables.

1. Java Virtual Machine (JVM)

The Java virtual machine (JVM) is a software implementation of a computer that executes programs like a real machine. The Java virtual machine is written specifically for a specific operating system, e.g., for Linux a special implementation is required as well as for Windows. Java programs are compiled by the Java compiler into bytecode. The Java virtual machine interprets this bytecode and executes the Java program.



2. Java Runtime Environment vs. Java Development Kit

A Java distribution typically comes in two flavors, the Java Runtime Environment (JRE) and the Java Development Kit (JDK). The JRE consists of the JVM and the Java class libraries. Those contain the necessary functionality to start Java programs. he JDK additionally contains the development tools necessary to create Java programs. The JDK therefore consists of a Java compiler, the Java virtual machine and the Java class libraries.

3. Development Process with Java

Java source files are written as plain text documents. The programmer typically writes Java source code in an Integrated Development Environment (IDE) for programming. An IDE supports the programmer in the task of writing code, e.g., it provides auto-formating of the source code, highlighting of the important keywords, etc.

At some point the programmer (or the IDE) calls the Java compiler (javac). The Java compiler creates the bytecode instructions. These instructions are stored in .class files and can be executed by the Java Virtual Machine.

4. Garbage collector

The JVM automatically re-collects the memory which is not referred to by other objects. The Java garbage collector checks all object references and finds the objects which can be automatically released.

While the garbage collector relieves the programmer from the need to explicitly manage memory, the programmer still need to ensure that he does not keep unneeded object references, otherwise the garbage collector cannot release the associated memory. Keeping unneeded object references are typically called memory leaks.

5. Classpath

The classpath defines where the Java compiler and Java runtime look for .class files to load. These instructions can be used in the Java program. For example, if you want to use an external Java library you have to add this library to your classpath to use it in your program.

6. Java is a platform independent language

Compiler (javac) converts source code (.java file) to the byte code (.class file). As mentioned above, JVM executes the bytecode produced by compiler. This byte code can run on any platform such as Windows, Linux, Mac OS etc. Which means a program that is compiled on windows can run on Linux and vice-versa. Each operating system has different JVM, however the output they produce after execution of bytecode is same across all operating systems. That is why we call java as platform independent language.

Object oriented programming is a way of organizing programs as collection of objects, each of which represents an instance of a class. 4 main concepts of Object-Oriented programming are:

1. Abstraction
2. Encapsulation
3. Inheritance
4. Polymorphism

Using java programming language, we can create distributed applications. RMI (Remote Method Invocation) and EJB (Enterprise Java Beans) are used for creating distributed applications in java. In simple words: The java programs can be distributed on more than one system that are connected to each other using internet connection. Objects on one JVM (java virtual machine) can execute procedures on a remote JVM.

As discussed above, java code that is written on one machine can run on another machine. The platform independent byte code can be carried to any platform for execution that makes java code portable.

Getting Started - Your First Java Program

Let us revisit the "Hello-world" program that prints a message "`Hello, world!`" to the display console.

Step 1: Write the Source Code: Enter the following source codes, which defines a *class* called "`Hello`", using a programming text editor. Do not enter the line numbers (on the left pane), which were added to aid 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`". Filename and classname are *case-sensitive*.

```
/*
 * First Java program, which says hello.
 */
public class Hello {   // Save as "Hello.java"
   public static void main(String[] args) {  // Program entry point
      System.out.println("Hello, world!");   // Print text message
   }
}
```

Step 2: Compile the Source Code: Compile the source code "`Hello.java`" into Java bytecode (or machine code) "`Hello.class`" using JDK's Java Compiler "`javac`".

Start a CMD Shell (Windows) or Terminal (UNIX/Linux/macOS) and issue these commands:

```
// Change directory (cd) to the directory (folder) containing the source file "Hello.java"
javac Hello.java
```

Step 3: Run the Program: Run the machine code using JDK's Java Runtime "`java`", by issuing this command:

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

How it Works

```
/* ...... */
// ... until the end of the current 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 (Single-Line) Comment*: begins with `//` and lasts until the end of the current line (as in Lines 4, 5, 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 braces `{......}` encloses the *body* of the class.

In Java, the name of the source file must be the same as the name of the class with a mandatory file extension of "`.java`". Hence, this file MUST be saved as "`Hello.java`" - case-sensitive.

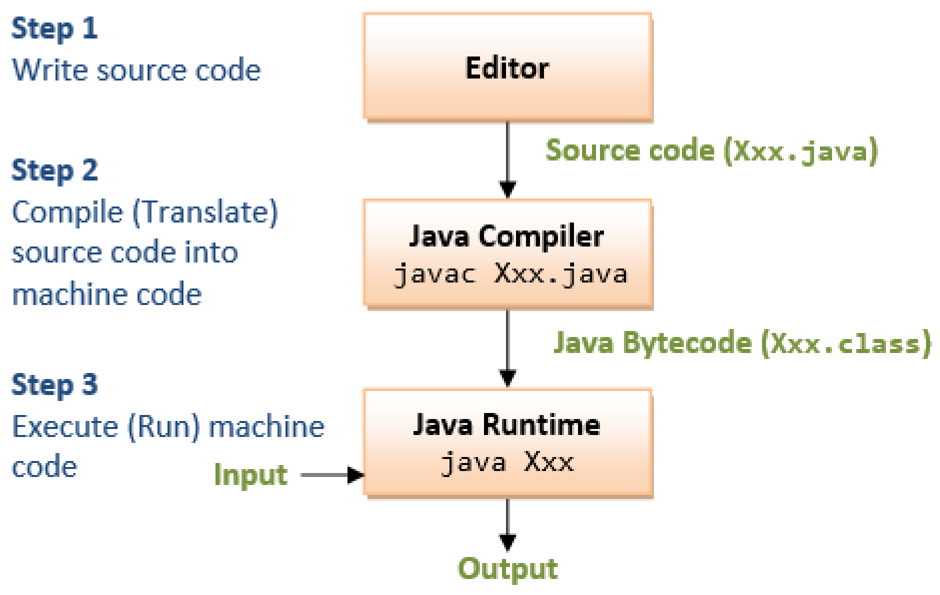
```
public static void main(String[] args) {
    ......
    }
```

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

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

In Line 6, the programming statement `System.out.println("Hello, world!")` is used to print the 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 ends with a semi-colon (`;`).

Java Programming Steps



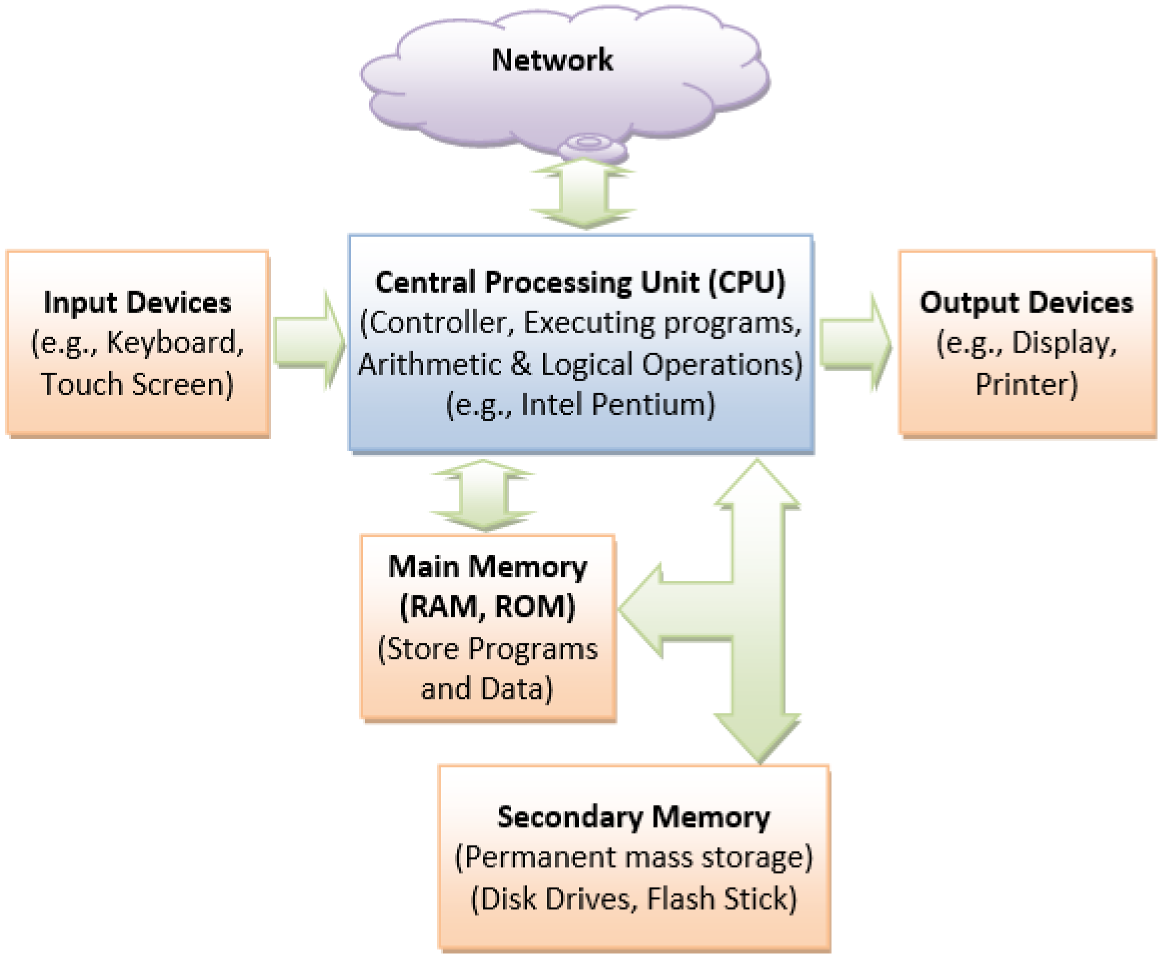
The steps in writing a Java program are illustrated as above:

Step 1: Write the source code "`Xxx.java`".

Step 2: Compile the source code "`Xxx.java`" into Java portable bytecode (or machine code) "`Xxx.class`" using the JDK's Java compiler by issuing the command "`javac Xxx.java`".

Step 3: Run the compiled bytecode "`Xxx.class`", using the JDK's Java Runtime by issuing the command "`java Xxx`".

Computer Architecture


Network
Input Devices (e.g., Keyboard, Touch Screen)
Central Processing Unit (CPU) (Controller, Executing programs, Arithmetic & Logical Operations) (e.g., Intel Pentium)
Output Devices (e.g., Display, Printer)
Main Memory (RAM, ROM) (Store Programs and Data)
Secondary Memory (Permanent mass storage) (Disk Drives, Flash Stick)

The *Central Processing Unit* (CPU) is the *heart* of a computer, which serves as the overall controller of the computer system. It fetches programs/data from main memory and executes the programs. It performs the arithmetic and logical operations (such as addition and multiplication).

The *Main Memory* stores the programs and data for execution by the CPU. It consists of RAM (Random Access Memory) and ROM (Read-Only Memory). RAM is volatile, which losses all its contents when the power is turned off. ROM is non-volatile, which retains its contents when the power is turned off. ROM is read-only and its contents cannot be changed once initialized. RAM is read-write. RAM and ROM are expensive. Hence, their amount is quite limited.

The *Secondary Memory*, such as disk drives and flash sticks, is less expensive and is used for *mass* and *permanent* storage of programs and data (including texts, images and video). However, the CPU can only run programs from the main memory, not the secondary memory.

When the power is turned on, a small program stored in ROM is executed to fetch the essential programs (called operating system) from the secondary memory to the main memory, in a process known as *booting*. Once the operating system is loaded into the main memory, the computer is ready for use. This is, it is ready to fetch the desired program from the secondary memory to the main memory for execution upon user's command.

The CPU can read data from the Input devices (such as keyboard or touch pad) and write data to the Output devices (such as display or printer). It can also read/write data through the network interfaces (wired or wireless).

You job as a programmer is to write programs, to be executed by the CPU to accomplish a specific task.

Java Terminology and Syntax

- **Comments**: A *multi-line comment* begins with `/*` and ends with `*/`, and may span multiple lines. An *end-of-line (single-line) comment* begins with `//` and lasts till the end of the current line. Comments are NOT executable statements and are ignored by the compiler. But they provide useful explanation and documentation. I strongly suggest that you write comments *liberally* to explain your thought and logic.
- **Statement**: A programming *statement* performs a single piece of programming action. It is 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 a pair of braces `{}`. This group of statements is treated as one single unit. There are two blocks in the above 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.
- **Whitespaces**: Blank, tab, and newline are collectively called *whitespace*. Extra whitespaces are ignored, i.e., only one whitespace is needed to separate the tokens. Nonetheless, extra whitespaces improve the readability, and I strongly suggest you use extra spaces and newlines to improve the readability of your code.
- **Case Sensitivity**: Java is *case sensitive* - a *ROSE* is NOT a *Rose*, and is NOT a *rose*. The *filename*, which is the same as the class name, is also case-sensitive.

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. Provide comments in your program!

```
  /*
   * Comment to state the purpose of the program
   */
  public class Classname {     // Choose a meaningful Classname. Save as
"Classname.java"
     public static void main(String[] args) {  // Entry point of the program
        // Your programming statements here!!!
     }
  }
```

Output via System.out.println() and System.out.print()

You can use `System.out.println()` (print-line) or `System.out.print()` to print text messages to the display console:

- `System.out.println(`*`aString`*`)` (print-line) prints *`aString`*, and advances the *cursor* to the beginning of the next line.
- `System.out.print(`*`aString`*`)` prints *`aString`* but places the cursor after the printed string.
- `System.out.println()` without parameter prints a *newline*.

Try the following program and explain the output produced:

```
/*
 * Test System.out.println() (print-line) and System.out.print().
 */
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("Hello world again!"); // Advance the cursor to the
beginning of next line after printing
      System.out.println();                      // Print an empty line
      System.out.print("Hello world!");            // Cursor stayed after the
printed string
      System.out.print("Hello world again!");     // Cursor stayed after the
printed string
      System.out.println();                      // Print an empty line
      System.out.print("Hello,");
      System.out.print(" ");                     // Print a space
      System.out.println("world!");
      System.out.println("Hello, world!");
   }
}
```

Save the source code as "`PrintTest.java`" (which is the same as the classname). Compile and run the program. The expected outputs are:

```
Hello world!
Hello world again!

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

Exercises 1

A- using `System.out.println()` write 4 programs, called

a) `PrintCheckerPattern`
b) `PrintSquarePattern`
c) `PrintTriangularPattern`
d) `PrintStarPattern`

to print each of the following patterns. Use ONE `System.out.println(...)` (print-line) statement for EACH LINE of outputs. Take note that you need to print the preceding blanks.

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

B- Write a program called `PrintSheepPattern` to print the following pattern:

```
          '__'
          (oo)
  /========//
 / || @@ ||
*  ||----||
   vv    vv   ''    ''
```

Let's Write a Program to Add a Few Numbers

Let us write a program to add FIVE integers and display their sum, as follows:

```java
/*
 * Add five integers and display their sum.
 */
public class FiveIntegerSum {   // Save as "FiveIntegerSum.java"
   public static void main(String[] args) {
      int number1 = 11;  // Declare 5 integer variables and assign a value
      int number2 = 22;
      int number3 = 33;
      int number4 = 44;
      int number5 = 55;
      int sum;  // Declare an integer variable called sum to hold the sum
      sum = number1 + number2 + number3 + number4 + number5;  // Compute sum
      System.out.print("The sum is ");  // Print a descriptive string
                                        // Cursor stays after the printed string
      System.out.println(sum);  // Print the value stored in variable sum
                                // Cursor advances to the beginning of next line
   }
}
```

Save the source code as "`FiveIntegerSum.java`" (which is the same as the classname). Compile and run the program. The expected output is:

```
The sum is 165
```

How It Works?

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

These five statements 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* `'='`.

Alternatively, you could declare many variables in one statement separated by 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 - its value is to be computed and assigned later.

```
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* (without the double quotes). The cursor stays after the printed string. Try using `println()` instead of `print()` and study the output.

Line 15 prints the *value* stored in the variable `sum` (in this case, the sum of the five integers). 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. The cursor advances to the beginning of next line after printing. Try using `print()` instead of `println()` and study the output.

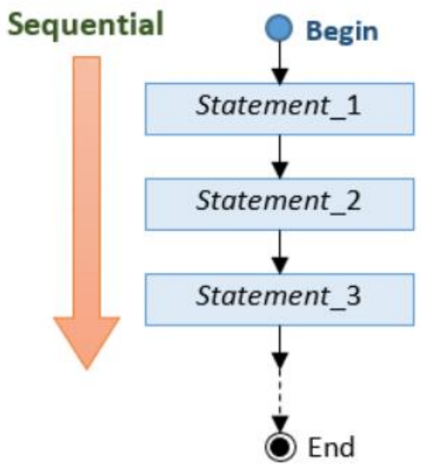
Exercise 2

1. Follow the above example, write a program called `SixIntegerSum` which includes a new variable called `number6` with a value of 66 and prints their sum.
2. Follow the above example, write a program called `SevenIntegerSum` which includes a new variable called `number7` with a value of 77 and prints their sum.
3. Follow the above example, write a program called `FiveIntegerProduct` to print the product of 5 integers. You should use a variable called `product` (instead of `sum`) to hold the product. Use symbol `*` for multiplication.

What is a Program

A *program* is *a sequence of instructions* (called *programming statements*), executing one after another in a *predictable* manner.

Sequential flow is the most common and straight-forward, where programming statements are executed in the order that they are written - from top to bottom in a *sequential* manner, as illustrated in the following flow chart.



Example

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

In this example, we use "`double`" which hold floating-point number (or real number with an optional fractional part) instead of "`int`" which holds *integer*.

```
/*
 * Print the area and circumference of a circle, given its radius.
 */
public class CircleComputation {  // Save as "CircleComputation.java"
   public static void main(String[] args) {
      // Declare 3 double variables to hold radius, area and circumference.
      // A "double" holds floating-point number with an optional fractional
part.
      double radius, area, circumference;
      // Declare a double to hold PI.
      // Declare as "final" to specify that its value cannot be changed (i.e.
constant).
      final double PI = 3.14159265;

      // Assign a value to radius. (We shall read in the value from the keyboard
later.)
      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 expected outputs are:

```
The radius is 1.2

The area is 4.523893416

The circumference is 7.5398223600000005
```

How It Works?

```
   double radius, area, circumference;
```

declare three `double` variables `radius`, `area` and `circumference`. A `double` variable can hold a real number or floating-point number with an optional fractional part. (In the previous example, we use `int`, which holds integer.)

```
   final double PI = 3.14159265;
```

declare a `double` variable called `PI` and assign a value. `PI` is declared `final` to specify that its value cannot be changed, i.e., a constant.

```
   radius = 1.2;
```

assigns a value (real number) to the `double` variable `radius`.

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

compute the `area` and `circumference`, based on the value of `radius` and `PI`.

```
   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()` method are executed one after another, in a *sequential* manner.

Exercises 3

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). You should use 4 `double` variables called `length`, `width`, `area` and `perimeter`.
2. Follow the above example, write a program called `CylinderComputation` to print the surface area, base area, and volume of a cylinder, given its radius and height (in `double`s). You should use 5 `double` variables called `radius`, `height`, `surfaceArea`, `baseArea` and `volume`. Take note that space (blank) is not allowed in variable names.

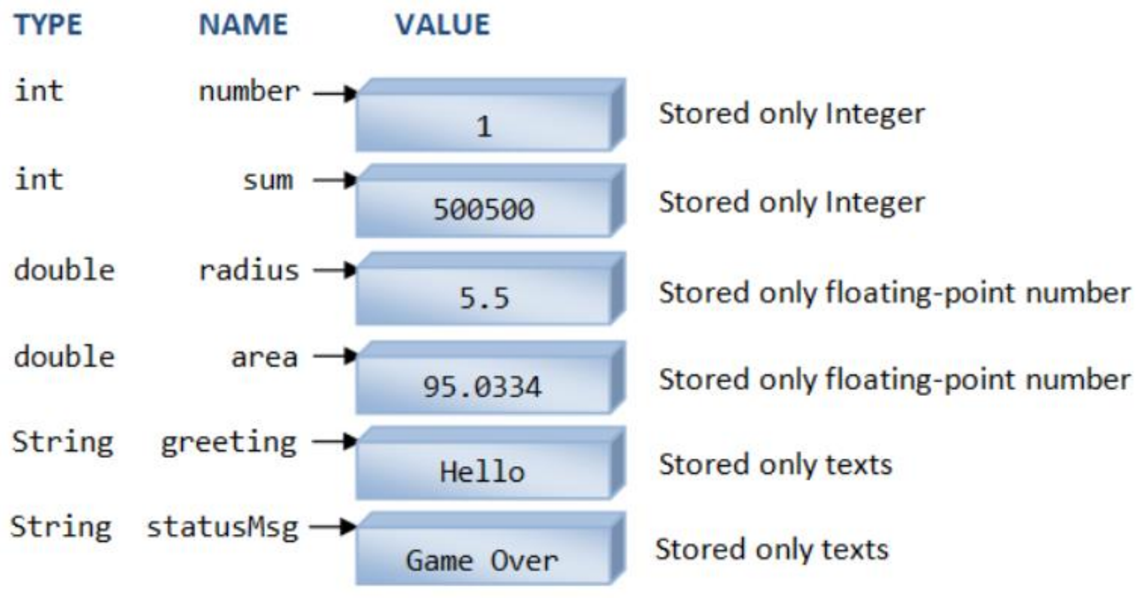
What is a Variable

A computer program manipulates (or processes) data. A variable is a storage location (like a house, a pigeon hole, a letter box) that stores a piece of data for processing. It is called variable because you can change the value stored inside. 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 particular type.

A variable has a name (aka identifier), e.g., radius, area, age, height, numStudnets. The name is needed to uniquely identify each variable, so as to assign a value to the variable (e.g., radius = 1.2), as well as to retrieve the value stored (e.g., radius * radius * 3.14159265). A variable has a type. Examples of type are:

- int: meant for integers (or whole numbers or fixed-point numbers) including zero, positive and negative integers, such as 123, -456, and 0;
- double: meant for floating-point numbers or real numbers, such as 3.1416, -55.66, having an optional decimal point and fractional part.
- String: meant for texts such as "Hello", "Good Morning!". Strings shall be enclosed with a pair of double quotes.

A variable can store a value of the declared 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 that particular type. For example, an int variable can store an integer value such as 123, but NOT a real number such as 12.34, nor texts such as "Hello". The concept of type was introduced into the early programming languages to simplify interpretation 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 (which includes integer as a special form of real number); a String variable stores texts.



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

Basic Arithmetic Operations

The basic arithmetic operations are:

Operator	Mode	Usage	Meaning	Example x=5; y=2
+	Binary Unary	x + y +x	Addition	x + y returns 7
-	Binary Unary	x - y -x	Subtraction	x - y returns 3
*	Binary	x * y	Multiplication	x * y returns 10
/	Binary	x / y	Division	x / y returns 2
%	Binary	x % y	Modulus (Remainder)	x % y returns 1
++	Unary Prefix Unary Postfix	++x x++	Increment by 1	++x or x++ (x is 6) Same as x = x + 1
--	Unary Prefix Unary Postfix	--x x--	Decrement by 1	--y or y-- (y is 1) Same as y = y - 1

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, --y) 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 5 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
```

The expected outputs are:

```
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
```

How It Works?

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

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

```
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.

```
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 18-31, 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 4

1. Combining Lines 18-31 into one single `println()` statement, using `'+'` to *concatenate* all the items together.

2. 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`.
3. Based on the above example, write a program called `SumProduct3Numbers`, which introduces 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.

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*.

```
   /*
    * Sum from a lowerbound to an upperbound using a while-loop
    */
   public class RunningNumberSum {  // Save as "RunningNumberSum.java"
      public static void main(String[] args) {
         final int LOWERBOUND = 1;      // Store the lowerbound
         final int UPPERBOUND = 1000;   // Store the upperbound
         int sum = 0;   // Declare an int variable "sum" to accumulate the numbers
                        // Set the initial sum to 0
         // Use a while-loop to repeatedly sum from the lowerbound to the upperbound
         int number = LOWERBOUND;
         while (number <= UPPERBOUND) {
               // number = LOWERBOUND, LOWERBOUND+1, LOWERBOUND+2, ..., UPPERBOUND
for each iteration
            sum = sum + number;  // Accumulate number into sum
            ++number;            // increment number
         }
         // Print the result
         System.out.println("The sum from " + LOWERBOUND + " to " + UPPERBOUND + "
is " + sum);
      }
   }
```

The expected output is:

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

How It Works?

```
final int LOWERBOUND = 1;
final int UPPERBOUND = 1000;
```

declare two `int` constants 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* 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 *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 example, 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 18).

A loop is typically controlled by an index variable. In this example, the index variable `number` takes the value `LOWERBOUND`, `LOWERBOUND+1`, `LOWERBOUND+2`, ..., `UPPERBOUND`, for each iteration of the loop.

In this example, the loop repeats `UPPERBOUND-LOWERBOUND+1` times. After the loop is completed, Line 18 prints the result with a proper description.

```
System.out.println("The sum from " + LOWERBOUND + " to " + UPPERBOUND + " is "
+ sum);
```

prints the results.

Exercises 5

1. Modify the above program (called `RunningNumberSum1`) to sum all the numbers from `9` to `899`. (*Ans*: `404514`)
2. Modify the above program (called `RunningNumberOddSum`) 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 (called `RunningNumberMod7Sum`) to sum all the numbers between `1` to `1000` that are divisible by `7`. (*Hint*: Modify the initialization statement to begin from 7 and post-processing statement to increment by 7. *Ans*: `71071`)
4. Modify the above program (called `RunningNumberSquareSum`) to find the sum of the *square* of all the numbers from `1` to `100`, i.e. `1*1 + 2*2 + 3*3 +...` (*Hint*: Modify the `sum = sum + number` statement. *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. Modify the `sum = sum + number` statement to do multiplication on variable `product`. *Ans*: `3628800`)

Conditional (or Decision)

What if you want to sum all the odd numbers and also all the even numbers between `1` and `1000`? You could declare two variables, `sumOdd` and `sumEven`, to keep the two sums. 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) {
         final int LOWERBOUND = 1;
         final int UPPERBOUND = 1000;
         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) {
               // number = LOWERBOUND, LOWERBOUND+1, LOWERBOUND+2, ..., UPPERBOUND
for each iteration
            if (number % 2 == 0) {  // Even
               sumEven += number;   // Same as sumEven = sumEven + number
            } else {                // Odd
               sumOdd += number;    // Same as sumOdd = sumOdd + number
            }
            ++number;  // Next number
         }
         // Print the result
         System.out.println("The sum of odd numbers from " + LOWERBOUND + " to " +
UPPERBOUND + " is " + sumOdd);
         System.out.println("The sum of even numbers from " + LOWERBOUND + " to "
+ UPPERBOUND + "  is " + sumEven);
         System.out.println("The difference between the two sums is " + (sumOdd -
sumEven));
      }
   }
```

The expected outputs are:

```
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
```

How It Works?

```
final int LOWERBOUND = 1;
final int UPPERBOUND = 1000;
```

declares and initializes the upperbound and lowerbound constants.

```
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 += number;
} else {
   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* or *modulus* operator (`%`) to compute the remainder of `number` divides by `2`. We then compare the remainder with `0` to test for even number.

Furthermore, `sumEven += number` is a *shorthand* for `sumEven = sumEven + number`.

Comparison Operators

There are six comparison (or relational) operators:

Operator	Mode	Usage	Meaning
==	Binary	x == y	Equal to
!=	Binary	x != y	Not equal to
>	Binary	x > y	Greater than
>=	Binary	x >= y	Greater than or equal to
<	Binary	x < y	Less than
<=	Binary	x <= y	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.

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 Java, 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	Mode	Usage	Meaning	Example
&&	Binary	x && y	Logical AND	(x >= 1) && (x <= 100)
||	Binary	x || y	Logical OR	(x < 1) || (x > 100)
!	Unary Prefix	!x	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 6

1. Write a program called **`ThreeFiveSevenSum`** to sum all the running integers from 1 and 1000, that are divisible by 3, 5 or 7, but NOT by 15, 21, 35 or 105.
2. Write a program called **`PrintLeapYears`** to print all the leap years between AD999 and AD2010. Also print the total number of leap years. (Hints: use a `int` variable called `count`, which is initialized to zero. Increment the `count` whenever a leap year is found.)