Java Programming Tutorial

Java Basics

This chapter explains the basic syntaxes of the Java programming language. I shall assume that you could write some simple programs. (Otherwise, read "Introduction To Java Programming for First-time Programmers".)

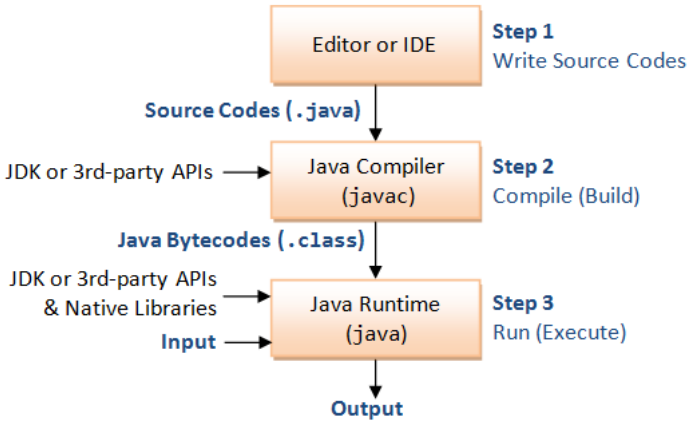
To be a proficient programmer, you need to master two things: (1) the systax of the programming language, and (2) the core libraries (i.e., API) associated with the language.

You may also try the "Exercises on Java Basics".

1. Basic Syntaxes

1.1 Revision

The steps in writing a Java program is illustrated as follows:



Step 1: Write the source codes (`.java`) using a programming text editor (such as Notepad++, Textpad, gEdit) or an IDE (such as Eclipse or NetBeans).

Step 2: Compile the source codes (`.java`) into Java portable bytecodes (`.class`) using the JDK compiler ("`javac`"). IDE (such as Eclipse or NetBeans) compiles the source codes automatically while they are entered.

Step 3: Run the compiled bytecodes (`.class`) with the input to produce the desired output, using the Java Runtime ("`java`").

Below is a simple Java program that demonstrates the three basic programming constructs: *sequential*, *loop*, and *conditional*. Read "Introduction To Java Programming for First-time Programmers" if you need help in understanding this program.

```
/*
 * 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;
      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) {
         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 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
```

1.2 Comments

Comments are used to document and explain your codes and program logic. Comments are not programming statements and are ignored by the compiler, but they VERY IMPORTANT for providing documentation and explanation for others to understand your program (and also for yourself three days later).

There are two kinds of comments in Java:

1. *Multi-line Comment*: begins with a `/*` and ends with a `*/`, and can span several lines.
2. *End-of-line Comment*: begins with `//` and lasts till the end of the current line.

I recommend that you use comments *liberally* to explain and document your codes. During program development, instead of deleting a chunk of statements irrevocably, you could *comment-out* these statements so that you could get them back later, if needed.

1.3 Statements and Blocks

Statement: A programming *statement* is the smallest indepedent unit in a program, just like a sentence in the english language. It performs *a piece of programming action*. A programming statement must be terminated by a semi-colon (`;`), just like an english sentence ends with a period. (Why not ends with a period like an english sentence? This is beacuse period crashes with decimal point - it is hard for the dumb computer to differentiate between period and decimal point!)

For examples,

```
// Each of the following lines is a programming statement, which ends with a semi-colon (;)
int number1 = 10;
int number2, number3=99;
int product;
product = number1 * number2 * number3;
System.out.println("Hello");
```

Block: A *block* (or a *compound statement*) is a group of statements surrounded by curly braces `{ }`. All the statements inside the block is treated as one single unit. Blocks are used as the *body* in constructs like class, method, if-else and for-loop, which may contain multiple statements but are treated as one unit. There is no need to put a semi-colon after the closing brace to end a compound statement. Empty block (no statement inside the braces) is permitted. For examples,

```
// Each of the followings is a "complex" statement comprising one or more blocks of statements.
// No terminating semi-colon needed after the closing brace to end the "complex" statement.
// Take note that a "complex" statement is usually written over a few lines for readability.
if (mark >= 50) {
```

```
   System.out.println("PASS");
   System.out.println("Well Done!");
   System.out.println("Keep it Up!");
}

if (number == 88) {
   System.out.println("Got it!");
} else {
   System.out.println("Try Again!");
}

i = 1;
while (i < 8) {
   System.out.print(i + " ");
   ++i;
}

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

1.4 White Spaces and Formatting Source Codes

White Spaces: *Blank*, *tab* and *newline* are collectively called *white spaces*. Java, like most of the computing languages, ignores *extra* white spaces. That is, multiple contiguous white spaces are treated as a *single* white space.

You need to use a white space to separate two keywords or tokens to avoid ambiguity, e.g.,

```
int sum=0;       // Cannot write intsum=0. Need at least one white space between "int" and "sum"
double average;  // Again, need at least a white space between "double" and "average"
```

Additional white spaces and extra lines are, however, ignored, e.g.,

```
// same as above
int  sum
 =  0      ;

   double  average  ;
```

Formatting Source Codes: As mentioned, extra white spaces are ignored and have no computational significance. However, proper indentation (with tabs and blanks) and extra empty lines greatly improves the readability of the program. This is extremely important for others (and yourself three days later) to understand your programs.

For example, the following one-line hello-world program works. But can you read and understand the program?

```
public class Hello{public static void main(String[] args){System.out.println("Hello, world!");}}
```

Braces: Java's convention is to place the beginning brace at the end of the line, and align the ending brace with the start of the statement.

Indentation: Indent each *level* of the body of a block by an extra 3 (or 4) spaces.

```
/*
 * Recommended Java programming style
 */
public class ClassName {      // Place the beginning brace at the end of the current line
   public static void main(String[] args) {  // Indent the body by an extra 3 or 4 spaces for each level

      // Use empty line liberally to improve readability
      // Sequential statements
      statement;
      statement;

      // Conditional statement
      if (test) {
         statements;
      } else {
         statements;
      }

      // loop
      init;
      while (test) {
         statements;
         update;
      }
   }
}   // ending brace aligned with the start of the statement
```

2. Variables and Types

2.1 Variables

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.

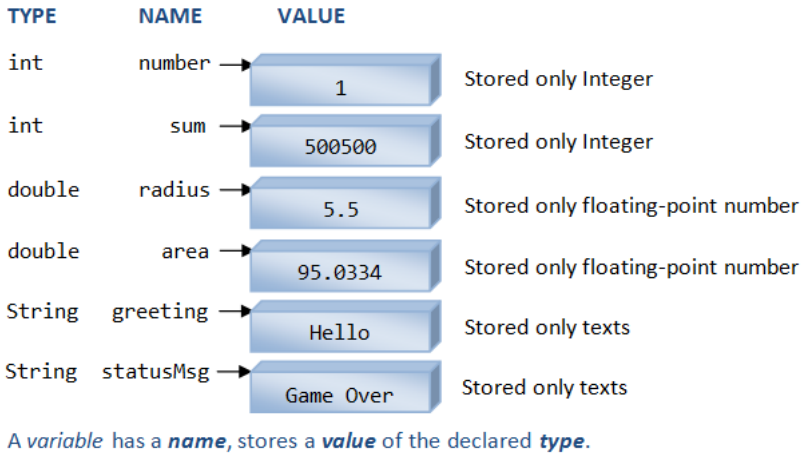
More precisely, a *variable* is a *named* storage location, that stores a *value* of a particular data *type*. In other words, a *variable* has a *name*, a *type* and stores a *value*.

- 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 (real numbers), such as `3.1416`, `-55.66`, having a decimal point and fractional part;
- `String`: for texts such as `"Hello"`, `"Good Morning!"`. Text strings are enclosed within a pair of double quotes.
- `char`: a single character, such as `'a'`, `'8'`. char is enclosed by single quotes;

- A variable can store a *value* of that particular data *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 in the early programming languages to simplify interpretation of data made up of binary numbers (`0`'s and `1`'s). The type determines the size and layout of the data, the range of its values, and the set of operations that can be applied.

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.



2.2 Identifiers

An *identifier* is needed to *name* a variable (or any other entity such as a method or a class). Java imposes the following *rules on identifiers*:

- An identifier is a sequence of characters, of any length, comprising uppercase and lowercase letters `(a-z, A-Z)`, digits `(0-9)`, underscore `"_"`, and dollar sign `"$"`.
- White space (blank, tab, newline) and other special characters (such as `+`, `-`, `*`, `/`, `@`, `&`, commas, etc.) are not allowed. Take note that blank and dash (`-`) are not allowed, i.e., `"max value"` and `"max-value"` are not valid names.
- An identifier must begin with a letter `(a-z, A-Z)` or underscore (`_`). It cannot begin with a digit `(0-9)`. Identifiers begin with dollar sign (`$`) are reserved for system-generated entities.
- An identifier cannot be a reserved keyword or a reserved literal (e.g., `class`, `int`, `double`, `if`, `else`, `for`, `true`, `false`, `null`).
- Identifiers are case-sensitive. A `rose` is NOT a `Rose`, and is NOT a `ROSE`.

Caution: Programmers don't use *blank* character in names. It is either not supported (e.g., in Java and C/C++), or will pose you more challenges.

Variable Naming Convention

A variable name is a noun, or a noun phrase made up of several words with no spaces between words. The first word is in lowercase, while the remaining words are initial-capitalized. For example, `thefontSize`, `roomNumber`, `xMax`, `yMin`, `xTopLeft` and `thisIsAVeryLongVariableName`. This convention is also known as *camel-case*.

Recommendations

1. It is important to choose a name that is *self-descriptive* and closely reflects the meaning of the variable, e.g., `numberOfStudents` or `numStudents`.
2. Do not use *meaningless* names like `a`, `b`, `c`, `d`, `i`, `j`, `k`, `i1`, `j99`.
3. Avoid *single-letter* names like `i`, `j`, `k`, `a`, `b`, `c`, which is easier to type but often meaningless. Exception are common names like `x`, `y`, `z` for coordinates, `i` for index. Long names are harder to type, but self-document your program. (I suggest you spend sometimes practicing your typing.)
4. Use *singular* and *plural* nouns prudently to differentiate between singular and plural variables. For example, you may use the variable `row` to refer to a single row number and the variable `rows` to refer to many rows (such as an array of rows - to be discussed later).

2.3 Variable Declaration

To use a variable in your program, you need to first "introduce" it by *declaring* its *name* and *type*, in one of the following syntaxes. The act of declaring a variable allocates a storage (of size capable of holding a value of the type).

Syntax	Example

`// Declare a variable of a specified type`
`type identifier;`
`// Declare multiple variables of the same type, separated by commas`
`type identifier1, identifier2, ..., identifierN;`
`// Declare a variable and assign an initial value`
`type identifier = initialValue;`
`// Declare multiple variables with initial values`
`type identifier1 = initValue1, ..., identifierN = initValueN;`
	`int option;`
	
	`double sum, difference, product, quotient;`
	
	`int magicNumber = 88;`
	
	`String greetingMsg = "Hi!", quitMsg = "Bye!";`

Take note that:

- Java is a "strongly type" language. A variable is declared with a *type*. Once the *type* of a variable is declared, it can only store a value belonging to 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"`.
- Each variable can only be declared once.

- You can declare a variable anywhere inside the program, as long as it is declared before used.
- The type of a variable cannot be changed inside the program.

2.4 Constants (`final` Variables)

Constants are *non-modifiable* variables, declared with keyword `final`. Their values cannot be changed during program execution. Also, constants must be initialized during declaration. For examples:

```
final double PI = 3.1415926;  // Need to initialize
```

Constant Naming Convention: Use uppercase words, joined with underscore. For example, `MIN_VALUE`, `MAX_SIZE`.

2.5 Expressions

An *expression* is a combination of *operators* (such as addition `'+'`, subtraction `'-'`, multiplication `'*'`, division `'/'`) and *operands* (variables or literals), that can be *evaluated to yield a single value of a certain type*. For example,

```
1 + 2 * 3           // evaluated to int 7

int sum, number;
sum + number        // evaluated to an int value

double principal, interestRate;
principal * (1 + interestRate)  // Evaluated to a double value
```

2.6 Assignment

An *assignment statement*:

1. assigns a literal value (of the RHS) to a variable (of the LHS), e.g., `x = 1`; or
2. evaluates an expression (of the RHS) and assign the resultant value to a variable (of the LHS), e.g., `x = (y + z) / 2`.

The syntax for assignment statement is:

Syntax	Example

`// Assign the literal value (of the RHS) to the variable (of the LHS)`
`variable` `=` *`literalValue`*`;`
`// Evaluate the expression (RHS) and assign the result to the variable (LHS)`
`variable` `=` *`expression`*`;`
	`number = 88;`
	
	`sum = sum + number;`

The assignment statement should be interpreted this way: The *expression* on the right-hand-side (RHS) is first evaluated to produce a resultant value (called *rvalue* or right-value). The *rvalue* is then assigned to the variable on the left-hand-side (LHS) or *lvalue*. Take note that you have to first evaluate the RHS, before assigning the resultant value to the LHS. For examples,

```
number = 8;           // Assign literal value of 8 to the variable number
number = number + 1;  // Evaluate the expression of number + 1,
                      //  and assign the resultant value back to the variable number
8 = number;           // INVALID
number + 1 = sum;     // INVALID
```

In programming, the equal symbol `'='` is known as the *assignment operator*. The meaning of `'='` in programming is different from Mathematics. It denotes *assignment of the LHS value to the RHS variable*, instead of *equality of the RHS and LHS*. The RHS shall be a literal value or an expression that evaluates to a value; while the LHS must be a variable.

Note that `x = x + 1` is valid (and often used) in programming. It evaluates `x + 1` and assign the resultant value to the variable `x`. `x = x + 1` illegal in Mathematics. While `x + y = 1` is allowed in Mathematics, it is invalid in programming (because the LHS of an assignment statement must be a variable). Some programming languages use symbol ":=", "->" or "<-" as the assignment operator to avoid confusion with equality.

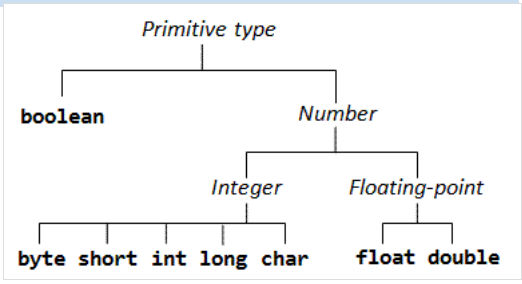
2.7 Primitive Types

In Java, there are two broad categories of *types*: *primitive types* (e.g., `int`, `double`) and *reference types* (e.g., objects and arrays). We shall describe the primitive types here and the reference types (classes and objects) in the later chapters on "Object-Oriented Programming".

TYPE	DESCRIPTION	
`byte`	Integer	8-bit signed integer The range is `[-2^7, 2^7-1] = [-128, 127]`
`short`		16-bit signed integer The range is `[-2^15, 2^15-1] = [-32768, 32767]`
`int`		32-bit signed integer The range is `[-2^31, 2^31-1] = [-2147483648, 2147483647]` (≈9 digits)
`long`		64-bit signed integer The range is `[-2^63, 2^63-1] = [-9223372036854775808, +9223372036854775807]` (≈19 digits)
`float`	Floating-Point Number	32-bit single precision floating-point number (≈6-7 significant decimal digits, in the range of `±[≈10^-45, ≈10^38]`)
`double`		64-bit double precision floating-point number (≈14-15 significant decimal digits, in the range of `±[≈10^-324, ≈10^308]`)
`char`	Character Represented in 16-bit Unicode `'\u0000'` to `'\uFFFF'`. Can be treated as 16-bit unsigned integers in the range of `[0, 65535]` in arithmetic operations.	

boolean	Binary Takes a value of either `true` or `false`. The size of `boolean` is not defined in the Java specification, but requires at least one bit.

Built-in Primitive Types



Primitive type are built-in to the languages. Java has eight *primitive types*, as listed in the above table:

- There are four integer types: 8-bit `byte`, 16-bit `short`, 32-bit `int` and 64-bit `long`. They are *signed integers* in *2's complement representation*, and can hold an integer value of the various ranges as shown in the table.
- There are two floating-point types: 32-bit single-precision `float` and 64-bit double-precision `double`, represented as specified by IEEE 754 standard. A `float` can represent a number between `±1.40239846×10^-45` and `±3.40282347×10^38`, approximated. A `double` can represented a number between `±4.94065645841246544×10^-324` and `±1.79769313486231570×10^308`, approximated. Take note that not all real numbers can be represented by `float` and `double`. This is because there are infinite real numbers even in a small range of `[1.1, 2.2]`, but there is a finite number of patterns in a n-bit representation. Many values are approximated.
- The type `char` represents a single character, such as `'0'`, `'A'`, `'a'`. In Java, char is represented using 16-bit Unicode (in UCS-2 format) to support internationalization (*i18n*). A `char` can be treated as a *16-bit unsigned integer* (in the range of `[0, 65535]`) in arithmetic operations. For example, character `'0'` is `48` (decimal) or `30H` (hexadecimal); character `'A'` is `65` (decimal) or `41H` (hexadecimal); character `'a'` is `97` (decimal) or `61H` (hexadecimal).
- Java introduces a new *binary* type called "`boolean`", which takes a value of either `true` or `false`.

Example: The following program can be used to print the *maximum*, *minimum* and *bit-length* of the primitive types. The maximum, minimum and bit-size of `int` are kept in constans `INTERER.MIN_VALUE`, `INTEGER.MAX_VALUE`, `INTEGER.SIZE`.

```
// Print the minimum, maximum and bit-length for primitive types
public class PrimitiveTypesMinMax {
   public static void main(String[] args) {
      // int (32-bit signed integer)
      System.out.println("int(min) = " + Integer.MIN_VALUE);
      System.out.println("int(max) = " + Integer.MAX_VALUE);
      System.out.println("int(bit-length) = " + Integer.SIZE);
      // byte (8-bit signed integer)
      System.out.println("byte(min) = " + Byte.MIN_VALUE);
      System.out.println("byte(max) = " + Byte.MAX_VALUE);
      System.out.println("byte(bit-length)=" + Byte.SIZE);
      // short (16-bit signed integer)
      System.out.println("short(min) = " + Short.MIN_VALUE);
      System.out.println("short(max) = " + Short.MAX_VALUE);
      System.out.println("short(bit-length) = " + Short.SIZE);
      // long (64-bit signed integer)
      System.out.println("long(min) = " + Long.MIN_VALUE);
      System.out.println("long(max) = " + Long.MAX_VALUE);
      System.out.println("long(bit-length) = " + Long.SIZE);
      // char (16-bit character or 16-bit unsigned integer)
      System.out.println("char(min) = " + (int)Character.MIN_VALUE);
      System.out.println("char(max) = " + (int)Character.MAX_VALUE);
      System.out.println("char(bit-length) = " + Character.SIZE);
      // float (32-bit floating-point)
      System.out.println("float(min) = " + Float.MIN_VALUE);
      System.out.println("float(max) = " + Float.MAX_VALUE);
      System.out.println("float(bit-length) = " + Float.SIZE);
      // double (64-bit floating-point)
      System.out.println("double(min) = " + Double.MIN_VALUE);
      System.out.println("double(max) = " + Double.MAX_VALUE);
      System.out.println("double(bit-length) = " + Double.SIZE);
   }
}
```

```
int(min) = -2147483648
int(max) = 2147483647
int(bit-length) = 32
byte(min) = -128
byte(max) = 127
byte(bit-length)=8
short(min) = -32768
short(max) = 32767
short(bit-length) = 16
long(min) = -9223372036854775808
long(max) = 9223372036854775807
long(bit-length) = 64
char(min) = 0
char(max) = 65535
char(bit-length) = 16
float(min) = 1.4E-45
float(max) = 3.4028235E38
float(bit-length) = 32
double(min) = 4.9E-324
double(max) = 1.7976931348623157E308
double(bit-length) = 64
```

(Advanced) Furthermore, the *constants* `Double.NaN` (Not-a-number, e.g., `0.0/0.0`), `Double.POSITIVE_INFINITY` (e.g., `1.0/0.0`), `Double.NEGATIVE_INFINITY` (e.g., `-1.0/0.0`), `Float.NaN`, `Float.POSITIVE_INFINITY`, `Float.NEGATIVE_INFINITY` denote the special values in the IEEE 754 floating-point representation.

String

Another commonly-used type is `String`, which represents texts (a sequence of characters) such as `"Hello, world"`. `String` is not a primitive type, and will be further elaborated later. In Java, a `char` is enclosed by single quotes (e.g., `'A'`, `'0'`), while a `String` is enclosed by double quotes (e.g., `"Hello"`). For example,

```
String message = "Hello, world!"; // strings are enclosed in double-quotes
char gender = 'm';                // char is enclosed in single-quotes
```

Choice of Data Types for Variables

As a programmer, you need to choose variables and decide on the type of the variables to be used in your programs. Most of the times, the decision is intuitive. For example, use an integer type for counting and whole number; a floating-point type for number with fractional part, `String` for text message, `char` for a single character, and `boolean` for binary outcomes.

Rules of Thumb

- Use `int` for integer and `double` for floating point numbers. Use `byte`, `short`, `long` and `float` only if you have a good reason to choose that specific precision.
- Use `int` for *counting* and *indexing*, NOT floating-point type (`float` or `double`). This is because integer type are precise and more efficient in operations.
- Use an integer type if possible. Use a floating-point type only if the number contains a fractional part.

Data Representation

Read "Data Representation" if you wish to understand how the numbers and characters are represented inside the computer memory. In brief, It is important to take note that char `'1'` is different from `int 1`, `byte 1`, `short 1`, `float 1.0`, `double 1.0`, and `String "1"`. They are represented differently in the computer memory, with different precision and interpretation. For example, `byte 1` is `"00000001"`, `short 1` is `"00000000 00000001"`, `int 1` is `"00000000 00000000 00000000 00000001"`, `long 1` is `"00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000001"`, `float 1.0` is `"0 01111111 0000000 00000000 00000000"`, `double 1.0` is `"0 01111111111 0000 00000000 00000000 00000000 00000000 00000000 00000000"`, `char '1'` is `"00000000 00110001"`, and `String "1"` is a complex object.

There is a subtle difference between `int 0` and `double 0.0`.

Furthermore, you MUST know the type of a value before you can interpret a value. For example, this value `"00000000 00000000 00000000 00000001"` cannot be interpreted unless you know the type.

Example (Variable Names and Types): Paul has bought a new notebook of brand "abc", with a processor speed of 3.2GHz, 4 GB of RAM, 500GB hard disk, with a 15-inch monitor, for $1650.45. He has chosen service plan 'B' among plans 'A', 'B' and 'C, plus on-site servicing. Identify the data types and name the variables.

Possible variable names and types are:

```
String name = "Paul";
String brand = "abc";
float processorSpeedInGHz = 3.2;  // or double
float ramSizeInGB = 4;            // or double
short harddiskSizeInGB = 500;     // or int
byte monitorInInch = 15;          // or int
double price = 1650.45;
char servicePlan = 'B';
boolean onSiteService = true;
```

Example (Variable Names and Types): You are asked to develop a software for a college. The system shall maintain information about students. This includes name, address, phone number, gender, date of birth, height, weight, degree pursued (e.g., B.Sc., B.A.), year of study, average GPA, with/without tuition grant, is/is not a president's scholar. Each student is assigned a unique 8-digit number as id. Identify the variables. Assign a suitable name to each variable and choose an appropriate type. Write the variable declaration statements.

2.8 Literals for Primitive Types and String

A *literal*, or *literal constant*, is a *specific constant value* or *raw data*, such as `123`, `-456`, `3.14`, `'a'`, `"Hello"`, that is used in the program source. It can be assigned directly to a variable; or used as part of an expression. They are called *literals* because they literally and explicitly identify their values. We call it *literal* to distinguish it from a *variable*.

Integer literals

A whole number, such as `123` and `-456`, is treated as an `int` by default. In Java, the range of 32-bit `int` literals is `-2,147,483,628` (`-2^31`) to `2,147,483,627` (`2^31-1`). For example,

```
int number = -123;
int sum = 1234567890;     // This value is within the range of int
int bigSum = 8234567890;  // ERROR: this value is outside the range of int
```

An `int` literal may precede with a plus (`+`) or minus (`-`) sign, followed by digits. No commas or special symbols (e.g., `$` or space) is allowed (e.g., `1,234` and `$123` are invalid). No preceding `0` is allowed too (e.g., `007` is invalid).

You can use a prefix `'0'` (zero) to denote a value in octal, and prefix `'0x'` (or `'0X'`) for a value in hexadecimal, e.g.,

```
int number = 1234;   // Decimal
int number = 01234;  // Octal 1234, Decimal 2322
int number = 0x1abc; // hexadecimal 1ABC, decimal 15274
```

(JDK 1.7) From JDK 7, you can use prefix '`0b`' or '`0B`' to specify a value in binary. You are also permitted to use underscore (`_`) to break the digits into groups to improve the readability. But you must start and end the literal with a digit. For example,

```
int number1 = 0b01010000101000101101000010100010;
int number2 = 0b0101_0000_1010_0010_1101_0000_1010_0010;  // break the digits with underscore (JDK 1.7)
int number3 = 2_123_456;  // break the digits with underscore (JDK 1.7)
```

A `long` literal above the `int` range requires a suffix `'L'` or `'l'` (avoid lowercase, which can be confused with the number one), e.g., `123456789012L`, `-98765432101`. In Java, the range of 64-bit `long` literals is `-9,223,372,036,854,775,808L` (`-2^63`) to `9,223,372,036,854,775,807L` (`2^63-1`). For example,

```
long bigNumber = 1234567890123L;  // Suffix 'L' needed
long sum = 123;                   // int 123 auto-casts to long 123L
```

No suffix is needed for `byte` and `short` literals. But you can only use integer values in the permitted range. For example,

```
byte smallNumber = 12345;     // ERROR: this value is outside the range of byte.
```

```
byte smallNumber = 123;       // This is within the range of byte
short midSizeNumber = -12345;
```

Floating-point literals

A number with a decimal point, such as `55.66` and `-33.44`, is treated as a `double`, by default. You can also express them in scientific notation, e.g., `1.2e34`, `-5.5E-6`, where `e` or `E` denotes the exponent in power of 10. You could precede the fractional part or exponent with a plus (`+`) or minus (`-`) sign. Exponent values are restricted to integer. There should be no space or other characters in the number.

You can *optionally* use suffix `'d'` or `'D'` to denote `double` literals.

You MUST use a suffix of `'f'` or `'F'` for `float` literals, e.g., `-1.2345F`. For example,

```
float average = 55.66;      // Error! RHS is a double. Need suffix 'f' for float.
float average = 55.66f;
```

Character Literals and Escape Sequences

A printable `char` literal is written by enclosing the character with a pair of *single quotes*, e.g., `'z'`, `'$'`, and `'9'`. In Java, characters are represented using 16-bit Unicode, and can be treated as a 16-bit *unsigned integer*s in arithmetic operations. In other words, `char` and 16-bit unsigned integer are interchangeable. You can also assign an integer in the range of `[0, 65535]` to a `char` variable.

For example,

```
char letter = 'a';                // Same as 97
char anotherLetter = 98;          // Same as the letter 'b'
System.out.println(letter);       // 'a' printed
System.out.println(anotherLetter); // 'b' printed instead of the number
anotherLetter += 2;               // 100 or 'd'
System.out.println(anotherLetter); // 'd' printed
```

Non-printable and control characters can be represented by a so-called *escape sequence*, which begins with a back-slash (`\`). The commonly-used escape sequences are:

Escape Sequence	Description	Unicode (Decimal)
`\n`	newline (or Line-feed)	`000AH (10D)`
`\r`	Carriage-return	`000DH (13D)`
`\t`	Tab	`0009H (9D)`
`\"`	Double-quote	`0022H (34D)`
`\'`	Single-quote	`0027H (39D)`
`\\`	Back-slash	`005CH (92D)`
`\uhhhh`	Unicode number *hhhh* (in hex), e.g., `\u000a` is newline, `\u60a8` is 您, `\u597d` is 好	*hhhh*H

Notes:

- newline (`000AH`) and carriage return (`000dH`), represented by `\n`, and `\r` respectively, are used as *line delimiter* (or *end-of-line*, or *EOL*). Take note that Unixes use `\n` as EOL, Windows use `\r\n`, while Macs use `\r`.
- Horizontal Tab (`0009H`) is represented as `\t`.
- To resolve ambiguity, characters back-slash (`\`), single-quote (`'`) and double-quote (`"`) are represented using escape sequences `\\`, `\'` and `\"`, respectively. This is because a single back-slash begins an escape sequence, while single-quotes and double-quotes are used to enclose character and string.
- Other less commonly-used escape sequences are: `\?` or `?`, `\a` for alert or bell, `\b` for backspace, `\f` for form-feed, `\v` for vertical tab. These may not be supported in some consoles.

String Literals

A `String` literal is composed of zero of more characters surrounded by a pair of *double quotes*, e.g., `"Hello, world!"`, `"The sum is "`. For example,

```
String directionMsg = "Turn Right";
String greetingMsg = "Hello";
String statusMsg = "";              // empty string
```

String literals may contains escape sequences. Inside a `String`, you need to use `\"` for double-quote to distinguish it from the ending double-quote, e.g. `"\"quoted\""`. Single quote inside a `String` does not require escape sequence. For example,

```
System.out.println("Use \\\" to place%n a \" within\ta\tstring");
```

```
Use \" to place
 a " within     a       string
```

TRY: Write a program to print the following animal picture using mutlitple `System.out.println()`. Take note that you need to use escape sequences to print special characters.

```
          '__'
          (oo)
   +========\/
  / || %%% ||
 *  ||-----||
    ""     ""
```

"boolean" Literals

There are only two `boolean` literals, i.e., `true` and `false`. For example,

```
boolean done = true;
```

```
boolean gameOver = false;
```

Example on Literals

```
public class LiteralTest {
   public static void main(String[] args) {
      String name = "Tan Ah Teck"; // String is double-quoted
      char gender = 'm';           // char is single-quoted
      boolean isMarried = true;    // true or false
      byte numChildren = 8;        // Range of byte is [-127, 128]
      short yearOfBirth = 1945;    // Range of short is [-32767, 32768]. Beyond byte
      int salary = 88000;          // Beyond the ranges of byte and short
      long netAsset = 8234567890L; // Need suffix 'L' for long. Beyond int
      double weight = 88.88;       // With fractional part
      float gpa = 3.88f;           // Need suffix 'f' for float

      // println() can be used to print value of any type
      System.out.println("Name is " + name);
      System.out.println("Gender is " + gender);
      System.out.println("Is married is " + isMarried);
      System.out.println("Number of children is " + numChildren);
      System.out.println("Year of birth is " + yearOfBirth);
      System.out.println("Salary is " + salary);
      System.out.println("Net Asset is " + netAsset);
      System.out.println("Weight is " + weight);
      System.out.println("GPA is " + gpa);
   }
}
```

```
Name is Tan Ah Teck
Gender is m
Is married is true
Number of children is 8
Year of birth is 1945
Salary is 88000
Net Asset is 1234567890
Weight is 88.88
Height is 188.8
```

3. Operations

3.1 Arithmetic Operators

Java supports the following arithmetic operators:

Operator	Description	Usage	Examples
`*`	Multiplication	*expr1* `*` *expr2*	`2 * 3` → `6` `3.3 * 1.0` → `3.3`
`/`	Division	*expr1* `/` *expr2*	`1 / 2` → `0` `1.0 / 2.0` → `0.5`

`%`	Remainder (Modulus)	*expr1* `%` *expr2*	`5 % 2` → `1`
			`-5 % 2` → `-1`
			`5.5 % 2.2` → `1.1`

`+`	Addition (or unary positive)	*expr1* `+` *expr2* `+`*expr*	`1 + 2` → `3` `1.1 + 2.2` → `3.3`
`-`	Subtraction (or unary negate)	*expr1* `-` *expr2* `-`*expr*	`1 - 2` → `-1` `1.1 - 2.2` → `-1.1`

All these operators are *binary* operators, i.e., they take two operands. However, '+' and '-' can also be interpreted as *unary* "positive" and "negative" operators. For example,

```
int number = -88;  // negate
int x = +5;        // '+' optional
```

3.2 Arithmetic Expressions

In programming, the following arithmetic expression:

$$\frac{1+2a}{3}+\frac{4(b+c)(5-d-e)}{f}-6\left(\frac{7}{g}+h\right)$$

must be written as `(1+2*a)/3 + (4*(b+c)*(5-d-e))/f - 6*(7/g+h)`. You cannot omit the multiplication symbol (`*`), as in Mathematics.

Like Mathematics, the multiplication (`*`), division (`/`) and ramainder (`%`) take precedence over addition (`+`) and subtraction (`-`). Unary `'+'` (positive) and `'-'` (negate) have higher precedence. Parentheses `()` have the highest precedence and can be used to change the order of evaluation. Within the same precedence level (e.g., addition and subtraction), the expression is evaluated from left to right (called *left-associative*). For example, `1+2-3+4` is evaluated as `((1+2)-3)+4` and `1*2%3/4` is `((1*2)%3)/4`.

3.3 Mixed-Type Operations

The arithmetic operators are only applicable to *primitive numeric types*: `byte`, `short`, `int`, `long`, `float`, `double`, and `char`. These operators do not apply to `boolean`.

If both operands are `int`, `long`, `float` or `double`, the arithmetic operations are carried in that type, and evaluated to a value of that type, i.e., `int 5 + int 6` → `int 11`; `double 2.1 + double 1.2` → `double 3.3`.

It is important to take note `int` division produces an `int`, i.e., `int/int → int`, with the result *truncated*, e.g., `1/2 → 0`, instead of `0.5`?!

If both operand are `byte`, `short` or `char`, the operations are carried out in `int`, and evaluated to a value of `int`. A `char` is treated as a 16-bit *unsigned* integer in the range of `[0, 65535]`. For example, `byte 127 + byte 1 → int 127 + int 1 → int 128`.

If the two operands belong to *different types*, the value of the *smaller* type is promoted automatically to the *larger* type (known as *implicit type-casting*). The operation is then carried out in the *larger* type, and evaluated to a value in the *larger* type.

- `byte`, `short` or `char` is first promoted to `int` before comparing with the type of the other operand.
- The order of promotion is: `int → long → float → double`.

For examples,

1. `int/double → double/double → double`. Hence, `1/2 → 0`, `1.0/2.0 → 0.5`, `1.0/2 → 0.5`, `1/2.0 → 0.5`.
2. `char + float → int + float → float + float → float`.
3. `9 / 5 * 20.1 → (9 / 5) * 20.1 → 1 * 20.1 → 1.0 * 20.1 → 20.1` (You probably don't expect this answer!)
4. `byte 1 + byte 2 → int 1 + int 2 → int 3` (The result is an `int`, NOT `byte`!)

```
byte b1 = 1, b2 = 2;
byte b3 = b1 + b2;  // Compliation Error: possible loss of precision
                    // b1+b2 returns an int, cannot be assigned to byte
```

The type-promotion rules for *binary* operations can be summarized as follows:

1. If one of the operand is `double`, the other operand is promoted to `double`;
2. Else If one of the operand is `float`, the other operand is promoted to `float`;
3. Else If one of the operand is `long`, the other operand is promoted to `long`;
4. Else both operands are promoted to `int`.

The type-promotion rules for *unary* operations (e.g., negate `'-'`) can be summaried as follows:

1. If the operand is `double`, `float`, `long` or `int`, there is no promotion.
2. Else (the operand is `byte`, `short`, `char`), the operand is promted to `int`.

For example,

```
byte b1 = 1;
byte b2 = -b1;  // Compilation Error: possible loss of precision
                // -b1 returns an int, cannot be assigned to byte
```

Remainder (Modulus)

To evaluate the remainder (for negative and floating-point operands), perform repeated subtraction until the absolute value of the remainder is less than the absolute value of the second operand.

For example,

- `-5 % 2 ⇒ -3 % 2 ⇒ -1`
- `5.5 % 2.2 ⇒ 3.3 % 2.2 ⇒ 1.1`

Take note that Java does not have an exponent operator (`'^'` is exclusive-or, not exponent).

3.4 Overflow/Underflow

Study the output of the following program:

```
public class OverflowTest {
   public static void main(String[] args) {
      // Range of int is [-2147483648, 2147483647]
      int i1 = 2147483647;  // maximum int
      System.out.println(i1 + 1);   // -2147483648 (overflow)
      System.out.println(i1 + 2);   // -2147483647
      System.out.println(i1 * i1);  // 1

      int i2 = -2147483648;  // minimum int
      System.out.println(i2 - 1);   // 2147483647 (underflow)
      System.out.println(i2 - 2);   // 2147483646
      System.out.println(i2 * i2);  // 0
   }
}
```

In arithmetic operations, the resultant value *wraps around* if it exceeds its range (i.e., overflow). Java runtime does NOT issue an error/warning message but produces an *incorrect* result.

On the other hand, integer division produces an truncated integer and results in so-called *underflow*. For example, `1/2` gives `0`, instead of `0.5`. Again, Java runtime does NOT issue an error/warning message, but produces an *imprecise* result.

It is important to take note that *checking of overflow/underflow is the programmer's responsibility*. i.e., your job!!!

Why computer does not flag overflow/underflow as error? It is due to the legacy design when the processors were very slow. Checking for overflow/underflow consumes computation power. Today, processors are fast. It is better to ask the computer to check for overflow/underflow (if you design a new language), because few humans expect such results.

To check for arithmetic overflow (known as *secure coding*) is tedious. Google for "INT32-C. Ensure that operations on signed integers do not result in overflow" @ www.securecoding.cert.org.

3.5 Type-Casting

In Java, you will get a *compilation error* if you try to assign a `double` value of to an `int` variable. This is because the fractional part would be lost. The compiler issues an error "possible loss in precision". For example,

```
double f = 3.5;
int i;
i = f;            // Compliation Error: possible loss of precision (assigning a double value to an int variable)
int sum = 55.66f; // Compliation Error: possible loss of precision (assigning a float value to an int variable)
```

Explicit Type-Casting and Type-Casting Operator

To assign the a `double` value to an `int` variable, you need to invoke the so-called *type-casting operator* - in the form of prefix `(int)` - to operate on the `double` operand and return a *truncated* value in `int` type. In other words, you concisely perform the truncation. You can then assign the truncated `int` value to the `int` variable. For example,

```
double f = 3.5;
int i;
i = (int) f;    // Cast double value of 3.5 to int 3. Assign the resultant value 3 to i
                // Casting from double to int truncates.
```

Type-casting forces an explicit conversion of the type of a value. Type-casting is an operation which takes one operand. It operates on its operand, and returns an equivalent value in the specified type. Take note that it is an operation that yield a resultant value, similar to an addition operation although addition involves two operands.

There are two kinds of type-casting in Java:

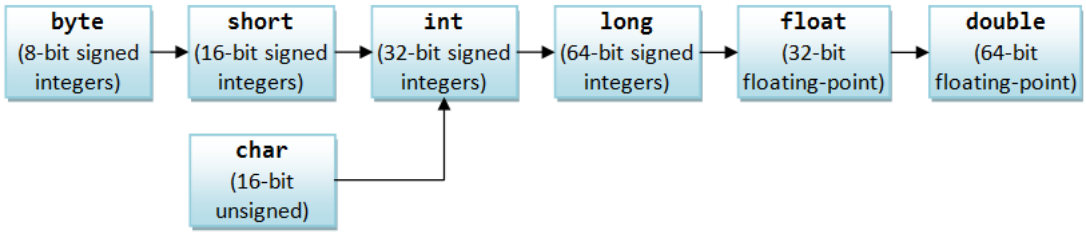
1. Explicit type-casting via a type-casting operator in the prefix form of `(new-type)operand`, as described above, and
2. Implicit type-casting performed by the compiler automatically, if there will be no loss of precision.

Implicit Type-Casting in Assignment

Explicit type-casting is not required if you assign an `int` value to a `double` variable, because there is no loss of precision. The compiler will perform the type-casting automatically (i.e., implicit type-casting). For example,,

```
int i = 3;
double f;
f = i;           // OK, no explicit type casting required
                 // f = 3.0
f = (double) i;  // Explicit type casting operator used here
double aDouble = 55;   // Compiler auto-casts to 55.0
double nought = 0;     // Compiler auto-casts to 0.0, int 0 and double 0.0 is different.
double average = (double)sum / count;  // Assume sum and count are int
```

The following diagram shows the order of implicit type-casting performed by compiler. The rule is to promote the smaller type to a bigger type to prevent loss of precision, known as widening conversion. Narrowing conversion requires explicit type-cast to inform the compiler that you are aware of the possible loss of precision. Take note that `char` is treated as an 16-bit unsigned integer in the range of `[0, 65535]`. `boolean` value cannot be type-casted (i.e., converted to non-`boolean`).



Orders of Implicit Type-Casting for Primitives

Example: Suppose that you want to find the average (in `double`) of the integers between `1` and `100`. Study the following codes:

```
1  public class Sum1To100 {
2     public static void main(String[] args) {
3        int sum = 0;
4        double average;
5        int number = 1;
6        while (number <= 100) {
7           sum += number;      // Final sum is int 5050
8           ++number;
9        }
10       average = sum / 100;   // Won't work (average = 50.0 instead of 50.5)
11       System.out.println("Average is " + average);  // Average is 50.0
12    }
13 }
```

This is because both the `sum` and `100` are `int`. The result of division is an `int`, which is then implicitly casted to `double` and assign to the `double` variable `average`. To get the correct answer, you can do either:

```
average = (double)sum / 100;     // Cast sum from int to double before division
average = sum / (double)100;     // Cast 100 from int to double before division
average = sum / 100.0;
average = (double)(sum / 100);   // Won't work. why?
```

3.6 Compound Assignment Operators

Besides the usual simple assignment operator `'='` described earlier, Java also provides the so-called *compound assignment operator*s as listed:

Operation	Description	Usage	Example
=	Assignment Assign the value of the LHS to the variable at the RHS	*var = expr*	`x = 5;`

+=	Compound addition and assignment	*var* += *expr* same as *var* = *var* + *expr*	x += 5; same as x = x + 5
-=	Compound subtraction and assignment	*var* -= *expr* same as *var* = *var* - *expr*	x -= 5; same as x = x - 5
*=	Compound multiplication and assignment	*var* *= *expr* same as *var* = *var* * *expr*	x *= 5; same as x = x * 5
/=	Compound division and assignment	*var* /= *expr* same as *var* = *var* / *expr*	x /= 5; same as x = x / 5
%=	Compound remainder (modulus) and assignment	*var* %= *expr* same as *var* = *var* % *expr*	x %= 5; same as x = x % 5

3.7 Increment/Decrement

Java supports these *unary* arithmetic operators: increment '++' and decrement '--' for all numeric primitive types (byte, short, char, int, long, float and double, except boolean).

Operator	Description	Example

++	Increment the value of the variable by 1
	same as x += 1 or x = x + 1
		int x = 5;
		x++;
		++x;

--	Decrement the value of the variable by 1 same as x -= 1 or x = x - 1	int y = 6; y--; --x;

The increment (++) and decrement (--) operate on its operand and store the result back to its operand. For example, x++ retrives x, increment and stores the result back to x. Writing x = x++ is a logical error!!

Unlike other unary operator (such as negate '-') which promotes byte, short and char to int, the increment and decrement do not promote its operand (because there is no such need).

The increment/decrement unary operator can be placed before the operand (prefix), or after the operands (postfix), which affects its *resultant value*.

- If these operators are used by themselves (e.g., i++ or ++i), the outcomes are the same for pre- and post-operators, because the resultant values are discarded.
- If '++' or '--' involves another operation (e.g., y=x++ or y=++x), then pre- or post-order is important to specify the order of the two operations:

Operator	Description	Example
++var	Pre-Increment Increment *var*, then use the new value of *var*	y = ++x; same as x=x+1; y=x;
var++	Post-Increment Use the old value of *var*, then increment *var*	y = x++; same as oldX=x; x=x+1; y=oldX;
--var	Pre-Decrement	y = --x; same as x=x-1; y=x;
var--	Post-Decrement	y = x--; same as oldX=x; x=x-1; y=oldX;

For examples,

```
x = 5;
System.out.println(x++);  // Print x (5), then increment x (=6). Output is 5. (x++ returns the oldX.)
x = 5;
System.out.println(++x);  // Increment x (=6), then print x (6). Output is 6. (++x returns x+1.)
```

Prefix operator (e.g, ++i) could be more efficient than postfix operator (e.g., i++)?!

3.8 Relational and Logical Operators

Very often, you need to compare two values before deciding on the action to be taken, e.g. if mark is more than or equals to 50, print "PASS!".

Java provides six *comparison operators* (or *relational operators*):

Operator	Description	Usage	Example (x=5, y=8)
==	Equal to	*expr1* == *expr2*	(x == y) → false
!=	Not Equal to	*expr1* != *expr2*	(x != y) → true
>	Greater than	*expr1* > *expr2*	(x > y) → false
>=	Greater than or equal to	*expr1* >= *expr2*	(x >= 5) → true
<	Less than	*expr1* < *expr2*	(y < 8) → false
<=	Less than or equal to	*expr1* >= *expr2*	(y <= 8) → true

In Java, these comparison operations returns a boolean value of either true or false.

Each comparison operation involves two operands, e.g., x <= 100. It is invalid to write 1 < x < 100 in programming. Instead, you need to break out the two comparison operations x > 1, x < 100, and join with with a logical AND operator, i.e., (x > 1) && (x < 100), where && denotes AND operator.

Java provides four logical operators, which operate on boolean operands only, in descending order of precedences, as follows:

Operator	Description	Usage

!	Logical NOT	*!booleanExpr*
^	Logical XOR	*booleanExpr1 ^ booleanExpr2*
&&	Logical AND	*booleanExpr1 && booleanExpr2*
||	Logical OR	*booleanExpr1 || booleanExpr2*

The truth tables are as follows:

AND (&&)	true	false
true	true	false
false	false	false

OR (||)	true	false
true	true	true
false	true	false

NOT (!)	true	false
Result	false	true

XOR (^)	true	false
true	false	true
false	true	false

Example:

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

// Return true if x is outside 0 and 100 (inclusive)
(x < 0) || (x > 100)   //or
!((x >= 0) && (x <= 100))

// 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)
```

Exercise: Study the following program, and explain its output.

```
public class RelationalLogicalOpTest {
   public static void main(String[] args) {
      int age = 18;
      double weight = 71.23;
      int height = 191;
      boolean married = false;
      boolean attached = false;
      char gender = 'm';

      System.out.println(!married && !attached && (gender == 'm'));
      System.out.println(married && (gender == 'f'));
      System.out.println((height >= 180) && (weight >= 65) && (weight <= 80));
      System.out.println((height >= 180) || (weight >= 90));
   }
}
```

Write an expression for all unmarried male, age between 21 and 35, with height above 180, and weight between 70 and 80.

Exercise: Given the year, month (1-12), and day (1-31), write a boolean expression which returns true for dates before October 15, 1582 (Gregorian calendar cut over date).

Ans: `(year < 1582) || (year == 1582 && month < 10) || (year == 1582 && month == 10 && day < 15)`

Operator Precedence

The precedence from highest to lowest is: `'!'` (unary), `'^'`, `'&&'`, `'||'`. But when in doubt, use parenthese!

```
System.out.println(true || true && false);    // true (same as below)
System.out.println(true || (true && false));  // true
System.out.println((true || true) && false);  // false

System.out.println(false && true ^ true);     // false (same as below)
System.out.println(false && (true ^ true));   // false
System.out.println((false && true) ^ true);   // true
```

Short-Circuit Operation

The logical AND (`&&`) and OR (`||`) operators are known as short-circuit operators, meaning that the right operand will not be evaluated if the result can be determined by the left operand. For example, `false && ...` gives `false`; `true || ...` give `true` without evaluating the right operand. This may have adverse consequences if you rely on the right operand to perform certain operations, e.g. `fase && (++i < 5)` but `++i` will not be evaluated.

4. Strings

A `String` is a sequence of characters. A string literal is surrounded by a pair of double quotes, e.g.,

```
String s1 = "Hi, This is a string!"  // String literals are enclosed in double quotes
String s2 = ""                       // An empty string
```

You need to use an escape sequence for special control characters (such as newline `\n` and tab `\t`), double-quote `\"` and backslash `\\` (due to conflict) and Unicode character `\uhhhh` (if your editor does not support Unicode input), e.g.,

```
String s3 = "A \"string" nested \\inside\\ a string"
String s4 = "Hello, \u60a8\u597d!"   // "Hello, 您好!"
```

Single-quote (`'`) does not require an escape sign.

```
String s5 = "Hi, I'm a string!"      // Single quote OK
```

4.1 `String` and `'+'` Operator

In Java, `'+'` is a special operator. It is *overloaded*. *Overloading* means that it carries out different operations depending on the types of its two operands.

- If both operands are numbers (`byte`, `short`, `int`, `long`, `float`, `double`, `char`), `'+'` performs the usual *addition*, e.g.,

  ```
  1 + 2 → 3
  1.2 + 2.2 → 3.4
  1 + 2.2 → 1.0 + 2.2 → 3.2
  ```

- If both operands are `String`s, `'+'` *concatenates* the two Strings and returns the concatenated `String`. E.g.,

  ```
  "Hello" + "world" → "Helloworld"
  "Hi" + ", " + "world" + "!" → "Hi, world!"
  ```

- If one of the operand is a `String` and the other is numeric, the numeric operand will be converted to `String` and the two `String`s concatenated, e.g.,

  ```
  "The number is " + 5 → "The number is " + "5" → "The number is 5"
  "The average is " + average + "!" (suppose average=5.5) → "The average is " + "5.5" + "!" → "The average is 5.5!"
  "How about " + a + b (suppose a=1, b=1) → "How about 11"
  ```

4.2 `String` Operations

The most commonly-used `String` methods are:

- `length()`: return the length of the string.
- `charAt(int index)`: return the `char` at the `index` position (index begins at `0` to `length()-1`).
- `equals()`: for comparing the contents of two strings. Take note that you cannot use `"=="` to compare two strings.

For examples,

```
String str = "Java is cool!";
System.out.println(str.length());       // return int 13
System.out.println(str.charAt(2));      // return char 'v'
System.out.println(str.charAt(5));      // return char 'i'

// Comparing two Strings
String anotherStr = "Java is COOL!";
System.out.println(str.equals(anotherStr));           // return boolean false
System.out.println(str.equalsIgnoreCase(anotherStr)); // return boolean true
System.out.println(anotherStr.equals(str));           // return boolean false
System.out.println(anotherStr.equalsIgnoreCase(str)); // return boolean true
// (str == anotherStr) to compare two Strings is WRONG!!!
```

To check all the available methods for `String`, open JDK API documentation ⇒ select *package* "`java.lang`" ⇒ select *class* "`String`" ⇒ choose *method*. For examples,

```
String str = "Java is cool!";
System.out.println(str.length());            // return int 13
System.out.println(str.charAt(2));           // return char 'v'
System.out.println(str.substring(0, 3));     // return String "Jav"
System.out.println(str.indexOf('a'));        // return int 1
System.out.println(str.lastIndexOf('a'));    // return int 3
System.out.println(str.endsWith("cool!"));   // return boolean true
System.out.println(str.toUpperCase());       // return a new String "JAVA IS COOL!"
System.out.println(str.toLowerCase());       // return a new String "java is cool!"
```

4.3 `String`/Primitive Conversion

"`String`" to "`int/byte/short/long`": You could use the JDK built-in methods `Integer.parseInt(anIntStr)` to convert a `String` containing a valid integer literal (e.g., "`1234`") into an `int` (e.g., `1234`). The runtime triggers a `NumberFormatException` if the input string does not contain a valid integer literal (e.g., "`abc`"). For example,

```
String inStr = "5566";
int number = Integer.parseInt(inStr);   // number <- 5566
// Input to Integer.parseInt() must be a valid integer literal
number = Integer.parseInt("abc");       // Runtime Error: NumberFormatException
```

Similary, you could use methods `Byte.parseByte(aByteStr)`, `Short.parseShort(aShortStr)`, `Long.parseLong(aLongStr)` to convert a string containing a valid `byte`, `short` or `long` literal to the primitive type.

"String" to "double/float": You could use `Double.parseDouble(aDoubleStr)` or `Float.parseFloat(aFloatStr)` to convert a `String` (containing a floating-point literal) into a `double` or `float`, e.g.

```
String inStr = "55.66";
float aFloat = Float.parseFloat(inStr);         // aFloat <- 55.66f
double aDouble = Double.parseDouble("1.2345");  // aDouble <- 1.2345
aDouble = Double.parseDouble("abc");            // Runtime Error: NumberFormatException
```

"String" to "char": You can use `aStr.charAt(index)` to extract individual character from a `String`, e.g.,

```
// Converting from binary to decimal
String msg = "101100111001!";
int pos = 0;
while (pos < msg.length()) {
   char binChar = msg.charAt(pos);   // Extract character at pos
   // Do something about the character
   .......
   ++pos;
}
```

"String" to "boolean": You can use method `Boolean.parseBoolean(aBooleanStr)` to convert string of "true" or "false" to `boolean` `true` or `false`, e.g.,

```
String boolStr = "true";
boolean done = Boolean.parseBoolean(boolStr);  // done <- true
boolean valid = Boolean.parseBoolean("false"); // valid <- false
```

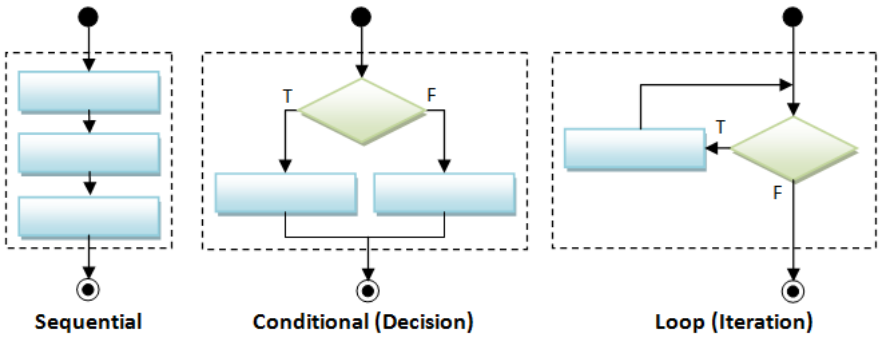
Primitive (int/double/float/byte/short/long/char/boolean) to "String": To convert a primitive to a `String`, you can use the '+' operator to concatenate the primitive with an *empty* `String` (`""`), or use the JDK built-in methods `String.valueOf(aPrimitve)`, `Integer.toString(anInt)`, `Double.toString(aDouble)`, `Character.toString(aChar)`, `Boolean.toString(aBoolean)`, etc. For example,

```
// String concantenation operator '+' (applicable to ALL primitive types)
String s1 = 123 + "";    // int 123 -> "123"
String s2 = 12.34 + "";  // double 12.34 -> "12.34"
String s3 = 'c' + "";    // char 'c' -> "c"
String s4 = true + "";   // boolean true -> "true"

// String.valueOf(aPrimitive) is applicable to ALL primitive types
String s1 = String.valueOf(12345);   // int 12345 -> "12345"
String s2 = String.valueof(true);    // boolean true -> "true"
double d = 55.66;
String s3 = String.valueOf(d);       // double 55.66 -> "55.66"

// toString() for each primitive type
String s4 = Integer.toString(1234);  // int 1234 -> "1234"
String s5 = Double.toString(1.23);   // double 1.23 -> "1.23"
char c1   = Character.toString('z'); // char 'z' -> "z"

// char to String
char c = 'a';
String s5 = c;       // Compilation Error: incompatible types
String s6 = c + "";  // Convert the char to String

// boolean to String
boolean done = false;
String s7 = done + "";    // boolean false -> "false"
String s8 = Boolean.toString(done);
String s9 = String.valueOf(done);
```

"char" to "int": You can convert char `'0'` to `'9'` to `int` `0` to `9` by subtracting the `char` with `'0'` (e.g., `'8'-'0'` → `8`).

5. Flow Control

There are three basic flow control constructs - *sequential*, *conditional* (or *decision*), and *loop* (or *iteration*), as illustrated below.



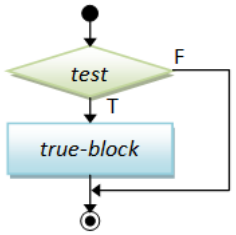
5.1 Sequential Flow Control

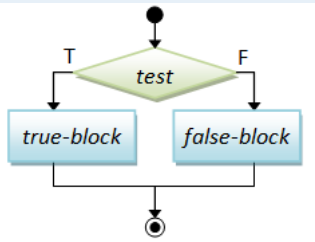
A program is a sequence of instructions. *Sequential* flow is the most common and straight-forward, where programming statements are executed in the order that they are written - from top to bottom in a sequential manner.

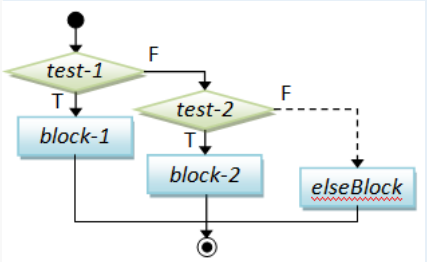
5.2 Conditional Flow Control

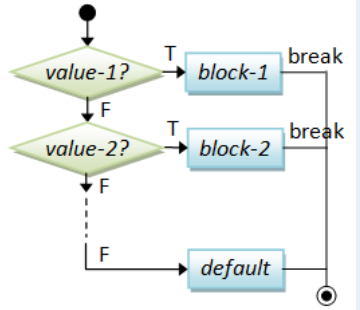
There are a few types of conditionals, *if-then*, *if-then-else*, *nested-if* (*if-elseif-elseif-...-else*), *switch-case*, and *conditional expression*.

Syntax	Example	Flowchart

`// if-then`
`if ( booleanExpression ) {`
`  true-block ;`
`}`
	`if (mark >= 50) {`
	`  System.out.println("Congratulation!");`
	`  System.out.println("Keep it up!");`
	`}`
		

`// if-then-else`
`if ( booleanExpression ) {`
`  true-block ;`
`} else {`
`  false-block ;`
`}`
	`if (mark >= 50) {`
	`  System.out.println("Congratulation!");`
	`  System.out.println("Keep it up!");`
	`} else {`
	`  System.out.println("Try Harder!");`
	`}`
		

`// nested-if`
`if ( booleanExpr-1 ) {`
`  block-1 ;`
`} else if ( booleanExpr-2 ) {`
`  block-2 ;`
`} else if ( booleanExpr-3 ) {`
`  block-3 ;`
`} else if ( booleanExpr-4 ) {`
`  ......`
`} else {`
`  elseBlock ;`
`}`
	`if (mark >= 80) {`
	`  System.out.println("A");`
	`} else if (mark >= 70) {`
	`  System.out.println("B");`
	`} else if (mark >= 60) {`
	`  System.out.println("C");`
	`} else if (mark >= 50) {`
	`  System.out.println("D");`
	`} else {`
	`  System.out.println("F");`
	`}`
		

`// switch-case-default`
`switch ( selector ) {`
`  case value-1:`
`  block-1; break;`
`  case value-2:`
`  block-2; break;`
`  case value-3:`
`  block-3; break;`
`  ......`
`  case value-n:`
`  block-n; break;`
`  default:`
`  default-block;`
`}`
	`char oper; int num1, num2, result;`
	`......`
	`switch (oper) {`
	`  case '+':`
	`  result = num1 + num2; break;`
	`  case '-':`
	`  result = num1 - num2; break;`
	`  case '*':`
	`  result = num1 * num2; break;`
	`  case '/':`
	`  result = num1 / num2; break;`
	`  default:`
	`  System.err.println("Unknown operator");`
	`}`
		

Braces: You could omit the braces `{ }`, if there is only one statement inside the block. However, I recommend that you keep the braces to improve the readability of your program. For example,

```
if (mark >= 50)
   System.out.println("PASS");   // Only one statement, can omit { } but NOT recommended
else {                          // More than one statements, need { }
   System.out.println("FAIL");
   System.out.println("Try Harder!");
}
```

"swtich-case-default"

"switch-case" is an alternative to the "nested-if". In a *switch-case* statement, a `break` statement is needed for each of the cases. If `break` is missing, execution will flow through the following case. You can use an `int`, `byte`, `short`, or `char` variable as the case-*selector*, but NOT `long`, `float`, `double` and `boolean`. (JDK 1.7 supports `String` as the case-selector).

5.3 Conditional Operator (? :)

A conditional operator is a ternary (3-operand) operator, in the form of *booleanExpr* `?` *trueExpr* `:` *falseExpr*. Depending on the *booleanExpr*, it evaluates and returns the value of *trueExpr* or *falseExpr*.

Syntax	Example

booleanExpr `?` *trueExpr* `:` *falseExpr*	`System.out.println((mark >= 50) ? "PASS" : "FAIL");`
	`max = (a > b) ? a : b;  // RHS returns a or b`
	`abs = (a > 0) ? a : -a;  // RHS returns a or -a`

5.4 Exercises on Conditional

LINK TO EXERCISES ON CONDITIONAL FLOW CONTROL

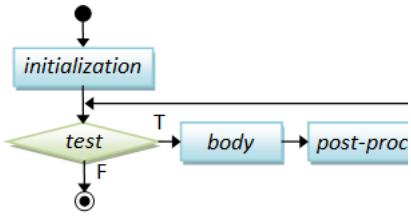
5.5 Loop Flow Control

Again, there are a few types of loops: *for*, *while-do*, and *do-while*.

Syntax	Example	Flowchart

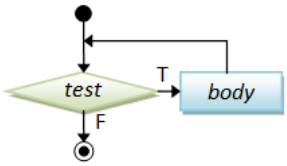
```
// for-loop
for (initialization; test; post-processing) {
   body;
}
```

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



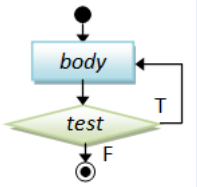

```
// while-do loop
while ( test ) {
   body;
}
```

```
int sum = 0, number = 1;
while (number <= 1000) {
   sum += number;
   ++number;
}
```




```
// do-while loop
do {
   body;
}
while ( test ) ;
```

```
int sum = 0, number = 1;
do {
   sum += number;
   ++number;
} while (number <= 1000);
```



The difference between *while-do* and *do-while* lies in the order of the *body* and *condition*. In *while-do*, the *condition* is tested first. The body will be executed if the *condition* is true and the process repeats. In *do-while*, the *body* is executed and then the *condition* is tested. Take note that the *body* of *do-while* will be executed at least once vs. possibly zero for *while-do*. Similarly, the for-loop's body could possibly not executed.

Example: Suppose that your program prompts user for a number between `1` to `10`, and checks for valid input. A do-while loop with a `boolean` flag could be more appropriate as it prompts for input at least once, and repeat again and again if the input is invalid.

```
// Input with validity check
boolean valid = false;
int number;
do {
  // prompt user to enter an int between 1 and 10
  ......
  // if the number entered is valid, set done to exit the loop
  if (number >=1 && number <= 10) {
     valid = true;
  }
} while (!valid);   // Need a semi-colon to terminate do-while
```

Example: Below is an example of using while-do with a `boolean` flag. The `boolean` flag is initialized to `false` to ensure that the loop is entered.

```
// Game loop
boolean gameOver = false;
while (!gameOver) {
   // play the game
   ......
   // Update the game state
   // Set gameOver to true if appropriate to exit the game loop
   ......
}
```

Empty for-loop

`for ( ; ; ) { body }` is known as an *empty for-loop*, with empty statement for initialization, test and post-processing. The body of the empty for-loop will execute continuouly (infinite loop). You need to use a `break` statement to break out the loop.

for-loop with Comma Separator

You could place more than one statement in the initialization and post-processing, separated with commas (instead of the usual semi-colon). For example,

```
for (int row = 0, col = 0; row < SIZE; ++row, ++col) {
   // Process diagonal elements
   ......
}
```

5.6 Exercises on Loops

LINK TO EXERCISES ON LOOP FLOW CONTROL

5.7 "`break`" and "`continue`" - Interrupting Loop Flow

The `break` statement breaks out and exits the current (innermost) loop.

The `continue` statement aborts the current iteration and continue to the next iteration of the current (innermost) loop.

`break` and `continue` are poor structures as they are hard to read and hard to follow. Use them only if absolutely necessary.

Example (`break`): The following program lists the non-prime numbers between 2 and an upperbound.

```
// List all non-prime numbers between 2 and an upperbound
public class NonPrimeList {
   public static void main(String[] args) {
      int upperbound = 100;
      for (int number = 2; number <= upperbound; ++number) {
         // Not a prime, if there is a factor between 2 and sqrt(number)
         int maxFactor = (int)Math.sqrt(number);
         for (int factor = 2; factor <= maxFactor; ++factor) {
            if (number % factor == 0) {   // Factor?
               System.out.println(number + " is NOT a prime");
               break;   // A factor found, no need to search for more factors
            }
         }
      }
   }
}
```

Let's rewrite the above program to list all the primes instead. A `boolean` flag called `isPrime` is used to indicate whether the current `number` is a prime. It is then used to control the printing.

```
// List all prime numbers between 2 and an upperbound
public class PrimeListWithBreak {
   public static void main(String[] args) {
      int upperbound = 100;
      for (int number = 2; number <= upperbound; ++number) {
         // Not a prime, if there is a factor between 2 and sqrt(number)
         int maxFactor = (int)Math.sqrt(number);
         boolean isPrime = true;  // boolean flag to indicate whether number is a prime
         for (int factor = 2; factor <= maxFactor; ++factor) {
            if (number % factor == 0) {   // Factor?
               isPrime = false;   // number is not a prime
               break;   // A factor found, no need to search for more factors
            }
         }
         if (isPrime) System.out.println(number + " is a prime");
      }
   }
}
```

Let's rewrite the above program without using `break` statement. A `while` loop is used (which is controlled by the `boolean` flag) instead of `for` loop with `break`.

```
// List all prime numbers between 2 and an upperbound
public class PrimeList {
   public static void main(String[] args) {
      int upperbound = 100;
      for (int number = 2; number <= upperbound; ++number) {
         // Not prime, if there is a factor between 2 and sqrt of number
         int maxFactor = (int)Math.sqrt(number);
         boolean isPrime = true;
         int factor = 2;
         while (isPrime && factor <= maxFactor) {
            if (number % factor == 0) {   // Factor of number?
                isPrime = false;
            }
            ++factor;
         }
         if (isPrime) System.out.println(number + " is a prime");
      }
   }
}
```

Example (`continue`):

```
// Sum 1 to upperbound, exclude 11, 22, 33,...
int upperbound = 100;
int sum = 0;
for (int number = 1; number <= upperbound; ++number) {
   if (number % 11 == 0) continue;  // Skip the rest of the loop body, continue to the next iteration
   sum += number;
}
// It is better to re-write the loop as:
for (int number = 1; number <= upperbound; ++number) {
   if (number % 11 != 0) sum += number;
}
```

Example (`break` and `continue`): Study the following program.

```
public class MysterySeries {
   public static void main(String[] args) {
      int number = 1;
      while(true) {
         ++number;
         if ((number % 3) == 0) continue;
         if (number == 133) break;
         if ((number % 2) == 0) {
            number += 3;
         } else {
            number -= 3;
         }
         System.out.print(number + " ");
      }
   }
}
```

Labeled `break`

In a nested loop, the `break` statement breaks out the innermost loop and continue into the outer loop. At times, there is a need to break out all the loops (or multiple loops). This is clumpy to achieve with `boolean` flag, but can be done easily via the so-called labeled `break`. You can add a `label` to a loop in the form of *`labelName: loop`*. For example,

```
level1:          // define a label for the level-1 loop
for (.....) {
level2:          // define a label for the level-2 loop
   for (.....) {
      for (......) {  // level-3 loop
         if (...) break level1;  // break all loops, continue after the loop
         if (...) break level2:  // continue into the next statement of level-1 loop
         ......
      }
   }
}
```

Labeled `continue`

In a nested loop, similar to labeled `break`, you can use labeled continue to continue into a specified loop. For example,

```
level1:          // define a label (with : suffix) for the level-1 loop
for (.....) {
level2:          // define a label (with : suffix) for the level-2 loop
   for (.....) {
      for (......) {  // level-3 loop
         if (...) continue level1;  // continue the next iteration of level-1 loop
         if (...) continue level2:  // continue the next iteration of level-2 loop
         ......
      }
   }
}
```

Again, labeled `break` and `continue` are not sturctured and hard to read. Use them only if absolutely necessary.

Example (Labeled `break`): Suppose that you are searching for a particular number in a 2D array.

```
1  public class TestLabeledBreak {
2     public static void main(String[] args) {
3        int[][] testArray = {
4           {1, 2, 3, 4},
5           {4, 3, 1, 4},
6           {9, 2, 3, 4}
7        };
8
9        int magicNumber = 8;
10       boolean found = false;
11       mainLoop:
12       for (int i = 0; i < testArray.length; ++i) {
13          for (int j = 0; j < testArray[i].length; ++j) {
14             if (testArray[i][j] == magicNumber) {
15                found = true;
16                break mainLoop;
17             }
18          }
19       }
20       System.out.println("Magic number " + (found ? "found" : "NOT found"));
21    }
22 }
```

5.8 Terminating Program

`System.exit(int exitCode)`: You could invoke the method `System.exit(int exitCode)` to terminate the program and return the control to the Java runtime. By convention, return code of zero indicates normal termination; while a non-zero `exitCode` indicates *abnormal termination*. For example,

```
if (errorCount > 10) {
   System.out.println("too many errors");
   System.exit(1);  // Terminate the program
}
```

The `return` statement: You could also use a "`return returnValue`" statement in the `main()` method to terminate the program and return control back to the Java Runtime. For example,

```
public static void main(String[] args) {
   ...
   if (errorCount > 10) {
      System.out.println("too many errors");
      return;  // Terminate and return control to Java Runtime from main()
   }
   ...
}
```

5.9 Nested Loops

Try out the following program, which prints a 8-by-8 checker box pattern using *nested loops*, as follows:

```
# # # # # # # #
# # # # # # # #
# # # # # # # #
```

```
# # # # # # # #
# # # # # # # #
# # # # # # # #
# # # # # # # #
# # # # # # # #
```

```
1  /*
2   * Print a square pattern
3   */
4  public class PrintSquarePattern {    // Save as "PrintSaurePattern.java"
5     public static void main(String[] args) {
6        int size = 8;
7        for (int row = 1; row <= size; ++row) {     // Outer loop to print all the rows
8           for (int col = 1; col <= size; ++col) {  // Inner loop to print all the columns of each row
9              System.out.print("# ");
10          }
11          System.out.println();   // A row ended, bring the cursor to the next line
12       }
13    }
14 }
```

This program contains two *nested* for-loops. The inner loop is used to print a row of eight "`# `", which is followed by printing a newline. The outer loop repeats the inner loop to print all the rows.

Suppose that you want to print this pattern instead (in program called `PrintCheckerPattern`):

```
# # # # # # # #
 # # # # # # # #
# # # # # # # #
 # # # # # # # #
# # # # # # # #
 # # # # # # # #
# # # # # # # #
 # # # # # # # #
```

You need to print an additional space for even-number rows. You could do so by adding the following statement before Line 8.

```
if ((row % 2) == 0) {   // print a leading space for even-numbered rows
   System.out.print(" ");
}
```

TRY:

1. Print these patterns using nested loop (in a program called `PrintPattern1x`). Use a variable called `size` for the size of the pattern and try out various sizes. You should use as few `print()` or `println()` statements as possible.

```
 # * # * # * # *     # # # # # # # #     # # # # # # # #     1                               1
# * # * # * # *      # # # # # # #         # # # # # # #     2 1                           1 2
 # * # * # * # *     # # # # # #             # # # # # #     3 2 1                       1 2 3
# * # * # * # *      # # # # #                 # # # # #     4 3 2 1                   1 2 3 4
 # * # * # * # *     # # # #                     # # # #     5 4 3 2 1               1 2 3 4 5
# * # * # * # *      # # #                         # # #     6 5 4 3 2 1           1 2 3 4 5 6
 # * # * # * # *     # #                             # #     7 6 5 4 3 2 1       1 2 3 4 5 6 7
# * # * # * # *      #                                 #     8 7 6 5 4 3 2 1   1 2 3 4 5 6 7 8
     (a)                   (b)                 (c)                 (d)                 (e)
```

Hints:
The equations for major and opposite diagonals are row = col and row + col = size + 1. Decide on what to print above and below the diagonal.

2. Print the timetable of 1 to 9, as follows, using nested loop. (Hints: you need to use an *if-else* statement to check whether the product is single-digit or double-digit, and print an additional space if needed.)

```
 1  2  3  4  5  6  7  8  9
 2  4  6  8 10 12 14 16 18
 ......
```

3. Print these patterns using nested loop.

```
# # # # # # #      # # # # # # #      # # # # # # #      # # # # # # #      # # # # # # #
#           #      #           #      #           #      #       #      # #       # #
#           #      #           #      #           #      #   #   #      #   #   #   #
#           #      #           #      #           #      #   #   #      #     #     #
#           #      #           #      #           #      #   #   #      #   #   #   #
#           #      #           #      #           #      #       #      # #       # #
# # # # # # #      # # # # # # #      # # # # # # #      # # # # # # #      # # # # # # #
     (a)                 (b)                 (c)                 (d)                 (e)
```

5.10 Exercises on Nested Loops

LINK TO MORE NESTED-LOOP EXERCISES

5.11 Some Issues in Flow Control

Dangling Else: The "dangling else" problem can be illustrated as follows:

```
if (i == 0)
   if (j == 0)
      System.out.println("i and j are zero");
else System.out.println("i is not zero");   // intend for the outer-if
```

The `else` clause in the above codes is syntactically applicable to both the outer-if and the inner-if. Java compiler always associate the `else` clause with the innermost if (i.e., the nearest if). Dangling else can be resolved by applying explicit parentheses. The above codes are logically incorrect and require explicit parentheses as shown below.

```
if ( i == 0) {
```

```
   if (j == 0) System.out.println("i and j are zero");
} else {
   System.out.println("i is not zero");   // non-ambiguous for outer-if
}
```

Endless loop: The following constructs:

```
while (true) { ..... }
```

is commonly used. It seems to be an endless loop (or infinite loop), but it is usually terminated via a `break` or `return` statement inside the loop body. This kind of code is hard to read - avoid if possible by re-writing the condition.

6. Writing Correct and Good Programs

It is important to write programs that produce the correct results. It is also important to write programs that others (and you yourself three days later) can understand, so that the programs can be maintained - I call these programs good programs - a good program is more than a correct program.

Here are the suggestions:

- Follow established convention so that everyone has the same basis of understanding. To program in Java, you MUTST read the "Code Convention for the Java Programming Language".
- Format and layout of the source code with appropriate indents, white spaces and white lines. Use 3 or 4 spaces for indent, and blank lines to separate sections of codes.
- Choose good names that are self-descriptive and meaningful, e.g., `row`, `col`, `size`, `xMax`, `numStudents`. Do not use meaningless names, such as `a`, `b`, `c`, `d`. Avoid single-alphabet names (easier to type but often meaningless), except common names likes x, `y`, `z` for co-ordinates and `i` for index.
- Provide comments to explain the important as well as salient concepts. Comment your codes liberally.
- Write your program documentation while writing your programs.
- Avoid *un-structured* constructs, such as `break` and `continue`, which are hard to follow.
- Use "mono-space" fonts (such as Consola, Courier New, Courier) for writing/displaying your program.

It is estimated that over the lifetime of a program, 20 percent of the effort will go into the original creation and testing of the code, and 80 percent of the effort will go into the subsequent maintenance and enhancement. Writing good programs which follow standard conventions is critical in the subsequent maintenance and enhancement!!!

6.1 Programming Errors

There are generally three classes of programming errors:

1. *Compilation Error* (or *Syntax Error*): can be fixed easily.
2. *Runtime Error*: program halts pre-maturely without producing the results - can also be fixed easily.
3. *Logical Error*: program completes but produces incorrect results. It is easy to detect if the program always produces wrong result. It is extremely hard to fix if the program produces the correct result most of the times, but incorrect result sometimes. For example,

   ```
   // Can compile and execute, but give wrong result - sometimes!
   if (mark > 50) {
      System.out.println("PASS");
   } else {
      System.out.println("FAIL");
   }
   ```

   This kind of errors is very serious if it is not caught before production. Writing good programs helps in minimizing and detecting these errors. A good *testing strategy* is needed to ascertain the correctness of the program. *Software testing* is an advanced topics which is beyond our current scope.

6.2 Debugging Programs

Here are the common debugging techniques:

1. Stare at the screen! Unfortunately, errors usually won't pop-up even if you stare at it extremely hard.
2. Study the error messages! Do not close the console when error occurs and pretending that everything is fine. This helps most of the times.
3. Insert print statements at appropriate locations to display the intermediate results. It works for simple toy program, but it is neither effective nor efficient for complex program.
4. Use a graphic debugger. This is the most effective means. Trace program execution step-by-step and watch the value of variables and outputs.
5. Advanced tools such as profiler (needed for checking memory leak and method usage).
6. Proper program testing to wipe out the logical errors.

6.3 Testing Your Program for Correctness

How to ensure that your program always produces correct result, 100% of the times? It is impossible to try out all the possible outcomes, even for a simple program. Program testing usually involves a set of representative test cases, which are designed to catch the major classes of errors. Program testing is beyond the scope of this writing.

7. Input/Output

7.1 Formatted Output via "`printf()`" (JDK 1.5)

`System.out.print()` and `println()` do not provide output formatting, such as controlling the number of spaces to print an `int` and the number of decimal places for a `double`.

Java SE 5 introduced a new method called `printf()` for formatted output (which is modeled after C Language's `printf()`). `printf()` takes the following form:

```
printf(formatting-string, arg1, arg2, arg3, ... );
```

Formatting-string contains both *normal texts* and the so-called *Format Specifiers*. Normal texts (including white spaces) will be printed as they are. Format specifiers, in the form of "`%[flags][width]conversion-code`", will be substituted by the arguments following the formatting-string, usually in a one-to-one and sequential manner. A format specifier begins with a '`%`' and ends with the conversion code, e.g., `%d` for integer, `%f` for floating-point number, `%c` for character and `%s` for string. Optional `[width]` can be inserted in between to specify the field-width. Similarly, optional `[flags]` can be used to control the alignment, padding and others. For examples,

- `%αd`: integer printed in α spaces (α is optional).
- `%αs`: String printed in α spaces (α is optional). If α is omitted, the number of spaces is the length of the string (to fit the string).
- `%α.βf`: Floating point number (`float` and `double`) printed in α spaces with β decimal digits (α and β are optional).
- `%n`: a system-specific new line (Windows uses "`\r\n`", Unix "`\n`", Mac "`\r`").

Examples:

```
System.out.printf("Hello%2d and %6s", 8, "HI!!!%n");
```

```
Hello*8 and ****HI!!!   // * denotes white-spaces inserted by format specifier
```

```
System.out.printf("Hi,%s%4d%n", "Hello", 88);
```

```
Hi,Hello**88
```

```
System.out.printf("Hi, %d %4.2f%n", 8, 5.556);
```

```
Hi, 8 5.56
```

```
System.out.printf("Hi,%-4s&%6.2f%n", "Hi", 5.5);  // '%-ns' for left-align String
```

```
Hi,Hi**&**5.50
```

```
System.out.printf("Hi, Hi, %.4f%n", 5.56);
```

```
Hi, Hi, 5.5600
```

Take note that `printf()` does not advance the cursor to the next line after printing. You need to explicitly print a newline characterat the end of the formatting-string to advance the cursor to the next line, if desires. [In C program, we often use '\n' to print a newline, which results in non-portable program. You should use format specifier "`%n`" instead.]

There are many more format specifiers in Java. Refer to JDK Documentation for the detailed descriptions.

(Also take note that `printf()` take a variable number of arguments (or *varargs*), which is a new feature introduced in JDK 1.5 in order to support `printf()`)

7.2 Input From Keyboard via "`Scanner`" (JDK 1.5)

Java, like all other languages, supports three standard input/output streams: `System.in` (standard input device), `System.out` (standard output device), and `System.err` (standard error device). The `System.in` is defaulted to be the keyboard; while `System.out` and `System.err` are defaulted to the console. They can be *re-directed* to other devices, e.g., it is quite common to redirect `System.err` to a disk file to save these error message.

You can read input from keyboard via `System.in` (standard input device).

Java SE 5 introduced a new class called `Scanner` in package `java.util` to simplify *formatted input* (and a new method `printf()` for formatted output described earlier). You can construct a `Scanner` to *scan* input from `System.in` (keyboard), and use methods such as `nextInt()`, `nextDouble()`, `next()` to *parse* the next `int`, `double` and `String` token (delimited by white space of blank, tab and newline).

```
import java.util.Scanner;    // Needed to use the Scanner
public class ScannerTest {
   public static void main(String[] args) {
      int num1;
      double num2;
      String str;
      // Construct a Scanner named "in" for scanning System.in (keyboard)
      Scanner in = new Scanner(System.in);
      System.out.print("Enter an integer: ");
      num1 = in.nextInt();         // Use nextInt() to read int
      System.out.print("Enter a floating point: ");
      num2 = in.nextDouble();      // Use nextDouble() to read double
      System.out.print("Enter a string: ");
      str  = in.next();            // Use next() to read a String token, up to white space
      // Formatted output via printf()
      System.out.printf("%s, Sum of %d & %.2f is %.2f%n", str, num1, num2, num1+num2);
   }
}
```

You can also use method `nextLine()` to read in the entire line, including white spaces, but excluding the terminating newline.

```
import java.util.Scanner;    // Needed to use the Scanner
public class ScannerNextLineTest {
   public static void main(String[] args) {
      Scanner in = new Scanner(System.in);
      System.out.print("Enter a string (with space): ");
      // Use nextLine() to read entire line including white spaces,
      //   but excluding the terminating newline.
      String str  = in.nextLine();
      System.out.printf("%s%n", str);
   }
}
```

Try not to mix `nextLine()` and `nextInt()|nextDouble()|next()` in a program (as you may need to flush the newline from the input buffer).

The `Scanner` supports many other input formats. Check the JDK documentation page, under package `java.util` ⇒ class `Scanner` ⇒ Method.

7.3 Input from Text File via "Scanner" (JDK 1.5)

Other than scanning `System.in` (keyboard), you can connect your `Scanner` to scan any input source, such as *a disk file* or *a network socket*, and use the same set of methods `nextInt()`, `nextDouble()`, `next()`, `nextLine()` to parse the next `int`, `double`, `String` and line. For example,

```
Scanner in = new Scanner(new File("in.txt"));  // Construct a Scanner to scan a text file
// Use the same set of methods
int anInt = in.nextInt();         // next String
double aDouble = in.nextDouble(); // next double
String str = in.next();           // next int
String line = in.nextLine();      // entire line
```

To open a file via `new File(filename)`, you need to handle the so-called `FileNotFoundException`, i.e., the file that you are trying to open cannot be found. Otherwise, you cannot compile your program. There are two ways to handle this exception: *throws* or *try-catch*.

```
// Technique 1: Declare "throws FileNotFoundException" in the enclosing main() method
import java.util.Scanner;               // Needed for using Scanner
import java.io.File;                    // Needed for file operation
import java.io.FileNotFoundException;   // Needed for file operation
public class TextFileScannerWithThrows {
   public static void main(String[] args)
         throws FileNotFoundException {  // Declare "throws" here
      int num1;
      double num2;
      String name;
      Scanner in = new Scanner(new File("in.txt"));  // Scan input from text file
      num1 = in.nextInt();      // Read int
      num2 = in.nextDouble();   // Read double
      name = in.next();         // Read String
      System.out.printf("Hi %s, the sum of %d and %.2f is %.2f%n", name, num1, num2, num1+num2);
   }
}
```

To run the above program, create a text file called `in.txt` containing:

```
1234
55.66
Paul
```

```
// Technique 2: Use try-catch to handle exception
import java.util.Scanner;               // Needed for using Scanner
import java.io.File;                    // Needed for file operation
import java.io.FileNotFoundException;   // Needed for file operation
public class TextFileScannerWithCatch {
   public static void main(String[] args) {
      int num1;
      double num2;
      String name;
      try {                                // try these statements
         Scanner in = new Scanner(new File("in.txt"));
         num1 = in.nextInt();      // Read int
         num2 = in.nextDouble();   // Read double
         name = in.next();         // Read String
         System.out.printf("Hi %s, the sum of %d and %.2f is %.2f%n", name, num1, num2, num1+num2);
      } catch (FileNotFoundException ex) {  // catch and handle the exception here
         ex.printStackTrace();              // print the stack trace
      }
   }
}
```

7.4 Formatted Output to Text File

Java SE 5.0 also introduced a so-called `Formatter` for formatted output (just like `Scanner` for formatted input). A `Formatter` has a method called `format()`. The `format()` method has the same syntax as `printf()`, i.e., it could use format specifiers to specify the format of the arguments. Again, you need to handle the `FileNotFoundException`.

```
// Technique 1: Declare "throws FileNotFoundException" in the enclosing main() method
import java.io.File;
import java.util.Formatter;             // <== note
import java.io.FileNotFoundException;   // <== note
public class TextFileFormatterWithThrows {
  public static void main(String[] args)
      throws FileNotFoundException {   // <== note
    // Construct a Formatter to write formatted output to a text file
    Formatter out = new Formatter(new File("out.txt"));
    // Write to file with format() method (similar to printf())
    int num1 = 1234;
    double num2 = 55.66;
    String name = "Paul";
    out.format("Hi %s,%n", name);
    out.format("The sum of %d and %.2f is %.2f%n", num1, num2, num1 + num2);
    out.close();                 // Close the file
    System.out.println("Done");  // Print to console
  }
}
```

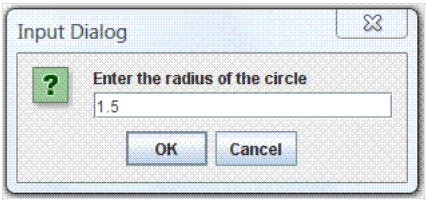
Run the above program, and check the outputs in text file "out.txt".

```
// Technique 2: Use try-catch to handle exception
import java.io.File;
import java.util.Formatter;           // <== note
```

```
import java.io.FileNotFoundException; // <== note

public class TextFileFormatterWithCatch {
   public static void main(String[] args) {
      try {    // try the following statements
         // Construct a Formatter to write formatted output to a text file
         Formatter out = new Formatter(new File("out.txt"));
         // Write to file with format() method (similar to printf())
         int num1 = 1234;
         double num2 = 55.66;
         String name = "Pauline";
         out.format("Hi %s,%n", name);
         out.format("The sum of %d and %.2f is %.2f%n", num1, num2, num1 + num2);
         out.close();               // Close the file
         System.out.println("Done"); // Print to console
      } catch (FileNotFoundException ex) {  // catch the exception here
         ex.printStackTrace();       // Print the stack trace
      }
   }
}
```

7.5 Input via a Dialog Box

You can also get inputs from users via a graphical dialog box, using the `JOptionPane` class. For example, the following program prompts the user to enter the radius of a circle, and computes the area.




```
1  import javax.swing.JOptionPane;   // Needed to use JOptionPane
2  public class JOptionPaneTest {
3     public static void main(String[] args) {
4        String radiusStr;
5        double radius, area;
6        // Read input String from dialog box
7        radiusStr = JOptionPane.showInputDialog("Enter the radius of the circle");
8        radius = Double.parseDouble(radiusStr);   // Convert String to double
9        area = radius*radius*Math.PI;
10       System.out.println("The area is " + area);
11    }
12 }
```

Dissecting the Program:

- In Line 1, the import statement is needed to use the `JOptionPane`.
- In Line 7, we use the method `JOptionPane.showInputDialog(promptMessage)` to prompt users for an input, which returns the input as a `String`.
- Line 8 converts the input `String` to a `double`, using the method `Double.parseDouble()`.

7.6 `java.io.Console` (JDK 1.6)

Java SE 6 introduced a new `java.io.Console` class to simplify character-based input/output to/from the system console. BUT, the `Console` class does not run under IDE (such as Eclipse/Netbeans)!!!

To use the new `Console` class, you first use `System.console()` to retrieve the `Console` object corresponding to the current system console.

```
Console con = System.console();
```

You can then use methods such as `readLine()` to read a line. You can optionally include a prompting message with format specifiers (e.g., `%d`, `%s`) in the prompting message.

```
String inLine = con.readLine();
String msg = con.readLine("Enter your message: ");        // readLine() with prompting message
String msg = con.readLine("%s, enter message: ", name);   // Prompting message with format specifier
```

You can use `con.printf()` for formatted output with format specifiers such as `%d`, `%s`. You can also connect the `Console` to a `Scanner` for formatted input, i.e., parsing primitives such as `int`, `double`, for example,

```
Scanner in = new Scanner(con.reader());  // Use Scanner to scan the Console
// Use the Scanner's methods such as nextInt(), nextDouble() to parse primitives
int anInt = in.nextInt();
double aDouble = in.nextDouble();
String str = in.next();
String line = in.nextLine();
```

Example:

```
import java.io.Console;
import java.util.Scanner;

public class ConsoleTest {
   public static void main(String[] args) {
      Console con = System.console();   // Retrieve the Console object
      // Console class does not work in Eclipse/Netbeans
      if (con == null) {
          System.err.println("Console Object is not available.");
          System.exit(1);
      }
```

```
      // Read a line with a prompting message
      String name = con.readLine("Enter your Name: ");
      con.printf("Hello %s%n", name);
      // Use the console with Scanner for parsing primitives
      Scanner in = new Scanner(con.reader());
      con.printf("Enter an integer: ");
      int anInt = in.nextInt();
      con.printf("The integer entered is %d%n", anInt);
      con.printf("Enter a floating point number: ");
      double aDouble = in.nextDouble();
      con.printf("The floating point number entered is %f%n", aDouble);
   }
}
```

The `Console` class also provides a secure mean for password entry via method `readPassword()`. This method disables input echoing and keep the password in a `char[]` instead of a `String`. The `char[]` containing the password can be and should be overwritten, removing it from memory as soon as it is no longer needed. (Recall that `String`s are immutable and cannot be overwritten. When they are longer needed, they will be garbage-collected at an unknown instance.)

```
import java.io.Console;
import java.util.Arrays;

public class ConsolePasswordTest {
   static String login;
   static char[] password;

   public static void main(String[] args) {
      Console con = System.console();
      if (con == null) {
         System.err.println("Console Object is not available.");
         System.exit(1);
      }

      login = con.readLine("Enter your login Name: ");
      password = con.readPassword("Enter your password: ");
      if (checkPassword(login, password)) {
         Arrays.fill(password, ' ');  // Remove password from memory
         // Continue ...

      }
   }

   static boolean checkPassword(String login, char[] password) {
       return true;
   }
}
```

7.7 Exercises on Input/Output

LINK TO EXERCISE ON INPUT

8. Arrays

Suppose that you want to find the average of the marks for a class of 30 students, you certainly do not want to create 30 variables: `mark1`, `mark2`, ..., `mark30`. Instead, You could use a single variable, called an *array*, with 30 elements.

An array is *an ordered collection of elements of the same type*, identified by a pair of square brackets `[ ]`. To use an array, you need to:

1. *Declare* the array with a *name* and a *type*. Use a plural name for array, e.g., `marks`, `rows`, `numbers`. All elements of the array belong to the same type.
2. A*llocate* the array using `new` operator, or through *initialization*, e.g.,

```
int[] marks;  // Declare an int array named marks
int marks[];  // Same as above, but the above syntax recommended
marks = new int[5];   // Allocate 5 elements via the "new" operator
// Declare and allocate a 20-element array in one statement via "new" operator
int[] factors = new int[20];
// Declare, allocate a 6-element array thru initialization
int[] numbers = {11, 22, 33, 44, 55, 66}; // size of array deduced from the number of items
```

When an array is constructed via the `new` operator, all the elements are initialized to their default value, e.g., `0` for `int`, `0.0` for `double`, `false` for `boolean`, and `null` for objects. [Unlike C/C++, which does NOT initialize the array contents.]

You can refer to an element of an array via an index (or subscript) enclosed within the square bracket `[ ]`. Java's array index begins with zero (`0`). For example, suppose that `marks` is an `int` array of 5 elements, then the 5 elements are: `marks[0]`, `marks[1]`, `marks[2]`, `marks[3]`, and `marks[4]`.

```
int[] marks = new int[5];   // Declare & allocate a 5-element int array
// Assign values to the elements
marks[0] = 95;
marks[1] = 85;
marks[2] = 77;
marks[3] = 69;
marks[4] = 66;
System.out.println(marks[0]);
System.out.println(marks[3] + marks[4]);
```

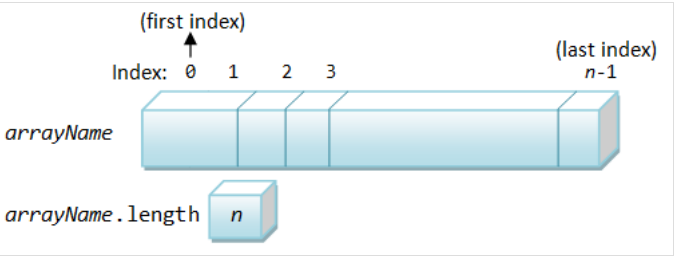
To create an array, you need to known the length (or size) of the array in advance, and allocate accordingly. Once an array is created, its length is fixed and cannot be changed. At times, it is hard to ascertain the length of an array (e.g., how many students?). Nonetheless, you need to estimate the length and allocate an upper bound. This is probably the major drawback of using an array.

In Java, the length of array is kept in an *associated variable* called `length` and can be retrieved using "*`arrayName`*`.length`", e.g.,

```
int[] factors = new int[5];       // Declare and allocate a 5-element int array
int numFactors = factors.length;  // numFactor is 5
```

The index of an array is between `0` and *`arrayName`*`.length - 1`.

Unlike languages like C/C++, Java performs array *index-bound check* at the *runtime*. In other words, for each reference to an array element, the index is checked against the array's length. If the index is outside the range of `[0, arrayName.legnth-1]`, the Java Runtime will signal an exception called `ArrayIndexOutOfBoundException`. It is important to note that checking array index-bound consumes computation power, which inevitably slows down the processing. However, the benefits gained in terms of good software engineering out-weight the slowdown in speed.



8.1 Array and Loop

Arrays works hand-in-hand with loops. You can process all the elements of an array via a loop, for example,

```
1  // Find the mean and standard deviation of numbers kept in an array
2  public class MeanStdArray {
3     public static void main(String[] args) {
4        int[] marks = {74, 43, 58, 60, 90, 64, 70};
5        int sum = 0;
6        int sumSq = 0;
7        int count = marks.length;
8        double mean, stdDev;
9        for (int i=0; i<count; ++i) {
10          sum += marks[i];
11          sumSq += marks[i]*marks[i];
12       }
13       mean = (double)sum/count;
14       System.out.printf("Mean is %.2f%n", mean);
15       stdDev = Math.sqrt((double)sumSq/count - mean*mean);
16       System.out.printf("Std dev is %.2f%n", stdDev);
17    }
18 }
```

8.2 Enhanced for-loop (or "for-each" Loop) (JDK 1.5)

JDK 1,5 introduces a new loop syntax known as *enhanced for-loop* (or *for-each* loop) to facilitate processing of arrays and collections. It takes the following syntax:

Syntax	Example

`for ( type item : anArray ) {`
`  body ;`
`}`
`// type must be the same as the`
`// anArray's type`
	`int[] numbers = {8, 2, 6, 4, 3};`
	`int sum = 0;`
	`for (int number : numbers) {  // for each int number in int[] numbers`
	`  sum += number;`
	`}`
	`System.out.println("The sum is " + sum);`

It shall be read as "for each element in the array...". The loop executes once for each element in the array, with the element's value copied into the declared variable. The for-each loop is handy to transverse all the elements of an array or a collection. It requires fewer lines of codes, eliminates the loop counter and the array index, and is easier to read. However, for array of primitive types (e.g., array of `int`s), it can *read* the elements only, and cannot *modify* the array's contents. This is because each element's value is copied into the loop's variable (pass-by-value), instead of working on its original copy.

In many situations, you merely want to transverse thru the array and read each of the elements. For these cases, enhanced for-loop is preferred and recommended over other loop constructs.

8.3 Exercises on Arrays

LINK TO EXERCISES ON ARRAY

8.4 Command-Line Arguments - An array of `String`

Java's `main()` method takes an argument: `String[] args`, i.e., a `String` array named `args`. This is known as "command-line arguments", which corresponds to the augments provided by the user when the java program is invoked. For example, a Java program called `Arithmetic` could be invoked with additional command-line arguments as follows (in a "cmd" shell):

```
> java Arithmetic 12 3456 +
```

Each argument, i.e., `"12"`, `"3456"` and `"+"`, is a `String`. Java runtime packs all the arguments into a `String` array and passes into the `main()` method as the parameter `args`. For this example, `args` has the following properties:

```
args = {"12", "3456", "+"}   // "args" is a String array
args.length = 3              // length of the array args
args[0] = "12"               // Each element of the array is a String
args[1] = "3456"
args[2] = "+"
args[0].length() = 2         // length of the String
args[1].length() = 4
args[2].length() = 1
```

Example: The program `Arithmetic` reads three parameters form the command-line, two integers and an arithmetic operator (`'+'`, `'-'`, `'*'`, or `'/'`), and performs the arithmetic operation accordingly. For example,

```
> java Arithmetic 3 2 +
3+2=5
> java Arithmetic 3 2 -
3-2=1
```

```
> java Arithmetic 3 2 /
3/2=1
```

```
public class Arithmetic {
   public static void main (String[] args) {
      int operand1, operand2;
      char theOperator;
      operand1 = Integer.parseInt(args[0]);  // Convert String to int
      operand2 = Integer.parseInt(args[1]);
      theOperator = args[2].charAt(0);       // Consider only 1st character
      System.out.print(args[0] + args[2] + args[1] + "=");
      switch(theOperator) {
         case ('+'):
            System.out.println(operand1 + operand2); break;
         case ('-'):
            System.out.println(operand1 - operand2); break;
         case ('*'):
            System.out.println(operand1 * operand2); break;
         case ('/'):
            System.out.println(operand1 / operand2); break;
         default:
            System.out.printf("%nError: Invalid operator!");
      }
   }
}
```

8.5 Exercises on Command-Line Arguments

LINK TO EXERCISES ON COMMAND-LINE ARGUMENTS

8.6 Multi-Dimensional Array

In Java, you can declare an array of arrays. For examples:

```
int grid[][] = new int[12][8];   // a 12×8 grid of int
grid[0][0] = 8;
grid[1][1] = 5;
System.out.println(grid.length);      // 12
System.out.println(grid[0].length);   // 8
System.out.println(grid[11].length);  // 8
```

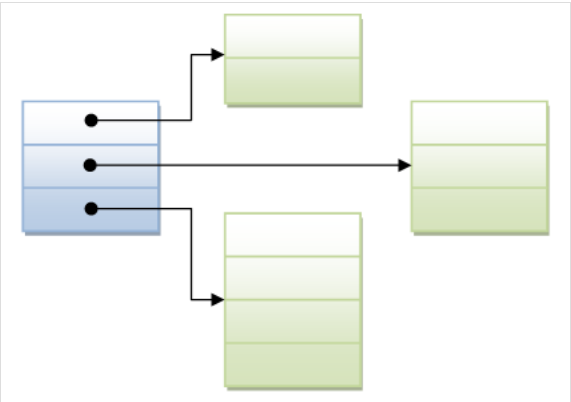
In the above example, `grid` is an array of 12 elements. Each of the elements (`grid[0]` to `grid[11]`) is an 8-element `int` array. In other words, `grid` is a "12-element array" of "8-element int arrays". Hence, `grid.length` gives `12` and `grid[0].length` gives `8`.

```
public class Array2DTest {
   public static void main(String[] args) {
      int[][] grid = new int[12][8];   // A 12x8 grid, in [row][col] or [y][x]
      int numRows = grid.length;       // 12
      int numCols = grid[0].length;    // 8

      // Fill in grid
      for (int row = 0; row < numRows; ++row) {
         for (int col = 0; col < numCols; ++col) {
            grid[row][col] = row*numCols + col + 1;
         }
      }

      // Print grid
      for (int row = 0; row < numRows; ++row) {
         for (int col = 0; col < numCols; ++col) {
            System.out.printf("%3d", grid[row][col]);
         }
         System.out.println();
      }
   }
}
```

To be precise, Java does not support multi-dimensional array directly. That is, it does not support syntax like `grid[3, 2]` like some languages. Furthermore, it is possible that the arrays in an array-of-arrays have different length.

Take note that the right way to view the "array of arrays" is as shown, instead of treating it as a 2D table, even if all the arrays have the same length.

For example,



```
public class Array2DWithDifferentLength {
   public static void main(String[] args) {
      int[][] grid = {
         {1, 2},
         {3, 4, 5},
         {6, 7, 8, 9}
      };
```

```
8
9        // Print grid
10       for (int y = 0; y < grid.length; ++y) {
11          for (int x = 0; x < grid[y].length; ++x) {
12             System.out.printf("%2d", grid[y][x]);
13          }
14          System.out.println();
15       }
16
17       int[][] grid1 = new int[3][];
18       grid1[0] = new int[2];
19       grid1[1] = new int[3];
20       grid1[2] = new int[4];
21
22       // Print grid - all elements init to 0
23       for (int y = 0; y < grid1.length; ++y) {
24          for (int x = 0; x < grid1[y].length; ++x) {
25             System.out.printf("%2d", grid1[y][x]);
26          }
27          System.out.println();
28       }
29    }
30 }
```

9. Methods

9.1 Why Methods?

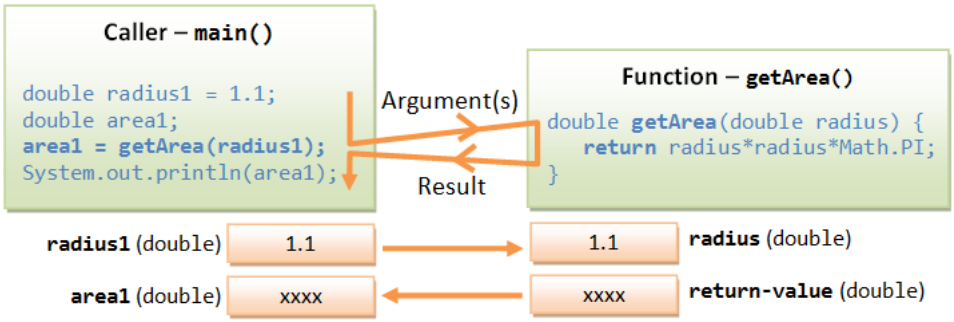
At times, a certain portion of codes has to be used many times. Instead of re-writing the codes many times, it is better to put them into a "subroutine", and "call" this "subroutine" many time - for ease of maintenance and understanding. Subroutine is called method (in Java) or function (in C/C++).

The benefits of using methods are:

1. *Divide and conquer*: construct the program from simple, small pieces or components. Modularize the program into self-contained tasks.
2. *Avoid repeating codes*: It is easy to copy and paste, but hard to maintain and synchronize all the copies.
3. *Software Reuse*: you can reuse the methods in other programs, by packaging them into library codes.

9.2 Using Methods

Two parties are involved in using a method: a *caller*, who calls the method, and the *method* to be called. The caller passes *arguments* to the method. The method receives these arguments, performs the programmed operations defined in the method's body, and returns the result back to the caller.



Example: Suppose that we need to evaluate the area of a circle many times, it is better to write a method called `getArea()`, and re-use it when needed.

```
1  public class MethodTest {
2     // The entry main method
3     public static void main(String[] args) {
4        double radius1 = 1.1, area1, area2;
5        // Call method getArea()
6        area1 = getArea(radius1);
7        System.out.println("area 1 is " + area1);
8        // Call method getArea()
9        area2 = getArea(2.2);
10       System.out.println("area 2 is " + area2);
11       // Call method getArea()
12       System.out.println("area 3 is " + getArea(3.3));
13    }
14
15    // Method getArea() definition
16    // Compute and return the area of circle given its radius
17    public static double getArea(double radius) {
18       return radius * radius * Math.PI;
19    }
20 }
```

```
area 1 is 3.8013271108436504
area 2 is 15.205308443374602
area 3 is 34.21194399759284
```

In the above example, a reusable method called `getArea()` is defined, which receives a parameter in `double` from the caller, performs the calculation, and return a `double` result to the caller. In the `main()`, we invoke `getArea()` methods thrice, each time with a different parameter.

Method Definition Syntax

The syntax for method definition is as follows:

```
public static returnValueType methodName ( arg-1-type arg-1, arg-2-type arg-2,... ) {
   body ;
}
```

Method Naming Convention

A method's name shall be a verb or verb phrase (action), comprising one or more words. The first word is in lowercase, while the rest are initial-capitalized (called *camel-case*). For example, `getArea()`, `setRadius()`, `moveDown()`, `isPrime()`, etc.

9.3 The "`return`" statement

Inside the method body, you could use a `return` statement to return a value (of the *`returnValueType`* declared in the method's signature) to return a value back to the caller. The syntax is:

```
return aReturnValue;   // of returnValueType declared in method's signature
return;                // return nothing (or void)
```

9.4 The "`void`" Return-Type

Suppose that you need a method to perform certain actions (e.g., printing) without a need to return a value to the caller, you can declare its return-value type as `void`. In the method's body, you could use a "`return;`" statement without a return value to return control to the caller. In this case, the `return` statement is optional. If there is no `return` statement, the entire body will be executed, and control returns to the caller at the end of the body.

Notice that `main()` is a method with a return-value type of `void`. `main()` is called by the Java runtime, perform the actions defined in the body, and return nothing back to the Java runtime.

9.5 Actual Parameters vs. Formal Parameters

Recall that a method receives arguments from its caller, performs the actions defined in the method's body, and return a value (or nothing) to the caller.

In the above example, the variable `(double radius)` declared in the signature of `getArea(double radius)` is known as *formal parameter*. Its scope is within the method's body. When the method is invoked by a caller, the caller must supply so-called *actual parameter*s or *arguments*, whose value is then used for the actual computation. For example, when the method is invoked via "`area1=getArea(radius1)`", `radius1` is the actual parameter, with a value of `1.1`.

9.6 Pass-by-Value for Primitive-Type Parameters

In Java, when an argument of primtive type is pass into a method, a *copy* is created and passed into the method. The invoked method works on the *cloned copy*, and cannot modify the original copy. This is known as *pass-by-value*.

For example,

```
1  public class PassingParameterTest {
2     public static void main(String[] args) {
3        int number = 8;                // primitive type
4        System.out.println("In caller, before calling the method, number is: " + number);  // 8
5        int result = increment(number); // invoke method
6        System.out.println("In caller, after calling the method, number is: " + number);   // 8
7        System.out.println("The result is " + result);  // 9
8     }
9
10    public static int increment(int number) {
11       System.out.println("Inside method, before operation, number is " + number); // 8
12       ++number;  // change the parameter
13       System.out.println("Inside method, after operation, number is " + number);  // 9
14       return number;
15    }
16 }
```

9.7 Varargs - Method with Variable Number of Formal Arguments (JDK 1.5)

Before JDK 1.5, a method has to be declared with a *fixed number of formal arguments*. C-like `printf()`, which take a *variable number of argument*, cannot not be implemented. Although you can use an array for passing a variable number of arguments, it is not neat and requires some programming efforts.

JDK 1.5 introduces variable arguments (or varargs) and a new syntax "*`Type`*`...`". For example,

```
public PrintWriter printf(String format, Object... args)
public PrintWriter printf(Local l, String format, Object... args)
```

Varargs can be used only for the last argument. The three dots (`...`) indicate that the last argument may be passed as an array or as a sequence of comma-separated arguments. The compiler automatically packs the varargs into an array. You could then retrieve and process each of these arguments inside the method's body as an array. It is possible to pass varargs as an array, becuase Java maintains the length of the array in an associated variable `length`.

```
1  public class VarargsTest {
2     // A method which takes a variable number of arguments (varargs)
3     public static void doSomething(String... strs) {
4        System.out.print("Arguments are: ");
5        for (String str : strs) {
6           System.out.print(str + ", ");
7        }
8        System.out.println();
9     }
10
11    // A method which takes exactly two arguments
12    public static void doSomething(String s1, String s2) {
13       System.out.println("Overloaded version with 2 args: " + s1 + ", " + s2);
14    }
15
```

```
16      // Cannot overload with this method - crash with varargs version
17      // public static void doSomething(String[] strs)
18
19      // Test main() method
20      // Can also use String... instead of String[]
21      public static void main(String... args) {
22         doSomething("Hello", "world", "again", "and", "again");
23         doSomething("Hello", "world");
24
25         String[] strs = {"apple", "orange"};
26         doSomething(strs);  // invoke varargs version
27      }
28   }
```

Notes:

- If you define a method that takes a varargs `String...`, you cannot define an overloaded method that takes a `String[]`.
- "varargs" will be matched *last* among the overloaded methods. The `varargsMethod(String, String)`, which is more specific, is matched before the `varargsMethod(String...)`.
- From JDK 1.5, you can also declare your `main()` method as:

```
public static void main(String... args) { .... }  // JDK 1.5 varargs
```

9.8 "boolean" Methods

A `boolean` method returns a `boolean` value to the caller.

Suppose that we wish to write a method called `isOdd()` to check if a given number is odd.

```
1   /**
2    *  Testing boolean method (method that returns a boolean value)
3    */
4   public class BooleanMethodTest {
5      // This method returns a boolean value
6      public static boolean isOdd(int number) {
7         if (number % 2 == 1) {
8            return true;
9         } else {
10           return false;
11        }
12     }
13
14     public static void main(String[] args) {
15        System.out.println(isOdd(5));  // true
16        System.out.println(isOdd(6));  // false
17        System.out.println(isOdd(-5)); // false
18     }
19  }
```

This seemingly correct codes produces `false` for `-5`, because `-5%2` is `-1` instead of `1`. You may rewrite the condition:

```
public static boolean isOdd(int number) {
   if (number % 2 == 0) {
      return false;
   } else {
      return true;
   }
}
```

The above produces the correct answer, but is poor. For boolean method, you can simply return the resultant `boolean` value of the comparison, instead of using a conditional statement, as follow:

```
public static boolean isEven(int number) {
   return (number % 2 == 0);
}
public static boolean isOdd(int number) {
   return !(number % 2 == 0);
}
```

9.9 Mathematical Methods

JDK provides many common-used Mathematical methods in a class called `Math`. The signatures of some of these methods are:

```
double Math.pow(double x, double y) // returns x raises to power of y
double Math.sqrt(double x)          // returns the square root of x
double Math.random()                // returns a random number in [0.0, 1.0)
double Math.sin()
double Math.cos()
```

The `Math` class also provide two constants:

```
Math.PI    // 3.141592653589793
Math.E     // 2.718281828459045
```

To check all the available methods, open JDK API documentation ⇒ select *package* "`java.lang`" ⇒ select *class* "`Math`" ⇒ choose *method*.

For examples,

```
int secretNumber = (int)Math.random()*100;  // Generate a random int between 0 and 99

double radius = 5.5;
double area = radius*radius*Math.PI;
```

```
area = Math.pow(radius, 2)*Math.PI;         // Not as efficient as above

int x1 = 1, y1 = 1, x2 = 2, y2 = 2;
double distance = Math.sqrt((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1));
int dx = x2 - x1;
int dy = y2 - y1;
distance = Math.sqrt(dx*dx + dy*dy);        // Slightly more efficient
```

9.10 Implicit Type-Casting for Method's Parameters

A method that takes a `double` parameter can accept any numeric primitive type, such as `int` or `float`. This is because implicit type-casting is carried out. However, a method that take a `int` parameter cannot accept a `double` value. This is because the implicit type-casting is always a widening conversion which prevents loss of precision. An explicit type-cast is required for narrowing conversion. Read "Type-Casting" on the conversion rules.

9.11 Exercises on Methods

LINK TO EXERCISES ON METHOD

10. Code Examples

10.1 Example: `Bin2Dec`

Convert a binary string into its equivalent decimal number.

Version 1:

```
/*
 * Prompt user for a binary string, and convert into its equivalent decimal number.
 */
import java.util.Scanner;

public class Bin2Dec {
   public static void main(String[] args) {
      String binStr;    // The input binary string
      int binStrLen;    // Length of binStr
      int dec = 0;      // Equivalent decimal number

      // Read input
      Scanner in = new Scanner(System.in);
      System.out.print("Enter a binary string: ");
      binStr = in.next();
      binStrLen = binStr.length();

      // Process the String from112 the right (i.e. LSB)
      for (int exp = 0; exp < binStrLen ; ++exp) {
         char binChar = binStr.charAt(binStrLen - 1 - exp);
         if (binChar == '1') {
            dec += (int)Math.pow(2, exp);
         } else if (binChar != '0') {
            System.out.println("Error: Invalid binary string \"" + binStr + "\"");
            System.exit(1);   // quit
         }
      }
      System.out.println("The equivalent decimal is " + dec);
   }
}
```

Version 2:

```
/*
 * Prompt user for a binary string, and convert into its equivalent decimal number.
 * Validate the input string.
 * Repeat the program, until user chooses to quit.
 * Allow blank in the binary string, e.g., "0100 1000".
 */
import java.util.Scanner;
public class Bin2DecIterative {
   public static void main(String[] args) {
      String inStr;    // The input binary string
      boolean done = false;

      // Set up the input Scanner
      Scanner in = new Scanner(System.in);

      while (!done) {
         // Prompt for the input string
         System.out.print("Enter a binary string or 'q' to quit: ");
         inStr = in.nextLine();  // read entire line including blanks
         if (inStr.equals("q")) {
            System.out.println("Bye!");
            done = true;
         } else if (!isValidBinStr(inStr)) {
            System.out.println("Error: Invalid binary string: \"" + inStr + "\", try again.");
         } else {
            System.out.println("The equivalent decimal number for \"" + inStr + "\" is " + bin2Dec(inStr));
         }
      }
   }
```

```
   // Return true if the given string contains only binary numbers and blanks.
   public static boolean isValidBinStr(String binStr) {
      for (int i = 0; i < binStr.length(); ++i) {
         char binChar = binStr.charAt(i);
         if (binChar != '0' && binChar != '1' && binChar != ' ') {
            return false;
         }
      }
      return true;
   }

   // Return the equivalent decimal number of the given binary string.
   // Blank allowed in the binStr, e.g., "0010 1000"
   public static int bin2Dec(String binStr) {
      int binStrLen = binStr.length(); // Length of binStr
      int dec = 0;                     // Equivalent decimal number

      // Process the String from112 the right (i.e. LSB)
      for (int exp = 0; exp < binStrLen ; ++exp) {
         char binChar = binStr.charAt(binStrLen - 1 - exp);
         if (binChar == '1') {
            dec += (int)Math.pow(2, exp);
         }
      }
      return dec;
   }
}
```

10.2 Example: Hex2Dec

Convert a hexadecimal string to its decimal equivalence.

```
/*
 * Prompt user for the hexadecimal string, and convert to its equivalent decimal number
 */
import java.util.Scanner;
public class Hex2Dec {
   public static void main(String[] args) {
      String hexStr;   // Input hexadecimal String
      int hexStrLen;   // Length of hexStr
      int dec = 0;     // Decimal equivalence

      // Read input
      Scanner in = new Scanner(System.in);
      System.out.print("Enter a Hexadecimal string: ");
      hexStr = in.next();
      hexStrLen = hexStr.length();

      // Process the string from the right
      for (int exp = 0; exp < hexStrLen; ++exp) {
         char hexChar = hexStr.charAt(hexStrLen - 1 - exp);
         int factor = (int)Math.pow(16, exp);
         if (hexChar >= '1' && hexChar <= '9') {
            dec += (hexChar - '0') * factor;
         } else if (hexChar >= 'a' && hexChar <= 'f') {
            dec += (hexChar - 'a' + 10) * factor;
         } else if (hexChar >= 'A' && hexChar <= 'F') {
            dec += (hexChar - 'A' + 10) * factor;
         } else {
            System.out.println("Error: Invalid hex string \"" + hexStr + "\"");
            System.exit(1);
         }
      }
      System.out.println("The equivalent decimal for \"" + hexStr + "\" is " + dec);
   }
}
```

10.3 Example: Dec2Hex

Convert a decimal number to its hexadecimal equivalence.

```
/*
 * Prompt user for an int, and convert to equivalent hexadecimal number.
 */
import java.util.Scanner;
public class Dec2Hex {
   public static void main(String[] args) {
      int dec;              // Input decimal number
      String hexStr = "";   // Equivalent hex String
      int radix = 16;       // Hex radix
      char[] hexChar = {'0','1','2','3','4','5','6','7','8','9',
          'A','B','C','D','E','F'};

      // Read input
      Scanner in = new Scanner(System.in);
      System.out.print("Enter a decimal number: ");
      dec = in.nextInt();

      // Repeated division and get the remainder
      while (dec > 0) {
         int hexDigit = dec % radix;
         hexStr = hexChar[hexDigit] + hexStr;  // append in front of the hex string
         dec = dec / radix;
```

```
      }
      System.out.println("The equivalent hexadecimal number is " + hexStr);
   }
}
```

10.4 Example: Hex2Bin

Convert a hexadecimal number to its binary equivalence.

```
/*
 * Prompt user for a hexadecimal string, and convert to its binary equivalence.
 */
import java.util.Scanner;
public class Hex2Bin {
   public static void main(String[] args) {
      String hexStr;     // Input hexadecimal String
      int hexStrLen;     // Length of hexStr
      String binStr =""; // equivalent binary String

      // Binary string corresponding to Hex '0' to 'F'
      String[] binStrs
         = {"0000","0001","0010","0011","0100","0101","0110","0111",
            "1000","1001","1010","1011","1100","1101","1110","1111"};

      // Read input
      Scanner in = new Scanner(System.in);
      System.out.print("Enter a Hexadecimal string: ");
      hexStr = in.next();
      hexStrLen = hexStr.length();

      // Process the string from the left
      for (int pos = 0; pos < hexStrLen; ++pos) {
         char hexChar = hexStr.charAt(pos);
         if (hexChar >= '0' && hexChar <= '9') {
            binStr += binStrs[hexChar-'0'];  // index into the binStrs array
         } else if (hexChar >= 'a' && hexChar <= 'f') {
            binStr += binStrs[hexChar-'a'+10];
         } else if (hexChar >= 'A' && hexChar <= 'F') {
            binStr += binStrs[hexChar-'A'+10];
         } else {
            System.err.println("Error: Invalid Hex string \"" + hexStr + "\"");
            System.exit(1);  // quit
         }
      }
      System.out.println("The equivalent binary for \"" + hexStr + "\" is \"" + binStr + "\"");
   }
}
```

10.5 Example: Guess A Number

Guess a number between 0 and 99.

```
import java.util.Scanner;
public class NumberGuess {
   public static void main(String[] args) {
      int secretNumber;     // Secret number to be guessed
      int numberIn;         // The guessed number entered
      int trialNumber = 0;  // Number of trials so far
      boolean done = false; // boolean flag for loop control

      // Set up the secret number
      // Math.random() generates a double in [0.0, 1.0)
      secretNumber = (int)(Math.random()*100);

      Scanner in = new Scanner(System.in);

      while (!done) {
         ++trialNumber;
         System.out.print("Enter your guess (between 0 and 99): ");
         numberIn = in.nextInt();
         if (numberIn == secretNumber) {
            System.out.println("Congratulation");
            done = true;
         } else if (numberIn < secretNumber) {
            System.out.println("Try higher");
         } else {
            System.out.println("Try lower");
         }
      }
      System.out.println("You got in " + trialNumber +" trials");
   }
}
```

10.6 More Exercises on Java Basics

LINK TO MORE EXERCISES ON JAVA BASICS

11. (Advanced) Bitwise Operations

11.1 Bitwise Logical Operations

Bitwise operators perform operations on one or two operands on a bit-by-bit basis, as follows, in descending order of precedences.

Operator	Description	Usage
~	Bitwise NOT (inversion)	*~expr*
&	Bitwise AND	*expr1* & *expr2*
^	Bitwise XOR	*expr1* ^ *expr2*
|	Bitwise OR	*expr1* | *expr2*

Example

```
public class TestBitwiseOp {
   public static void main(String[] args) {
      int x = 0xAAAA_5555;                // a negative number (sign bit (msb) = 1)
      int y = 0x5555_1111;                // a positive number (sign bit (msb) = 0)
      System.out.printf("%d%n", x);       // -1431677611
      System.out.printf("%d%n", y);       // 1431638289
      System.out.printf("%08X%n", ~x);    // 5555AAAAH
      System.out.printf("%08X%n", x & y); // 00001111H
      System.out.printf("%08X%n", x | y); // FFFF5555H
      System.out.printf("%08X%n", x ^ y); // FFFF4444H
   }
}
```

Compound operator `&=`, `|=` and `^=` are also available, e.g., `x &= y` is the same as `x = x & y`.

Take note that:

1. `'&'`, `'|'` and `'^'` are applicable when both operands are integers (`int`, `byte`, `short`, `long` and `char`) or `boolean`s. When both operands are integers, they perform bitwise operations. When both operands are `boolean`s, they perform logical AND, OR, XOR operations (i.e., same as logical `&&`, `||` and `^`). They are not applicable to `float` and `double`. On the other hand, logical AND (`&&`) and OR (`||`) are applicable to `boolean`s only.

   ```
   System.out.println(true & true);   // logical -> true
   System.out.println(0x1 & 0xffff);  // bitwise -> 1
   System.out.println(true && true);  // logical -> true
   ```

2. The bitwise NOT (or bit inversion) operator is represented as '~', which is different from logical NOT (`!`).
3. The bitwise XOR is represented as `'^'`, which is the same as logical XOR (^).
4. The operators' precedence is in this order: `'~'`, `'&'`, `'^'`, `'|'`, `'&&'`, `'||'`. For example,

   ```
   System.out.println(true | true & false);  // true | (true & false) -> true
   System.out.println(true ^ true & false);  // true ^ (true & false) -> true
   ```

Bitwise operations are powerful and yet extremly efficient. [Example on advanced usage.]

11.2 Bit-Shift Operations

Bit-shift operators perform left or right shift on an operand by a specified number of bits. Right-shift can be either signed-extended (>>) (padded with signed bit) or unsigned-extended (>>>) (padded with zeros). Left-shift is always padded with zeros (for both signed and unsigned).

Operator	Usage	Description
<<	`operand << number`	Left-shift and padded with zeros
>>	`operand >> number`	Right-shift and padded with sign bit (signed-extended right-shift)
>>>	`operand >>> number`	Right-shift and padded with zeros (unsigned-extended right-shift)

Since all the Java's integers (byte, short, int and long) are signed integers, left-shift << and right-shift >> operators perform signed-extended bit shift. Signed-extended right shift >> pads the most significant bits with the sign bit to maintain its sign (i.e., padded with zeros for positive numbers and ones for negative numbers). Operator >>> (introduced in Java, not in C/C++) is needed to perform unsigned-extended right shift, which always pads the most significant bits with zeros. There is no difference between the signed-extended and unsigned-extended left shift, as both operations pad the least significant bits with zeros.

Example

```
public class BitShiftTest {
   public static void main(String[] args) {
      int x = 0xAAAA5555;               // a negative number (sign bit (msb) = 1)
      int y = 0x55551111;               // a positive number (sign bit (msb) = 0)
      System.out.printf("%d%n", x);     // -1431677611
      System.out.printf("%d%n", y);     // 1431638289
      System.out.printf("%08X%n", x<<1);  // 5554AAAAH
      System.out.printf("%08X%n", x>>1);  // D5552AAAH
      System.out.printf("%d%n", x>>1);    // negative
      System.out.printf("%08X%n", y>>1);  // 2AAA8888H
      System.out.printf("%08d%n", y>>1);  // positive
      System.out.printf("%08X%n", x>>>1); // 55552AAAH
      System.out.printf("%d%n", x>>>1);   // positive
      System.out.printf("%08X%n", y>>>1); // 2AAA8888
      System.out.printf("%d%n", y>>>1);   // positive

      // More efficient to use signed-right-right to perform division by 2, 4, 8,...
      int i1 = 12345;
      System.out.println("i1 divides by 2 is " + (i1 >> 1));
      System.out.println("i1 divides by 4 is " + (i1 >> 2));
      System.out.println("i1 divides by 8 is " + (i1 >> 3));
      int i2 = -12345;
      System.out.println("i2 divides by 2 is " + (i2 >> 1));
```

```
24          System.out.println("i2 divides by 4 is " + (i2 >> 2));
25          System.out.println("i2 divides by 8 is " + (i2 >> 3));
26       }
27    }
```

As seen from the example, it is more efficient to use sign-right-shift to perform division by 2, 4, 8... (power of 2), as integers are stored in binary.

[More example on advanced usage.]

11.3 Types and Bitwise Operations

The bitwise operators are applicable to integral primitive types: `byte`, `short`, `int`, `long` and `char`. `char` is treated as unsigned 16-bit integer. There are not applicable to `float` and `double`. The `'&'`, `'|'`, `'^'`, when apply to two `boolean`s, perform logical operations. Bit-shift operators are not applicable to `boolean`s.

Like binary arithmetic operations:

- `byte`, `short` and `char` operands are first promoted to `int`.
- If both the operands are of the same type (`int` or `long`), they are evaluated in that type and returns a result of that type.
- If the operands are of different types, the smaller operand (`int`) is promoted to the larger one (`long`). It then operates on the larger type (`long`) and returns a result in the larger type (`long`).

12. Algorithms

Before writing a program to solve a problem, you have to first develop the steps involved, called *algorithm*, and then translate the *algorithm* into programming statements. This is the hardest part in programming, which is also hard to teach because the it involves intuition, knowledge and experience.

An *algorithm* is a step-by-step instruction to accomplice a task, which may involve decision and iteration. It is often expressed in English-like *pseudocode*, before translating into programming statement of a particular programming language. There is no standard on how to write pseudocode - simply write something that you, as well as other people, can understand the steps involved, and able to translate into a working program.

12.1 Algorithm for Prime Testing

Ancient Greek mathematicians like Euclid and Eratosthenes (around 300-200 BC) had developed many *algorithms* (or step-by-step instructions) to work on prime numbers. By definition, a *prime* is a positive integer that is divisible by one and itself only.

To test whether a number x is a prime number, we could apply the definition by dividing x by 2, 3, 4, ..., up to $x-1$. If no divisor is found, then x is a prime number. Since divisors come in pair, there is no need to try all the factors until x-1, but up to $\sqrt{x}$.

```
// To test whether an int x is a prime
int maxFactor = (int)Math.sqrt(x);   // find the nearest integral square root of x
assume x is a prime;
for (int factor = 2; factor <= maxFactor; ++factor) {
   if (x is divisible by factor) {
      x is not a prime;
      break;    // a factor found, no need to find more factors
   }
}
```

TRY: translate the above pseudocode into a Java program called `PrimeTest`.

12.2 Algorithm for Perfect Numbers

A positive integer is called a *perfect number* if the sum of all its proper divisor is equal to its value. For example, the number `6` is perfect because its proper divisors are `1`, `2`, and `3`, and `6=1+2+3`; but the number `10` is not perfect because its proper divisors are `1`, `2`, and `5`, and `10≠1+2+5`. Other perfect numbers are `28`, `496`, ...

The following algorithm can be used to test for perfect number:

```
// To test whether int x is a perfect number
int sum = 0;
for (int i = 1; i < x; ++i) {
   if (x is divisible by i) {
      i is a proper divisor;
      add i into the sum;
   }
}
if (sum == x)
   x is a perfect number
else
   x is not a perfect number
```

TRY: translate the above pseudocode into a Java program called `PerfectNumberTest`.

12.3 Algorithm on Computing Greatest Common Divisor (GCD)

Another early algorithm developed by ancient Greek mathematician Euclid (300 BC) is to find the Greatest Common Divisor (GCD) (or Highest Common Factor (HCF)) of two integers. By definition, `GCD(a,b)` is the largest factor that divides both `a` and `b`.

Assume that `a` and `b` are positive integers and `a >= b`, the Euclidean algorithm is based on these two properties:

```
1.  GCD(a, 0) = a
2.  GCD(a, b) = GCD(b, a mod b), where "a mod b" denotes the remainder of a divides by b.
```

For example,

```
GCD(15, 5) = GCD(5, 0) = 5
GCD(99,88) = GCD(88,11) = GCD(11,0) = 11
GCD(3456,1233) = GCD(1233,990) = GCD(990,243) = GCD(243,18) = GCD(18,9) = GCD(9,0) = 9
```

The Euclidean algorithm is as follows:

```
GCD(a, b)  // assume that a >= b
while (b != 0) {
   // Change the value of a and b: a ← b, b ← a mod b, and repeat until b is 0
   temp ← b
   b ← a mod b
   a ← temp
}
// after the loop completes, i.e., b is 0, we have GCD(a, 0)
GCD is a
```

Before explaining the algorithm, suppose we want to exchange (or swap) the values of two variables x and y. Explain why the following code does not work.

```
int x = 55, y=66;
// swap the values of x and y
x = y;
y = x;
```

To swap the values of two variables, we need to define a temporary variable as follows:

```
int x = 55, y=66;
int temp;
// swap the values of x and y
temp = y;
y = x;
x = temp;
```

Let us look into the Euclidean algorithm, `GCD(a, b) = a`, if `b` is `0`. Otherwise, we replace `a` by `b`; `b` by `(a mod b)`, and compute `GCD(b, a mod b)`. Repeat the process until the second term is `0`. Try this out on pencil-and-paper to convince yourself that it works.

TRY: Write a program called `GCD`, based on the above algorithm.

12.4 Exercises on Algorithm

LINK TO EXERCISES ON ALGORITHMS

13. Summary

This chapter covers the Java programming basics:

- Variables, literals, expressions and statements.
- The concept of type and Java's eight primitive types: `byte`, `short`, `int`, `long`, `float`, `double`, `char`, and `boolean`.
- Implicit and explicit type-casting.
- Operators: assignment (=), arithmetic operators (+, -, *, /, %), increment/decrement (++, --) relational operators (==, !=, >, >=, <, <=), logical operators (&&, ||, !, ^) and conditional (? :).
- Three flow control constructs: sequential, condition (`if`, `if-else`, `switch-case` and nested-if) and loops (`while`, `do-while`, `for` and nested loops).
- Input (via `Scanner`) & Output (`print()`, `println()` and `printf()`) operations.
- Arrays and the enhanced for-loop.
- Methods and passing parameters into methods.
- The advanced bitwise logical operators (&, |, ~, ^) and bit-shift operators (<<, >>, >>>).
- Developing algorithm for solving problems.

Link to Java References and Resources

More References and Resources

1. "Code Conventions for the Java Programming Language" @ http://www.oracle.com/technetwork/java/codeconv-138413.html (MUST READ), Sun Microsystems (now Oracle).

Latest version tested: JDK 1.7.0_17
Last modified: April, 2013