



Object-Oriented Programming and Design with Java

HaQT



HUS
VNU UNIVERSITY OF SCIENCE

TABLE OF CONTENTS

7

PART I

Java Basics

| | | |
|------|---|-----|
| 1 | Introduction to Java Programming | 9 |
| 1.1 | Getting Started - Your First Java Program | 9 |
| 1.2 | Java Programming Steps | 11 |
| 1.3 | Computer Architecture | 12 |
| 1.4 | Java Terminology and Syntax | 13 |
| 1.5 | Java Program Template | 13 |
| 1.6 | Output via System.out.println() and System.out.print() | 14 |
| 1.7 | Let's Write a Program to Add a Few Numbers | 17 |
| 1.8 | What is a Program? | 19 |
| 1.9 | What is a Variable? | 22 |
| 1.10 | Basic Arithmetic Operations | 25 |
| 1.11 | What If Your Need To Add a Thousand Numbers? Use a Loop | 29 |
| 1.12 | Conditional (or Decision) | 32 |
| 1.13 | Summary | 35 |
| 2 | Java Basics | 37 |
| 2.1 | Basic Syntaxes | 37 |
| 2.2 | Variables and Types | 44 |
| 2.3 | Primitive Types and String | 50 |
| 2.4 | Basic Operations | 64 |
| 2.5 | Flow Control | 80 |
| 2.6 | Input/Output | 108 |

2.7 Writing Correct and Good Programs 129

2.8 More on Loops - Nested-Loops, break & continue 133

2.9 String and char operations 148

2.10 Arrays 167

2.11 Methods (Functions) 184

2.12 Command-Line Arguments 210

2.13 (Advanced) Bitwise Operations 214

2.14 Algorithms 217

2.15 Summary 223

2.16 Exercises 224

241 | PART II
Object-Oriented Programming

243 | PART III
Collections Framework

245 | PART IV
Correctness, Robustness, Efficiency

247 | PART V
Design Patterns

249 | PART VI
Java Generics

251

PART VII

Java Concurrency

253

PART VIII

Java GUI

255

PART IX

References

Part I Java Basics

1 Introduction to Java Programming

JDK

You should have already installed Java Development Kit (JDK) and written a "HelloWorld" program. Otherwise, Read "How to Install JDK".

Programming Text Editor

Do NOT use Notepad (Windows) or TextEdit (macOS) for programming. Install a programming text editor, which does syntax color highlighting. For example,

- For Windows: Sublime Text, Atom, NotePad++, TextPad.
- For macOS: Sublime Text, Atom, jEdit, gEdit.
- For Ubuntu: gEdit.

1.1 Getting Started - Your First Java Program

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

Step 1. Write the Source Code: Enter the following source codes, which defines a class called "Hello", using a programming text editor. Do not enter the line numbers (on the left pane), which were added to aid in the explanation.

Save the source file as "Hello.java". A Java source file should be saved with a file extension of ".java". The filename shall be the same as the classname - in this case "Hello". Filename and classname are *case-sensitive*.

Hello.java



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

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

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

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

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

```
Command window
java Hello
2 Hello, world!
```

How it Works

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

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

1. *Multi-Line Comment*: begins with `/*` and ends with `*/`, and may span more than one lines (as in Lines 1 – 3).
2. *End-of-Line (Single-Line) Comment*: begins with `//` and lasts until the end of the current line (as in Lines 4, 5, and 6).

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

The basic unit of a Java program is a *class*. A class called "Hello" is defined via the keyword "class" in lines 4 – 8. The braces encloses the *body* of the class.

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

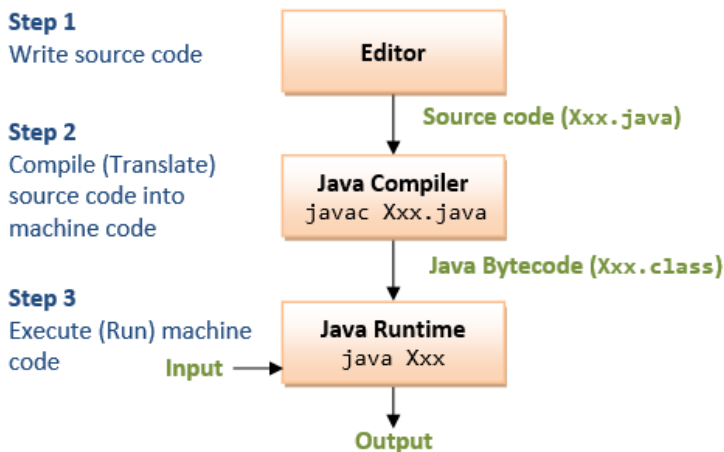
```
public static void main(String[] args) {  
    .....  
}
```

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

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

In line 6, the programming statement `System.out.println("Hello, world!")` is used to print the string "Hello, world!" to the display console. A string is surrounded by a pair of double quotes and contain texts. The text will be printed as it is, without the double quotes. A programming statement ends with a semi-colon (;).

1.2 Java Programming Steps



Java Programming Steps

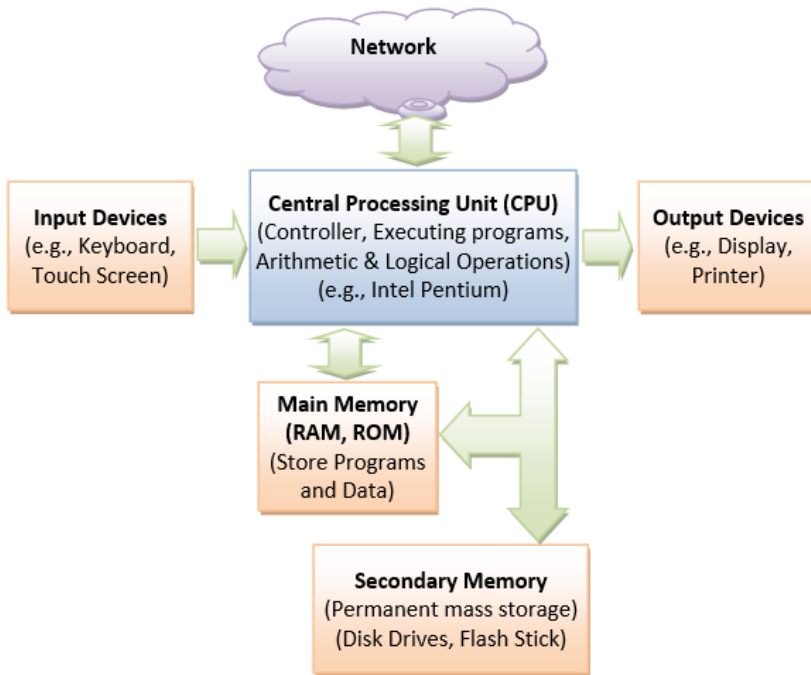
The steps in writing a Java program is illustrated as above:

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

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

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

1.3 Computer Architecture



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

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

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

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

fetch the desired program from the secondary memory to the main memory for execution upon user's command.

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

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

1.4 Java Terminology and Syntax

Comments: A *multi-line comment* begins with `/*` and ends with `*/`, and may span multiple lines. An *end-of-line (single-line) comment* begins with `//` and lasts till the end of the current line. Comments are NOT executable statements and are ignored by the compiler. But they provide useful explanation and documentation. I strongly suggest that you write comments *liberally* to explain your thought and logic.

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

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

White spaces: Blank, tab, and newline are collectively called *whitespace*. Extra whitespaces are ignored, i.e., only one whitespace is needed to separate the tokens. Nonetheless, extra whitespaces improve the readability, and I strongly suggest you use extra spaces and newlines to improve the readability of your code.

Case Sensitivity: Java is *case sensitive* - a **ROSE** is NOT a *Rose*, and is NOT a *rose*. The *filename*, which is the same as the class name, is also case-sensitive.

1.5 Java Program Template

You can use the following *template* to write your Java programs. Choose a meaningful "*Classname*" that reflects the *purpose* of your program, and write your programming statements inside the body of the `main()` method. Don't worry about the other terms and keywords now. It will be explained later. Provide comments in your program!

Classname.java



```

1  /**
2   * Comment to state the purpose of the program
3   */
4   public class Classname { // Choose a meaningful Classname.
5       // Save as "Classname.java"
6       public static void main(String[] args) { // Entry point of the program
7           // Your programming statements here!!!
8       }
9   }

```

1.6

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

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

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

Try the following program and explain the output produced:



```

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

```



```

15 System.out.print("Hello, ");
    System.out.print(" ");           // Print a space
17 System.out.println("world!");
    System.out.println("Hello, world!");
19 }
    }

```

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

Command window

Hello world!
2 Hello world again!

4 Hello world!Hello world again!
Hello, world!
6 Hello, world!

1.6.1 Exercises using System.out.println()

1. Write a program called **AmericanFlag** to print an American flag on the screen.

Command window

```

* * * * * =====
2  * * * * * =====
* * * * * =====
4  * * * * * =====
* * * * * =====
6  * * * * * =====
* * * * * =====
8  * * * * * =====
* * * * * =====
10 =====
=====
12 =====
=====
14 =====
=====

```

2. Write a program called **PrintSheepPattern** to print the following pattern on the screen.

```

Command window
1      '  '
      _
      (oo)
3  /=====//
  /  ||  @@  ||
5  *  ||----||
      VV    VV
7      '  '  '  '

```

3. Write 4 programs, called **PrintCheckerPattern**, **PrintSquarePattern**, **PrintTriangularPattern** and **PrintStarPattern** to print each of the following patterns. Use ONE *System.out.println(...)* (print-line) statement for EACH LINE of outputs. Take note that you need to print the preceding blanks.

```

Command window
1  * * * * *      * * * * *      * * * * *      *
2  * * * * *      *       *      *       *      * * * * *
3  * * * * *      *       *      *       *      * *
4  * * * * *      *       *      *       *      * *
5  * * * * *      * * * * *      *       *      * *
      (a)          (b)          (c)          (d)

```

4. Write a program called **JavaPattern** to display the following pattern on the screen.

```

Command window
      J   a   v       v   a
2     J   a a   v   v   a a
   J J   aaaaa   VV   aaaaa
4    JJ  a     a   V a     a

```

5. Write a program called **Face** to print a face on the screen.

```

Command window
      +-----+
2     [  o o  ]
      |   ^   |
4     | ' _ ' |
      +-----+

```


1.7 Let's Write a Program to Add a Few Numbers

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



```
1  /**
   * FiveIntegerSum.java
   * Add five integers and display their sum.
   */
5  public class FiveIntegerSum {    // Save as "FiveIntegerSum.java"
    public static void main(String[] args) {
6      // Declare 5 integer variables and assign a value
7      int number1 = 11;
9      int number2 = 22;
10     int number3 = 33;
11     int number4 = 44;
12     int number5 = 55;
13
14     int sum; // Declare an integer variable called sum to hold the sum
15     sum = number1 + number2 + number3 + number4 + number5; // Compute sum
16     System.out.print("The sum is "); // Print a descriptive string
17     // Cursor stays after the printed string
18     System.out.println(sum); // Print the value stored in variable sum
19     // Cursor advances to the beginning of next line
20 }
21 }
```

Save the source code as "FiveIntegerSum.java" (which is the same as the classname). Compile and run the program.

The expected output is:

Command window

```
1 The sum is 165
```

How It Works?

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

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

Alternatively, you could declare many variables in one statement separated by commas, e.g.,



```
1 // Declare many variables in one statement separated by commas
  int number1 = 11, number2 = 22, number3 = 33, number4 = 44, number5 = 55;
```

```
int sum;
```

declares an *int* (integer) variable called *sum*, without assigning an initial value - its value is to be computed and assigned later.

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

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

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

```
System.out.println(sum);
```

Line 16 prints a descriptive string. A *String* is surrounded by double quotes, and will be printed as it is (without the double quotes). The cursor stays after the printed string. Try using *println()* instead of *print()* and study the output.

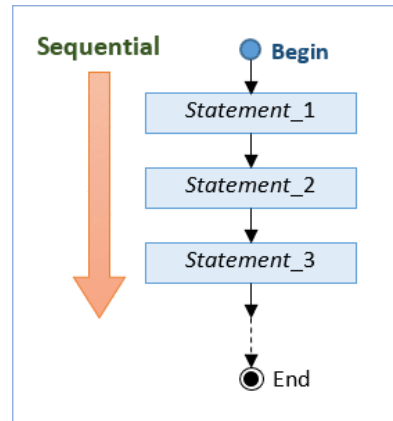
Line 18 prints the *value* stored in the variable *sum* (in this case, the sum of the five integers). You should not surround a variable to be printed by double quotes; otherwise, the text will get printed instead of the value stored in the variable. The cursor advances to the beginning of next line after printing. Try using *print()* instead of *println()* and study the output.

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

1.8 What is a Program?

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

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



Example

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

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



```

1  /**
2   * CircleComputation.java
3   * Print the area and circumference of a circle , given its radius .
4   */
5
6   public class CircleComputation { // Save as "CircleComputation.java"
7       public static void main(String[] args) {
8           // Declare 3 double variables to hold radius, area and circumference.
9           // A "double" holds floating-point number with an optional fractional part.
10          double radius;           // The variable to hold radius value
11          double area;             // The variable to hold radius area
12          double circumference;    // The variable to hold radius circumference
13
14          // Declare a double to hold PI. Declare as "final" to specify that
15          // its value cannot be changed (i.e. constant).
16          final double PI = 3.14159265;
17
18          // Assign a value to radius. (We shall read in the value from the keyboard later.)
19          radius = 1.2;
20
21          // Compute area and circumference
  
```



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

24    // Print results
    System.out.print("The radius is "); // Print description
26    System.out.println(radius); // Print the value stored in the variable
    System.out.print("The area is ");
28    System.out.println(area);
    System.out.print("The circumference is ");
30    System.out.println(circumference);
    }
32 }
```

The expected outputs are:

```
Command window
The radius is 1.2
2 The area is 4.523893416
The circumference is 7.5398223600000005
```

How It Works?

double radius, area, circumference;

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

final double PI = 3.14159265;

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

radius = 1.2;

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

area = radius * radius * PI;

circumference = 2.0 * radius * PI;

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

System.out.print("The radius is ");

System.out.println(radius);

System.out.print("The area is ");

System.out.println(area);

System.out.print("The circumference is ");

System.out.println(circumference);

print the results with proper descriptions.

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

1.8.1 Exercises

1. Write a program called **Operations** to print the results of the following operations.

- $-5 + 8 * 6$
- $(55+9)$
- $20 + -3*5 / 8$
- $5 + 15 / 3 * 2 - 8$

Expected Output:



```
Command window
1 43
  1
3 19
  13
```

2. Write a program called **SpecifiedFormula** to compute a specified formula.

Specified Formula: $4.0 * (1 - (1.0/3) + (1.0/5) - (1.0/7) + (1.0/9) - (1.0/11))$

Expected Output:



```
Command window
2.9760461760461765
```

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

1.9 What is a Variable?

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

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

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

A variable has a type. Examples of *type* are:

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

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

| TYPE | NAME | VALUE | |
|--------|-----------|-----------|-----------------------------------|
| int | number | 1 | Stored only Integer |
| int | sum | 500500 | Stored only Integer |
| double | radius | 5.5 | Stored only floating-point number |
| double | area | 95.0334 | Stored only floating-point number |
| String | greeting | Hello | Stored only texts |
| String | statusMsg | Game Over | Stored only texts |

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

The above diagram illustrates three types of variables: *int*, *double* and *String*.

- An *int* variable stores an integer (whole number);
- a *double* variable stores a real number (which includes integer as a special form of real number);
- a *String* variable stores texts.

Declaring and Using Variables

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



```

1 // Declare a variable of a type
  type varName;
3
  // Declare multiple variables of the SAME type
5 type varName1, varName2 ,...;

7 // Declare a variable of a type, and assign an initial value
  type varName = initialValue ;
9
  // Declare variables with initial values
11 type varName1 = initialValue1 , varName2 = initialValue2 , ...;
```

For examples,



```

1 // Declare a variable named "sum" of the type "int" for storing an integer .
  // Terminate the statement with a semi-colon.
3 int sum;

5 // Declare a variable named "average" of the type "double" for storing
  // a real number.
7 double average;

9 // Declare 2 "int" variables named "number1" and "number2", separated by a comma.
  int number1, number2;
11
  // Declare an "int" variable , and assign an initial value.
13 int height = 20;

15 // Declare a "String" variable , and assign an initial value.
  String msg = "Hello";
```

Take note that:

- Each *variable declaration statement* begins with a *type*, and applicable for only that type. That is, you cannot mix 2 types in one variable declaration statement.
- Each *statement* is terminated with a semi-colon (;).
- In multiple-variable declaration, the variable names are separated by commas (,).
- The symbol '=', known as the *assignment operator*, can be used to assign an initial value to a variable, in a declaration statement.

More examples,



```
1 int number;    // Declare a variable named "number" of the type "int" (integer).
2 number = 99;   // Assign an integer value of 99 to the variable "number".
3 number = 88;   // Re-assign a value of 88 to "number".
4 number = number + 1; // Evaluate "number + 1", and assign the result back to "number".

6 int sum = 0;    // Declare an int variable named "sum" and assign an initial value of 0.
7 sum = sum + number; // Evaluate "sum + number", and assign the result
8                      // back to "sum", i.e. add number into sum.
9 int num1 = 5, num2 = 6; // Declare and initialize 2 int variables in
10                      // one statement, separated by a comma.

12 double radius = 1.5; // Declare a variable named "radius", and initialize to 1.5.

14 String msg;      // Declare a variable named msg of the type "String ".
15 msg = "Hello";    // Assign a double-quoted text string to the String variable.

16
17 int number;      // ERROR: The variable named "number" has already been declared.
18 sum = 55.66;     // ERROR: The variable "sum" is an int. It cannot be assigned a double.
19 sum = "Hello";   // ERROR: The variable "sum" is an int. It cannot be assigned a string.
```

Take note that:

- Each variable can only be declared once. (You cannot have two houses with the same address.)
- In Java, you can declare a variable anywhere inside your program, as long as it is declared before it is being used. (Some older languages require you to declare all the variables at the beginning of the program.)
- Once a variable is declared, you can assign and re-assign a value to that variable, via the assignment operator '='.

- Once the type of a variable is declared, it can only store a value of that 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".
- Once declared, the type of a variable CANNOT be changed.

x = x + 1?

Assignment in programming (denoted as '=') is different from equality in Mathematics (also denoted as '='). For example, " $x = x + 1$ " is invalid in Mathematics. However, in programming, it means compute the value of x plus 1, and *assign* the result back to variable x .

" $x + y = 1$ " is valid in Mathematics, but is invalid in programming. In programming, the RHS (Right-Hand Side) of '=' has to be evaluated to a value; while the LHS (Left-Hand Side) shall be a variable. That is, evaluate the RHS first, then assign the result to LHS.

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

1.10

Basic Arithmetic Operations

The basic *arithmetic operations* are:

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

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

Example

The following program illustrates these arithmetic operations:



```

1  /**
   * ArithmeticTest.java
3  * Test Arithmetic Operations
   */
5  public class ArithmeticTest { // Save as "ArithmeticTest.java"
    public static void main(String[] args) {
7      int number1 = 98; // Declare an int variable number1 and initialize it to 98
      int number2 = 5; // Declare an int variable number2 and initialize it to 5
9      // Declare 5 int variables to hold results
      int sum, difference, product, quotient, remainder;
11
      // Perform arithmetic Operations
13      sum = number1 + number2;
      difference = number1 - number2;
15      product = number1 * number2;
      quotient = number1 / number2;
17      remainder = number1 % number2;
19
      // Print results
      System.out.print("The sum, difference, product, quotient and remainder of ");
21      System.out.print(number1); // Print the value of the variable
      System.out.print(" and ");
23      System.out.print(number2);
      System.out.print(" are ");
25      System.out.print(sum);
      System.out.print(", ");
27      System.out.print(difference);
      System.out.print(", ");
29      System.out.print(product);
      System.out.print(", ");
31      System.out.print(quotient);
      System.out.print(", and ");
33      System.out.println(remainder);
35
      ++number1; // Increment the value stored in the variable "number1" by 1
      // Same as "number1 = number1 + 1"
37
      --number2; // Decrement the value stored in the variable "number2" by 1
      // Same as "number2 = number2 - 1"
39
41      // Print description and variable
      System.out.println("number1 after increment is " + number1);

```



```

43 System.out.println("number2 after decrement is " + number2);

45 quotient = number1 / number2;
System.out.println("The new quotient of " + number1 + " and "
47 + number2 + " is " + quotient);
    }
49 }

```

The expected outputs are:

Command window

```

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

```

How It Works?

```
int number1 = 98;
```

```
int number2 = 5;
```

```
int sum, difference, product, quotient, remainder;
```

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

```
sum = number1 + number2;
```

```
difference = number1 - number2;
```

```
product = number1 * number2;
```

```
quotient = number1 / number2;
```

```
remainder = number1 % number2;
```

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

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

```
.....
```

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

For example,



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

`++number1;`

`--number2;`

illustrate the increment and decrement operations. Unlike '+', '-', '**', '/' and '%', which work on two operands (*binary operators*), '++' and '--' operate on only one operand (*unary operators*). ++x is equivalent to x = x + 1, i.e., increment x by 1.

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

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

print the new values stored after the increment/decrement operations. Take note that instead of using many print() statements as in Lines 18-31, we could simply place all the items (text strings and variables) into one println(), with the items separated by '+'. In this case, '+' does not perform *addition*. Instead, it *concatenates* or *joins* all the items together.

1.10.1 Exercises

1. Combining lines 18 – 31 into one single `println()` statement, using '+' to *concatenate* all the items together.
2. In Mathematics, we could omit the multiplication sign in an arithmetic expression, e.g., $x = 5a + 4b$. In programming, you need to explicitly provide all the operators, i.e., $x = 5 * a + 4 * b$. Try printing the sum of 31 times of number1 and 17 times of number2.
3. Write a program called Multiplication that takes a number as input and prints its multiplication table up to 10.

Expected Output :

```
Command window
Input a number: 8
2 8 x 1 = 8
  8 x 2 = 16
4 8 x 3 = 24
  ...
6 8 x 10 = 80
```

1.11

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

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

Example

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



```

1  /**
2  * RunningNumberSum.java
3  * Sum from a lowerbound to an upperbound using a while-loop
4  */
5  public class RunningNumberSum { // Save as "RunningNumberSum.java"
6  public static void main(String[] args) {
7      final int LOWERBOUND = 1; // Store the lowerbound
8      final int UPPERBOUND = 1000; // Store the upperbound
9      int sum = 0; // Declare an int variable "sum" to accumulate the numbers
10     // Set the initial sum to 0
11
12     // Use a while-loop to repeatedly sum from the lowerbound to the upperbound
13     int number = LOWERBOUND;
14     while (number <= UPPERBOUND) {
15         // number = LOWERBOUND, LOWERBOUND+1, LOWERBOUND+2, ...,
16         // UPPERBOUND for each iteration
17         sum = sum + number; // Accumulate number into sum
18         ++number; // increment number
19     }
20
21     // Print the result
22     System.out.println("The sum from " + LOWERBOUND + " to "
23         + UPPERBOUND + " is " + sum);
24 }

```

The expected output is:

Command window

The sum from 1 to 1000 is 500500

How It Works?

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

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

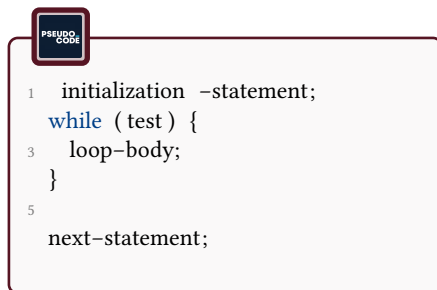
```
int sum = 0;
```

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

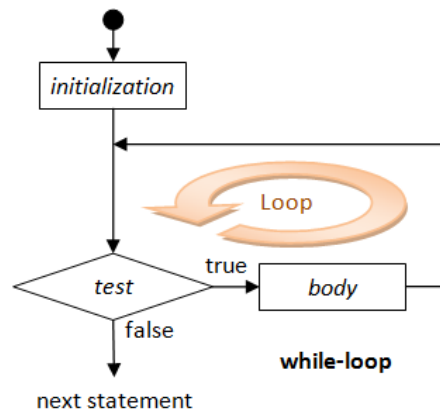
```
int number = LOWERBOUND;  
while (number <= UPPERBOUND) {  
    sum = sum + number;  
    ++number;  
}
```

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

Syntax



Flowchart



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

In our example, the *initialization* statement declares an `int` variable named `number` and initializes it to `LOWERBOUND`. The *test* checks if `number` is equal to or less than the `UPPERBOUND`. If it is *true*, the current value of `number` is added into the `sum`, and the statement `++number` increases the value of `number` by 1. The *test* is then checked again and the process repeats until the *test* is *false* (i.e., `number` increases to `UPPERBOUND + 1`), which causes the loop to terminate. Execution then continues to the next statement (in line 21).

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

In this example, the loop repeats `UPPERBOUND - LOWERBOUND + 1` times. After the loop is completed, Line 19 prints the result with a proper description.

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

1.11.1 Exercises

1. Modify the above program (called **RunningNumberSum1**) to sum all the numbers from 9 to 899. (Ans: 404514)
2. Modify the above program (called **RunningNumberOddSum**) to sum all the odd numbers between 1 to 1000. (Hint: Change the post-processing statement to "`number = number + 2`". Ans: 250000)
3. Modify the above program (called **RunningNumberMod7Sum**) to sum all the numbers between 1 to 1000 that are divisible by 7. (Hint: Modify the initialization statement to begin from 7 and post-processing statement to increment by 7. Ans: 71071)
4. Modify the above program (called **RunningNumberSquareSum**) to find the sum of the square of all the numbers from 1 to 100, i.e. $1 * 1 + 2 * 2 + 3 * 3 + \dots$. (Hint: Modify the `sum = sum + number` statement. Ans: 338350)
5. Modify the above program (called **RunningNumberProduct**) to compute the product of all the numbers from 1 to 10. (Hint: Use a variable called `product` instead of `sum` and initialize `product` to 1. Modify the `sum = sum + number` statement to do multiplication on variable `product`. Ans: 3628800)
6. Write a program called **FactorCounter** to accept an integer and count the factors of the number.
7. Write a program called **PrimesSum** to compute the sum of the first 100 prime numbers.

Sample Output:

```
Command window
Sum of the first 100 prime numbers: 24133
```

1.12 Conditional (or Decision)

What if you want to sum all the odd numbers and also all the even numbers between 1 and 1000? You could declare two variables, *sumOdd* and *sumEven*, to keep the two sums. You can then use a conditional statement to check whether the number is odd or even, and accumulate the number into the respective sums.



```

1  /**
   * Sum the odd numbers and the even numbers from a lowerbound to an upperbound.
3  */
   public class OddEvenSum { // Save as "OddEvenSum.java"
5     public static void main(String[] args) {
        final int LOWERBOUND = 1;
7         final int UPPERBOUND = 1000;
        int sumOdd = 0;           // For accumulating odd numbers, init to 0
9         int sumEven = 0;       // For accumulating even numbers, init to 0
        int number = LOWERBOUND;
11        while (number <= UPPERBOUND) {
            // number = LOWERBOUND, LOWERBOUND + 1, LOWERBOUND + 2, ...,
13            // UPPERBOUND for each iteration
            if (number % 2 == 0) { // Even
15                sumEven += number; // Same as sumEven = sumEven + number
            } else { // Odd
17                sumOdd += number; // Same as sumOdd = sumOdd + number
            }
19            ++number; // Next number
        }

21        // Print the result
23        System.out.println("The sum of odd numbers from " + LOWERBOUND
            + " to " + UPPERBOUND + " is " + sumOdd);
25        System.out.println("The sum of even numbers from " + LOWERBOUND
            + " to " + UPPERBOUND + " is " + sumEven);
27        System.out.println("The difference between the two sums is "
            + (sumOdd - sumEven));
29    }
}

```

The expected outputs are:

Command window

```

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

```


How It Works?

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

declares and initializes the upperbound and lowerbound constants.

```
int sumOdd = 0;
int sumEven = 0;
```

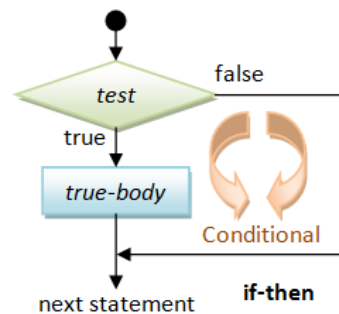

declare two int variables named *sumOdd* and *sumEven* and initialize them to 0, for accumulating the odd and even numbers, respectively.

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

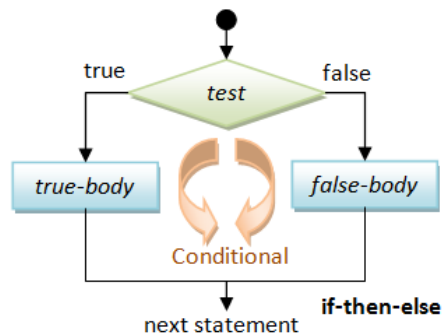
This is a *conditional* statement. The conditional statement can take one these forms: *if-then* or *if-then-else*.

if-then Syntax


```
1 // if-then
2 if ( test ) {
3     true-body;
4 }
```

Flowchart**if-then-else Syntax**


```
1 // if-then-else
2 if ( test ) {
3     true-body;
4 } else {
5     false-body;
6 }
```

Flowchart

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

In our program, we use the *remainder* or *modulus* operator (%) to compute the remainder of number divides by 2. We then compare the remainder with 0 to test for even number. Furthermore, *sumEven += number* is a *shorthand* for *sumEven = sumEven + number*.

Comparison Operators

There are six comparison (or relational) operators:

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

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

Combining Simple Conditions

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

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

| Operator | Mode | Usage | Meaning | Example |
|----------|--------------|--------|-------------|------------------------|
| && | Binary | x && y | Logical AND | (x >= 1) && (x <= 100) |
| | Binary | x y | Logical OR | (x < 1) (x > 100) |
| ! | Unary Prefix | !x | Logical NOT | !(x == 8) |

For examples:



```

// Return true if x is between 0 and 100 ( inclusive )
2 (x >= 0) && (x <= 100) // AND (&&)
// Incorrect to use 0 <= x <= 100
4
// Return true if x is outside 0 and 100 ( inclusive )
6 (x < 0) || (x > 100) // OR (||)
!((x >= 0) && (x <= 100)) // NOT (!), AND (&&)
8
// Return true if "year" is a leap year. A year is a leap year if
10 // it is divisible by 4 but not by 100, or it is divisible by 400.
((year % 4 == 0) && (year % 100 != 0)) || (year % 400 == 0)

```

1.12.1 Exercises

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

Expected Output:

Command window

```

1 Input an integer number less than ten billion : 125463
Number of digits in the number: 6

```

1.13

Summary

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

2 Java Basics

This chapter explains the basic syntaxes of the Java programming language. To be proficient in a programming language, you need to master two things:

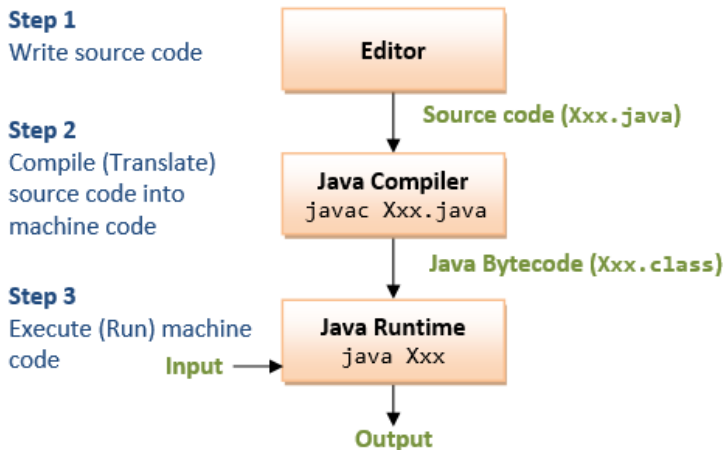
- The syntax of the programming language: Not too difficult to learn a small set of keywords and syntaxes. For examples, JDK 1.8 has 48 keywords; C11 has 44, and C++11 has 73.
- The Application Program Interface (API) libraries associated with the language: You don't want to write everything from scratch yourself. Instead, you can re-use the available code in the library. Learning library could be difficult as it is really huge, evolving and could take on its own life as another programming language.

The first few sections, I have to explain the basic concepts with some details.

2.1 Basic Syntaxes

2.1.1 Steps in Writing a Java Program

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



Step 1. Write the source code Xxx.java using a programming text editor (such as Sublime Text, Atom, Notepad++, Textpad, gEdit) or an IDE (such as Eclipse, NetBeans, IntelliJ IDEA).

Step 2. Compile the source code Xxx.java into Java portable bytecode Xxx.class using the JDK Compiler by issuing command:

Command window

```
javac Xxx.java
```

Step 3. Run the compiled bytecode Xxx.class with the input to produce the desired output, using the Java Runtime by issuing command:

Command window

```
1 java Xxx
```

2.1.2 Java Program Template

You can use the following *template* to write your Java programs. Choose a meaningful "Classname" that reflects the *purpose* of your program, and write your programming statements inside the body of the *main()* method. Don't worry about the other terms and keywords now. They will be explained later. Provide comments in your program!



```
1  /**
   * Classname.java
   * Comment to state the purpose of the program
   */
5  public class Classname { // Choose a meaningful Classname.
                           // Save as "Classname.java"
7      public static void main(String[] args) { // Entry point of the program
9          // Your programming statements here!!!
11     }
    }
```

2.1.3 A Sample Program Illustrating Sequential, Decision and Loop Constructs

Below is a simple Java program that demonstrates the three basic programming constructs: *sequential*, *loop*, and *conditional*.



```

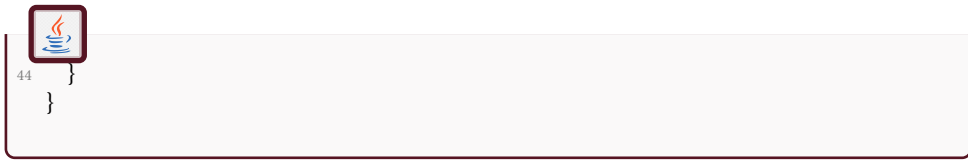
/**
2  * OddEvenSum.java
  * Find the sums of the running odd numbers and even numbers from a given lowerbound
4  * to an upperbound. Also compute their absolute difference.
  */
6  public class OddEvenSum { // Save as "OddEvenSum.java"
    public static void main(String[] args) {
8      // Declare variables
        final int LOWERBOUND = 1;
10     final int UPPERBOUND = 1000; // Define the bounds
        int sumOdd = 0;    // For accumulating odd numbers, init to 0
12     int sumEven = 0;    // For accumulating even numbers, init to 0
        int absDiff;      // Absolute difference between the two sums
14
        // Use a while loop to accumulate the sums from LOWERBOUND to UPPERBOUND
16     int number = LOWERBOUND; // loop init
        while (number <= UPPERBOUND) { // loop test
18         // number = LOWERBOUND, LOWERBOUND + 1, ..., UPPERBOUND
            // A if-then-else decision
20         if (number % 2 == 0) { // Even number
                sumEven += number; // Same as sumEven = sumEven + number
22         } else { // Odd number
                sumOdd += number; // Same as sumOdd = sumOdd + number
24         }
            ++number; // loop update for next number
26     }

28     // Another if-then-else Decision
        if (sumOdd > sumEven) {
30         absDiff = sumOdd - sumEven;
        } else {
32         absDiff = sumEven - sumOdd;
        }
34

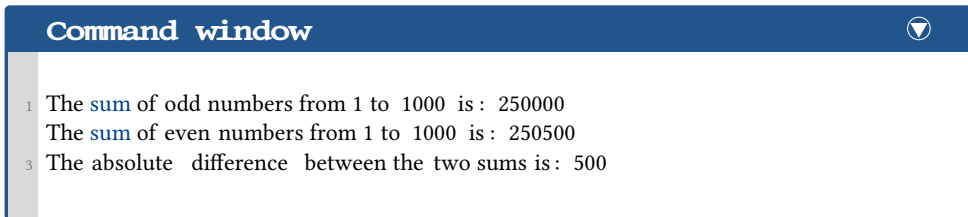
        // OR using one liner conditional expression
36     // absDiff = (sumOdd > sumEven) ? sumOdd - sumEven : sumEven - sumOdd;

38     // Print the results
        System.out.println("The sum of odd numbers from " + LOWERBOUND
40         + " to " + UPPERBOUND + " is: " + sumOdd);
        System.out.println("The sum of even numbers from " + LOWERBOUND
42         + " to " + UPPERBOUND + " is: " + sumEven);
        System.out.println("The absolute difference between the two sums is: " + absDiff);

```



The expected outputs are:



2.1.4 Comments

Comments are used to document and explain your code and your program logic. Comments are not programming statements. They are ignored by the compiler and have no consequences to the program execution. Nevertheless, comments are VERY IMPORTANT for providing documentation and explanation for others to understand your programs (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 multiple lines. `/** */` is a *special documentation* comment. These comment can be extracted to produce documentation.
2. *End-of-Line (Single-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 code.

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.

2.1.5 Statements and Blocks

Statement

A programming *statement* is the smallest independent 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 because period crashes with decimal point - it is challenging for the dumb

computer to differentiate between period and decimal point in the early days of computing!). For examples,



```

1 // Each of the following lines is a programming statement, which ends with a semi-colon (;).
  // A programming statement performs a piece of programming action.
3 int number1 = 10;
  int number2, number3 = 99;
5 int product;
  number2 = 8;
7 product = number1 * number2 * number3;
  System.out.println("Hello");

```

Block

A *block* is a group of programming statements surrounded by a pair of 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 loop, which may contain multiple statements but are treated as one unit (one body). There is no need to put a semi-colon after the closing brace to end a compound statement. Empty block (i.e., no statement inside the braces) is permitted. For examples,



```

  // Each of the followings is a "compound" statement comprising one or more blocks
2 // of statements. No terminating semi-colon needed after the closing brace to
  // end the "compound" statement.
4 // Take note that a "compound" statement is usually written over a few lines for readability.
  if (mark >= 50) { // A if statement
6     System.out.println("PASS");
      System.out.println("Well Done!");
8     System.out.println("Keep it Up!");
  }
10
  if (input != -1) { // A if-else statement
12     System.out.println("Continue");
  } else {
14     System.out.println("Exit");
  }
16
  i = 1;
18 while (i < 8) { // A while-loop statement
      System.out.print(i + " ");
20     ++i;
  }
22
  public static void main(String[] args) { // A method definition statement
24     ... statements ...

```



```
26  
public class Hello { // A class definition statement  
28 ... statements ...  
}
```

2.1.6 White Spaces and Formatting Source Code

White Spaces

Blank, *tab* and *newline* are collectively called *white spaces*.

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



```
1 // Cannot write "intsum". Need at least one white space between "int" and "sum"  
int sum = 0;  
3  
// Again, need at least a white space between "double" and "average"  
5 double average;
```

Java, like most of the programming languages, ignores extra white spaces. That is, multiple contiguous white spaces are treated as a single white space. Additional white spaces and extra lines are ignored, e.g.,



```
1 // Same as above with many redundant white spaces. Hard to read.  
int sum  
3 =0 ;  
5 double  
average  
7 ;  
9 // Also same as above with minimal white space. Also hard to read  
int sum=0;double average;
```

Formatting Source Code

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 to align the ending brace with the start of the statement. Pair-up the properly. Unbalanced is one of the most common syntax errors for beginners.

Indentation

Indent each level of the body of a block by an extra 3 or 4 spaces according to the hierarchy of the block. Don't use tab because tab-spaces is editor-dependent.



```
1  /**
   * ClassName.java
3  * Recommended Java programming style (Documentation comments about the class)
   */
5  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
7
   // Use empty line liberally to improve readability
9   // Sequential statements
   statement-1;
11  statement-2;

13  // A if-else statement
   if ( test ) {
15      true-statements;
   } else {
17      false-statements;
   }

19
   // A loop statement
21  init ;
   while ( test ) {
23      body-statements;
       update;
25  }
```



```
27 } // Ending brace aligned with the start of the statement
```

"Code is read much more often than it is written."

Hence, you have to make sure that your code is readable (by others and yourself a few days later), by following convention and recommended coding style.

2.2 Variables and Types

2.2.1 Variables - Name, Type and Value

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* (aka *identifier*), e.g., radius, area, age, height and num-Students. The name is needed to *uniquely* identify and reference each variable. You can use the *name* to assign a value to the variable (e.g., radius = 1.2), and to retrieve the value stored (e.g., radius*radius*3.1419265).
- A variable has a *data type*. The frequently-used Java *data types* are:
 - *int*: meant for integers (whole numbers) such as 123 and -456.
 - *double*: meant for floating-point number (real numbers) having an optional decimal point and fractional part, such as 3.1416, -55.66, 1.2e3, or -4.5E-6, where *e* or *E* denotes exponent of base 10.
 - *String*: meant for texts such as "Hello" and "Good Morning!". *Strings* are enclosed within a pair of double quotes.
 - *char*: meant for a single character, such as 'a', '8'. A *char* is enclosed by a pair of single quotes.
- In Java, you need to declare the *name* and the *type* of a variable before using a variable. For examples,



```
1 byte nybbles; // Declare an "byte" variable named "nybbles"  
2 long hexBytes; // Declare an "long" variable named "hexBytes"  
3 int sum; // Declare an "int" variable named "sum"
```



```
double average; // Declare a "double" variable named "average"
5 String message; // Declare a "String" variable named "message"
char grade; // Declare a "char" variable named "grade"
7 boolean ok; // Declare an "boolean" variable named "ok"
```

- A variable can store a *value* of the declared 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 that particular type. For example, an `int` variable can store an integer value such as 123, but NOT floating-point number such as 12.34, nor string such as "Hello".
- The concept of *type* was introduced in the early programming languages to *simplify* interpretation of data made up of binary sequences (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 (or whole number or fixed-point number); a *double* variable stores a floating-point number (or real number); a *String* variable stores texts.

| TYPE | NAME | VALUE | |
|--------|-----------|-----------|-----------------------------------|
| int | number | 1 | Stored only Integer |
| int | sum | 500500 | Stored only Integer |
| double | radius | 5.5 | Stored only floating-point number |
| double | area | 95.0334 | Stored only floating-point number |
| String | greeting | Hello | Stored only texts |
| String | statusMsg | Game Over | Stored only texts |

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

2.2.2 Identifiers (or Names)

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. (This is because blank creates two tokens and dash crashes with minus sign!)
- An identifier must begin with a letter (a-z, A-Z) or underscore (_). It cannot begin with a digit (0-9) (because that could confuse with a number). 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.

For example, *abc*, *_xyz*, \$123, *_1_2_3* are valid identifiers. But *1abc*, *min-value*, *surface area*, *ab@c* are NOT valid identifiers.

Caution: Programmers don't use *blank* character in any names (filename, project name, variable name, etc.). It is either not supported (e.g., in Java and C/C++), or will pose you many 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 examples, *radius*, *area*, *fontSize*, *numStudents*, *xMax*, *yMin*, *xTopLeft*, *isValidInput*, 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*, but not *n* or *x*, to store the number of students. It is alright to use abbreviations.
2. **Do not use meaningless names** like *a*, *b*, *c*, *i*, *j*, *k*, *n*, *i1*, *i2*, *i3*, *j99*, *exercise85* (what is the purpose of this exercise?), and *example12* (what is this example about?).
3. *Avoid single-letter names* like *i*, *j*, *k*, *a*, *b*, *c*, which are easier to type but often meaningless. Exceptions 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).

2.2.3 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; | <i>int</i> sum; <i>double</i> average; <i>String</i> statusMsg; |
| // Declare multiple variables of the SAME type, separated by commas type identifier1, identifier2, ..., identifierN; | <i>int</i> number, count; <i>double</i> sum, difference, product, quotient; <i>String</i> helloMsg, gameOverMsg; |
| // Declare a variable and assign an initial value type identifier = initialValue; | <i>int</i> magicNumber = 99; <i>double</i> pi = 3.14169265; <i>String</i> helloMsg = "hello,"; |
| // Declare multiple variables of the SAME type, with initial values type identifier1 = initValue1, ..., identifierN = initValueN; | <i>int</i> sum = 0, product = 1; <i>double</i> height = 1.2; length = 3.45; <i>String</i> greetingMsg = "hi!", quitMsg = "bye!"; |

Take note that:

- A variable is declared with a type. Once the type of a variable is declared, it can only store a value belonging to that 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 because identifier shall be unique.
- You can declare a variable anywhere inside the program, as long as it is declared before being used.
- The type of a variable cannot be changed inside the program, once it is declared.
- A variable declaration statement begins with a type, and works for only that type. In other words, you cannot declare variables of two different types in a single declaration statement.
- Java is a *statically-typed language*. This means that the type is resolved at compile time and never changes.

2.2.4 Constants (final variables)

Constants are *non-modifiable (immutable) variables*, declared with keyword *final*. You can only assign values to final variables ONCE. Their values cannot be changed

during program execution. There are several cases where using constants can be helpful. They are as follows:

- To prevent a variable from being changed accidentally
- To enforce the immutability of an object



```

1 final double PI = 3.14159265; // Declare and initialize the constant
2
3 final int SCREEN_X_MAX = 1280;
4 SCREEN_X_MAX = 800; // compilation error: cannot assign a value to final variable
5
6 // You can only assign value to final variables ONCE
7 final int SCREEN_Y_MIN;
8 SCREEN_Y_MIN = 0; // First assignment
9 SCREEN_Y_MIN = 10; // Compilation error: variable might already have been assigned

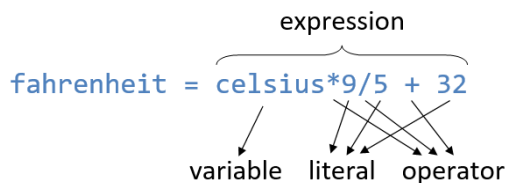
```

Constant Naming Convention

Use uppercase words, joined with underscore. For example, MIN_VALUE, MAX_SIZE, and INTEREST_RATE_6_MTH.

2.2.5 Expressions

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



For example,



```

1 // "int" literals
2 ((1 + 2) * 3 / 4) % 6 // This expression is evaluated to an "int" value
3 // "double" literals
4 3.45 + 6.7 // This expression is evaluated to a "double" value
5 // Assume that variables sum and number are "int"
6 sum + number * number // evaluates to an "int" value
7 // Assume that variables principal and interestRate are "double"
8 principal * (1.0 + interestRate) // evaluates to a "double" value


```


2.2.6 Assignment (=)

An *assignment statement* evaluates the RHS (Right-Hand Side) and assigns the resultant value to the variable of the LHS (Left-Hand Side). The syntax for assignment statement is:

| Syntax | Example |
|--|--|
| // Assign the RHS literal value to the LHS variable variable = literalValue; | int number; number = 9; |
| // Evaluate the RHS expression and assign the result to the LHS variable variable = expression; | int sum = 0; int number = 8; sum = sum + number; |

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



```
int number;  
2 number = 8;           // Assign RHS literal value of 8 to the LHS variable number  
number = number + 1;    // Evaluate the RHS expression (number + 1), and assign  
4                        // the resultant value back to the LHS variable number  
8 = number;             // Invalid in Programming, LHS shall be a variable  
6 number + 1 = sum;     // Invalid in Programming, LHS shall be a variable
```

Operator '=' is Assignment, NOT Equality

In Java, the equal symbol '=' is known as the *assignment operator*. The meaning of '=' in programming is different from Mathematics. It denotes *assignment of the RHS value to the LHS variable*, NOT *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 the RHS expression $x + 1$ and assigns the resultant value to the LHS variable x . On the other hand, $x = x + 1$ is illegal in Mathematics.

While $x + y = 1$ is allowed in Mathematics, it is invalid in programming because the LHS of an assignment statement shall be a variable.

Some programming languages use symbol ':=' , '>-' or '<-' as the assignment operator to avoid confusion with equality.

2.3

Primitive Types and String

In Java, there are two broad categories of *data types*:

- 1. Primitive types (e.g., *int*, *double*),
- 2. Reference types (e.g., objects and arrays).

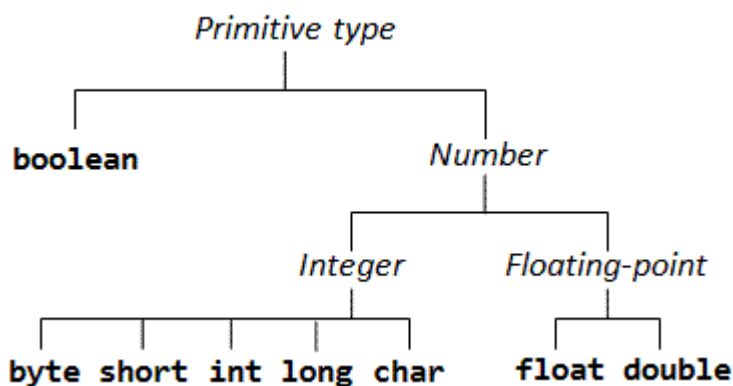
We shall describe the primitive types here. We will cover the reference types (classes and objects) in the later chapters on "Object-Oriented Programming".

2.3.1 Built-in Primitive Types

| Type | Description | |
|----------------|--|---|
| <i>byte</i> | Integer | 8-bit signed integer The range is $[-2^7, 2^7 - 1] = [-128, 127]$ |
| <i>short</i> | | 16-bit signed integer The range is $[-2^{15}, 2^{15} - 1] = [-32768, 32767]$ |
| <i>int</i> | | 32-bit signed integer The range is $[-2^{31}, 2^{31} - 1] = [-2147483648, 2147483647]$ (≈ 9 digits, $\pm 2G$) |
| <i>long</i> | | 64-bit signed integer The range is $[-2^{63}, 2^{63} - 1] = [-9223372036854775808, 9223372036854775807]$ (≈ 19 digits) |
| <i>float</i> | Floating-Point Number $F \times 2^E$ | 32-bit single precision floating-point number ($\approx 6 - 7$ significant decimal digits, in the range of $\pm[1.4 \times 10^{-45}, 3.4028235 \times 10^{38}]$) |
| <i>double</i> | | 64-bit double precision floating-point number ($\approx 14 - 15$ significant decimal digits, in the range of $\pm[4.9 \times 10^{-324}, 1.7976931348623157 \times 10^{308}]$) |
| <i>char</i> | Character | Represented in 16-bit Unicode '\u0000' to '\uFFFF'. Can be treated as integer in the range of $[0, 65535]$ in arithmetic operations. (Unlike C/C++, which uses 8-bit ASCII code.) |
| <i>boolean</i> | Binary | Takes a literal value of either true or false. The size of boolean is not defined in the Java specification, but requires at least one bit. Booleans are used in test in decision and loop, not applicable for arithmetic operations. (Unlike C/C++, which uses integer 0 for false, and non-zero for true.) |

The Java programming language is statically-typed, which means that all variables must first be declared before they can be used. A primitive data type specifies the size and type of variable values, and it has no additional methods. Primitive type are built-into the language for maximum efficiency, in terms of both space and computational efficiency.

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 a zero, positive and negative 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. They are represented in scientific notation of $F \times 2^E$ where the fraction (F) and exponent (E) are stored separately (as specified by the IEEE 754 standard).

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 say $[1.0, 1.1]$, but there is a finite number of patterns in a n -bit representation. Most of the floating-point values are approximated to their nearest representation.

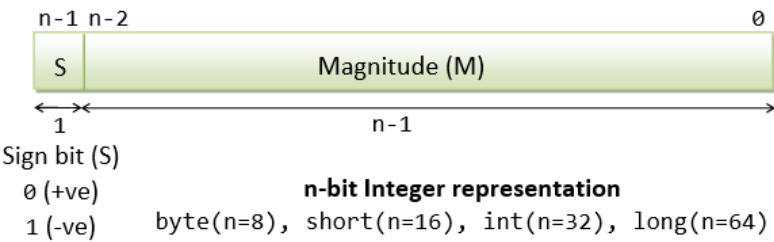
- The type *char* represents a single character, such as '0', 'A', 'a'. In Java, *char* is represented using 16-bit Unicode (in UCS-2 format) to support internationalization (*i18n*). A *char* can be treated as an integer in the range of $[0, 65535]$ in arithmetic operations. For example, character '0' 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 literal value of either *true* or *false*. *booleans* are used in *test* in decision and loop. They are not applicable to arithmetic operations (such as addition and multiplication).

2.3.2 Integers vs. Floating-Point Numbers

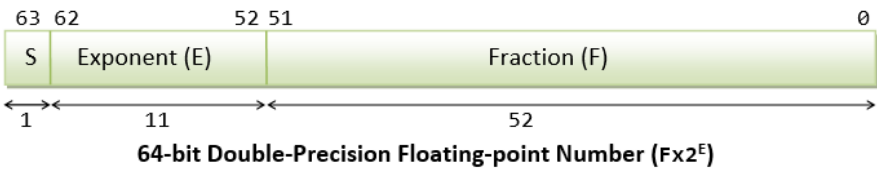
In computer programming, integers (such as 123, -456) and floating-point numbers (such as 1.23, -4.56, 1.2e3, -4.5e - 6) are TOTALLY different.

- 1. Integers and floating-point numbers are *represented* and *stored* differently.
- 2. Integers and floating-point numbers are *operated* differently.

How Integers and Floating-Point Numbers are Represented and Stored in Computer Memory?



Integers are represented in a so called 2's complement scheme as illustrated. The most-significant bit is called *Sign Bit (S)*, where $S = 0$ represents positive integer and $S = 1$ represents negative integer. The remaining bits represent the magnitude of the integers. For positive integers, the magnitude is the same as the binary number, e.g., if $n = 16$ (short), 0000000000000010 is $+2_{10}$. Negative integers require 2's complement conversion.



Floating-point numbers are represented in scientific form of $F \times 2^E$, where Fraction (F) and Exponent (E) are stored separately. For example, to store 12.75_{10} ; first convert to binary of 1100.11_2 ; then normalize to $1.10011_2 \times 2^3$; we have $F = 1.1011$ and $E = 3_{10} = 11_2$ which are then stored with some scaling.

How Integers and Floating-Point Numbers are Operated?

Integers and floating-point numbers are operated differently using different hardware circuitries. Integers are processed in CPU (Central Processing Unit), while floating-point numbers are processed in FPU (Floating-point Co-processor Unit).

Integer operations are straight-forward. For example, integer addition is carried out as illustrated:

$$123 + 4567 = ?$$

$$\begin{array}{r} 123 \\ 4567(+ \\ \hline 4690 \end{array}$$

On the other hand, floating-point addition is complex, as illustrated:

$$12.3 + 456.7 = ?$$

$$\begin{array}{l} 12.3 \rightarrow 1.23 \times 10^1 \\ 456.7 \rightarrow 4.567 \times 10^2 \end{array}$$

$$\begin{array}{ccc} 1.23 \times 10^1 & \xrightarrow{\text{Align Exponents}} & 0.123 \times 10^2 \\ \underline{4.567 \times 10^2 (+)} & & \underline{4.567 \times 10^2 (+)} \\ & & 4.690 \times 10^2 \end{array}$$

It is obvious that integer operations (such as addition) is much faster than floating-point operations.

Furthermore, integer are precise. All numbers within the range can be represented accurately. For example, a 32-bit int can represent ALL integers from -2147483648 to $+2147483647$ with no gap in between. On the other hand, floating-point are NOT precise, but close approximation. This is because there are infinite floating-point numbers in any interval (e.g., between 0.1 to 0.2). Not ALL numbers can be represented using a finite precision (32-bit float or 64-bit double).

You need to treat integers and floating-point numbers as two DISTINCT types in programming!

Use integer if possible (it is faster, precise and uses fewer bits). Use floating-point number only if a fractional part is required.

2.3.3 Data Representation

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. They are also processed differently. For examples,

- *byte* 1 is "00000001" (8-bit).
- *short* 1 is "00000000 00000001" (16-bit).
- *int* 1 is "00000000 00000000 00000000 00000001" (32-bit).
- *long* 1 is "00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000001" (64-bit).
- *float* 1.0 is "0 01111111 00000000 00000000 00000000" (32-bit).
- *double* 1.0 is "0 0111111111 0000 00000000 00000000 00000000 00000000 00000000 00000000" (64-bit).
- *char* '1' is "00000000 00110001" (16-bit) (Unicode number 49).
- *String* "1" is a complex object (many many bits).

There is a subtle difference between *int* 0 and *double* 0.0 as they have different bit-lengths and internal representations.

Furthermore, you **MUST** know the type of a value before you can interpret a value. For example, this bit-pattern "00000000 00000000 00000000 00000001" cannot be interpreted unless you know its type (or its representation).

2.3.4 Maximum/Minimum Values of Primitive Number Types

The following program can be used to print the *maximum*, *minimum* and *bit-length* of the primitive types. For example, the maximum, minimum and bit-size of *int* are kept in built-in constants `Integer.MIN_VALUE`, `Integer.MAX_VALUE`, `Integer.SIZE`.



```

1  /**
2   * PrimitiveTypesMinMaxBitLen.java
3   * Print the minimum, maximum and bit-length of all primitive types (except boolean)
4   */
5   public class PrimitiveTypesMinMaxBitLen {
6       public static void main(String[] args) {
7           /* int (32-bit signed integer) */
8           System.out.println("int (min) = " + Integer.MIN_VALUE);
9           // int (min) = -2147483648
10          System.out.println("int (max) = " + Integer.MAX_VALUE);
11          // int (max) = 2147483647
12          System.out.println("int (bit-length) = " + Integer.SIZE);
13          // int (bit-length) = 32
14
15          // byte (8-bit signed integer)
16          System.out.println("byte(min) = " + Byte.MIN_VALUE);

```



```

18 // byte(min) = -128
System.out.println("byte(max) = " + Byte.MAX_VALUE);
// byte(max) = 127
20 System.out.println("byte(bit-length) = " + Byte.SIZE);
// byte(bit-length) = 8
22
// short (16-bit signed integer)
24 System.out.println("short(min) = " + Short.MIN_VALUE);
// short(min) = -32768
26 System.out.println("short(max) = " + Short.MAX_VALUE);
// short(max) = 32767
28 System.out.println("short(bit-length) = " + Short.SIZE);
// short(bit-length) = 16
30
// long (64-bit signed integer)
32 System.out.println("long(min) = " + Long.MIN_VALUE);
// long(min) = -9223372036854775808
34 System.out.println("long(max) = " + Long.MAX_VALUE);
// long(max) = 9223372036854775807
36 System.out.println("long(bit-length) = " + Long.SIZE);
// long(bit-length) = 64
38
// char (16-bit character or 16-bit unsigned integer)
40 System.out.println("char(min) = " + (int)Character.MIN_VALUE);
// char(min) = 0
42 System.out.println("char(max) = " + (int)Character.MAX_VALUE);
// char(max) = 65535
44 System.out.println("char(bit-length) = " + Character.SIZE);
// char(bit-length) = 16
46
// float (32-bit floating-point)
48 System.out.println("float(min) = " + Float.MIN_VALUE);
// float(min) = 1.4E-45
50 System.out.println("float(max) = " + Float.MAX_VALUE);
// float(max) = 3.4028235E38
52 System.out.println("float(bit-length) = " + Float.SIZE);
// float(bit-length) = 32
54
// double (64-bit floating-point)
56 System.out.println("double(min) = " + Double.MIN_VALUE);
// double(min) = 4.9E-324
58 System.out.println("double(max) = " + Double.MAX_VALUE);
// double(max) = 1.7976931348623157E308
60 System.out.println("double(bit-length) = " + Double.SIZE);
// double(bit-length) = 64
62
// No equivalent constants for boolean type
64 }
}

```

2.3.5 One More Important Type - String

Beside the 8 primitive types, another important and frequently-used type is *String*. A *String* is a sequence of characters (texts) such as "Hello, world". *String* is not a primitive type (this will be elaborated later).

In Java, a *char* is a single character enclosed by single quotes (e.g., 'A', '0', '\$'); while a *String* is a sequence of characters enclosed by double quotes (e.g., "Hello"). For example,



```
1 String greetingMsg = "hello , world"; // String is enclosed in double-quotes
   char gender = 'm'; // char is enclosed in single-quotes
3 String statusMsg = ""; // An empty String
```

2.3.6 Choice of Data Types for Variables

As a programmer, you need to 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.

It is important to take note that your programs will have data of DIFFERENT types.

Rules of Thumb for Choosing Data Types

- For numbers, use an integer type if possible. Use a floating-point type only if the number contains a fractional part. Although floating-point numbers includes integers (e.g., 1.0, 2.0, 3.0), floating-point numbers are approximation (not precise) and require more resources (computational and storage) for operations.
- Although there are 4 integer types: 8-bit *byte*, 16-bit *short*, 32-bit *int* and 64-bit *long*, we shall use *int* for integers in general. Use *byte*, *short*, and *long* only if you have a good reason to choose that particular precision.
- Among there are two floating-point types: 32-bit *float* and 64-bit *double*, we shall use *double* in general. Use *float* only if you wish to conserve storage and do not need the precision of *double*. *char*, *boolean* and *String* have their specific usage.

Example (Variable Names and Types) Paul has bought a new notebook of "idol" brand, with a processor speed of 2.66GHz, 8 GB of RAM, 500GB hard disk, with a 15-inch monitor, for \$1760.55. He has chosen service plan 'C' among plans 'A', 'B',

'C', and 'D', plus on-site servicing but did not choose extended warranty. Identify the data types and name the variables.

The possible variable names and types are:



```
1 String name = "Paul";  
   String brand = "idol";  
  
3  
   double processorSpeedInGHz = 2.66; // or float  
5   double ramSizeInGB = 8;           // or float  
   double price = 1760.55;  
  
7  
   int harddiskSizeInGB = 500;        // or short  
9   int monitorInInch = 15;          // or byte  
  
11  char servicePlan = 'C';  
   char vowel = 'e';  
  
13  
   boolean onSiteService = true;  
15  boolean extendedWarranty = false;
```

Exercise (Variable Names and Types)

1. 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 scholar. Each student is assigned a unique 8-digit number as id.

You are required to identify the variables, assign a suitable name to each variable and choose an appropriate type. Write the variable declaration statements as in the above example.

2.3.7 Literals for Primitive Types and String

Literal

A *literal*, or *literal constant*, is a specific constant value, such as 123, -456 , 3.1416, $-1.2E3$, $4.5e - 6$, '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 (*int*, *long*, *short*, *byte*) literals

A whole number literal, such as 123 and -456 , is treated as an *int* by default. For example,



```

1 int number = -123;
  int sum = 1234567890;    // This value is within the range of int
3 int bigSum = 8234567890; // error: this value is outside the range of int
  int intRate = 6%;        // error: no percent sign
5 int pay = $1234;         // error: no dollar sign
  int product = 1,234,567; // error: no grouping commas

```

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, 567, \$123 and 12% are invalid).

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



```

  int number1 = 1234;    // The usual decimal
2 int number2 = 01234;   // Octal 1234, Decimal 2322
  int number3 = 0017;    // Octal 17, Decimal 15
4 int number4 = 0x1abc;  // Hexadecimal 1ABC, decimal 15274

```

From JDK 7, you can use prefix '0b' or '0B' to specify an integer literal 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, not underscore. For example,



```

  // JDK 7
2 int number1 = 0b01010000101000101101000010100010 ;

4 // Break the bits with underscore
  int number2 = 0b0101_0000_1010_0010_1101_0000_1010_0010 ;

6
  // Break the decimal digits with underscore
8 int number3 = 2_123_456;

10 // Error: cannot begin or end with underscore
   int number4 = _123_456;

```

A long literal outside the *int* range requires a suffix 'L' or 'l' (avoid lowercase 'l', which could be confused with the number one '1'), e.g., 123456789012L, -9876543210l. For example,



```
1 long sum = 123;           // Within the "int" range, no need for suffix 'L'
   long bigSum = 1234567890123L; // Outside "int" range, suffix 'L' needed
```

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



```
byte smallNumber1 = 123;           // This is within the range of byte [-128, 127]
byte smallNumber2 = -1234;         // Error: this value is out of range

short midSizeNumber1 = -12345;     // This is within the range of short [-32768, 32767]
short midSizeNumber2 = 123456;     // Error: this value is out of range
```

Floating-point (*double*, *float*) literals

A literal 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., $1.2e3$, $-5.5E-6$, where e or E denotes the exponent in base 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 are reminded that floating-point numbers are stored in scientific form of $F \times 2^E$, where F (Fraction) and E (Exponent) are stored separately. 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,



```
1 float average = 55.66;           // Error: RHS is a double. Need suffix 'f' for float.
   float average = 55.66F;         // float literal needs suffix 'f' or 'F'

3
   float rate = 1.2e-3;           // Error: RHS is a double. Need suffix 'f' for float.
5 float rate = 1.2e-3f;           // float literal needs suffix 'f' or 'F'
```

Character (char) Literals and Escape Sequences

A printable char literal (such as letters, digits and special symbols) is written by enclosing the character with a pair of *single quotes*, e.g., 'A', 'z', '0', '9', '\$', and '@'. Special char literals (such as tab, newline) are represented using so-called *escape sequences* (to be described later).

In Java, chars are represented using 16-bit Unicode. Printable characters for English letters (a-z, A-Z), digits (0-9) and symbols (+, -, @, etc.) are assigned to code numbers 32 – 126 (20H – 7EH), as tabulated below (arranged in decimal and hexadecimal).

| Dec | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|-----|---|---|----|---|---|---|----|---|---|---|
| 3 | | | SP | ! | " | # | \$ | % | & | ' |
| 4 | (|) | * | + | , | - | . | / | 0 | 1 |
| 5 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | : | ; |
| 6 | < | = | > | ? | @ | A | B | C | D | E |
| 7 | F | G | H | I | J | K | L | M | N | O |
| 8 | P | Q | R | S | T | U | V | W | X | Y |
| 9 | Z | [| \ |] | ^ | _ | ` | a | b | c |
| 10 | d | e | f | g | h | i | j | k | l | m |
| 11 | n | o | p | q | r | s | t | u | v | w |
| 12 | x | y | z | { | | } | | | | |

| Hex | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|-----|----|---|---|---|----|---|---|---|---|---|---|---|---|---|---|---|
| 2 | SP | ! | " | # | \$ | % | & | ' | (|) | * | + | , | - | . | / |
| 3 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | : | ; | < | = | > | ? |
| 4 | @ | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 5 | P | Q | R | S | T | U | V | W | X | Y | Z | [| \ |] | ^ | _ |
| 6 | ` | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o |
| 7 | p | q | r | s | t | u | v | w | x | y | z | { | | } | | |

In Java, a *char* can be treated as its underlying integer in the range of [0, 65535] in arithmetic operations. In other words, *char* and integer are *interchangeable* in arithmetic operations. You can treat a *char* as an *int*, you can also assign an integer value in the range of [0, 65535] to a char variable. For example,




```
1 char letter = 'a';           // Same as 97
  char anotherLetter = 98;     // Same as the letter 'b'
3 // You can assign an integer in the range of [0, 65535] to char
  System.out.println ( letter ); // 'a' printed
5 System.out.println ( anotherLetter ); // 'b' printed instead of the number,
                                         // because the type is char
7 anotherLetter += 2;           // 100 or 'd'
  System.out.println ( anotherLetter ); // 'd' printed
```

Special characters are represented by so-called escape sequence, which begins with a back-slash (\) followed by a pattern, e.g., \t for tab, \n for newline. The commonly-used escape sequences are:

| Escape Sequence | Description | Unicode (Decimal) | Unicode (Hex) |
|-----------------|---|-------------------|---------------|
| \t | Tab | 9 | 0009H |
| \n | Newline (or Line-feed) | 10 | 000AH |
| \r | Carriage-return | 13 | 000DH |
| \" | Double-quote (Needed to be used inside double-quoted <i>String</i>) | - | - |
| \' | Single-quote (Needed to be used inside single-quoted char, i.e., '\'') | - | - |
| \\ | Back-slash (Needed as back-slash is given a special meaning) | - | - |
| \uhhhh | Unicode number hhhh (in hex) | - | hhhhH |


For examples,



```
char tabChar = '\t';           // tab
2 char anotherTabChar = 9;      // Code number 9 is tab
char newlineChar = '\n';       // newline, code number 10
4 char backSlashChar = '\\';    // Since back-slash is given a special meaning,
                               // to write a back-slash, use double back-slash
6 char singleQuoteChar = '\'';  // Need to use escape sequence to resolve conflict
char doubleQuoteChar = '\"';    // No conflict. No need for escape sequence
8 System.out.println("A tab " + tabChar + " before this; end with two newlines!"
    + newlineChar + newlineChar);
```

String Literals and Escape Sequences

A *String* is a sequence of characters. A *String* literal is composed of zero or more characters surrounded by a pair of *double quotes*. For examples,



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

You need to use an escape sequence for special non-printable characters, such as

newline (`\n`) and tab (`\t`). You also need to use escape sequence for double-quote (`\`) and backslash (`\\`) due to conflict. For examples,



```
1 String str1 = "hello\tworld\n";    // tab and newline
   String str2 = "a double quoted \" hello \" ";
3 String str3 = "1 back-slash \\\, another 2 back-slashes \\\\ ";
   String str1 = "A \" string \" nested \\ inside \\ a string "
5 // A "string" nested \inside\ a string

7 String str2 = "Hello, \u60a8\u597d!"
```

Single-quote (`'`) inside a *String* does not require an escape sequence because there is no ambiguity, e.g.,



```
1 String str3 = "Hi, I'm a string !" // Single quote OK
```

It is important to take note that `\t` or `\` is ONE single character, NOT TWO!

Exercise

1. Write a program to print the following animal picture using `System.out.println()`. Take note that you need to use escape sequences to print some characters, e.g., `\` for `"`, `\\` for `\`.

```
Command window
1
   ' '
   (oo)
3  +=====V
   / || %%% ||
5  * ||-----||
   ""      ""
```

End-of-Line (EOL)

Newline (0AH) and Carriage-Return (0DH), represented by the escape sequence `\n`, and `\r` respectively, are used as line delimiter (or end-of-line, or EOL) for text files. Take note that Unix and macOS use `\n` (0AH) as EOL, while Windows use `\r\n` (0D0AH).

boolean Literals

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



```

boolean done = true;
2 boolean gameOver = false;

4 boolean isValid;
  isValid = false;

```

Example on Literals



```

1 /**
   * TestLiterals .java
   * Test literals for various primitive types
   */
2 public class TestLiterals {
   public static void main(String[] args) {
3       String name = "Tan Ah Teck"; // String is double-quoted
4       char gender = 'm';           // char is single-quoted
5       boolean isMarried = true;    // boolean of either true or false
6       byte numChildren = 8;        // Range of byte is [-127, 128]
7       short yearOfBirth = 1945;    // Range of short is [-32767, 32768]. Beyond byte
8       int salary = 88000;           // Beyond the ranges of byte and short
9       long netAsset = 8234567890L; // Need suffix 'L' for long. Beyond int
10      double weight = 88.88;        // With fractional part
11      float gpa = 3.88f;            // Need suffix 'f' for float

12
13      // println () can be used to print value of any type
14      System.out.println("Name is: " + name);
15      // Name is: Tan Ah Teck
16      System.out.println("Gender is: " + gender);
17      // Gender is: m
18      System.out.println("Is married is: " + isMarried);
19      // Is married is: true
20      System.out.println("Number of children is: " + numChildren);
21      // Number of children is: 8
22      System.out.println("Year of birth is: " + yearOfBirth);
23      // Year of birth is: 1945
24      System.out.println("Salary is: " + salary);
25      // Salary is: 88000
26      System.out.println("Net Asset is: " + netAsset);
27      // Net Asset is: 8234567890
28      System.out.println("Weight is: " + weight);
29      // Weight is: 88.88
30      System.out.println("GPA is: " + gpa);
31      // GPA is: 3.88
32  }
33 }

```

2.3.8 var - Local Variable Type Inference (JDK 10)

JDK 10 introduces a new way to declare variables via a new keyword `var`, for examples,



```
1 var v1 = 0;           // type inferred to "int"
  var v2 = 0.0;         // type inferred to "double"
3 var v3 = 1.0f;        // type inferred to "float"
  var v4 = '0';         // type inferred to "char"
5 var v5 = "hello";     // type inferred to "String"
  // var v6;           // compilation error: cannot use 'var' on variable without initializer
```

Clearly, you need to initialize the variable, so that the compiler can infer its type.

2.4

Basic Operations

2.4.1 Arithmetic Operators

Java supports the following binary/unary arithmetic operations:

| Operator | Mode | Usage | Description | Examples |
|----------|--------|----------|------------------------|--|
| + | Binary | $x + y$ | Addition | $1 + 2 \Rightarrow 3$ |
| | Unary | $+x$ | Unary positive | $1.1 + 2.2 \Rightarrow 3.3$ |
| - | Binary | $x - y$ | Subtraction | $1 - 2 \Rightarrow -1$ |
| | Unary | $-x$ | Unary negate | $1.1 - 2.2 \Rightarrow -1.1$ |
| * | Binary | $x * y$ | Multiplication | $2 * 3 \Rightarrow 6$ $3.3 * 1.0 \Rightarrow 3.3$ |
| / | Binary | x / y | Division | $1 / 2 \Rightarrow 0$ $1.0 * 2.0 \Rightarrow 0.5$ |
| % | Binary | $x \% y$ | Modulus (Remainder) | $5 \% 2 \Rightarrow 1$ $-5 \% 2 \Rightarrow -1$ $5.5 \% 2.2 \Rightarrow 1.1$ |

These operators are typically *binary infix* operators, i.e., they take two operands with the operator in between the operands (e.g., $11 + 12$). However, `'-'` and `'+'` can also be interpreted as *unary* "negate" and "positive" *prefix* operator, with the operator in front of the operand. For examples,



```
1 int number = -9;
2 number = -number; // Unary negate
```


2.4.2 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 sign (*) as in Mathematics.

Rules on Precedence

Like Mathematics:

1. Parentheses () have the highest precedence and can be used to change the order of evaluation.
2. Unary '-' (negate) and '+' (positive) have next higher precedence.
3. The multiplication (*), division (/) and modulus (%) have the same precedence. They take precedence over addition (+) and subtraction (-). For example, $1 + 2*3 - 4/5 + 6\%7$ is interpreted as $1 + (2*3) - (4/5) + (6\%7)$.
4. Within the same precedence level (i.e., addition/subtraction and multiplication/division/modulus), the expression is evaluated from left to right (called left-associative). For examples, $1 + 2 - 3 + 4$ is evaluated as $((1 + 2) - 3) + 4$, and $1*2\%3/4$ is $((1*2)\%3)/4$.

2.4.3 Type Conversion in Arithmetic Operations

Your program typically contains data of many types, e.g., count and sum are *int*, average and gpa are *double*, and message is a *String*. Hence, it is important to understand how Java handles types in your programs.

The arithmetic operators (+, -, *, /, %) are only applicable to primitive number types: *byte*, *short*, *int*, *long*, *float*, *double*, and *char*. They are not applicable to *boolean*.

Same-Type Operands of *int*, *long*, *float*, *double*

If BOTH operands are *int*, *long*, *float* or *double*, the binary arithmetic operations are carried in that type, and evaluate to a value of that type, i.e.,

1. $int \oplus int \Rightarrow int$, where \oplus denotes a binary arithmetic operators such as +, -, *, /, %.
2. $long \oplus long \Rightarrow long$
3. $float \oplus float \Rightarrow float$
4. $double \oplus double \Rightarrow double$

int Division

It is important to take note *int* division produces an *int*, i.e., $int / int \Rightarrow int$, with the result truncated. For example, $1/2 \Rightarrow 0$ (*int*), but $1.0/2.0 \Rightarrow 0.5$ (*double* / *double* \Rightarrow *double*).

Same-Type Operands of *byte*, *short*, *char*: Convert to *int*

If BOTH operands are *byte*, *short* or *char*, the binary operations are carried out in *int*, and evaluate to a value of *int*. A *char* is treated as an integer of its underlying Unicode number in the range of [0, 65535]. That is,

- $byte \oplus byte \Rightarrow int \oplus int \Rightarrow int$, where \oplus denotes a binary arithmetic operators such as $+$, $-$, $*$, $/$, $\%$.
- $short \oplus short \Rightarrow int \oplus int \Rightarrow int$
- $char \oplus char \Rightarrow int \oplus int \Rightarrow int$

Take note that NO arithmetic operations are carried out in *byte*, *short* or *char*. For examples,



```
byte b1 = 5, b2 = 9, b3;
2 // byte + byte -> int + int -> int
  b3 = b1 + b2; // error: RHS is "int ", cannot assign to LHS of "byte"
4 b3 = (byte)(b1 + b2); // Need explicit type casting (to be discussed later)
```

However, if compound arithmetic operators ($+=$, $-=$, $*=$, $/=$, $\%=$) (to be discussed later) are used, the result is automatically converted to the LHS. For example,



```
byte b1 = 5, b2 = 9;
2 b2 += b1; // Result in "int ", but automatically converted back to "byte"
```

Mixed-Type Arithmetic Operations

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. (In Java, no operations are carried out in *byte*, *short* or *char*.)

- The order of promotion is: $int \Rightarrow long \Rightarrow float \Rightarrow double$.

For examples,

1. $int/double \Rightarrow double/double$. Hence, $1/2 \Rightarrow 0$, $1.0/2.0 \Rightarrow 0.5$, $1.0/2 \Rightarrow 0.5$, $1/2.0 \Rightarrow 0.5$
2. $9/5 * 20.1 \Rightarrow (9/5) * 20.1 \Rightarrow 1 * 20.1 \Rightarrow 1.0 * 20.1 \Rightarrow 20.1$ (You probably don't expect this answer!)
3. $char '0' + int 2 \Rightarrow int 48 + int 2 \Rightarrow int 50$ (Result is an *int*, need to explicitly cast back to *char* '2' if desired.)
4. $char \oplus float \Rightarrow int \oplus float \Rightarrow float \oplus float \Rightarrow float$
5. $byte \oplus double \Rightarrow int \oplus double \Rightarrow double \oplus double \Rightarrow double$

Summary: Type-Conversion Rules for Binary Operations

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*.

Summary: Type-Conversion Rules for Unary Operations

The type-promotion rules for unary operations (e.g., negate '-') can be summarized 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 promoted to *int*.

2.4.4 More on Arithmetic Operators

Modulus (Remainder) Operator

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,

- $-5 \% 2 \Rightarrow -3 \% 2 \Rightarrow -1$
- $5.5 \% 2.2 \Rightarrow 3.3 \% 2.2 \Rightarrow 1.1$

Exponent?

Java does not have an exponent operator. (The ^ operator denotes exclusive-or, NOT exponent). You need to use JDK method `Math.exp(x, y)` to evaluate `x` raises to power `y`; or write your own code.

2.4.5 Overflow/Underflow

Study the output of the following program.



```

1  /**
2  * TestOverflow.java
3  * Illustrate "int" overflow
4  */
5
6  public class TestOverflow {
7      public static void main(String[] args) {
8          // Range of int is [-2147483648, 2147483647]
9          int i1 = 2147483647;           // maximum int
10         System.out.println(i + 1);    // -2147483648 (overflow)
11         System.out.println(i + 2);    // -2147483647 (overflow)
12         System.out.println(i + 3);    // -2147483646 (overflow)
13         System.out.println(i * 2);    // -2 (overflow)
14         System.out.println(i * i);    // 1 (overflow)
15
16         int i2 = -2147483648;          // minimum int
17         System.out.println(i2 - 1);    // 2147483647 (overflow)
18         System.out.println(i2 - 2);    // 2147483646 (overflow)
19         System.out.println(i2 * i2);  // 0 (overflow)
20     }
21 }

```

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 a 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 an error? This 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.

2.4.6 More on Integer vs. Floating-Point Numbers

Integers (*byte*, *short*, *int*, *long*) are precise (exact). But float and double are not precise but close approximation. Study the results of the following program:



```

1  /**
2  * TestPreciseness .java
3  * Test preciseness for int/ float /double
4  */
5
6  public class TestPreciseness {
7
8      public static void main(String[] args) {
9          // doubles are NOT precise
10         System.out.println (2.2 + 4.4);          // 6.6000000000000005
11         System.out.println (6.6 - 2.2 - 4.4); // -8.881784197001252E-16 (NOT Zero!)
12
13         // Compare two doubles
14         System.out.println ((6.6) == (2.2 + 4.4)); // false
15
16         // int is precise , float /double are NOT!
17         int i1 = 123456789;
18         System.out.println (i1*10); // 1234567890 (exact within the range)
19
20         // Test float
21         float f1 = 123456789.0f; // float keeps 6-7 significant digits
22         System.out.println (f1); // 1.23456792E8 (=123456792 close but not exact)
23         System.out.println (f1*10); // 1.23456794E9 (=1234567940)
24     }
25 }

```

Always use *int* if you do not need the fractional part, although double can also represent most of the integers (e.g., 1.0, 2.0, 3.0). This is because:

- *int* is more efficient (faster) than *double* in arithmetic operations.
- 32-bit *int* takes less memory space than 64-bit *double*.
- *int* is exact (precise) in representing ALL integers within its range. *double* is an approximation - NOT ALL integer values can be represented by *double*.

2.4.7 Type Casting

In Java, you will get a compilation "error: incompatible types: possible lossy

conversion from *double*|*float*|*long* to *int*" if you try to assign a *double*, *float*, or *long* value of to an *int* variable. This is because the fractional part would be truncated and lost. For example,



```
1 // Assign a "double" value to an "int" variable
  double d = 3.5;
3 int i = d;      // Compilation error: incompatible types: possible lossy conversion
                  // from double to int
5
  // Assign a "float" value to an "int" variable
7 int sum = 55.66f; // Compilation error: incompatible types: possible lossy conversion
                  // from float to int
9
  // Assign a "long" value to an "int" variable
11 long lg = 123;
   int count = lg; // Compilation error: incompatible types: possible lossy conversion
                  // from long to int
13
```

There are two kinds of type-casting in Java:

1. **Explicit type-casting** via a type-casting operator.
2. **Implicit type-casting** performed by the compiler automatically, if there is no loss of precision.

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 *(int)doubleOperand* - to operate on the double operand and return a *truncated* value in *int*. In other words, you tell the compiler you consciously perform the truncation and you are fully aware of the "possible lossy conversion". You can then assign the truncated *int* value to the *int* variable. For example,



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

Type casting is an operation which takes one operand. It operates on its operand, and returns an equivalent value in the specified type. The syntax is:



```
(type) variable    // e.g., (int)height
2 (type) literal    // e.g., (int)55.66
```

Implicit Type-Casting in Assignment

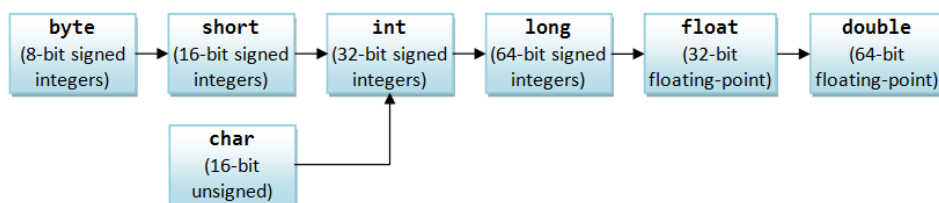
Explicit type-casting is not required if you assign an `int` value to a *double* variable, because there is no loss of precision. The compiler will perform the type-casting automatically (i.e., implicit type-casting). For example,



```
int i = 3;
2 double d;
d = i;           // OK, no explicit type casting required
4 // d = 3.0
d = (double)i;   // Explicit type casting operator used here
6
double aDouble = 55; // Compiler auto-casts int 55 to double 55.0
8 double nought = 0;  // Compiler auto-casts int 0 to double 0.0
                      // int 0 and double 0.0 are different .
```

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 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

Type casting is also called type conversion:

1. **Widening Primitive Conversion (Implicit):** Conversion of narrow (smaller) data type value to wider (larger) data type is called widening primitive conversion which happens implicitly.

2. **Narrowing Primitive Conversion (Explicit):** Conversion of wider data type to narrower data type is called narrowing primitive conversion which requires explicitly type conversion.

Example

Suppose that you want to find the average (in *double*) of the running integers from 1 and 100. Study the following code:



```

1  /**
   * Average1To100.java
   * Compute the average of running numbers 1 to 100
   */
2  public class Average1To100 {
3      public static void main(String[] args) {
4          int sum = 0;
5          for (int number = 1; number <= 100; ++number) {
6              sum += number;    // Final sum is int 5050
7          }
8
9          double average = sum / 100;    // Won't work (average = 50.0 instead of 50.5)
10         System.out.println("Average is " + average);    // Average is 50.0
11     }
12 }

```

The average of 50.0 is incorrect. This is because both the sum and 100 are *int*. The result of *int/int* 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:



```

1  // Cast sum from int to double before division ,
   // double / int -> double / double -> double
2  average = (double)sum / 100;

3
4  // Cast 100 from int to double before division ,
   // int / double -> double / double -> double
5  average = sum / (double)100;

6
7  // int / double -> double / double -> double
   average = sum / 100.0;

8
9  // Division int before cast
10 average = (double)(sum / 100);    // Won't work. why?

```


2.4.8 Compound Assignment Operators

Besides the usual simple assignment operator (=) described earlier, Java also provides the so-called *compound assignment operators* as listed:

| Operator | Mode | Usage | Description | Examples |
|----------|--------|--|--|---|
| = | Binary | $var = expr$ | Assignment Assign the LHS value to the RHS variable | <code>x = 5</code> |
| += | Binary | $var += expr$ same as: $var = var + expr$ | Compound addition and assignment | <code>x += 5;</code> same as: <code>x = x + 5</code> |
| -= | Binary | $var -= expr$ same as: $var = var - expr$ | Compound subtraction and assignment | <code>x -= 5;</code> same as: <code>x = x - 5</code> |
| *= | Binary | $var *= expr$ same as: $var = var * expr$ | Compound multiplication and assignment | <code>x *= 5;</code> same as: <code>x = x * 5</code> |
| /= | Binary | $var /= expr$ same as: $var = var / expr$ | Compound division and assignment | <code>x /= 5;</code> same as: <code>x = x / 5</code> |
| %= | Binary | $var \% = expr$ same as: $var = var \% expr$ | Compound division and assignment | <code>x \% = 5;</code> same as: <code>x = x \% 5</code> |

One subtle difference between simple and compound operators is in *byte*, *short*, *char* binary operations. For examples,



```

1 byte b1 = 5, b2 = 8, b3;

3 // byte + byte -> int + int -> int, need to explicitly cast back to "byte"
  b3 = (byte)(b1 + b2);

5

6 b3 = b1 + b2;           // error: RHS is int, cannot assign to byte
7 b1 += b2;               // implicitly casted back to "byte"

9 char c1 = '0', c2;

11 // char + int -> int + int -> int, need to explicitly cast back to "char"
   c2 = (char)(c1 + 2);

13

14 c2 = c1 + 2;           // error: RHS is int, cannot assign to char
15 c1 += 2;               // implicitly casted back to "char"

```

2.4.9 Increment/Decrement

Java supports these *unary* arithmetic operators: increment (++) and decrement (--) for all primitive number types (*byte*, *short*, *char*, *int*, *long*, *float* and *double*, except *boolean*). The increment/decrement unary operators can be placed before the operand (prefix), or after the operands (postfix). These operators were introduced in C++ to shorthand `x = x + 1` to `x++` or `++x`.

| Operator | Mode | Usage | Description | Examples |
|-------------------|-------------------------------|------------|---|--|
| ++ (Increment) | Unary Prefix Unary Postfix | ++x x++ | Increment the value of the operand by 1. x++ or ++x is the same as x += 1 or x = x + 1 | int x = 5; x++; // x is 6 ++x; // x is 7 |
| -- (Decrement) | Unary Prefix Unary Postfix | --x x-- | Decrement the value of the operand by 1. x-- or --x is the same as x -= 1 or x = x - 1 | int y = 6; y--; // y is 5 --y; // y is 4 |

The increment (++) and decrement (--) operate on its sole operand and store the result back to its operand. For example, ++x retrieves x, increment and stores the result back to x.

In Java, there are 4 ways to increment/decrement a variable:



```
1 int x = 5;
   // 4 ways to increment by 1
3 x = x + 1;    // x is 6
  x += 1;      // x is 7
5 x++;         // x is 8
  ++x;        // x is 9
7 // 4 ways to decrement by 1
  x = x - 1;   // x is 8
9 x -= 1;      // x is 7
  x--;        // x is 6
11 --x;       // x is 5
```

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 may affect the outcome.

- If these operators are used by themselves (standalone) in a statement (e.g., `x++`; or `++x`), the outcomes are the SAME for pre- and post-operators.

- If ++ or -- involves another operation in the SAME statement, e.g., `y = x++`; or `y = ++x`; where there are two operations in the same statement: assignment and increment, then pre- or post-order is important to specify the order of these two operations, as tabulated below:

| Operator | Description | Example | Same As |
|--|---|----------------------|--|
| <code>++var</code> (Pre-Increment) | Increment <i>var</i> , and return the incremented <i>var</i> for the other operation in the same statement. | <code>y = ++x</code> | <code>x = x + 1;</code> <code>y = x</code> |
| <code>var++</code> (Post-Increment) | Return the old value of <i>var</i> for the other operation in the same statement, then increment <i>var</i> . | <code>y = x++</code> | <code>oldX = x;</code> <code>x = x + 1</code> <code>y = oldX;</code> |
| <code>--var</code> (Pre-Decrement) | Decrement <i>var</i> , and return the decremented <i>var</i> for the other operation in the same statement. | <code>y = --x</code> | <code>x = x - 1;</code> <code>y = x</code> |
| <code>var--</code> (Post-Decrement) | Return the old value of <i>var</i> for the other operation in the same statement, then decrement <i>var</i> . | <code>y = x--</code> | <code>oldX = x;</code> <code>x = x - 1</code> <code>y = oldX;</code> |

For examples,



```

1 // Two operations in the statement: increment and assignment
  x = 5;
3 y = ++x;    // Increment x (= 6), then assign x to y (= 6). (++x returns x+1)
  x = 5;
5 y = x++;    // Assign x to y (=5), then increment x (=6). (x++ returns the oldX)
  // After the operations, x gets the SAME value, but the other operation has different
  // outcomes

7
  // Two operations in the statement: increment and println ()
9 x = 5;
  System.out.println(++x);    // Increment x (=6), then print x (=6). (++x returns x+1)
11
  x = 5;
13 System.out.println(x++);    // Print x (=5), then increment x (=6).
                               // (x++ returns the oldX)

```

Notes:

- Prefix operator (e.g., `++i`) could be more efficient than postfix operator (e.g., `i++`)?!
- What is `i = i++`? Try it out!

2.4.10 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). All these operators are *binary operators* (that takes two operands) and return a boolean value of either true or false.

| Operator | Mode | Usage | Description | Examples (x = 5, y = 8) |
|----------|--------|--------|--------------------------|----------------------------|
| == | Binary | x == y | Equal to | (x == y) ⇒ <i>false</i> |
| != | Binary | x != y | Not equal to | (x != y) ⇒ <i>true</i> |
| > | Binary | x > y | Greater than | (x > y) ⇒ <i>false</i> |
| >= | Binary | x >= y | Greater than or equal to | (x >= 5) ⇒ <i>true</i> |
| < | Binary | x < y | Less than | (x < 8) ⇒ <i>false</i> |
| <= | Binary | x <= y | Less than or equal to | (x <= 8) ⇒ <i>true</i> |

Take note that the comparison operators are *binary infix* operators, that operate on two operands with the operator in between the operands, e.g., x <= 100. It is invalid to write 1 < x < 100 (non-binary operations). Instead, you need to break out the two binary comparison operations x > 1, x < 100, and join 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 precedence, as follows:

| Operator | Mode | Usage | Description | Examples |
|----------|--------|--------|----------------------------|----------|
| ! | Unary | !x | Logical NOT | |
| && | Binary | x && y | Logical AND | |
| | Binary | x y | Logical OR | |
| ^ | Binary | x ^ y | Logical Exclusive-OR (XOR) | |

The truth tables are as follows:

| NOT (!) | true | false |
|---------|--------------|-------------|
| Result | <i>false</i> | <i>true</i> |

| AND (&&) | true | false |
|--------------|--------------|--------------|
| <i>true</i> | <i>true</i> | <i>false</i> |
| <i>false</i> | <i>false</i> | <i>false</i> |

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

| XOR (^) | true | false |
|---------|-------|-------|
| true | false | true |
| false | true | false |

Examples



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

4
// Return true if x is outside 0 and 100 ( inclusive )
6 (x < 0) || (x > 100)
// or
8 !((x >= 0) && (x <= 100))

10 // 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.
12 ((year % 4 == 0) && (year % 100 != 0)) || (year % 400 == 0)
```

Exercise

Study the following program, and explain its output.



```
/**
2 * TestRelationalLogicalOp .java
* Test relational and logical operators
4 */
public class TestRelationalLogicalOp {
6 public static void main(String[] args) {
    int age = 18;
8 double weight = 71.23;
    int height = 191;
10 boolean married = false ;
    boolean attached = false ;
12 char gender = 'm';

14 System.out.println (!married && !attached && (gender == 'm')); // true
    System.out.println (married && (gender == 'f')); // false
```



```

16 System.out.println (( height >= 180) && (weight >= 65)
    && (weight <= 80)); // true
18 System.out.println (( height >= 180) || (weight >= 90)); // true
    }
20 }

```

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

Exercise

1. 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)

Equality Comparison ==

You can use == to compare two integers (*byte*, *short*, *int*, *long*) and *char*. But do NOT use == to compare two floating-point numbers (*float* and *double*) because they are NOT precise. To compare floating-point numbers, set a threshold for their difference, e.g.,



```

/**
2  * TestFloatComparison.java
   */
4  public class TestFloatComparison {
    public static void main(String[] args) {
6      // floating -point numbers are NOT precise
        double d1 = 2.2 + 4.4;
8      double d2 = 6.6;
        System.out.println (d1 == d2); // false
10     System.out.println (d1);        // 6.6000000000000005

12     // Set a threshold for comparison with ==
        final double EPSILON = 1e-7;
14     System.out.println (Math.abs(d1 - d2) < EPSILON); // true
    }
16 }

```

You also CANNOT use == to compare two *Strings* because *Strings* are objects. You need to use `str1.equals(str2)` instead. This will be elaborated later.

Logical Operator Precedence

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



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

Short-Circuit Operations

The binary 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 && rightOperand` gives `false` and `true || rightOperand` 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. `false && (++i < 5)` but `++i` will not be evaluated.

2.4.11 String and '+' Concatenation Operator

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

- If both operands are numeric (*byte*, *short*, *int*, *long*, *float*, *double*, *char*), '+' performs the usual *addition*. For examples,



```
1 1 + 2 ⇒ 3                // int + int ⇒ int
2 1.2 + 2.2 ⇒ 3.4          // double + double ⇒ double
1 1 + 2.2 ⇒ 1.0 + 2.2 ⇒ 3.2 // int + double ⇒ double + double ⇒ double
4 '0' + 2 ⇒ 48 + 2 ⇒ 50     // char + int ⇒ int + int ⇒ int
                          // (need to cast back to char '2')
```

- If both operands are *Strings*, '+' *concatenates* the two *Strings* and returns the concatenated *String*. For examples,



```
"Hello" + "world" ⇒ "Helloworld"
2 "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 Strings concatenated, e.g.,



```

1 "The number is " + 5 ⇒ "The number is " + "5" ⇒ "The number is 5"

3 "The average is " + average + "!" (suppose average=5.5) ⇒ "The average is
   " + "5.5" + "!" ⇒ "The average is 5.5!"

5 "How about " + a + b (suppose a=1, b=1) ⇒ "How about 1" + b ⇒ "How about
   11" ( left - associative )

7 "How about " + (a + b) (suppose a=1, b=1) ⇒ "How about " + 2 ⇒ "How
   about 2"

```

We use *String* concatenation operator '+' frequently in the `print()` and `println()` to produce the desired output String. For examples,



```

1 System.out.println("The sum is: " + sum); // Value of "sum" converted
                                           // to String and concatenated

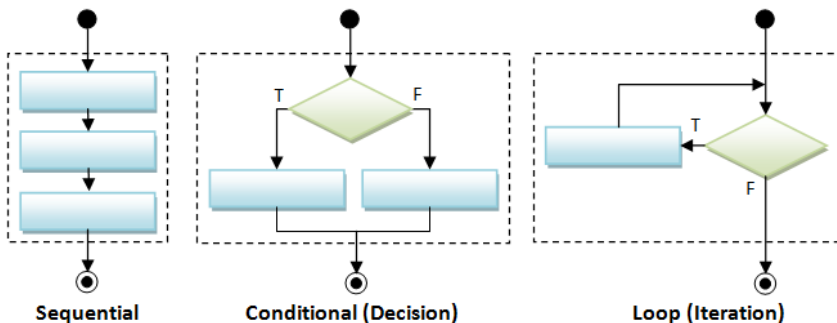
4 System.out.println("The square of " + input + " is " + squareInput);

```

2.5

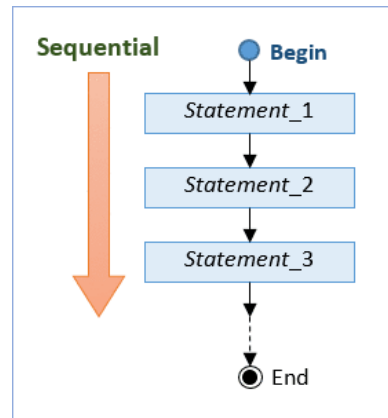
Flow Control

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



2.5.1 Sequential Flow Control

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



2.5.2 Conditional Flow Control

There are a few types of conditionals, if-then, if-then-else, nested-if, switch-case-default, and conditional expression.

if-then

Syntax

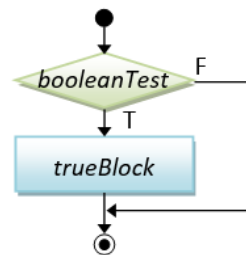


```

// if-then
2 if (booleanTest) {
    trueBlock;
4 }

6 // next statement
  
```

Flowchart



Example



```

// Test if-then with int
2 int mark = 50;    // Assume that mark is [0, 100]
  if (mark >= 50) { // [50, 100]
4   System.out.println("Well Done!");
    System.out.println("Keep it up!");
6 }
  System.out.println("Life goes on!");
  
```



```

8
// Test if-then with double
10 double temperature = 80.1;
    if (temperature > 80) {
12     System.out.println ("Too Hot!");
    }
14 System.out.println ("yummy!");

```

if-then-else

Syntax



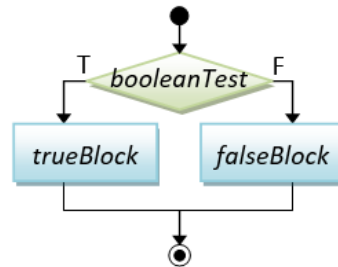
```

// if-then-else
2 if (booleanTest) {
    trueBlock;
4 } else {
    falseBlock;
6 }

8 // next statement

```

Flowchart



Example



```

// Test if-then-else with int
2 int mark = 50; // Assume that mark is [0, 100]
    if (mark >= 50) { // [50, 100]
4     System.out.println ("Congratulation!");
        System.out.println ("Keep it up!");
    } else { // [0, 49]
6     System.out.println ("Try Harder!");
    }
8     System.out.println ("Life goes on!");

10
// Test if-then-else with double
12 double temperature = 80.1;
    if (temperature > 80) {
14     System.out.println ("Too Hot!");
    } else {
16     System.out.println ("Too Cold!");
    }
18 System.out.println ("yummy!");

```

Braces

You could omit the braces {}, if there is only one statement inside the block.



```

1  if (condition)
2    statements;
3  else
4    statements;

```

For example,



```

1  // if-then
2  int absValue = -5;
3  if (absValue < 0)
4    absValue = -absValue;    // Only one statement in the block, can omit { }

6  int min = 0;
7  int value = -5;
8  if (value < min) {    // More than one statements in the block, need { }
9    min = value;
10   System.out.println ("Found new min");
11 }

12 // if-then-else
14 int mark = 50;
15 if (mark >= 50)
16   System.out.println ("PASS");    // Only one statement in the block, can omit {}
17 else {    // More than one statements in the block, need { }
18   System.out.println ("FAIL");
19   System.out.println ("Try Harder!");
20 }

22 // Harder to read without the braces
23 int number1 = 8;
24 int number2 = 9;
25 int absDiff;
26 if (number1 > number2)
27   absDiff = number1 - number2;
28 else
29   absDiff = number2 - number1;

```

However, I recommend that you keep the braces to improve the readability of your program, even if there is only one statement in the block.

Nested-if

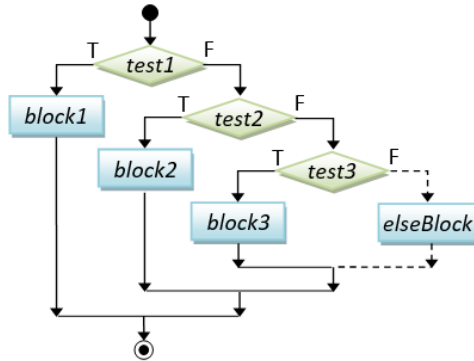
Syntax



```

1 // nested-if
2 if (booleanTest1) {
3     block1;
4 } else if (booleanTest2) {
5     block2;
6 } else if (booleanTest3) {
7     block3;
8 } else if (booleanTest4) {
9     .....
10 } else {
11     elseBlock;
12 }
13 // next statement
  
```

Flowchart



Example



```

1 int mark = 62; // Assume that mark is [0, 100]
2 if (mark >= 80) { // [80, 100]
3     System.out.println("A");
4 } else if (mark >= 65) { // [65, 79]
5     System.out.println("B");
6 } else if (mark >= 50) { // [50, 64]
7     System.out.println("C");
8 } else { // [0, 49]
9     System.out.println("F");
10 }
11 System.out.println("Life goes on!");

12
13 double temperature = 61;
14 if (temperature > 80) { // > 80
15     System.out.println("Too Hot!");
16 } else if (temperature > 75) { // (75, 80]
17     System.out.println("Just right!");
18 } else { // <= 75
19     System.out.println("Too Cold!");
20 }
21 System.out.println("yummy!");
  
```

Java does not provide a separate syntax for nested-if (e.g., with keywords like `elseif`), but supports nested-if with nested if-else statements, which is interpreted as below. Take note that you need to put a space between `else` and `if`. Writing `elseif` causes a syntax error.

Different approach

Syntax

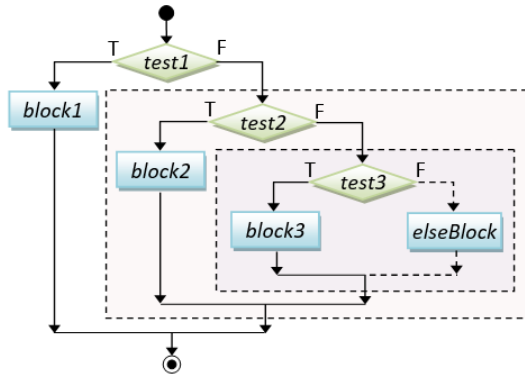


```

1  if ( booleanTest1 ) {
    block1;
3  } else {
    if ( booleanTest2 ) {
5      block2;
    } else {
7      if (booleanTest3) {
        block3;
9      } else {
        if ( booleanTest4 ) {
11         .....
        } else {
13         elseBlock;
        }
15     }
17 }
    // This alignment is hard to read!

```

Flowchart



However, for readability, it is recommended to align the nest-if statement as written in the syntax/examples.

Take note that the blocks are exclusive in a nested-if statement; only one of the blocks will be executed. Also, there are two ways of writing nested-if, for example,



```

// Assume that mark is [0, 100]
2  if (mark >= 80) {           // [80, 100]
    System.out.println("A");
4  } else if (mark >= 65) {    // [65, 79]
    System.out.println("B");
6  } else if (mark >= 50) {    // [50, 64]
    System.out.println("C");
8  } else {                   // [0, 49]
    System.out.println("F");
10 }

12 // OR
13 if (mark < 50) {           // [0, 49]
    System.out.println("F");
14 } else if (mark < 65) {    // [50, 64]
    System.out.println("C");
16 } else if (mark < 80) {    // [65, 79]

```



```

18 System.out.println("B");
   } else {           // [80, 100]
20 System.out.println("A");
   }

```

Dangling-else Problem

The "dangling-else" problem can be illustrated as follows:



```

1 int i = 0, j = 0;
  if (i == 0)           // outer-if
3   if (j == 0)         // inner-if
      System.out.println("i and j are zero");
5 else System.out.println("xxx"); // This else can pair with the inner-if and outer-if?!

```

The else clause in the above code is syntactically applicable to both the outer-if and the inner-if, causing the dangling-else problem.

Java compiler resolves the dangling-else problem by associating the else clause with the *innermost-if* (i.e., the nearest-if). Hence, the above code shall be interpreted as:



```

1 int i = 0, j = 0;
  if (i == 0)
3   if (j == 0)
      System.out.println("i and j are zero");
5 else // associated with if (j == 0) - the nearest if
      System.out.println("xxx");

```

Dangling-else can be prevented by applying explicit parentheses. For example, if you wish to associate the else clause with the outer-if, do this:



```

// Force the else-clause to associate with the outer-if with parentheses
2 int i = 0, j = 0;
  if (i == 0) {
4   if (j == 0)
      System.out.println("i and j are zero");
6 } else {
    System.out.println("i is not zero"); // non-ambiguous for outer-if

```



```

8 }

10 // Force the else-clause to associate with the inner-if with parentheses
   int i = 0, j = 0;
12 if (i == 0) {
   if (j == 0) {
14     System.out.println("i and j are zero");
   } else {
16     System.out.println("i is zero, j is not zero"); // non-ambiguous for inner-if
   }
18 }

```

Nested-if vs. Sequential-if

Study the following code:



```

// Assume mark is between 0 and 100
2 // This "sequential-if" works but NOT efficient !
   // Try mark = 81, which will run thru ALL the if's.
4 int mark = 81;
   if (mark > 80) { // [81, 100]
6     grade = 'A';
   }
8   if (mark > 65 && mark <= 80) { // [66, 80]
     grade = 'B';
10  }
12   if (mark >= 50 && mark <= 65) { // [50, 65]
     grade = 'C';
   }
14   if (mark < 50) { // [0, 49]
     grade = 'F';
16  }

18 // This "nested-if" is BETTER
   // Try mark = 81, which only run thru only the first if.
20 int mark = 81;
   if (mark > 80) { // [81, 100]
22     grade = 'A';
   } else if (mark > 65 && mark <= 80) { // [66, 80]
24     grade = 'B';
   } else if (mark >= 50 && mark <= 65) { // [50, 65]
26     grade = 'C';
   } else {
28     grade = 'F'; // [0, 49]
   }
30

// This "nested-if" is the BEST with fewer tests

```



```

32 int mark = 81;
   if (mark > 80) {           // [81, 100]
34     grade = 'A';
   } else if (mark > 65) {    // [66, 80]
36     grade = 'B';
   } else if (mark >= 50) {   // [50, 65]
38     grade = 'C';
   } else {                  // [0, 49]
40     grade = 'F';
   }

```

switch-case-default

Syntax

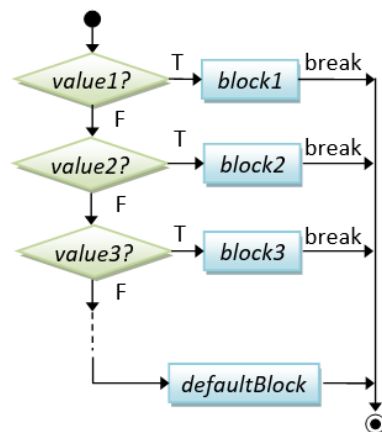


```

1 // switch-case-default
  switch ( selector ) {
3   case value1:
      block1;
5   break;
   case value2:
      block2;
7   break;
   case value3:
      block3;
9   break;
   .....
13  case valueN:
      blockN;
15  break;
   default: // not the above
17     defaultBlock;
   }
19 // next statement
   // Selector Types: byte, short, int, char, String

```

Flowchart



Example



```

// Print number in word
2 int number = 3;
  switch (number) { // "int" selector
4   case 1:         // "int" value

```




```

        System.out.println ("ONE");
6      break;
      case 2:
8        System.out.println ("TWO");
        break;
10     case 3:
        System.out.println ("THREE");
12     break;
      default :
14     System.err.println ("OTHER");
    }

```

"switch-case-default" is an alternative to the "nested-if" for fixed-value tests (but not applicable for range tests). 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.

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, which is typically a mistake. However, we could use this property to handle multiple-value selector. For example,



```

1 // Converting Phone keypad letter to digit
  char inChar = 'x';
3 switch (inChar) {
    case 'a':
5     case 'b':
    case 'c': // 'a' and 'b' (without break) fall thru 'c'
7       System.out.print (2);
        break;
9     case 'd':
    case 'e':
11    case 'f':
        System.out.print (3);
13    break;
    case 'g':
15    case 'h':
    case 'i':
17    System.out.print (4);
        break;
19    case 'j':
    case 'k':
21    case 'l':
        System.out.print (5);
23    break;

```



```

.....
25  default :
      System.out.println("Invalid Input");
27  }

```

Conditional Expression (... ? ... : ...)

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



```

1  // Conditional Expression
   booleanExpr ? trueExpr : falseExpr
3  // An expression that returns the value of trueExpr or falseExpr

```

Examples



```

1  int num1 = 9;
   int num2 = 8;
3  int max = (num1 > num2) ? num1 : num2; // RHS returns num1 or num2
   // same as
5  if (num1 > num2) {
       max = num1;
7  } else {
       max = num2;
9  }

11 int value = -9;
   int absValue = (value > 0) ? value : -value; // RHS returns value or -value
13 // same as
   if (value > 0) {
15     absValue = value;
   } else {
17     absValue = -value;
   }

19
   int mark = 48;
21 System.out.println((mark >= 50) ? "PASS" : "FAIL"); // Return "PASS" or "FAIL"
   // same as

```



```

23 if (mark >= 50) {
    System.out.println("PASS");
25 } else {
    System.out.println("FAIL");
27 }

```

Conditional expression is a short-hand for if-else. But you should use it only for one-liner, for readability.

2.5.3 Exercises on Getting Started and Conditional

1. CheckPassFail

Write a program called **CheckPassFail** which prints "PASS" if the int variable "mark" is more than or equal to 50; or prints "FAIL" otherwise. The program shall always print "DONE" before exiting.

Hints

Use \geq for greater than or equal to comparison.



```

1  /**
   * CheckPassFail.java
3  * Trying if-else statement.
   */
5  public class CheckPassFail { // Save as "CheckPassFail.java"
    public static void main(String[] args) { // Program entry point
6      int mark = 49; // Set the value of "mark" here!
      System.out.println("The mark is " + mark);
9
      // if-else statement
11     if ( ..... ) {
        System.out.println( ..... );
13     } else {
        System.out.println( ..... );
15     }

17     System.out.println( ..... );
    }
19 }

```

Try mark = 0, 49, 50, 51, 100 and verify your results.

Take note of the source-code **indentation**!!! Whenever you open a block with '{', indent all the statements inside the block by 3 (or 4 spaces). When the block ends, un-indent the closing '}' to align with the opening statement.

2. CheckOddEven

Write a program called **CheckOddEven** which prints "Odd Number" if the int variable "number" is odd, or "Even Number" otherwise. The program shall always print "Bye!" before exiting.

Hints

n is an even number if $(n \% 2)$ is 0; otherwise, it is an odd number. Use `==` for comparison, e.g., $(n \% 2) == 0$.



```

1  /**
   * CheckOddEven.java
   * Trying if-else statement and modulus (%) operator.
   */
5  public class CheckOddEven { // Save as "CheckOddEven.java"
    public static void main(String[] args) { // Program entry point
7      int number = 49; // Set the value of "number" here!
        System.out.println("The number is " + number);
9      if ( ..... ) {
            System.out.println( ..... ); // even number
11     } else {
            System.out.println( ..... ); // odd number
13     }

15     System.out.println( ..... );
    }
17 }

```

Try number = 0, 1, 88, 99, -1, -2 and verify your results.

Again, take note of the source-code indentation! Make it a good habit to indent your code properly, for ease of reading your program.

3. PrintNumberInWord

Write a program called **PrintNumberInWord** which prints "ONE", "TWO", ..., "NINE", "OTHER" if the int variable "number" is 1, 2, ..., 9, or other, respectively. Use (a) a "nested-if" statement; (b) a "switch-case-default" statement.

Hints



```

1  /**
   * PrintNumberInWord.java
3  * Trying nested-if and switch-case statements.
   */
5  public class PrintNumberInWord { // Save as "PrintNumberInWord.java"
    public static void main(String[] args) {
6      int number = 5; // Set the value of "number" here!

7

9      // Using nested-if
      if (number == 1) { // Use == for comparison
10         System.out.println( ..... );
      } else if ( ..... ) {
11
12         .....
      } else if ( ..... ) {
13
14         .....
15         .....
16         .....
17     } else {
18         .....
19     }

20
21     // Using switch-case-default
22     switch(number) {
23     case 1:
24         System.out.println( ..... );
25         break; // Don't forget the "break" after each case!
26     case 2:
27         System.out.println( ..... );
28         break;
29         .....
30         .....
31         .....
32     default :
33         System.out.println( ..... );
34     }
35 }
36 }
37 }

```

Try number = 0, 1, 2, 3, ..., 9, 10 and verify your results.

4. PrintDayInWord

Write a program called **PrintDayInWord** which prints "Sunday", "Monday", ... "Saturday" if the *int* variable "*dayNumber*" is 0, 1, ..., 6, respectively. Otherwise, it shall print "Not a valid day". Use (a) a "nested-if" statement; (b) a "switch-case-default" statement.

Try dayNumber = 0, 1, 2, 3, 4, 5, 6, 7 and verify your results.

2.5.4 Loop Flow Control

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

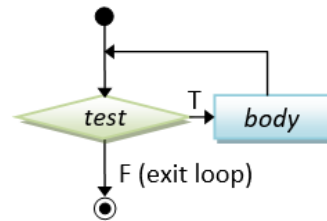
while-do

Syntax



```
1 // while-do loop
  while (booleanTest) {
3   body;
  }
5 // next statement
```

Flowchart



Example



```
1 /**
   * WhileDoLoopExample.java
3  */
  public class WhileDoLoopExample {
5   public static void main(String[] args) {
      // Sum from 1 to upperbound
7   int sum = 0;
      final int UPPERBOUND = 100;
9   int number = 1; // init
      while (number <= UPPERBOUND) {
11    // number = 1, 2, 3, ..., UPPERBOUND for each iteration
        sum += number;
13    ++number; // update
      }
15   System.out.println("sum is: " + sum);

17   // Factorial of n (= 1*2*3*...* n)
      int n = 5;
19   int factorial = 1;
      int number = 1; // init
21   while (number <= n) {
        // num = 1, 2, 3, ..., n for each iteration
23    factorial *= number;
        ++num; // update
25   }
      System.out.println(" factorial is: " + factorial );
27  }
  }
```

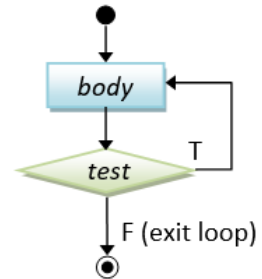
do-while**Syntax**

```

// do-while loop
2 do {
    body;
4 } while (booleanTest);

6 // next statement
// Need a semi-colon to terminate do-while statement

```

Flowchart**Example**

```

// Sum from 1 to upperbound
2 int sum = 0;
  final int UPPERBOUND = 100;
4 int number = 1; // init
  do {
6   // number = 1, 2, 3, ..., UPPERBOUND for each iteration
    sum += number;
8   ++number; // update
  } while (number <= UPPERBOUND);
10 System.out.println("sum is: " + sum);

12 // Factorial of n (=1*2*3*...* n)
  int n = 5;
14 int factorial = 1;
  int number = 1; // init
16 do {
    // num = 1, 2, 3, ..., n for each iteration
18   factorial *= number;
    ++number; // update
20 } while (number <= n);
  System.out.println("factorial is: " + factorial);

```

The difference between *while-do* and *do-while* lies in the order of the *body* and *test*. In *while-do*, the *test* is carried out first. The body will be executed if the *test* is true and the process repeats. In *do-while*, the *body* is executed and then the *test* is carried out. Take note that the *body* of *do-while* is executed at least once (1+); but the body of *while-do* is possibly zero (0+). Similarly, the for-loop's body could possibly not be executed (0+).

for-loop

Syntax

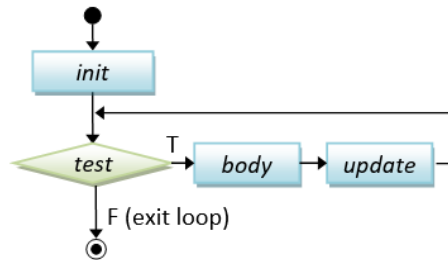


```

1 // for-loop
  for ( init ; booleanTest ; update ) {
3   body;
  }
5 // next statement

```

Flowchart



Example



```

1 /**
   * ForLoopExample.java
3  */
   public class ForLoopExample {
5     public static void main(String[] args) {
       // Sum from 1 to upperbound
7       int sum = 0;
       final int UPPERBOUND = 100;
9       for ( int number = 1; number <= UPPERBOUND; ++number ) {
           // num = 1, 2, 3, ..., UPPERBOUND for each iteration
11          sum += number;
       }
13       System.out.println("sum is: " + sum);

15       // Factorial of n (=1*2*3*...* n)
       int n = 5;
       int factorial = 1;
17       for ( int number = 1; number <= n; ++number ) {
19           // number = 1, 2, 3, ..., n for each iteration
           factorial *= number;
21       }
       System.out.println(" factorial is: " + factorial );
23     }
   }

```

for-loop is a shorthand for *while-do* with fewer lines of code. It is the most commonly-used loop especially if the number of repetitions is known. But its syntax is harder to comprehend. Make sure that you understand *for-loop* by going through the flow-chart and examples.

Loop's Index/Counter Variable

A loop is typically controlled by an *index* or *counter* variable. For example,



```

public class LoopCounter {
2  public static void main() {
    // Sum from 1 to UPPERBOUND using for-loop
4  int sum = 0;
    final int UPPERBOUND = 100;
6  for (int number = 1; number <= UPPERBOUND; ++number) {
    // number = 1, 2, 3, ..., UPPERBOUND for each iteration
8      sum += number;
    }

10
    // Sum from 1 to UPPERBOUND using while-loop
12  int sum = 0;
    final int UPPERBOUND = 100;
14  int number = 1;
    while (number <= UPPERBOUND) {
16      // number = 1, 2, 3, ..., UPPERBOUND for each iteration
        sum += number;
18      ++number;
    }
20 }
}

```

In the above examples, the variable `number` serves as the index variable, which takes on the values 1, 2, 3, ..., `UPPERBOUND` for each iteration of the loop. You need to increase/decrease/modify the index variable explicitly (e.g., via `++number`). Otherwise, the loop becomes an endless loop, as the test (`number <= UPPERBOUND`) will return the same outcome for the same value of `number`.

Observe that `for`-loop is a shorthand of `while`-loop. Both the `for`-loop and `while`-loop have the same set of statements, but `for`-loop re-arranges the statements.

For the `for`-loop, the index variable `number` is declared inside the loop, and therefore is only available inside the loop. You cannot access the variable after the loop, as it is destroyed after the loop. On the other hand, for the `while`-loop, the index variable `number` is available inside and outside the loop.

For the `for`-loop, you can choose to declare the index variable inside the loop or outside the loop. We recommend that you declare it inside the loop, to keep the life-span of this variable to where it is needed, and not any longer.

Code Example: Sum and Average of Running Integers

The following program sums the running integers from a given lowerbound to an upperbound. Also compute their average.



```

1  /**
   * SumAverageRunningNumbers.java
3  * Sum the running integers from lowerbound to an upperbound.
   * Also compute the average.
5  */
   public class SumAverageRunningNumbers {
7     public static void main(String[] args) {
           // Declare variables
9         int sum = 0;           // store the accumulated sum
           final int LOWERBOUND = 1;
11        final int UPPERBOUND = 1000;
           double average;

13
           // Use a for-loop to accumulate the sum
15        for (int number = LOWERBOUND; number <= UPPERBOUND; ++number) {
               // number = LOWERBOUND, LOWERBOUND+1, LOWERBOUND+2, ...,
               // UPPERBOUND for each iteration
17            sum += number;
           }
19        average = (double)sum / (UPPERBOUND - LOWERBOUND + 1); // need to cast int
               ↪ to double first
           // Print results
21        System.out.println("The sum from " + LOWERBOUND + " to "
               + UPPERBOUND + " is: " + sum);
23        //The sum from 1 to 1000 is: 500500
           System.out.println("The average is: " + average);
25        //The average is: 500.5

27        // Sum only the ODD numbers
           int count = 0;           // counts of odd numbers
29        sum = 0;           // reset sum for accumulation again
           // Adjust the LOWERBOUND to the next odd number if it is a even number
31        final int ADJUSTED_LOWERBOUND = LOWERBOUND % 2 == 0 ?
               LOWERBOUND + 1 : LOWERBOUND;
33        // Use a for-loop to accumulate the sum with step size of 2
           for (int number = ADJUSTED_LOWERBOUND; number <= UPPERBOUND;
               ↪ number += 2) {
35            // number = ADJUSTED_LOWERBOUND, ADJUSTED_LOWERBOUND+2,
               ADJUSTED_LOWERBOUND+4, ..., UPPERBOUND for each iteration
               ++count;
37            sum += number;
           }
39        average = (double)sum / count;
           System.out.println("The sum of odd numbers is: " + sum);
41        //The sum of odd numbers is: 250000
           System.out.println("The average of odd numbers is: " + average);
43        //The average of odd numbers is: 500.0
           }
45    }

```

Using boolean Flag for Loop Control

Besides using an index variable for loop control, another common way to control the loop is via a boolean flag.

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.



```

1 // Game loop
  boolean gameOver = false;
3 while (!gameOver) {
    // play the game
5     .....
    .....
7    // Update the game state
    // Set gameOver to true if appropriate to exit the game loop
9    if ( ..... ) {
        gameOver = true;    // exit the loop upon the next iteration test
11   }
   }

```

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
2 boolean isValid = false ;
  int number;
4 do {
    // prompt user to enter an int between 1 and 10
6     .....
    // if the number entered is valid , set done to exit the loop
8    if (number >= 1 && number <= 10) {
        isValid = true;    // exit the loop upon the next iteration test
10    // Do the operations
        .....
12    } else {
        // Print error message and repeat ( isValid remains false )
14    .....
    }
16 } while (! isValid );    // Repeat for invalid input

```

for-loop with Comma Separator

You could place more than one statement in the *init* and *update*, separated with commas. For example,



```
// for (init ; test ; update) { ..... }
2 for (int row = 0, col = 0; row < SIZE; ++row, ++col) {
    // Process diagonal elements (0,0) , (1,1) , (2,2) ,...
4     .....
    }
```

The *test* must be a *boolean* expression that returns a *boolean true* or *false*.

2.5.5 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,



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

The return statement

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



```
public static void main(String[] args) {
2     ...
    if (errorCount > 10) {
4         System.out.println ("too many errors");
        return; // Terminate and return control to Java Runtime from main()
6     }
    ...
8 }
```

2.5.6 Exercises on Decision and Loop

1. SumAverageRunningInt

Write a program called **SumAverageRunningInt** to produce the sum of 1, 2, 3, ..., to 100. Store 1 and 100 in variables *lowerbound* and *upperbound*, so that we can change their values easily. Also compute and display the average. The output shall look like:

Command window
⌵

```

1 The sum of 1 to 100 is 5050
2 The average is 50.5

```

Hints



```

/**
2  * SumAverageRunningInt.java
  * Compute the sum and average of running integers from a lowerbound
4  * to an upperbound using loop.
  */
6 public class SumAverageRunningInt { // Save as "SumAverageRunningInt.java"
    public static void main(String[] args) {
8        // Define variables
        int sum = 0;           // The accumulated sum, init to 0
10       double average;      // average in double
        final int LOWERBOUND = 1;
12       final int UPPERBOUND = 100;

14       // Use a for-loop to sum from lowerbound to upperbound
        for (int number = LOWERBOUND; number <= UPPERBOUND; ++number) {
16           // The loop index variable number = 1, 2, 3, ..., 99, 100
            sum += number;    // same as "sum = sum + number"
18         }

20         // Compute average in double. Beware that int / int produces int !
        .....

22         // Print sum and average
        .....
24     }
26 }

```

Try

- (a) Modify the program to use a "while-do" loop instead of "for" loop.



```

1 int sum = 0;
2 int number = LOWERBOUND; // declare and init loop index variable
while (number <= UPPERBOUND) { // test
3     sum += number;
4     ++number;                // update
5 }

```

- (b) Modify the program to use a "do-while" loop.



```

1 int sum = 0;
2 int number = LOWERBOUND; // declare and init loop index variable
3 do {
4     sum += number;
5     ++number;                // update
6 } while (number <= UPPERBOUND); // test

```

- (c) What is the difference between "for" and "while-do" loops? What is the difference between "while-do" and "do-while" loops?
- (d) Modify the program to sum from 111 to 8899, and compute the average. Introduce an *int* variable called *count* to count the numbers in the specified range (to be used in computing the average).



```

1 int count = 0; // Count the number within the range, init to 0
for ( ...; ...; ... ) {
2     .....
3     ++count;
4 }

```

- (e) Modify the program to find the "sum of the squares" of all the numbers from 1 to 100, i.e. $1 * 1 + 2 * 2 + 3 * 3 + \dots + 100 * 100$.
- (f) Modify the program to produce two sums: sum of odd numbers and sum of even numbers from 1 to 100. Also compute their absolute difference.



```

// Define variables
2 int sumOdd = 0; // Accumulating sum of odd numbers

```



```

1 int sumEven = 0; // Accumulating sum of even numbers
4 int absDiff;    // Absolute difference between the two sums
   .....
6
   // Compute sums
8 for (int number = ...; ...; ...) {
   if (.....) {
10     sumOdd += number;
   } else {
12     sumEven += number;
   }
14 }

16 // Compute Absolute Difference
   if (sumOdd > sumEven) {
18     absDiff = .....;
   } else {
20     absDiff = .....;
   }
22
   // OR use one liner conditional expression
24 absDiff = (sumOdd > sumEven) ? ..... : .....;

```

2. Product1ToN (or Factorial)

Write a program called **Product1ToN** to compute the product of integers from 1 to 10 (i.e., $1 \times 2 \times 3 \times \cdots \times 10$), as an *int*. Take note that It is the same as factorial of *N*.

Hints

Declare an int variable called product, initialize to 1, to accumulate the product.



```

1 // Define variables
   int product = 1; // The accumulated product, init to 1
3 final int LOWERBOUND = 1;
   final int UPPERBOUND = 10;

```

Try

- Compute the product from 1 to 11, 1 to 12, 1 to 13 and 1 to 14. Write down the product obtained and decide if the results are correct.

Hints: Factorial of 13 (= 6227020800) is outside the range of int [-2147483648, 2147483647]. Take note that computer programs may not produce the correct result even though the code seems correct!

- Repeat the above, but use long to store the product. Compare the products obtained with int for $N = 13$ and $N = 14$.

Hints: With *long*, you can store factorial of up to 20.

3. HarmonicSum

Write a program called **HarmonicSum** to compute the sum of a harmonic series, as shown below, where $n = 50000$. The program shall compute the sum from left-to-right as well as from the right-to-left. Are the two sums the same? Obtain the absolute difference between these two sums and explain the difference. Which sum is more accurate?

$$\text{harmonic}(n) = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}.$$

Hints



```

1  /**
2   * HarmonicSum.java
3   * Compute the sum of harmonics series from left-to-right and right-to-left.
4   */
5   public class HarmonicSum { // Save as "HarmonicSum.java"
6       public static void main(String[] args) {
7           final int MAX_DENOMINATOR = 50000; // Use a more meaningful name
8               ↳ instead of n
9           double sumL2R = 0.0;           // Sum from left-to-right
10          double sumR2L = 0.0;           // Sum from right-to-left
11          double absDiff;                 // Absolute difference between the two sums
12
13          // for-loop for summing from left-to-right
14          for (int denominator = 1; denominator <= MAX_DENOMINATOR; ++
15               ↳ denominator) {
16              // denominator = 1, 2, 3, 4, 5, ..., MAX_DENOMINATOR
17              .....
18              // Beware that int/int gives int, e.g., 1/2 gives 0.
19          }
20          System.out.println("The sum from left-to-right is: " + sumL2R);
21
22          // for-loop for summing from right-to-left
23          .....
24
25          // Find the absolute difference and display
26          if (sumL2R > sumR2L) .....
27          else .....
28      }
29  }

```

4. ComputePI

Write a program called **ComputePI** to compute the value of π , using the following series expansion. Use the maximum denominator (MAX_DENOMINATOR) as the terminating condition. Try MAX_DENOMINATOR of 1000, 10000, 100000, 1000000 and compare the *PI* obtained. Is this series suitable for computing *PI*? Why?

$$\pi = 4 \times \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \frac{1}{13} - \frac{1}{15} + \dots \right).$$

Hints

Add to sum if the denominator % 4 is 1, and subtract from sum if it is 3.



```

1 double sum = 0.0;
  int MAX_DENOMINATOR = 1000; // Try 10000, 100000, 1000000
3 for (int denom = 1; denom <= MAX_DENOMINATOR; denom += 2) {
    // denominator = 1, 3, 5, 7, ..., MAX_DENOMINATOR
5     if (denom % 4 == 1) {
        sum += .....;
7     } else if (denom % 4 == 3) {
        sum -= .....;
9     } else { // remainder of 0 or 2
        System.out.println("Impossible !!!");
11    }
    }
13    .....

```

Try

- (a) Instead of using maximum denominator as the terminating condition, rewrite your program to use the maximum number of terms (MAX_TERM) as the terminating condition.



```

1 int MAX_TERM = 10000; // number of terms used in computation
  int sum = 0.0;
3 for (int term = 1; term <= MAX_TERM; term++) {
    // term = 1, 2, 3, 4, ..., MAX_TERM
5     if (term % 2 == 1) { // odd term number: add
        sum += 1.0 / (term * 2 - 1);
7     } else { // even term number: subtract
        .....
9     }
    }

```

- (b) JDK maintains the value of π in a built-in double constant called *Math.PI* ($= 3.141592653589793$). Add a statement to compare the values obtained and the *Math.PI*, in percents of *Math.PI*, i.e., $(piComputed / Math.PI) * 100$.

5. Fibonacci

Write a program called **Fibonacci** to print the first 20 Fibonacci numbers $F(n)$, where

$$F(n) = F(n-1) + F(n-2) \quad \text{and} \quad F(1) = F(2) = 1.$$

Also compute their average. The output shall look like:

Command window
▼

```

The first 20 Fibonacci numbers are:
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765
The average is 885.5
```

Hints



```

1  /**
   * Fibonacci.java
   * Print first 20 Fibonacci numbers and their average
   */
5  public class Fibonacci {
    public static void main(String[] args) {
7      int n = 3; // The index n for F(n), starting from n = 3, as n = 1 and n =
                // 2 are pre-defined
        int fn; // F(n) to be computed
9      int fnMinus1 = 1; // F(n-1), init to F(2)
        int fnMinus2 = 1; // F(n-2), init to F(1)
11     int nMax = 20; // maximum n, inclusive
        int sum = fnMinus1 + fnMinus2; // Need sum to compute average
13     double average;

15     System.out.println("The first " + nMax + " Fibonacci numbers are:");
        .....

17     while (n <= nMax) { // n starts from 3
19         // n = 3, 4, 5, ..., nMax
        // Compute F(n), print it and add to sum
21         .....

23         // Increment the index n and shift the numbers for the next iteration
        ++n;
25         fnMinus2 = fnMinus1;
```



```

27     fnMinus1 = fn;
    }

29     // Compute and display the average (=sum/nMax).
    // Beware that int/int gives int.
31     .....
    }
33 }
```

Try

- (a) Tribonacci numbers are a sequence of numbers $T(n)$ similar to Fibonacci numbers, except that a number is formed by adding the three previous numbers, i.e., $T(n) = T(n-1) + T(n-2) + T(n-3)$, $T(1) = T(2) = 1$, and $T(3) = 2$. Write a program called **Tribonacci** to produce the first twenty Tribonacci numbers.

6. ExtractDigits

Write a program called **ExtractDigits** to extract each digit from an *int*, in the reverse order. For example, if the int is 15423, the output shall be "3 2 4 5 1", with a space separating the digits.

Hints

The coding pattern for extracting individual digits from an integer n is:

- Use $(n \% 10)$ to extract the last (least-significant) digit.
- Use $n = n / 10$ to drop the last (least-significant) digit.
- Repeat if $(n > 0)$, i.e., more digits to extract.

Take note that n is destroyed in the process. You may need to clone a copy.



```

1  int n = ...;
   while (n > 0) {
3     int digit = n % 10; // Extract the least - significant digit

5     // Print this digit
     .....
7
     n = n / 10; // Drop the least - significant digit and repeat the loop
9 }
```

2.6 Input/Output

2.6.1 Formatted Output via "printf()" (JDK 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:



```
1 printf( formattingString , 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]conversionCode`", will be substituted by the arguments following the *formattingString*, usually in a one-to-one and sequential manner. A format specifier begins with a '`%`' and ends with the *conversionCode*, e.g., `%d` for integer, `%f` for floating-point number (*float* and *double*), `%c` for char and `%s` for String. An optional width can be inserted in between to specify the field-width. Similarly, an optional flags can be used to control the alignment, padding and others.

For examples,

- `%d`, `% α d`: integer printed in α spaces (α is optional), right-aligned. If α is omitted, the number of spaces is the length of the integer.
- `%s`, `% α s`: String printed in α spaces (α is optional), right-aligned. If α is omitted, the number of spaces is the length of the string (to fit the string).
- `%f`, `% α . β f`, `%. β f`: Floating point number (*float* and *double*) printed in α spaces with β decimal digits (α and β are optional). If α is omitted, the number of spaces is the length of the floating-point number.
- `%n`: a system-specific new line (Windows uses `"\r\n"`, Unix and macOS `"\n"`).

Example



```
1 // Without specifying field-width
  System.out.printf( "Hi, %s| %d| %f|, @xyz%n", "Hello", 123, 45.6 );
3
  // Specifying the field-width and decimal places for double
```



```

5 System.out.printf("Hi,| %6s| %6d| %6.2f|, @xyz%n", "Hello", 123, 45.6);

7 // Various way to format integers :
  // flag '-' for left-align, '0' for padding with 0
9 System.out.printf("Hi,| %d| %5d| %-5d| %05d|, @xyz%n", 111, 222, 333, 444);

11 // Various way to format floating-point numbers:
  // flag '-' for left-align
13 System.out.printf("Hi,| %f| %7.2f| %.2f| %-7.2f|, @xyz%n", 11.1, 22.2, 33.3, 44.4);

15 // To print a '%', use %% (as % has special meaning)
  System.out.printf("The rate is: %.2f%%. %n", 1.2);

```

Command window

```

Hi,| Hello |123|45.600000|, @xyz
2 Hi,| Hello|   123|  45.60|, @xyz
  Hi,|111|   222|333   |00444|, @xyz
4 Hi,|11.100000|   22.20|33.30|44.40   |, @xyz
  The rate is: 1.20%.

```

Take note that `printf()` does not advance the cursor to the next line after printing. You need to explicitly print a newline character (via `%n`) at the end of the formatting-string to advance the cursor to the next line, if desired, as shown in the above examples.

There are many more format specifiers in Java. Refer to JDK Documentation for the detailed descriptions (@ <https://docs.oracle.com/javase/10/docs/api/java/util/Formatter.html> for JDK 10). (Also take note that `printf()` take a variable number of arguments (or varargs), which is a new feature introduced in JDK 5 in order to support `printf()`)

2.6.2 Input From Keyboard via "Scanner" (JDK 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 display 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).

JDK 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).



```

1 import java.util.Scanner;    // Needed to use the Scanner

3 /**
   * ScannerTest.java
5  * Test input scanner
   */
7 public class ScannerTest {
   public static void main(String[] args) {
9       // Read inputs from keyboard
       // Construct a Scanner named "in" for scanning System.in (keyboard)
11      Scanner in = new Scanner(System.in);

13      System.out.print("Enter an integer: "); // Show prompting message
       int num1 = in.nextInt();                // Use nextInt() to read an int

15
       System.out.print("Enter a floating point: "); // Show prompting message
17      double num2 = in.nextDouble();          // Use nextDouble() to read a double

19
       System.out.print("Enter a string: "); // Show prompting message
       String str = in.next();                // Use next() to read a String token, up to white space
21      in.close();                            // Scanner not longer needed, close it

23      // Formatted output via printf()
       System.out.printf("%s, Sum of %d & %.2f is %.2f%n",
25          str, num1, num2, num1+num2);
   }
27 }

```

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



```

1 import java.util.Scanner;    // Needed to use the Scanner

3 /**
   * TestScannerNextLine.java
5  * Test Scanner's nextLine()
   */
7 public class TestScannerNextLine {

```



```

public static void main(String[] args) {
9    // Read inputs from keyboard
    // Construct a Scanner named "in" for scanning System.in (keyboard)
11   Scanner in = new Scanner(System.in);

13   System.out.print("Enter a string (with space): ");
    // Use nextLine() to read entire line including white spaces,
15   // but excluding the terminating newline.
    String str = in.nextLine();
17   in.close();
    System.out.printf("%s\n", str);
19 }
}

```

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 module *java.base* ⇒ package *java.util* ⇒ class **Scanner** ⇒ Method (@ <https://docs.oracle.com/javase/17/docs/api/java/util/Scanner.html> for JDK 17).

2.6.3 Code Example: Prompt User for Two Integers and Print their Sum

The following program prompts user for two integers and print their sum. For examples,

Command window

```

Enter first integer : 8
2 Enter second integer : 9
The sum is: 17

```



```

1 import java.util.Scanner; // For keyboard input

3 /**
 * 1. Prompt user for 2 integers
5 * 2. Read inputs as "int"
 * 3. Compute their sum in "int"
7 * 4. Print the result
 */

```



```
9 public class Add2Integer { // Save as "Add2Integer.java"
    public static void main (String [] args) {
11     // Put up prompting messages and read inputs as "int"
        Scanner in = new Scanner(System.in); // Scan the keyboard for input
13
        System.out.print ("Enter first integer: "); // No newline for prompting message
15     int number1 = in.nextInt (); // Read next input as "int"

17     System.out.print ("Enter second integer: ");
        int number2 = in.nextInt ();
19     in.close ();

21     // Compute sum
        int sum = number1 + number2;
23
        // Display result
25     System.out.println ("The sum is: " + sum); // Print with newline
    }
27 }
```

2.6.4 Code Example: Income Tax Calculator

The progressive income tax rate is mandated as follows:

| Taxable Income | Rate (%) |
|----------------|----------|
| First \$20,000 | 0 |
| Next \$20,000 | 10 |
| Next \$20,000 | 20 |
| The remaining | 30 |

For example, suppose that the taxable income is \$85000, the income tax payable is $\$20000 * 0\% + \$20000 * 10\% + \$20000 * 20\% + \$25000 * 30\%$. Write a program called `IncomeTaxCalculator` that reads the taxable income (in *int*). The program shall calculate the income tax payable (in *double*); and print the result rounded to 2 decimal places.

Command window

1

Enter the taxable income: \$41234
The income tax payable is : \$2246.80

3

5

Enter the taxable income: \$67891
The income tax payable is : \$8367.30

Command window

```

1 Enter the taxable income: $85432
  The income tax payable is : $13629.60
3
  Enter the taxable income: $12345
5 The income tax payable is : $0.00

```



```

1 import java.util.Scanner; // For keyboard input

3 /**
   * IncomeTaxCalculator.java
   * 1. Prompt user for the taxable income in integer .
   * 2. Read input as "int ".
   * 3. Compute the tax payable using nested-if in "double".
   * 4. Print the values rounded to 2 decimal places .
   */
10 public class IncomeTaxCalculator {
11     public static void main(String[] args) {
12         // Declare constants first (variables may use these constants)
13         final double TAX_RATE_ABOVE_20K = 0.1;
14         final double TAX_RATE_ABOVE_40K = 0.2;
15         final double TAX_RATE_ABOVE_60K = 0.3;

17         // Prompt and read inputs as "int "
18         Scanner in = new Scanner(System.in);
19         System.out.print("Enter the taxable income: $");
20         int taxableIncome = in.nextInt();
21         in.close();

23         // Compute tax payable in "double" using a nested-if to handle 4 cases
24         double taxPayable;
25         if (taxableIncome <= 20000) { // [0, 20000]
26             taxPayable = 0;
27         } else if (taxableIncome <= 40000) { // [20001, 40000]
28             taxPayable = (taxableIncome - 20000) * TAX_RATE_ABOVE_20K;
29         } else if (taxableIncome <= 60000) { // [40001, 60000]
30             taxPayable = 20000 * TAX_RATE_ABOVE_20K
31                 + (taxableIncome - 40000) * TAX_RATE_ABOVE_40K;
32         } else { // >=60001
33             taxPayable = 20000 * TAX_RATE_ABOVE_20K
34                 + 20000 * TAX_RATE_ABOVE_40K
35                 + (taxableIncome - 60000) * TAX_RATE_ABOVE_60K;
36         }

37         // Alternatively , you could use the following nested-if conditions
38         // but the above follows the table data
39         // if (taxableIncome > 60000) { // [60001, ]

```



```

41 // .....
   // } else if (taxableIncome > 40000) { // [40001, 60000]
43 // .....
   // } else if (taxableIncome > 20000) { // [20001, 40000]
45 // .....
   // } else { // [0, 20000]
47 // .....
   // }

49 // Print result rounded to 2 decimal places
51 System.out.printf("The income tax payable is: $%.2f%n", taxPayable);
   }
53 }

```

2.6.5 Code Example: Income Tax Calculator with Sentinel

Based on the previous example, write a program called `IncomeTaxCalculatorSentinel` which shall repeat the calculations until user enter `-1`. For example,

Command window

```

1 Enter the taxable income: $41000
  The income tax payable is: $2200.00
3 Enter the taxable income: $62000
  The income tax payable is: $6600.00
5 Enter the taxable income: $73123
  The income tax payable is: $9936.90
7 Enter the taxable income: $84328
  The income tax payable is: $13298.40
9 Enter the taxable income: $-1
  bye!

```

The `-1` is known as the *sentinel value*. (In programming, a *sentinel value*, also referred to as a flag value, trip value, rogue value, signal value, or dummy data, is a special value which uses its presence as a condition of termination.)



```

import java.util.Scanner; // For keyboard input

2
/**
4  * IncomeTaxCalculatorSentinel.java
  * 1. Prompt user for the taxable income in integer.
6  * 2. Read input as "int".

```



```

* 3. Compute the tax payable using nested-if in "double".
8 * 4. Print the values rounded to 2 decimal places.
* 5. Repeat until user enter -1.
10 */
12 public class IncomeTaxCalculatorSentinel {
13     public static void main(String[] args) {
14         // Declare constants first (variables may use these constants)
15         final double TAX_RATE_ABOVE_20K = 0.1;
16         final double TAX_RATE_ABOVE_40K = 0.2;
17         final double TAX_RATE_ABOVE_60K = 0.3;
18         final int SENTINEL = -1; // Terminating value for input
19
20         // Declare variables
21         int taxableIncome;
22         double taxPayable;
23
24         Scanner in = new Scanner(System.in);
25         // Read the first input to "seed" the while loop
26         System.out.print("Enter the taxable income: $");
27         taxableIncome = in.nextInt();
28
29         while (taxableIncome != SENTINEL) {
30             // Compute tax payable in "double" using a nested-if to handle 4 cases
31             if (taxableIncome > 60000) {
32                 taxPayable = 20000 * TAX_RATE_ABOVE_20K
33                     + 20000 * TAX_RATE_ABOVE_40K
34                     + (taxableIncome - 60000) * TAX_RATE_ABOVE_60K;
35             } else if (taxableIncome > 40000) {
36                 taxPayable = 20000 * TAX_RATE_ABOVE_20K
37                     + (taxableIncome - 40000) * TAX_RATE_ABOVE_40K;
38             } else if (taxableIncome > 20000) {
39                 taxPayable = (taxableIncome - 20000) * TAX_RATE_ABOVE_20K;
40             } else {
41                 taxPayable = 0;
42             }
43
44             // Print result rounded to 2 decimal places
45             System.out.printf("The income tax payable is: $%.2f%n", taxPayable);
46
47             // Read the next input
48             System.out.print("Enter the taxable income: $");
49             taxableIncome = in.nextInt();
50             // Repeat the loop body, only if the input is not the SENTINEL value.
51             // Take note that you need to repeat these two statements inside/outside the loop!
52         }
53         System.out.println("bye!");
54         in.close(); // Close Scanner
55     }
56 }

```

Notes: The *coding pattern* for handling input with sentinel (terminating) value is as follows:



```

1 // Get first input to "seed" the while loop
  input = .....;
3 while (input != SENTINEL){
    // Process input
5     .....
    .....
7 // Get next input and repeat the loop
  input = .....;    // Need to repeat these statements
9 }
    .....

```

2.6.6 Code Example: Guess A Number

Guess a number between 0 and 99.



```

import java . util .Scanner;

2
/**
4  * NumberGuess.java
  * Guess a secret number between 0 and 99.
6  */
public class NumberGuess {
8  public static void main(String[] args) {
    // Define variables
10  final int SECRET_NUMBER; // Secret number to be guessed
    int numberIn;           // The guessed number entered
12  int trialNumber = 0;     // Number of trials so far
    boolean done = false ;  // boolean flag for loop control
14
    Scanner in = new Scanner(System.in);

16
    // Set up the secret number: Math.random() generates a double in [0.0, 1.0)
18  SECRET_NUMBER = (int)(Math.random()*100);

20
    // Use a while-loop to repeatedly guess the number until it is correct
    while (!done) {
22        ++trialNumber;
        System.out. print ("Enter your guess (between 0 and 99): ");
24        numberIn = in. nextInt ();
        if (numberIn == SECRET_NUMBER) {
26            System.out. println ("Congratulation");
            done = true ;

```



```

28     } else if (numberIn < SECRET_NUMBER) {
        System.out.println ("Try higher");
30     } else {
        System.out.println ("Try lower");
32     }
    }
34
    System.out.println ("You got in " + trialNumber + " trials ");
36    in.close ();
    }
38 }

```

Notes: The above program uses a boolean flag to control the loop, in the following coding pattern:



```

boolean done = false ;
2 while (!done) {
    if (.....) {
4         done = true;    // exit the loop upon the next iteration
        .....
6     }

8     .....                // done remains false . repeat loop
    }

```

2.6.7 Exercises on Decision/Loop with Input

1. SumProductMinMax3

Write a program called **SumProductMinMax3** that prompts user for three integers. The program shall read the inputs as int; compute the sum, product, minimum and maximum of the three integers; and print the results. For examples,

Command window

```

1 Enter 1st integer : 8
  Enter 2nd integer : 2
3 Enter 3rd integer : 9
  The sum is: 19
5 The product is : 144
  The min is : 2

```

Command window

The max is: 9

Hints



```
1 // Declare variables
  // The 3 input integers
3 int number1;
  int number2;
5 int number3;

7 // To compute these
  int sum;
9 int product;
  int min;
11 int max;

13 // Prompt and read inputs as "int"
  Scanner in = new Scanner(System.in); // Scan the keyboard
15 .....
  .....
17 in.close();

19 // Compute sum and product
  sum = .....
21 product = .....

23 // Compute min
  // The "coding pattern" for computing min is:
25 // 1. Set min to the first item
  // 2. Compare current min with the second item and update min if second item is
     ↪ smaller
27 // 3. Repeat for the next item
  min = number1; // Assume min is the 1st item
29 if (number2 < min) { // Check if the 2nd item is smaller than current min
    min = number2; // Update min if so
31 }
  if (number3 < min) { // Continue for the next item
33   min = number3;
  }
35

  // Compute max – similar to min
37 .....

39 // Print results
  .....
```

Try

- (a) Write a program called **SumProductMinMax5** that prompts user for five integers. The program shall read the inputs as *int*; compute the sum, product, minimum and maximum of the five integers; and print the results. Use five *int* variables: *number1*, *number2*, ..., *number5* to store the inputs.

2. CircleComputation

Write a program called **CircleComputation** that prompts user for the radius of a circle in floating point number. The program shall read the input as *double*; compute the diameter, circumference, and area of the circle in *double*; and print the values rounded to 2 decimal places. Use System-provided constant *Math.PI* for pi. The formulas are:

Command window

```
diameter = 2.0 * radius;  
2 area = Math.PI * radius * radius;  
circumference = 2.0 * Math.PI * radius;
```

Hints



```
1 // Declare variables  
double radius;  
3 double diameter;  
double circumference;  
5 double area; // inputs and results - all in double  
.....  
7  
// Prompt and read inputs as "double"  
9 System.out.print("Enter the radius: ");  
radius = in.nextDouble(); // read input as double  
11  
// Compute in "double"  
13 .....  
  
15 // Print results using printf() with the following format specifiers :  
// %.2f for a double with 2 decimal digits  
17 // %n for a newline  
System.out.printf("Diameter is: %.2f%n", diameter);  
19 .....
```

Try

- (a) Write a program called **SphereComputation** that prompts user for the radius of a sphere in floating point number. The program shall read the input as *double*; compute the volume and surface area of the sphere in *double*; and print the values rounded to 2 decimal places. The formulas are:

```
surfaceArea = 4 * Math.PI * radius * radius;  
volume = 4 / 3 * Math.PI * radius * radius * radius; // But this does not  
work in programming?! Why?
```

Take note that you cannot name the variable surface area with a space or surface-area with a dash. Java's naming convention is *surfaceArea*. Other languages recommend *surface_area* with an underscore.

- (b) Write a program called **CylinderComputation** that prompts user for the base radius and height of a cylinder in floating point number. The program shall read the inputs as *double*; compute the base area, surface area, and volume of the cylinder; and print the values rounded to 2 decimal places. The formulas are:

```
baseArea = Math.PI * radius * radius;  
surfaceArea = 2.0 * Math.PI * radius + 2.0 * baseArea;  
volume = baseArea * height;
```

3. PensionContributionCalculator

Both the employer and the employee are mandated to contribute a certain percentage of the employee's salary towards the employee's pension fund. The rate is tabulated as follows:

| Employee's Age | Employee Rate (%) | Employer Rate (%) |
|----------------|-------------------|-------------------|
| 55 and below | 20 | 17 |
| above 55 to 60 | 13 | 13 |
| above 60 to 65 | 7.5 | 9 |
| above 65 | 5 | 7.5 |

However, the contribution is subjected to a salary ceiling of \$6,000. In other words, if an employee earns \$6,800, only \$6,000 attracts employee's and employer's contributions, the remaining \$800 does not.

Write a program called **PensionContributionCalculator** that reads the monthly salary and age (in *int*) of an employee. Your program shall calculate the employee's, employer's and total contributions (in *double*); and print the results rounded to 2 decimal places. For examples,

Command window

```

1 Enter the monthly salary: $3000
  Enter the age: 30
3 The employee's contribution is: $600.00
  The employer's contribution is: $510.00
5 The total contribution is: $1110.00

```

Hints

```

1 // Declare constants
  final int SALARY_CEILING = 6000;
3 final double EMPLOYEE_RATE_55_AND_BELOW = 0.2;
  final double EMPLOYER_RATE_55_AND_BELOW = 0.17;
5 final double EMPLOYEE_RATE_55_TO_60 = 0.13;
  final double EMPLOYER_RATE_55_TO_60 = 0.13;
7 final double EMPLOYEE_RATE_60_TO_65 = 0.075;
  final double EMPLOYER_RATE_60_TO_65 = 0.09;
9 final double EMPLOYEE_RATE_65_ABOVE = 0.05;
  final double EMPLOYER_RATE_65_ABOVE = 0.075;
11
  // Declare variables
13 int salary ;
  int age;    // to be input
15
  int contributableSalary ;
17 double employeeContribution;
  double employerContribution;
19 double totalContribution ;
  .....
21 .....

23 // Check the contribution cap
  contributableSalary = .....
25
  // Compute various contributions in "double" using a nested-if
27 // to handle 4 cases .
  if (age <= 55) {           // 55 and below
29     .....
  } else if (age <= 60) {    // (60, 65]
31     .....
  } else if (age <= 65) {    // (55, 60]
33     .....
  } else {                  // above 65
35     .....
  }
37
  // Alternatively ,

```



```
39 // if (age > 65) .....  
    // else if (age > 60) .....  
41 // else if (age > 55) .....  
    // else .....
```

4. PensionContributionCalculatorWithSentinel

Based on the previous **PensionContributionCalculator**, write a program called **PensionContributionCalculatorWithSentinel** which shall repeat the calculations until user enter -1 for the salary. For examples,

Command window

```
Enter the monthly salary (or -1 to end): $5123  
2 Enter the age: 21  
The employee's contribution is: $1024.60  
4 The employer's contribution is: $870.91  
The total contribution is: $1895.51  
6  
Enter the monthly salary (or -1 to end): $5123  
8 Enter the age: 64  
The employee's contribution is: $384.22  
10 The employer's contribution is: $461.07  
The total contribution is: $845.30  
12  
Enter the monthly salary (or -1 to end): USD-1  
14 Bye!
```

Hints



```
// Read the first input to "seed" the while loop  
2 System.out.print("Enter the monthly salary (or -1 to end): $");  
   salary = in.nextInt();  
4  
while (salary != SENTINEL){  
6   // Read the remaining  
   System.out.print("Enter the age: ");  
8   age = in.nextInt();  
   .....  
10  
   // Read the next input and repeat  
12 System.out.print("Enter the monthly salary (or -1 to end): $");  
   salary = in.nextInt();
```



14 }

5. ReverseInt

Write a program that prompts user for a positive integer. The program shall read the input as *int*; and print the "reverse" of the input integer. For examples,

Command window

Enter a positive integer : 12345
2 The reverse is : 54321

Hints

Use the following coding pattern which uses a while-loop with repeated modulus/divide operations to extract and drop the last digit of a positive integer.



```

// Declare variables
2 int inNumber; // to be input
  int inDigit;  // each digit
4  .....
   .....
6
// Extract and drop the "last" digit repeatably using a while-loop
8 // with modulus/divide operations
while (inNumber > 0) {
10  inDigit = inNumber % 10; // extract the "last" digit
    // Print this digit (which is extracted in reverse order)
12  .....

14  inNumber /= 10;          // drop "last" digit and repeat
    }
16  .....

```

6. InputValidation

Your program often needs to validate the user's inputs, e.g., marks shall be between 0 and 100.

Write a program that prompts user for an integer between 0 – 10 or 90 – 100. The program shall read the input as *int*; and repeat until the user enters a valid input. For examples,

Command window

```

Enter a number between 0-10 or 90-100: -1
2 Invalid input, try again ...
Enter a number between 0-10 or 90-100: 50
4 Invalid input, try again ...
Enter a number between 0-10 or 90-100: 101
6 Invalid input, try again ...
Enter a number between 0-10 or 90-100: 95
8 You have entered: 95

```

Hints

Use the following coding pattern which uses a do-while loop controlled by a *boolean* flag to do input validation. We use a do-while instead of while-do loop as we need to execute the body to prompt and process the input at least once.



```

// Declare variables
2 int numberIn;    // to be input
  boolean isValid; // boolean flag to control the loop
4 .....
  .....
6
  // Use a do-while loop controlled by a boolean flag
8 // to repeatedly read the input until a valid input is entered.
  isValid = false; // default assuming input is not valid
10 do {
    // Prompt and read input
12 .....
    .....
14
    // Validate input by setting the boolean flag accordingly
16 if (numberIn ..... ) {
    isValid = true; // exit the loop
18 } else {
    System.out.println (.....) ; // Print error message and repeat
20 }
  } while (! isValid);
22
  .....

```

7. AverageWithInputValidation

Write a program that prompts user for the mark (between 0 – 100 in *int*) of 3 students; computes the average (in *double*); and prints the result rounded to 2 decimal places. Your program needs to perform input validation. For examples,

Command window

```

1 Enter the mark (0-100) for student 1: 56
Enter the mark (0-100) for student 2: 101
3 Invalid input, try again ...
Enter the mark (0-100) for student 2: -1
5 Invalid input, try again ...
Enter the mark (0-100) for student 2: 99
7 Enter the mark (0-100) for student 3: 45
The average is : 66.67

```

Hints



```

// Declare constant
2 final int NUM_STUDENTS = 3;

4 // Declare variables
int numberIn;
6 boolean isValid; // boolean flag to control the input validation loop
int sum = 0;
8 double average;
.....

10 for (int studentNo = 1; studentNo <= NUM_STUDENTS; ++studentNo) {
12 // Prompt user for mark with input validation
.....

14     isValid = false; // reset assuming input is not valid
16     do {
.....
18         } while (! isValid);

20     sum += .....;
22 }

24 .....

```

2.6.8 Input from Text File via "Scanner" (JDK 5)

Other than scanning `System.in` (keyboard), you can connect your Scanner to scan any input sources, 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,



```
// Construct a Scanner to scan a text file
2 Scanner in = new Scanner(new File("in.txt"));

4 // Use the same set of methods to read from the file
int anInt = in.nextInt(); // next String
6 double aDouble = in.nextDouble(); // next double
String str = in.next(); // next int
8 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*.



```
import java.util.Scanner; // Needed for using Scanner
2 import java.io.File; // Needed for file operation
import java.io.FileNotFoundException; // Needed for file operation

4
/**
6 * TextFileScannerWithThrows.java
* Input from File .
8 * Technique 1: Declare "throws FileNotFoundException" in the enclosing main() method
*/
10 public class TextFileScannerWithThrows {
// Declare "throws" here
12 public static void main(String[] args) throws FileNotFoundException {
Scanner in = new Scanner(new File("in.txt")); // Scan input from text file
14 int num1 = in.nextInt(); // Read int
double num2 = in.nextDouble(); // Read double
16 String name = in.next(); // Read String
System.out.printf("Hi %s, the sum of %d and %.2f is %.2f%n",
18 name, num1, num2, num1 + num2);
in.close();
20 }
}
```

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

```
1234
55.66
Paul
```



```

1 import java.util.Scanner;           // Needed for using Scanner
import java.io.File;                 // Needed for file operation
3 import java.io.FileNotFoundException; // Needed for file operation

5 /**
 * TextFileScannerWithCatch.java
7 * Input from File .
 * Technique 2: Use try-catch to handle exception
9 */
public class TextFileScannerWithCatch {
11     public static void main(String[] args) {
        try {                          // try these statements
13         Scanner in = new Scanner(new File("in.txt"));
            int num1 = in.nextInt();    // Read int
15         double num2 = in.nextDouble(); // Read double
            String name = in.next();    // Read String
17         System.out.printf("Hi %s, the sum of %d and %.2f is %.2f%n",
            name, num1, num2, num1 + num2);
19         in.close();
        } catch (FileNotFoundException ex) { // catch and handle the exception here
21         ex.printStackTrace();          // print the stack trace
        }
23     }
}

```

2.6.9 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**.



```

import java.io.File;
2 import java.util.Formatter;         // <== note
import java.io.FileNotFoundException; // <== note

4
/**
6 * TextFileFormatterWithThrows.java
 * Output to File .
8 * Technique 1: Declare "throws FileNotFoundException" in the enclosing main() method
 */
10 public class TextFileFormatterWithThrows {
    public static void main(String[] args) throws FileNotFoundException {

```



```

12 // Construct a Formatter to write formatted output to a text file
    Formatter out = new Formatter(new File("out.txt"));

14

16 // Write to file with format() method (similar to printf())
    int num1 = 1234;
    double num2 = 55.66;
18    String name = "Paul";
    out.format("Hi %s,%n", name);
20    out.format("The sum of %d and %.2f is %.2f%n", num1, num2, num1 + num2);
    out.close(); // Close the file
22    System.out.println("Done"); // Print to console
    }
24 }

```

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



```

import java.io.File;
2 import java.util.Formatter; // <== note
import java.io.FileNotFoundException; // <== note

4

6 /**
 * TextFileFormatterWithCatch.java
 * Output to File .
8 * Technique 2: Use try-catch to handle exception
 */
10 public class TextFileFormatterWithCatch {
    public static void main(String[] args) {
12        try { // try the following statements
            // Construct a Formatter to write formatted output to a text file
14            Formatter out = new Formatter(new File("out.txt"));

16            // Write to file with format() method (similar to printf())
            int num1 = 1234;
18            double num2 = 55.66;
            String name = "Pauline";
20            out.format("Hi %s,%n", name);
            out.format("The sum of %d and %.2f is %.2f%n", num1, num2, num1 + num2);
22            out.close(); // Close the file
            System.out.println("Done"); // Print to console
24        } catch (FileNotFoundException ex) { // catch the exception here
            ex.printStackTrace(); // Print the stack trace
26        }
    }
28 }

```


2.7 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 MUST 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 code.
- 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 like *x*, *y*, *z* for coordinates and *i* for index.
- Provide comments to explain the important as well as salient concepts. Comment your code liberally.
- Write your program documentation while writing your programs. Avoid unstructured constructs, such as `break` and `continue`, which are hard to follow.
- Use "mono-space" fonts (such as Consolas, 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!!!

2.7.1 Programming Errors: Compilation, Runtime and Logical Errors

There are generally three classes of programming errors:

1. **Compilation Error (or Syntax Error):** The program cannot compile. This can be fixed easily by checking the compilation error messages. For examples,



```
Sys.out.print("Hello"); // System instead of Sys
```

Command window



```
error: package Sys does not exist
2 Sys.out.print("Hello");
  ^
```



```
1 System.out.print("Hello") // Missing semi-colon
```

Command window



```
error: ';' expected
2 System.out.print("Hello")
  ^
```

2. **Runtime Error:** The program can compile, but fail to run successfully. This can also be fixed easily, by checking the runtime error messages. For examples,



```
1 int count = 0, sum = 100, average;
  average = sum / count; // Divide by 0. Runtime error
```

Command window



```
1 Exception in thread "main" java.lang.ArithmeticException: / by zero
```




```
// Wrong conversion code, %d for integer
2 System.out.printf("The sum is: %d%n", 1.23);
```

Command window

```
1 Exception in thread "main" java.util.IllegalFormatConversionException: d !=  
    java.lang.Double
```

3. **Logical Error:** The program can compile and run, but produces incorrect results (always or sometimes). This is the hardest error to fix as there is no error messages - you have to rely on checking the output. It is easy to detect if the program always produces wrong output. 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 run, but give wrong result - sometimes!  
2 if (mark > 50) {  
    System.out.println ("PASS");  
4 } else {  
    System.out.println ("FAIL");  
6 }
```

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.

2.7.2 Debugging Programs

Here are the common debugging techniques:

1. Stare at the screen! Unfortunately, nothing will 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. Perform program testing to wipe out the logical errors. "Write test first, before writing program".

2.7.3 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 for adding two integers (because there are too many combinations of two integers). Program testing usually involves a set of representative test cases, which are designed to catch all classes of errors. Program testing is beyond the scope of this writing.

2.7.4 Exercises on Debugging/Tracing Programs using a Graphic Debugger

1. Factorial (Using a graphic debugger)

The following program computes and prints the factorial of n ($= 1 * 2 * 3 * \dots * n$). The program, however, has a logical error and produce a wrong answer for $n = 20$ ("The Factorial of 20 is -2102132736" – negative?!).

Use the graphic debugger of Eclipse/NetBeans/IntelliJ IDEA to debug the program by single-step through the program and tabulating the values of i and factorial at the statement marked by (*).

You should try out debugging features such as "Breakpoint", "Step Over", "Watch variables", "Run-to-Line", "Resume", "Terminate", among others.



```

1  /**
   * Factorial . java
3  * Print factorial of n.
   */
5  public class Factorial {
   public static void main(String[] args) { // Set an initial breakpoint at
        ↪ this statement
7      int n = 20;
        int factorial = 1;
9
        // n! = 1*2*3...* n
11     for (int i = 1; i <= n; i++) { // i = 1, 2, 3, ..., n
            factorial = factorial * i; // *
13     }
        System.out.println("The Factorial of " + n + " is " + factorial );
15 }
   }

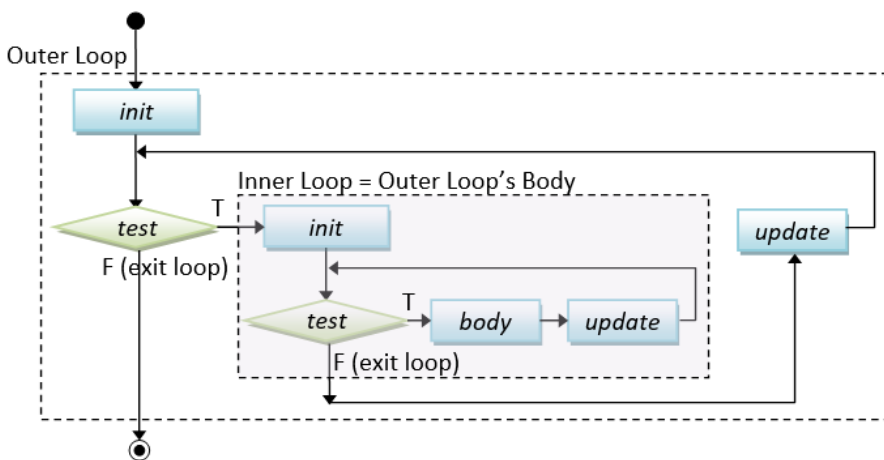
```

2.8 More on Loops - Nested-Loops, break & continue

2.8.1 Nested Loops

Nested loops are needed to process 2-dimensional (or N-dimensional) data, such as printing 2D patterns. A nested-for-loop takes the following form:

nested-for-loop



Syntax



```

1  for (...; ...; ...) {    // outer loop
2  // Before running the inner loop
3  .....
4  for (...; ...; ...) {    // inner loop
5  .....
6  }
7
8  // After running the inner loop
9  .....
10 }
```

2.8.2 Code Examples: Print Square Pattern

The following program prompts user for the size of the pattern, and prints a square pattern using nested-loops. For example,

Command window

Enter the size : 6

```

2 * * * * *
  * * * * *
4 * * * * *
  * * * * *
6 * * * * *
  * * * * *

```



```

1 import java . util .Scanner;

3 /**
   * PrintSquarePattern .java
   * Prompt user for the size ; and print Square pattern
   */
7 public class PrintSquarePattern {
   public static void main (String [] args) {
9       // Declare variables
       final int SIZE;    // size of the pattern to be input

11
       // Prompt user for the size and read input as "int"
13       Scanner in = new Scanner(System.in);

15       System.out.print ("Enter the size : ");
       SIZE = in .nextInt () ;
17       in .close () ;

19       // Use nested-loop to print a 2D pattern
       // Outer loop to print ALL the rows
21       for (int row = 1; row <= SIZE; row++) {
           // Inner loop to print ALL the columns of EACH row
23           for (int col = 1; col <= SIZE; col++) {
               System.out.print ("* ");
25           }

27           // Print a newline after all the columns
               System.out.println () ;
29       }
31 }

```

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

Coding Pattern: Print 2D Patterns

The *coding pattern* for printing 2D patterns is as follows. I recommend using row and col as the loop variables which is self-explanatory, instead of i and j, x and y.



```

1 for (int row = 1; row <= ROW_SIZE; row++) { // outer loop for rows
    ..... // before each row
3   for (int col = 1; col <= COL_SIZE; col++) { // inner loop for columns
       if (.....) {
5         System.out.print (.....) ; // without newline
       } else {
7         System.out.print (.....) ;
       }
9   }
    ..... // after each row
11  System.out.println (); // Print a newline after all the columns
    }

```

2.8.3 Code Examples: Print Checker Board Pattern

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

Command window

Enter the size : 6

2

* * * * *

* * * * *

4

* * * * *

* * * * *

6

* * * * *

* * * * *

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



```

1 if ((row % 2) == 0) { // print a leading space for even-numbered rows
    System.out.print(" ");
3 }

```

2.8.4 Code Example: Print Multiplication Table

The following program prompts user for the size, and print the multiplication table as follows:

Command window

```

1 Enter the size : 10
  * | 1 2 3 4 5 6 7 8 9 10
3 -----
  1 | 1 2 3 4 5 6 7 8 9 10
  2 | 2 4 6 8 10 12 14 16 18 20
  3 | 3 6 9 12 15 18 21 24 27 30
  4 | 4 8 12 16 20 24 28 32 36 40
  5 | 5 10 15 20 25 30 35 40 45 50
  6 | 6 12 18 24 30 36 42 48 54 60
  7 | 7 14 21 28 35 42 49 56 63 70
  8 | 8 16 24 32 40 48 56 64 72 80
  9 | 9 18 27 36 45 54 63 72 81 90
 10 | 10 20 30 40 50 60 70 80 90 100

```



```

1 import java.util.Scanner;

3 /**
  * PrintTimeTable.java
  * Prompt user for the size and print the multiplication table.
  */
7 public class PrintTimeTable {
  public static void main(String[] args) {
9      // Prompt for size and read input as "int"
      Scanner in = new Scanner(System.in);
11     System.out.print("Enter the size: ");
      final int SIZE = in.nextInt();
13     in.close();

15     // Print header row
      System.out.print(" * |");
17     for (int col = 1; col <= SIZE; ++col) {
        System.out.printf("%4d", col);
19     }
      System.out.println(); // End row with newline

21     // Print separator row
23     System.out.print("----");
      for (int col = 1; col <= SIZE; ++col) {
25         System.out.printf("%4s", "----");
      }
27     System.out.println(); // End row with newline

```




```

29 // Print body using nested-loops
30 for (int row = 1; row <= SIZE; ++row) { // outer loop
31     System.out.printf("%2d |", row); // print row header first
32     for (int col = 1; col <= SIZE; ++col) { // inner loop
33         System.out.printf("%4d", row*col);
34     }
35     System.out.println(); // print newline after all columns
36 }
37 }

```

2.8.5 Exercises on Nested Loops with Input

1. SquarePattern

Write a program called **SquarePattern** that prompts user for the *size* (a non-negative integer in *int*); and prints the following square pattern using two nested for-loops.

Command window
▼

```

Enter the size : 5
2  #####
   #####
4  #####
   #####
6  #####

```

Hints

The code pattern for printing 2D patterns using nested loops is:



```

// Outer loop to print each of the rows
2 for (int row = 1; row <= size; row++){ // row = 1, 2, 3, ..., size
// Inner loop to print each of the columns of a particular row
4 for (int col = 1; col <= size; col++) { // col = 1, 2, 3, ..., size
    System.out.print( ..... ); // Use print() without newline inside the
        ↪ inner loop
6     .....
    }
8 // Print a newline after printing all the columns
    System.out.println();
10 }

```

Notes

- (a) You should name the loop indexes *row* and *col*, NOT *i* and *j*, or *x* and *y*, or *a* and *b*, which are meaningless.
- (b) The *row* and *col* could start at 1 (and upto *size*), or start at 0 (and upto *size* - 1). As computer counts from 0, it is probably more efficient to start from 0. However, since humans counts from 1, it is easier to read if you start from 1.

Try

- (a) Rewrite the above program using nested while-do loops.

2. CheckerPattern

Write a program called **CheckerPattern** that prompts user for the size (a non-negative integer in *int*); and prints the following checkerboard pattern.

Command window
▼

```

Enter the size : 7
2 # # # # # # #
   # # # # # # #
4 # # # # # # #
   # # # # # # #
6 # # # # # # #
   # # # # # # #
8 # # # # # # #

```

Hints

```

// Outer loop to print each of the rows
2 for (int row = 1; row <= size; row++) { // row = 1, 2, 3, ..., size
    // Inner loop to print each of the columns of a particular row
    4 for (int col = 1; col <= size; col++) { // col = 1, 2, 3, ..., size
        if ((row % 2) == 0) { // row 2, 4, 6, ...
            6 .....
        }
        8 System.out.print( ..... ); // Use print() without newline inside the
            ↳ inner loop
            .....
    10 }
    // Print a newline after printing all the columns
    12 System.out.println();
}

```

3. TimeTable

Write a program called **TimeTable** that prompts user for the *size* (a positive integer in *int*); and prints the multiplication table as shown:

Command window

```
1 Enter the size : 10
2 * | 1 2 3 4 5 6 7 8 9 10
3 -----
4 1 | 1 2 3 4 5 6 7 8 9 10
5 2 | 2 4 6 8 10 12 14 16 18 20
6 3 | 3 6 9 12 15 18 21 24 27 30
7 4 | 4 8 12 16 20 24 28 32 36 40
8 5 | 5 10 15 20 25 30 35 40 45 50
9 6 | 6 12 18 24 30 36 42 48 54 60
10 7 | 7 14 21 28 35 42 49 56 63 70
11 8 | 8 16 24 32 40 48 56 64 72 80
12 9 | 9 18 27 36 45 54 63 72 81 90
13 10 | 10 20 30 40 50 60 70 80 90 100
```

Hints

- (a) Use *printf()* to format the output, e.g., each cell is %4d.

4. TriangularPattern

Write 4 programs called **TriangularPatternX** (X = A, B, C, D) that prompts user for the *size* (a non-negative integer in *int*); and prints each of the patterns as shown:

Command window

```
1 Enter the size : 8
2
3 #           # # # # # # # # # # # # # # # #
4 # #         # # # # # # # # # # # # # # # #
5 # # #       # # # # # # # # # # # # # # # #
6 # # # #     # # # # # # # # # # # # # # # #
7 # # # # #   # # # # # # # # # # # # # # # #
8 # # # # # # # # # # # # # # # # # # # # # #
9 # # # # # # # # # # # # # # # # # # # # # #
10 # # # # # # # # # # # # # # # # # # # # # #
11 (a)          (b)          (c)          (d)
```

Hints

- (a) On the main diagonal, $row = col$. On the opposite diagonal, $row + col = size + 1$, where row and col begin from 1.

- (b) You need to print the leading blanks, in order to push the # to the right. The trailing blanks are optional, which does not affect the pattern.
- (c) For pattern (a), if ($row \geq col$) print #. Trailing blanks are optional.
- (d) For pattern (b), if ($row + col \leq size + 1$) print #. Trailing blanks are optional.
- (e) For pattern (c), if ($row \geq col$) print #; else print blank. Need to print the leading blanks.
- (f) For pattern (d), if ($row + col \geq size + 1$) print #; else print blank. Need to print the leading blanks.
- (g) The coding pattern is:



```

1 // Outer loop to print each of the rows
  for (int row = 1; row <= size; row++) { // row = 1, 2, 3, ..., size
3 // Inner loop to print each of the columns of a particular row
    for (int col = 1; col <= size; col++) { // col = 1, 2, 3, ..., size
5     if (.....) {
        System.out.print("# ");
7     } else {
        System.out.print(" "); // Need to print the "leading" blanks
9     }
    }
11 // Print a newline after printing all the columns
    System.out.println();
13 }

```

5. BoxPattern

Write 4 programs called **BoxPatternX** (X = A, B, C, D) that prompts user for the *size* (a non-negative integer in *int*); and prints the pattern as shown:

Command window
▼

Enter the size : 8

```

2
# # # # # # # # # # # # # # # # # # # # # # # # # # # #
4
#           #           #           #           #
#           #           #           #           #
6
#           #           #           #           #
#           #           #           #           #
8
#           #           #           #           #
# # # # # # # # # # # # # # # # # # # # # # # # # # # #
10
(a)           (b)           (c)           (d)

```

Hints

- (a) On the main diagonal, $row = col$. On the opposite diagonal, $row + col = size + 1$, where row and col begin from 1.
- (b) For pattern (a), if $(row == 1 \ || \ row == size \ || \ col == 1 \ || \ col == size)$ print #; else print blank. Need to print the intermediate blanks.
- (c) For pattern (b), if $(row == 1 \ || \ row == size \ || \ row == col)$ print #; else print blank.

6. HillPattern

Write 3 programs called **HillPatternX** ($X = A, B, C, D$) that prompts user for the *size* (a non-negative integer in *int*); and prints the pattern as shown:

```

Command window
Enter the rows: 5
2
# # # # # # # # # # # # # # # #
4
# # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # # #
6
# # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # # #
8
(a) (b) # # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # # #
10
# # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # # #
12
(c) (d)

```

Hints

- (a) For pattern (a):



```

for (int row = 1; ..... ) {
2 // numCol = 2*numRows - 1
  for (int col = 1; ..... ) {
4    if ((row + col >= numRows + 1) && (row >= col - numRows + 1)) {
        .....;
6    } else {
        .....;
8    }
    }
10 .....;
  }
}

```

or, use 2 sequential inner loops to print the columns:



```

for (int row = 1; row <= rows; row++) {
2   for (int col = 1; col <= rows; col++) {
        if ((row + col >= rows + 1)) {
4           .....
        } else {
6           .....
        }
8   }

10  for (int col = 2; col <= rows; col++) { // skip col = 1
        if (row >= col) {
12           .....
        } else {
14           .....
        }
16  }

18  .....
}

```

2.8.6 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.

Endless 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 continuously (infinite loop). You need to use a break statement to break out the loop.

Similar, while (true) *body* and do *body* while (true) are endless loops.



```

for ( ; ; ) {
2   ..... // Need break inside the loop body
}

4   while (true) {

```



```

6 .....    // Need break inside the loop body
7 }
8
9 do {
10 .....    // Need break inside the loop body
11 } while (true);

```

Endless loop is typically a mistake especially for new programmers. You need to break out the loop via a break statement inside the loop body..

Example (break)

The following program lists the non-prime numbers between 2 and an upperbound.



```

1 /**
2  * NonPrimeList.java
3  * List all non-prime numbers between 2 and an upperbound
4  */
5 public class NonPrimeList {
6     public static void main(String[] args) {
7         final int UPPERBOUND = 100;
8         for (int number = 2; number <= UPPERBOUND; ++number) {
9             // Not a prime, if there is a factor between 2 and sqrt(number)
10            int maxFactor = (int) Math.sqrt(number);
11            for (int factor = 2; factor <= maxFactor; ++factor) {
12                if (number % factor == 0) { // Factor?
13                    System.out.println(number + " is NOT a prime");
14                    break; // A factor found, no need to search for more factors
15                }
16            }
17        }
18    }
19 }

```

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.



```

1 /**
2  * PrimeListWithBreak.java
3  * List all prime numbers between 2 and an upperbound
4  */

```



```

5 public class PrimeListWithBreak {
    public static void main(String[] args) {
6         final int UPPERBOUND = 100;
        for (int number = 2; number <= UPPERBOUND; ++number) {
9             // Not a prime, if there is a factor between 2 and sqrt(number)
            int maxFactor = (int) Math.sqrt(number);
11            boolean isPrime = true; // boolean flag to indicate whether number is a prime
            for (int factor = 2; factor <= maxFactor; ++factor) {
13                if (number % factor == 0) { // Factor?
                    isPrime = false; // number is not a prime
15                    break; // A factor found, no need to search for more factors
                }
17            }

19            if (isPrime) {
                System.out.println(number + " is a prime");
21            }
        }
23    }
}

```

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*.



```

1  /**
2   * PrimeList.java
3   * List all prime numbers between 2 and an upperbound
4   */
    public class PrimeList {
6        public static void main(String[] args) {
            final int UPPERBOUND = 100;
8            for (int number = 2; number <= UPPERBOUND; ++number) {
                // Not prime, if there is a factor between 2 and sqrt of number
10                int maxFactor = (int) Math.sqrt(number);
                boolean isPrime = true;
12                int factor = 2;
                while (isPrime && factor <= maxFactor) {
14                    if (number % factor == 0) { // Factor of number?
                        isPrime = false;
16                    }

18                    ++factor;
                }

20                if (isPrime) {
22                    System.out.println(number + " is a prime");

```




```

    }
24 }
    }
26 }
```

Example (continue)



```

/**
2  * SumDivisibleEleven.java
  * Sum 1 to upperbound, exclude 11, 22, 33,...
4  */
public class SumDivisibleEleven {
6  public static void main(String[] args) {
    final int UPPERBOUND = 100;
8  int sum = 0;
    for (int number = 1; number <= UPPERBOUND; ++number) {
10     if (number % 11 == 0) {
        continue; // Skip the rest of the loop body, continue to the next iteration
12     }
        sum += number;
14     }

16     // It is better to re-write the loop as:
    for (int number = 1; number <= UPPERBOUND; ++number) {
18         if (number % 11 != 0) {
            sum += number;
20         }
        }
22     }
}
```

Example (break and continue)



```

1 /**
  * MysterySeries.java
3  * A mystery series created using break and continue
  */
5 public class MysterySeries {
    public static void main(String[] args) {
7        int number = 1;
        while(true) {
```



```

9      ++number;
      if ((number % 3) == 0) {
11         continue;
      }

13
      if (number == 133) {
15         break;
      }

17
      if ((number % 2) == 0) {
19         number += 3;
      } else {
21         number -= 3;
      }
23     System.out.print(number + " ");
    }
25 }
27 // Can you figure out the output?
    // break and continue are hard to read, use it with great care!

```

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 clumsy 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
2  for (.....) {
    level2 :       // define a label for the level-2 loop
4  for (.....) {
    for (.....) {  // level-3 loop
6      if (...) {
          break level1 ; // break all loops, continue after the loop
8      }

10     if (...) {
          break level2 : // continue into the next statement of level-1 loop
12     }
        .....
14     }
    }
16 }

```

Labeled continue

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



```

1  level1 :      // define a label (with : suffix ) for the level-1 loop
2  for (.....) {
3      level2 :  // define a label (with : suffix ) for the level-2 loop
4      for (.....) {
5          for (.....) {          // level-3 loop
6              if (...) {
7                  continue level1 ; // continue the next iteration of level-1 loop
8              }
9
10             if (...) {
11                 continue level2 : // continue the next iteration of level-2 loop
12             }
13             .....
14         }
15     }
16 }
```

Again, labeled break and continue are not structured 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  /**
2   * TestLabeledBreak.java
3   */
4  public class TestLabeledBreak {
5      public static void main(String[] args) {
6          int [][] testArray = {
7              {1, 2, 3, 4},
8              {4, 3, 1, 4},
9              {9, 2, 3, 4}
10         };

11
12         final int MAGIC_NUMBER = 8;
13         boolean found = false ;
14         mainLoop:
15         for (int i = 0; i < testArray.length; ++i) {
16             for (int j = 0; j < testArray[i].length; ++j) {
17                 if (testArray[i][j] == MAGIC_NUMBER){
18                     break mainLoop;
19                 }
20             }
21         }
22     }
23 }
```



```

18         found = true;
           break mainLoop;
20     }
22 }
    System.out.println("Magic number " + (found ? "found" : "NOT found"));
24 }
    }

```

2.9 String and char operations

2.9.1 char Arithmetic Operations

In Java, each char is represented by a 16-bit Unicode number. For examples, *char* '0' is represented by code number 48 (30H), *char* '1' by 49 (31H), *char* 'A' by 65 (41H). *char* 'a' by 97 (61H). Take note that *char* '0' is NOT *int* 0, *char* '1' is NOT *int* 1.

Chars can take part in arithmetic operations. A *char* is treated as its underlying *int* (in the range of [0, 65535]) in arithmetic operations. In other words, *char* and *int* are interchangeable. *char* '0' \Leftrightarrow *int* 48, *char* '1' \Leftrightarrow *int* 49, *char* 'A' \Leftrightarrow *int* 65, *char* 'a' \Leftrightarrow *int* 97. For examples,



```

1 char c1 = '0';      // Code number 48
  char c2 = 'A';      // Code number 65
3 char c3;

5 // char <-> int (interchangeable)
  System.out.println((int)c1); // Print int 48
7 System.out.println((int)c2); // Print int 65
  c3 = 97;             // Code number for 'a'
9 System.out.println(c3);    // Print char 'a'
  System.out.println((char)97); // Print char 'a'

```

In arithmetic operations, *char* (and *byte*, and *short*) is first converted to *int*. In Java, arithmetic operations are only carried out in *int*, *long*, *float*, or *double*; NOT in *byte*, *short*, and *char*.

Hence, *char* \oplus *char* \Rightarrow *int* \oplus *int* \Rightarrow *int*, where \oplus denotes an binary arithmetic operation (such as +, -, *, / and %). You may need to explicitly cast the resultant *int* back to *char*. For examples,



```

1 char c1 = '0';      // Code number 48
  char c2 = 'A';      // Code number 65
3 char c3;

5 // char + char -> int + int -> int
  // c3 = c1 + c2; // error: RHS evaluated to "int", cannot assign to LHS of "char"
7 c3 = (char)(c1 + c2); // Need explicit type casting
                        // return char 'q' (code number 113)
9 System.out.println(c3); // Print 'q', as c3 is a char
  System.out.println(c1 + c2); // Print int 113
11 System.out.println((char)(c1 + c2)); // Print char 'q'

```

Similar, $\text{char} \oplus \text{int} \Rightarrow \text{int} \oplus \text{int} \Rightarrow \text{int}$. You may need to explicitly cast the resultant `int` back to `char`. For examples,



```

char c1 = '0';      // Code number 48
2 char c2 = 'A';      // Code number 65
  char c3;

4 // char + int -> int + int -> int
6 // c3 = c1 + 5; // error: RHS evaluated to "int", cannot assign to LHS of "char"
  c3 = (char)(c1 + 5); // Need explicit type casting, return char '5' (code number 53)
8 System.out.println(c3); // Print '5', as c3 is a char
  System.out.println(c1 + 5); // Print int 53

10 // Print the code number for 'a' to 'z'
12 for (int codeNum = 'a'; codeNum <= 'z'; ++codeNum) {
    System.out.println((char)codeNum + ": " + codeNum);
14 }

```

However, for compound operators (such as `+=`, `-=`, `*=`, `/=`, `%=`), the evaluation is carried out in `int`, but the result is casted back to the LHS automatically. For examples,



```

1 char c4 = '0';      // Code number 48
  c4 += 5;             // Automatically cast back to char '5'
3 System.out.println(c4); // Print char '5'

```

For increment (`++`) and decrement (`--`) of `char` (and `byte`, and `short`), there is no promotion to `int`. For examples,



```
// Print char '0' to '9' via increment
2 for (char c = '0'; c <= '9'; ++c) { // ++c remains as "char"
    System.out.println(c);
4 }
```

2.9.2 Converting *char* to *int*

You can convert *char* '0' to '9' to *int* 0 to 9 by subtracting the *char* with the base '0', e.g., '8' - '0' \Rightarrow 8. That is, suppose *c* is a *char* between '0' and '9', (*c* - '0') is the corresponding *int* 0 to 9.

The following program illustrates how to convert a hexadecimal character (0-9, A-F or a-f) to its decimal equivalent (0-15), by subtracting the appropriate base *char*.



```
// Converting a hex char (0-9|A-F|a-f) to its equivalent decimal (0-15)
2 char hexChar = 'a';
   int dec;
4
   if (hexChar >= '0' && hexChar <= '9') {
6       dec = hexChar - '0'; // int 0-9
   } else if (hexChar >= 'A' && hexChar <= 'F') {
8       dec = hexChar - 'A' + 10; // int 10-15
   } else if (hexChar >= 'a' && hexChar <= 'f') {
10      dec = hexChar - 'a' + 10; // int 10-15
   } else {
12      dec = -1; // to overcome variable have not been initialized error
        System.out.println("Invalid hex char");
14 }
   System.out.println(hexChar + ": " + dec);
```

2.9.3 String Operations

The most commonly-used *String* methods are as follows, suppose that *str*, *str1*, *str2* are *String* variables:

- *str.length()*: return the length of the *str*.
- *str.charAt(intindex)*: return the char at the index position of the *str*. Take note that index begins at 0, and up to *str.length()*-1.
- *str1.equals(str2)*: for comparing the contents of *str1* and *str2*. Take note that you cannot use "*str1* == *str2*" to compare two *Strings*. This is because "==" is only applicable to primitive types, but *String* is not a primitive type.

For examples,



```

1 String str = "Java is cool!";
  System.out.println (str.length()); // return int 13
3 System.out.println (str.charAt(2)); // return char 'v'
  System.out.println (str.charAt(5)); // return char 'i'
5
  // Comparing two Strings
7 String anotherStr = "Java is COOL!";
  System.out.println (str.equals(anotherStr)); // return boolean false
9 System.out.println (str.equalsIgnoreCase(anotherStr)); // return boolean true
  System.out.println (anotherStr.equals(str)); // return boolean false
11 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 Documentation ⇒ Select "API documentation" ⇒ Click "FRAMES" (top menu) ⇒ From "Modules" (top-left pane), select "java.base" ⇒ From "java.base Packages" (top-left pane), select "java.lang" ⇒ From "Classes" (bottom-left pane), select "String" ⇒ choose "SUMMARY" "METHOD" (right pane) (@ <https://docs.oracle.com/javase/17/docs/api/java/lang/String.html> for JDK 17). For examples,



```

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

```

2.9.4 Converting String to Primitive

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

Similarly, 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";
2 float aFloat = Float.parseFloat(inStr); // aFloat ← 55.66f
  double aDouble = Double.parseDouble("1.2345"); // aDouble ← 1.2345
4 aDouble = Double.parseDouble("1.2e-3"); // aDouble ← 0.0012
  // Input to Double.parseDouble() must be a valid double literal
6 // aDouble = Double.parseDouble("abc"); // Runtime Error: NumberFormatException
```

String to char

You can use *aStr.charAt(index)* to extract individual character from a *String*, where index begins at 0 and up to *aStr.length()* - 1, e.g.,



```
// Extract each char
2 String msg = "Hello, world";
  char msgChar;
4 for (int idx = 0; idx < msg.length(); ++idx) {
    msgChar = msg.charAt(idx);
6    // Do something about the extracted char
    .....
8 }
```

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";
2 boolean done = Boolean.parseBoolean(boolStr); // done ← true
boolean valid = Boolean.parseBoolean(" false "); // valid ← false
```

2.9.5 Converting Primitive to String

To convert a primitive to a *String*, you can:

1. Use the '+' operator to concatenate the primitive with an empty *String* "".
2. Use the JDK built-in methods **String.valueOf(aPrimitive)**, which is applicable to all primitives.
3. Use the *toString()* methods of the respective wrapper class, such as **Integer.toString(anInt)**, **Double.toString(aDouble)**, **Character.toString(aChar)**, **Boolean.toString(aBoolean)**, etc.

For examples,



```
1 // Using String concatenation operator '+' with an empty String
  // (applicable to ALL primitive types)
3 String str1 = 123 + ""; // int 123 → String "123"
  String str2 = 12.34 + ""; // double 12.34 → String "12.34"
5 String str3 = 'c' + ""; // char 'c' → String "c"
  String str4 = true + ""; // boolean true → String "true"
7
  // Using String.valueOf(aPrimitive) (applicable to ALL primitive types)
9 String str5 = String.valueOf(12345); // int 12345 → String "12345"
  String str6 = String.valueOf(true); // boolean true → String "true"
11 String str7 = String.valueOf(55.66); // double 55.66 → String "55.66"

13 // Using toString() for each primitive type
  String str8 = Integer.toString(1234); // int 1234 → String "1234"
15 String str9 = Double.toString(1.23); // double 1.23 → String "1.23"
  String str10 = Character.toString('z'); // char 'z' → String "z"
```

2.9.6 Formatting Strings - String.format()

You can use *printf()* to create a formatted string and send it to the display console, e.g.,



```
// Send the formatted String to console
2 System.out.printf("Hi, %d, %.1f%n", 11, 22.22);
```

There is a similar function called **String.format()** which returns the formatted string, instead of sending to the console, e.g.,



```
// Returns a String "1.2" (for further operations)
2 String str = String.format("%.1f", 1.234);
```

String.format() has the same form as *printf()*.

2.9.7 Code Example: Reverse String

The following program prompts user a *String*, and prints the input *String* in the reverse order. For examples,

Command window

```
Enter a String : abcdefg
2 The reverse is : gfedcba
```



```
import java.util.Scanner;

2
/**
4  * ReverseString.java
  * Prompt user for a string; and print the input string in reverse order.
6  */
public class ReverseString {
8  public static void main(String[] args) {
    // Prompt and read input as "String"
10  Scanner in = new Scanner(System.in);
    System.out.print("Enter a String: ");
12  String inStr = in.next(); // input String
    int inStrLen = inStr.length(); // length of the input String
14  in.close();

16  System.out.print("The reverse is: ");
```



```
// Use a for-loop to extract each char in reverse order
18 for (int inCharIdx = inStrLen - 1; inCharIdx >= 0; --inCharIdx) {
    System.out.print(inStr.charAt(inCharIdx));
20 }

22 System.out.println();
    }
24 }
```

2.9.8 Code Example: Validating Binary String

The following program prompts user for a string, and checks if the input is a valid binary string, consisting of '0' and '1' only. For example,

Command window

```
Enter a binary string : 1011000
2 "1011000" is a binary string

4 Enter a binary string : 10001900
  "10001900" is NOT a binary string
```

Version 1: With a boolean flag



```
1 import java.util.Scanner;

3 /**
   * ValidateBinString.java
   * Check if the input string is a valid binary string.
   */
7 public class ValidateBinString {
    public static void main(String[] args) {
9        // Prompt and read input as "String"
        Scanner in = new Scanner(System.in);
11       System.out.print("Enter a binary string : ");
        String inStr = in.next(); // The input string
13       int inStrLen = inStr.length(); // The length of the input string
        in.close();

15       boolean isValid; // "is" or "is not" a valid binary string?
17       isValid = true; // Assume that the input is valid, unless our check fails
        for (int inCharIdx = 0; inCharIdx < inStrLen; ++inCharIdx) {
```



```

19     inChar = inStr.charAt(inCharIdx);
    if (!(inChar == '0' || inChar == '1')) {
21         isValid = false;
        break; // break the loop upon first error, no need to continue for more errors
23         // If this is not encountered, isValid remains true after the loop.
    }
25 }

27 System.out.println("\"" + inStr + "\" is " + (isValid ?
    "" : "NOT") + " a binary string");
29 }
}

```

Version 2



```

import java.util.Scanner;

2
/**
4  * ValidateBinStringV2.java
  * Check if the input string is a valid binary string.
6  */
public class ValidateBinStringV2 {
8     public static void main(String[] args) {
        // Prompt and read input as "String"
10     Scanner in = new Scanner(System.in);
        System.out.print("Enter a binary string: ");
12     String inStr = in.next(); // The input string
        int inStrLen = inStr.length(); // The length of the input string
14     in.close();

16     char inChar; // Each char of the input string
        for (int inCharIdx = 0; inCharIdx < inStrLen; ++inCharIdx) {
18         inChar = inStr.charAt(inCharIdx);
            if (!(inChar == '0' || inChar == '1')) {
20             System.out.println("\"" + inStr + "\" is NOT a binary string");
                return; // exit the program upon the first error detected
22         }
        }

24     // for-loop completed. No error detected.
26     System.out.println("\"" + inStr + "\" is a binary string");
    }
28 }

```

This version, although shorter, are harder to read, and harder to maintain.

2.9.9 Code Example: Binary to Decimal (Bin2Dec)

The following program prompts user for a binary string, and converts into its equivalent decimal number. For example,

Command window
▼

```

Enter a binary string : 10001001
2 The equivalent decimal for "10001001" is 137

```



```

import java.util.Scanner;

2
/**
4  * Bin2Dec.java
  * Prompt user for a binary string, and convert into its equivalent decimal number.
6  */
public class Bin2Dec {
8  public static void main(String[] args) {
    // Prompt and read input as "String"
10  Scanner in = new Scanner(System.in);
    System.out.print("Enter a binary string : ");
12  String binStr = in.next(); // The input binary string
    int binStrLen = binStr.length(); // The length of binStr
14  in.close();

16  // Process char by char from the right (i.e. Least-significant bit)
    // using exponent as loop index.
18  int dec = 0; // The decimal equivalent, to accumulate from 0
    char binChar; // Each individual char of the binStr
20  for (int exp = 0; exp < binStrLen; ++exp) {
        binChar = binStr.charAt(binStrLen - 1 - exp);
22  // 3 cases: '1' (add to dec), '0' (valid but do nothing), other (error)
        if (binChar == '1') {
24  dec += (int) Math.pow(2, exp); // cast the double result back to int
        } else if (binChar == '0') {
26  } else {
            System.out.println("error: invalid binary string \"" + binStr + "\"");
28  return; // or System.exit(1);
        }
30  }

32  // Print result
    System.out.println("The equivalent decimal for \""
34  + binStr + "\" is " + dec);
    }
36  }

```

Notes

1. The conversion formula is:



$$\begin{aligned} \text{binStr} &= b_{n-1}b_{n-2}\dots b_1b_0, \quad b_i \in \{0, 1\} \quad \text{where } b_0 \text{ is the least-significant bit} \\ \text{dec} &= b_{n-1}2^{n-1} + b_{n-2}2^{n-2} + \dots + b_12^1 + b_02^0 \end{aligned}$$

2. We use `binStr.charAt(idx)` to extract each individual *char* from the *binStr*. The *idx* begins at zero, and increases from left-to-right. On the other hand, the exponent number increases from right-to-left, as illustrated in the following example:



```

1 binStr           : 1 0 1 1 1 0 0 1
  charAt(idx)      : 0 1 2 3 4 5 6 7  (idx increases from the left)
3 Math.pow(2, exp) : 7 6 5 4 3 2 1 0  (exp increases from the right)

5 binStr.length() = 8
  idx + exp = binStr.length() - 1

```

3. This code uses `exp` as the loop index, and computes the `idx` for `charAt()` using the relationship `idx + exp = binStr.length() - 1`. You could also use the `idx` as the loop index (see next example).
4. We use the built-in function `Math.pow(x, y)` to compute the exponent, which takes two doubles and return a double. We need to explicitly cast the resultant double back to int for `dec`.
5. There are 3 cases to handle: '1' (add to `dec`), '0' (valid but do nothing for multiply by 0) and other (error). We can write the nested-if as follows, but that is harder to read:



```

1 if (binChar == '1') {
    dec += (int)Math.pow(2, exp); // cast the double result back to int
3 } else if (binChar != '0') {
    System.out.println("error: invalid binary string \"" + binStr + "\"");
5 return; // or System.exit(1);
    } // else (binChar == '0') do nothing

```

6. You can use Scanner's *nextInt(int radix)* method to read an *int* in the desired radix. Try reading a binary number (radix of 2) and print its decimal equivalent. For example,



```
1 number = in.nextInt(2);      // Input in binary e.g., 10110100
  System.out.println(number); // 180
```

2.9.10 Code Example: Hexadecimal to Decimal (Hex2Dec)

The following program prompts user for a hexadecimal string and converts into its equivalent decimal number. For example,

Command window

```
1 Enter a Hexadecimal string: 10aB
  The equivalent decimal for "10aB" is 4267
```



```
import java.util.Scanner;

2
/**
3  * Hex2Dec.java
4  * Prompt user for the hexadecimal string, and convert to its equivalent decimal number
5  */
6
public class Hex2Dec {
7
8  public static void main(String[] args) {
9      // Prompt and Read input as "String"
10     Scanner in = new Scanner(System.in);
11     System.out.print("Enter a Hexadecimal string: ");
12     String hexStr = in.next();      // The input hexadecimal String
13     int hexStrLen = hexStr.length(); // The length of hexStr
14     in.close();

15
16     // Process char by char from the left (most-significant digit)
17     int dec = 0;      // The decimal equivalent, to accumulate from 0
18     for (int charIdx = 0; charIdx < hexStrLen; ++charIdx) {
19         char hexChar = hexStr.charAt(charIdx);
20         int expFactor = (int) Math.pow(16, hexStrLen - 1 - charIdx);
21         // 23 cases: '0'-'9', 'a'-'f', 'A'-'F', other (error)
22         if (hexChar == '0') {
23             // Valid but do nothing
24         } else if (hexChar >= '1' && hexChar <= '9') {
```



```

26     dec += (hexChar - '0') * expFactor; // Convert char '0'-'9' to int 0-9
    } else if (hexChar >= 'a' && hexChar <= 'f') {
28         dec += (hexChar - 'a' + 10) * expFactor; // Convert char 'a'-'f' to int 10-15
    } else if (hexChar >= 'A' && hexChar <= 'F') {
30         dec += (hexChar - 'A' + 10) * expFactor; // Convert char 'A'-'F' to int 10-15
    } else {
32         System.out.println("error: invalid hex string \"" + hexStr + "\"");
        return; // or System.exit(1);
    }
34 }

36 System.out.println("The equivalent decimal for \"" + hexStr
    + "\" is " + dec);
38 }
}

```

Notes

1. The conversion formula is:



1 $hexStr = h_{n-1}h_{n-2} \dots h_2h_1h_0$, $h_i \in \{0, \dots, 9, A, \dots, F\}$ where h_0 is the least-significant bit

3 $dec = h_{n-1}16^{n-1} + h_{n-2}16^{n-2} + \dots h_1 + 16^1 + h_016^0$

2. In this example, we use the *charIdx* as the loop index, and compute the exponent via the relationship $charIdx + exp = hexStr.length() - 1$ (See the illustration in the earlier example).
3. You could write a big switch of 23 cases (0-9, A-F, a-f, and other). But take note how they are reduced to 5 cases.
 - (a) To convert *hexChar* '1' to '9' to int 1 to 9, we subtract the *hexChar* by the base '0'.
 - (b) Similarly, to convert *hexChar* 'a' to 'f' (or 'A' to 'F') to int 10 to 15, we subtract the *hexChar* by the base 'a' (or 'A') and add 10.
4. You may use *str.toLowerCase()* to convert the input string to lowercase to further reduce the number of cases. But You need to keep the original String for output in this example (otherwise, you could use *in.next().toLowerCase()* directly).

2.9.11 Exercises on String and char operations

1. ReverseString

Write a program called **ReverseString**, which prompts user for a *String*, and prints the reverse of the *String* by extracting and processing each character. The output shall look like:

Command window
▼

```

Enter a String: abcdef
2 The reverse of the String "abcdef" is "fedcba".

```

Hints

For a *String* called *inStr*, you can use *inStr.length()* to get the length of the *String*; and *inStr.charAt(idx)* to retrieve the *char* at the *idx* position, where *idx* begins at 0, up to *inStr.length() - 1*.



```

// Define variables
2 String inStr;           // input String
  int inStrLen;           // length of the input String
4 .....

6 // Prompt and read input as "String"
  System.out.print("Enter a String: ");
8 inStr = in.next();      // use next() to read a String
  inStrLen = inStr.length();

10
12 // Use inStr.charAt(index) in a loop to extract each character
12 // The String's index begins at 0 from the left .
12 // Process the String from the right
14 for (int charIdx = inStrLen - 1; charIdx >= 0; --charIdx) {
14     // charIdx = inStrLen-1, inStrLen-2, ... ,0
16     .....
16 }

```

2. CountVowelsDigits

Write a program called **CountVowelsDigits**, which prompts the user for a *String*, counts the number of vowels (a, e, i, o, u, A, E, I, O, U) and digits (0 – 9) contained in the *String*, and prints the counts and the percentages (rounded to 2 decimal places). For example,

Command window

```
1 Enter a String : testing12345
   Number of vowels: 2 (16.67 %)
3   Number of digits : 5 (41.67 %)
```

Hints

- (a) To check if a *char* *c* is a digit, you can use *boolean* expression $(c \geq '0' \ \&\& \ c \leq '9')$; or use built-in *boolean* function *Character.isDigit(c)*.
- (b) You could use *in.next().toLowerCase()* to convert the input *String* to lowercase to reduce the number of cases.
- (c) To print a % using *printf()*, you need to use *%%*. This is because % is a prefix for format specifier in *printf()*, e.g., %d and %f.

3. PhoneKeyPad

On your phone keypad, the alphabets are mapped to digits as follows: ABC(2), DEF(3), GHI(4), JKL(5), MNO(6), PQRS(7), TUV(8), WXYZ(9). Write a program called **PhoneKeyPad**, which prompts user for a *String* (case insensitive), and converts to a sequence of keypad digits. Use (a) a nested-if, (b) a switch-case-default.

Hints

- (a) You can use *in.next().toLowerCase()* to read a *String* and convert it to lowercase to reduce your cases.
- (b) In switch-case, you can handle multiple cases by omitting the break statement, e.g.,



```
1 switch (inChar) {
   case 'a':
3   case 'b':
   case 'c': // No break for 'a' and 'b', fall thru 'c'
5     System.out.print(2);
     break;
7   case 'd':
   case 'e':
9   case 'f':
     .....
11  default :
     .....
13 }
```

4. Caesar's Code

Caesar's Code is one of the simplest encryption techniques. Each letter in the plaintext is replaced by a letter some fixed number of position (n) down the alphabet cyclically. In this exercise, we shall pick $n = 3$. That is, 'A' is replaced by 'D', 'B' by 'E', 'C' by 'F', ..., 'X' by 'A', ..., 'Z' by 'C'.

Write a program called **CaesarCode** to cipher the Caesar's code. The program shall prompt user for a plaintext *String* consisting of mix-case letters only; compute the ciphertext; and print the ciphertext in uppercase. For example,

```
Command window
Enter a plaintext string : Testing
2 The ciphertext string is : WHVWLQJ
```

Hints

- (a) Use `in.next().toUpperCase()` to read an input string and convert it into uppercase to reduce the number of cases.
- (b) You can use a big nested-if with 26 cases ('A' - 'Z'). But it is much better to consider 'A' to 'W' as one case; 'X', 'Y' and 'Z' as 3 separate cases.
- (c) Take note that char 'A' is represented as Unicode number 65 and char 'D' as 68. However, 'A' + 3 gives 68. This is because `char + int` is implicitly casted to `int + int` which returns an int value. To obtain a char value, you need to perform explicit type casting using `(char)('A' + 3)`. Try printing ('A' + 3) with and without type casting.

5. Decipher Caesar's Code

Write a program called **DecipherCaesarCode** to decipher the Caesar's code described in the previous exercise. The program shall prompts user for a ciphertext string consisting of mix-case letters only; compute the plaintext; and print the plaintext in uppercase. For example,

```
Command window
Enter a ciphertext string : wHVwLQJ
2 The plaintext string is : TESTING
```

6. Exchange Cipher

This simple cipher exchanges 'A' and 'Z', 'B' and 'Y', 'C' and 'X', and so on.

Write a program called **ExchangeCipher** that prompts user for a plaintext string consisting of mix-case letters only. Your program shall compute the ciphertext; and print the ciphertext in uppercase. For examples,

Command window
▼

```
Enter a plaintext string : abcXYZ
2 The ciphertext string is : ZYXCBA
```

Hints

- (a) Use `in.next().toUpperCase()` to read an input string and convert it into uppercase to reduce the number of cases.
- (b) You can use a big nested-if with 26 cases ('A' - 'Z'), or use the following relationship:

```
'A' + 'Z' == 'B' + 'Y' == 'C' + 'X' == ... == plainTextChar + cipher-
TextChar
Hence, cipherTextChar = 'A' + 'Z' - plainTextChar
```

7. TestPalindromicWord and TestPalindromicPhrase

A word that reads the same backward as forward is called a palindrome, e.g., "mom", "dad", "racecar", "madam", and "Radar" (case-insensitive). Write a program called **TestPalindromicWord**, that prompts user for a word and prints "'xxx' is | is not a palindrome".

A phrase that reads the same backward as forward is also called a palindrome, e.g., "Madam, I'm Adam", "A man, a plan, a canal - Panama!" (ignoring punctuation and capitalization). Modify your program (called **TestPalindromicPhrase**) to check for palindromic phrase. Use `in.nextLine()` to read a line of input.

Hints

- (a) Maintain two indexes, forwardIndex (*fldx*) and backwardIndex (*bldx*), to scan the phrase forward and backward.



```
int fldx = 0;
2 int bldx = strLen - 1;
while (fldx < bldx) {
```



```

4      .....
      ++fldx;
6      --bldx;
      }

8
      // or
10     for (int fldx = 0, bldx = strLen - 1; fldx < bldx; ++fldx, --bldx) {
          .....
12     }

```

- (b) You can check if a *char* *c* is a letter either using built-in *boolean* function *Character.isLetter(c)*; or boolean expression ($c \geq 'a' \ \&\& \ c \leq 'z'$). Skip the index if it does not contain a letter.

8. CheckHexadecimalString

The hexadecimal (hex) number system uses 16 symbols, 0 – 9 and A - F (or a - f). Write a program to verify a hex *String*. The program shall prompt user for a hex *String*; and decide if the input *String* is a valid hex *String*. For examples,

Command window

```

1 Enter a hex string : 123aBc
  "123aBc" is a hex string
3
  Enter a hex string : 123aBcx
5 "123aBcx" is NOT a hex string

```

Hints



```

1 if (!((inChar >= '0' && inChar <= '9')
      || (inChar >= 'A' && inChar <= 'F')
3      || (inChar >= 'a' && inChar <= 'f')))) { // Use positive logic and then
      ↪ reverse
      .....
5 }

```

9. BinanyToDecimal

Write a program called **BinanyToDecimal** to convert an input binary *String* into its equivalent decimal number. Your output shall look like:

```
Command window
1 Enter a Binary string : 1011
  The equivalent decimal number for binary "1011" is : 11
3
  Enter a Binary string : 1234
5 error: invalid binary string "1234"
```

10. HexadecimalToDecimal

Write a program called **HexadecimalToDecimal** to convert an input hexadecimal *String* into its equivalent decimal number. Your output shall look like:

```
Command window
1 Enter a Hexadecimal string : 1a
  The equivalent decimal number for hexadecimal "1a" is : 26
3
  Enter a Hexadecimal string : 1y3
5 error: invalid hexadecimal string "1y3"
```

11. OctalToDecimal

Write a program called **OctalToDecimal** to convert an input Octal *String* into its equivalent decimal number. For example,

```
Command window
1 Enter an Octal string : 147
  The equivalent decimal number "147" is : 103
```

12. RadixNToDecimal

Write a program called **RadixNToDecimal** to convert an input *String* of any radix (≤ 16) into its equivalent decimal number.

```
Command window
  Enter the radix : 16
2 Enter the string : 1a
  The equivalent decimal number "1a" is : 26
```

2.10 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 (or items).

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. *Allocate* the array using `new` operator, or through *initialization*, e.g.,



```

1 int [] marks; // Declare an int array named "marks"
  // "marks" is assigned to a special value called "null" before allocation
3 int marks[]; // Same as above, but the above syntax recommended
  marks = new int [5]; // Allocate 5 elements via the "new" operator
5
  // Declare and allocate a 20-element array in one statement via "new" operator
7 int [] factors = new int [20];

9 // 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.]

When an array is declared but not allocated, it has a special value called *null*.

2.10.1 Array Index

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]`.



```

1 int [] marks = new int [5]; // Declare & allocate a 5-element int array

3 // Assign values to the elements
  marks[0] = 95;
  
```



```
5 marks[1] = 85;
   marks[2] = 77;
7 marks[3] = 69;
   marks[4] = 66;

9
   // Retrieve elements of the array
11 System.out.println(marks[0]);
    System.out.println(marks[3] + marks[4]);
```

2.10.2 Array's length

To create an array, you need to know 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 during runtime.

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. Suppose you set the length to 30 (for a class of students) and there are 31 students, you need to allocate a new array (of length 31), copy the old array to the new array, and delete the old array. In other words, the length of an array cannot be dynamically adjusted during runtime. This is probably the major drawback of using an array. (There are other structures that can be dynamically adjusted.)

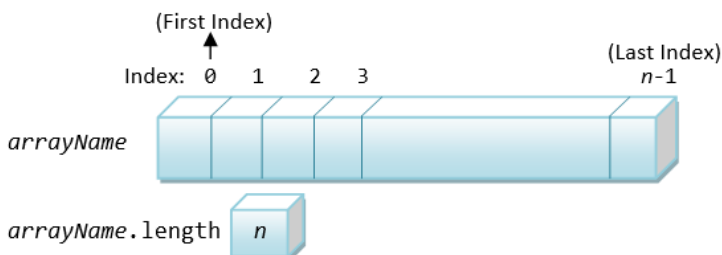
In Java, the length of array is kept in an *associated variable* called *length* and can be retrieved using "*arrayName.length*", e.g.,



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

Index and Length

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.length - 1]`, Java Runtime will signal an exception called `ArrayIndexOutOfBoundsException`.

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-weigh the slow down in speed.

2.10.3 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  /**
2  * MeanSDArray.java
3  * Find the mean and standard deviation of numbers kept in an array
4  */
5  public class MeanSDArray {
6      public static void main(String[] args) {
7          // Declare variable
8          int[] marks = {74, 43, 58, 60, 90, 64, 70};
9          int sum = 0;
10         int sumSquare = 0;
11
12         // Compute sum and square-sum using loop
13         for (int i = 0; i < marks.length; ++i) {
14             sum += marks[i];
15             sumSquare += marks[i] * marks[i];
16         }
17         double mean = (double)sum / marks.length;
18         double stdardDeviation = Math.sqrt(((double)sumSquare / marks.length
19             - mean * mean));
20
21         // Print results
22         System.out.printf("Mean is: %.2f%n", mean);
23         System.out.printf("Standard deviation is: %.2f%n", standardDeviation);
24     }
25 }

```

2.10.4 Enhanced for-loop (or "for-each" Loop) (JDK 5)

JDK 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



```
1 for (type item : anArray) {  
    body ;  
3 }  
    // type must be the same as the anArray's type
```

Example



```
int [] numbers = {8, 2, 6, 4, 3};  
2 int sum = 0;  
int sumSq = 0;  
4 for (int number : numbers) {    // for each int number in int [] numbers  
    sum += number;  
6    sumSq += number * number;  
}  
8 System.out.println ("The sum is: " + sum);  
System.out.println ("The square sum is: " + sumSq);
```

This loop 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. It requires fewer lines of code, eliminates the loop counter and the array index, and is easier to read. However, for array of primitive types (e.g., array of ints), 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, 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.

Code Example: Read and Print Array

The following program prompts user for the length and all the elements of an array, and print the array in the form of [a0, a1, ..., an]. For examples,

Command window

```
1 Enter the number of items: 5  
Enter the value of all items (separated by space): 7 9 1 6 2  
3 The values are: [7, 9, 1, 6, 2]
```



```

1 import java.util.Scanner;

3 /**
   * ReadPrintArray.java
5  * Prompt user for the length and all the elements of an array; and print [a1, a2, ..., an]
   */
7 public class ReadPrintArray {
   public static void main(String[] args) {
9       Scanner in = new Scanner(System.in);
       // Prompt for a non-negative integer for the number of items;
11      // and read the input as "int". No input validation .
       System.out.print("Enter the number of items: ");
13      final int NUM_ITEMS = in.nextInt();

15      // Allocate the array
       // Declare array name, to be allocated after numItems is known
17      int[] items = new int[NUM_ITEMS];

19      // Prompt and read the items into the "int" array, only if array length > 0
       if (items.length > 0) {
21          System.out.print("Enter the value of all items (separated by space): ");
          for (int i = 0; i < items.length; ++i) {
23              items[i] = in.nextInt();
          }
25      }
       in.close();

27

       /*
29      * Print array contents, need to handle first item
       * and subsequent items differently .
31      */
       System.out.print("The values are: [");
33      for (int i = 0; i < items.length; ++i) {
          if (i == 0) {
35              // Print the first item without a leading commas
               System.out.print(items[0]);
37          } else {
               // Print the subsequent items with a leading commas
39              System.out.print(", " + items[i]);
          }
41      }
       System.out.println("]");
43  }
}

```

Arrays.toString() (JDK 5)

JDK 5 provides an built-in methods called `Arrays.toString(anArray)`, which returns a `String` in the form `[a0, a1, ..., an]`. You need to import `java.util.Arrays`. For examples,



```

import java.util.Arrays;    // Needed to use Arrays.toString ()

2
/**
4  * TestArrayToString.java
  * Use Arrays.toString() to print an array in the form of [a1, a2, ..., an]
6  */
public class TestArrayToString {
8  public static void main(String[] args) {
    // Declare and allocate test arrays
10  int[] a1 = {6, 1, 3, 4, 5};    // Allocate via initialization
    int[] a2 = {};                // Empty array with length = 0
12  double[] a3 = new double[1]; // One-Element array, init to 0.0

14  System.out.println (Arrays.toString(a1)); // [6, 1, 3, 4, 5]
    System.out.println (Arrays.toString(a2)); // []
16  System.out.println (Arrays.toString(a3)); // [0.0]
    a3[0] = 2.2;
18  System.out.println (Arrays.toString(a3)); // [2.2]
    }
20 }

```

2.10.5 Code Example: Horizontal and Vertical Histograms

The following program prompts user for the number of students, and the grade of each student. It then print the histogram, in horizontal and vertical forms, as follows:

Command window

```

Enter the grade for student 1: 98
2 Enter the grade for student 2: 100
Enter the grade for student 3: 9
4 Enter the grade for student 4: 3
Enter the grade for student 5: 56
6 Enter the grade for student 6: 58
Enter the grade for student 7: 59
8 Enter the grade for student 8: 87

10 0 - 9: **
    10 - 19:
12 20 - 29:
    30 - 39:
14 40 - 49:
    50 - 59: ***
16 60 - 69:

```

Command window

```

70 - 79:
2 80 - 89: *
90 - 100: **
4
*
6 *           *           *
*           *           *
8 0-9  10-19 20-29 30-39 40-49 50-59 60-69 70-79 80-89 90-100

```



```

import java.util.Scanner;
2 import java.util.Arrays; // for Arrays.toString ()

4 /**
 * GradesHistograms.java
6 * Print the horizontal and vertical histograms of grades.
 */
8 public class GradesHistograms {
    public static void main(String[] args) {
10         // Declare variables
        int[] bins = new int[10]; // int array of 10 histogram bins for 0-9, 10-19, ...,
12                                // 90-100

14         Scanner in = new Scanner(System.in);
        // Prompt and read the number of students as "int"
16         System.out.print("Enter the number of students: ");
        int numStudents = in.nextInt();

18         // Allocate the array
20         int[] grades = new int[numStudents]; // Declare array name, and to be
                                                // allocated after numStudents is known
22
        // Prompt and read the grades into the int array "grades"
24         for (int i = 0; i < grades.length; ++i) {
            System.out.print("Enter the grade for student " + (i + 1) + ": ");
26             grades[i] = in.nextInt();
        }
28         in.close();

30         // Print array for debugging
        System.out.println(Arrays.toString(grades));
32
        // Populate the histogram bins
34         for (int grade : grades) {
            if (grade == 100) { // Need to handle 90-100 separately as it has 11 items.
36                 ++bins[9];
            } else {

```



```

38         ++bins[grade/10];
39     }
40 }
41 // Print array for debugging
42 System.out.println(Arrays.toString(bins));
43
44 /*
45  * Print the horizontal histogram
46  * Rows are the histogram bins[0] to bins[9]
47  * Columns are the counts in each bins[i]
48  */
49 for (int binIdx = 0; binIdx < bins.length; ++binIdx) {
50     // Print label
51     if (binIdx != 9) { // Need to handle 90-100 separately as it has 11 items
52         System.out.printf("%2d-%3d: ", binIdx*10, binIdx*10+9);
53     } else {
54         System.out.printf("%2d-%3d: ", 90, 100);
55     }
56     // Print columns of stars
57     for (int itemNo = 0; itemNo < bins[binIdx]; ++itemNo) { // one star per item
58         System.out.print(" ");
59     }
60     System.out.println();
61 }
62
63 // Find the max value among the bins
64 int binMax = bins[0];
65 for (int binIdx = 1; binIdx < bins.length; ++binIdx) {
66     if (binMax < bins[binIdx]) {
67         binMax = bins[binIdx];
68     }
69 }
70
71 /*
72  * Print the Vertical histogram
73  * Columns are the histogram bins[0] to bins[9]
74  * Rows are the levels from binMax down to 1
75  */
76 for (int level = binMax; level > 0; --level) {
77     for (int binIdx = 0; binIdx < bins.length; ++binIdx) {
78         if (bins[binIdx] >= level) {
79             System.out.print("  * ");
80         } else {
81             System.out.print("    ");
82         }
83     }
84     System.out.println();
85 }
86
87 // Print label
88 for (int binIdx = 0; binIdx < bins.length; ++binIdx) {

```



```

90     System.out. printf ( "%3d-%3d", binIdx*10, (binIdx != 9) ?
        binIdx * 10 + 9 : 100);    // Use '-' flag for left-aligned
92     }
        System.out. println () ;
94     }
    }

```

Notes

1. We use two arrays in this exercise, one for storing the grades of the students (of the length numStudents) and the other to storing the histogram counts (of length 10).
2. We use a 10-element int arrays called bins, to keep the histogram counts for grades of [0, 9], [10, 19], ..., [90, 100]. Take note that there are 101 grades between [0, 100], and the last bin has 11 grades (instead of 10 for the rest). The bins's index is grade/10, except grade of 100.

2.10.6 Code Example: Hexadecimal to Binary (Hex2Bin)

The following program prompts user for a hexadecimal string and convert it to its binary equivalence. For example,

Command window

```

Enter a Hexadecimal string: 1bE3
2 The equivalent binary for "1bE3" is "0001101111100011"

```



```

import java. util .Scanner;

2
/**
4  * Hex2Bin.java
  * Prompt user for a hexadecimal string , and print its binary equivalent
6  */
public class Hex2Bin {
8  public static void main(String[] args) {
    // The equivalent binary String, to accumulate from an empty String
10     String binStr = "";

12     /*
    * Lookup table for the binary sub-string corresponding to

```



```

14      * Hex digit '0' (index 0) to 'F' (index 15)}*)
15      */
16      final String [] BIN_STRS = {
17          "0000", "0001", "0010", "0011",
18          "0100", "0101", "0110", "0111",
19          "1000", "1001", "1010", "1011",
20          "1100", "1101", "1110", "1111"
21      };
22
23      // Prompt and read input as "String"
24      Scanner in = new Scanner(System.in);
25      System.out.print("Enter a Hexadecimal string: ");
26      String hexStr = in.next();          // The input hexadecimal String
27      int hexStrLen = hexStr.length();    // The length of hexStr
28      in.close();
29
30      // Process the string from the left (most-significant hex digit)
31      char hexChar;                       // Each char in the hexStr
32      for (int charIdx = 0; charIdx < hexStrLen; ++charIdx) {
33          hexChar = hexStr.charAt(charIdx);
34          if (hexChar >= '0' && hexChar <= '9') {
35              binStr += BIN_STRS[hexChar - '0']; // index into the BIN_STRS
36                                              // array and concatenate
37          } else if (hexChar >= 'a' && hexChar <= 'f') {
38              binStr += BIN_STRS[hexChar - 'a' + 10];
39          } else if (hexChar >= 'A' && hexChar <= 'F') {
40              binStr += BIN_STRS[hexChar - 'A' + 10];
41          } else {
42              System.err.println("error: invalid hex string \"" + hexStr + "\"");
43              return; // or System.exit(1);
44          }
45      }
46
47      System.out.println("The equivalent binary for \""
48          + hexStr + "\" is \"" + binStr + "\"");
49  }
50 }

```

Notes

1. We keep the binary string corresponding to hex digit '0' to 'F' in an array with indexes of 0 – 15, used as look-up table.
2. We extract each hexChar, find its array index (0 – 15), and retrieve the binary string from the array based on the index.
 - a. To convert hexChar '1' to '9' to *int* 1 to 9, we subtract the hexChar by the base '0'.
 - b. Similarly, to convert hexChar 'a' to 'f' (or 'A' to 'F') to *int* 10 to 15, we subtract the hexChar by the base 'a' (or 'A') and add 10.

2.10.7 Code Example: Decimal to Hexadecimal (Dec2Hex)

The following program prompts user for an integer, reads as *int*, and prints its hexadecimal equivalent. For example,

Command window

```
Enter a decimal number: 1234
2 The equivalent hexadecimal number is 4D2
```



```
import java.util.Scanner;

2
/**
4  * Dec2Hex.java
  * Prompt user for an int, and print its equivalent hexadecimal number.
6  */
public class Dec2Hex {
8  public static void main(String[] args) {
    // Declare variables
10  int dec;           // The input decimal number in "int"
    String hexStr = ""; // The equivalent hex String, to accumulate from an empty
                        // String
12  int radix = 16;    // Hex radix

14  // Use this array as lookup table for converting 0-15 to 0-9A-F
    final char[] HEX_CHARS =
16  { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F' };

18  // Prompt and read input as "int"
    Scanner in = new Scanner(System.in);
20  System.out.print("Enter a decimal number: ");
    dec = in.nextInt();
22  in.close();

24  // Repeated modulus/division and get the hex digits (0-15) in reverse order
    while (dec > 0) {
26      int hexDigit = dec % radix;           // 0-15
        hexStr = HEX_CHARS[hexDigit] + hexStr; // Append in front of the hex string
                                                // corresponds to reverse order
28      dec = dec / radix;
30  }

32  System.out.println("The equivalent hexadecimal number is " + hexStr);
    }
34 }
```

Notes

1. We use modulus/divide algorithm to get the hex digits (0-15) in reserve order.
See "Number System Conversion".
2. We look up the hex digit '0'-'F' from an array using index 0-15.

2.10.8 Multi-Dimensional Array

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



```
1 int grid [][] = new int [12][8];           // a 12x8 grid of int
2 grid [0][0] = 8;
  grid [1][1] = 5;
4 System.out. println (grid .length);        // 12
  System.out. println (grid [0].length);     // 8
6 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.



```
1  /**
2   * Array2DTest.java
3   * The Array2DTest class
4   */
5   public class Array2DTest {
6       public static void main(String[] args) {
7           int [][] grid = new int [12][8];           // A 12x8 grid, in [row][col] or [y][x]
8           final int NUM_ROWS = grid.length;         // 12
9           final int NUM_COLS = grid[0].length;      // 8
10
11          // Fill in grid
12          for (int row = 0; row < NUM_ROWS; ++row) {
13              for (int col = 0; col < NUM_COLS; ++col) {
14                  grid[row][col] = row*NUM_COLS + col + 1;
15              }
16          }
17
18          // Print grid
19          for (int row = 0; row < NUM_ROWS; ++row) {
20              for (int col = 0; col < NUM_COLS; ++col) {
21                  System.out. printf ("%3d", grid[row][col]);
22              }
23              System.out. println ();
24          }
25      }
26  }
```



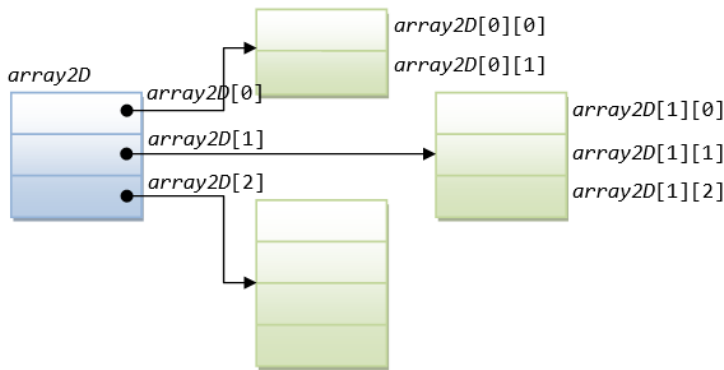
```

24     }
    }
26 }

```

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,



```

/**
2  * Array2DWithDifferentLength.java
  * The Array2DWithDifferentLength class
4  */
public class Array2DWithDifferentLength {
6  public static void main(String[] args) {
    // Allocate grid
8  int [][] grid = {
    {1, 2},
10 {3, 4, 5},
    {6, 7, 8, 9}
12 };

14 // Print grid
    for (int y = 0; y < grid.length; ++y) {
16     for (int x = 0; x < grid[y].length; ++x) {
        System.out.printf("%2d", grid[y][x]);
18     }
    }
}

```



```

    System.out.println ();
20 }

22 // Another 2D array
    int [][] grid1 = new int [3][];
24 grid1 [0] = new int [2];
    grid1 [1] = new int [3];
26 grid1 [2] = new int [4];

28 // Print grid - all elements init to 0
    for (int y = 0; y < grid1.length; ++y) {
30         for (int x = 0; x < grid1[y].length; ++x) {
                System.out.printf ("%2d", grid1[y][x]);
32         }
        System.out.println ();
34     }
    }
36 }

```

2.10.9 Exercises on Arrays

1. PrintArray

Write a program called **PrintArray** which prompts user for the number of items in an array (a non-negative integer), and saves it in an int variable called NUM_ITEMS. It then prompts user for the values of all the items and saves them in an int array called items. The program shall then print the contents of the array in the form of $[x_1, x_2, \dots, x_n]$. For example,

Command window

```

Enter the number of items: 5
2 Enter the value of all items (separated by space): 3 2 5 6 9
The values are: [3, 2, 5, 6, 9]

```

Hints



```

1  /**
    * PrintArray.java
3  */
    public class PrintArray {
5      public static void main(String[] args) {
            // Declare variables

```



```

7  final int NUM_ITEMS;

9  // Declare array name, to be allocated after NUM_ITEMS is known
   int[] items;
11  .....

13  // Prompt for the number of items and read the input as "int"
   .....
15  NUM_ITEMS = .....

17  // Allocate the array
   items = new int[NUM_ITEMS];
19

21  // Prompt and read the items into the "int" array, if array length > 0
   if (items.length > 0) {
       .....
23     for (int i = 0; i < items.length; ++i) { // Read all items
25         .....
       }
27     }

29     // Print array contents, need to handle first item
31     // and subsequent items differently
       .....
33     for (int i = 0; i < items.length; ++i) {
35         if (i == 0) {
37             // Print the first item without a leading commas
               .....
           } else {
39                 // Print the subsequent items with a leading commas
                   .....
           }

41         // or, using a one liner
43         //System.out.print ((i == 0) ? ..... : .....);
       }
45     }
}

```

2. PrintArrayInStars

Write a program called **PrintArrayInStars** which prompts user for the number of items in an array (a non-negative integer), and saves it in an *int* variable called NUM_ITEMS. It then prompts user for the values of all the items (non-negative integers) and saves them in an *int* array called items. The program shall then print the contents of the array in a graphical form, with the array index and values represented by number of stars. For examples,

Command window

```

Enter the number of items: 5
2 Enter the value of all items (separated by space): 7 4 3 0 7
0: ******(7)
4 1: ****(4)
2: ***(3)
6 3: (0)
4: ******(7)

```

Hints



```

1  /**
   * PrintArrayInStars .java
3  */
   public class PrintArrayInStars {
5     public static void main(String[] args) {
        // Declare variables
7     final int NUM_ITEMS;
        // Declare array name, to be allocated after NUM_ITEMS is known
9     int[] items;
        .....
11
        // Print array in "index: number of stars" using a nested-loop
13     // Take note that rows are the array indexes and columns are the value
        // in that index
15     for (int idx = 0; idx < items.length; ++idx) { // row
        System.out.print(idx + ": ");
17         // Print value as the number of stars
        for (int starNo = 1; starNo <= items[idx]; ++starNo) { // column
19             System.out.print("*");
        }
21         .....
        }
23         .....
    }
25 }

```

3. GradesStatistics

Write a program which prompts user for the number of students in a class (a non-negative integer), and saves it in an *int* variable called *numStudents*. It then prompts user for the grade of each of the students (integer between 0 to 100) and saves them in an *int* array called *grades*. The program shall then compute and print the average (in *double* rounded to 2 decimal places) and minimum/maximum (in *int*).

```

Command window
Enter the number of students : 5
2 Enter the grade for student 1: 98
Enter the grade for student 2: 78
4 Enter the grade for student 3: 78
Enter the grade for student 4: 87
6 Enter the grade for student 5: 76
The average is : 83.40
8 The minimum is: 76
The maximum is: 98

```

4. HexToBinary

Write a program called **HexToBinary** that prompts user for a hexadecimal *String* and print its equivalent binary *String*. The output shall look like:

```

Command window
1 Enter a Hexadecimal string : 1abc
The equivalent binary for hexadecimal "1abc" is : 0001 1010 1011 1100

```

Hints

- (a) Use an array of 16 Strings containing binary strings corresponding to hexadecimal number 0 – 9A – F (or a – f), as follows:



```

final String[] HEX_BITS = {"0000", "0001", "0010", "0011",
2  "0100", "0101", "0110", "0111",
  "1000", "1001", "1010", "1011",
4  "1100", "1101", "1110", "1111"};

```

5. DecimalToHexadecimal

Write a program called **DecimalToHexadecimal** that prompts user for a positive decimal number, read as *int*, and print its equivalent hexadecimal string. The output shall look like:

```

Command window
1 Enter a decimal number: 1234
The equivalent hexadecimal number is 4D2

```

2.11 Methods (Functions)

2.11.1 Why Methods?

At times, a certain portion of code has to be used many times. Instead of re-writing the code 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 code: 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 code (or API).

2.11.2 Using Methods

Two parties are involved in using a method: a caller, who calls (or invokes) the method, and the method called. The process is:

1. The caller invokes a method and passes arguments to the method.
2. The method:
 - a. receives the arguments passed by the caller,
 - b. performs the programmed operations defined in the method's body, and
 - c. returns a result back to the caller.
3. The caller receives the result, and continue its operations.

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.



```
/*  
2  * EgMethodGetArea.java  
   * The EgMethodGetArea class  
4  */
```




```

public class EgMethodGetArea {
6  // The entry main method
  public static void main(String [] args) {
8    double r = 1.1;

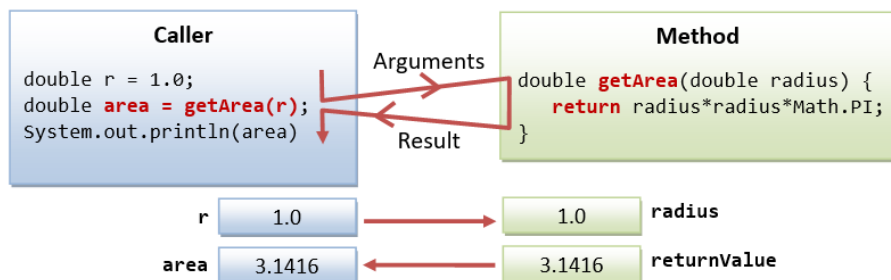
10   // Call (Invoke) method getArea() and return
    double area = getArea(r);
12   System.out.println("area is " + area);

14   // Call method getArea() again and return
    double area2 = getArea(2.2);
16   System.out.println("area 2 is " + area2);

18   // Call method getArea() one more time and return
    System.out.println("area 3 is " + getArea(3.3));
20  }

22  /*
   * Method getArea() Definition .
24  * Compute and return the area (in double) of circle given its radius (in double).
   */
26  public static double getArea(double radius) {
    return radius *radius *Math.PI;
28  }
}

```



The expected outputs are:

Command window

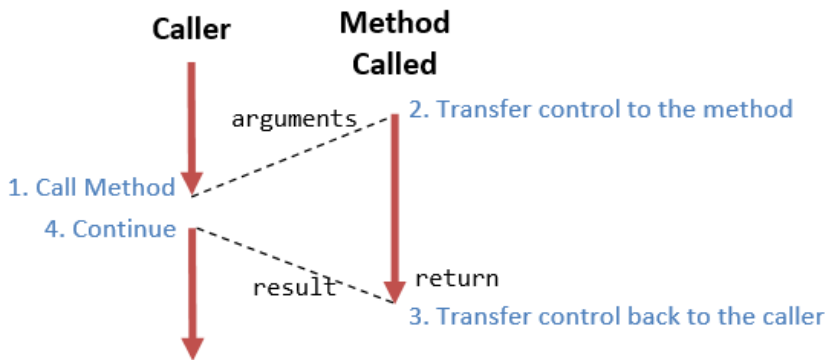
```

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

```

In the above example, a reusable method called *getArea()* is defined, which receives an argument 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.

Take note that there is a transfer of control from the caller to the method called, and from the method back to the caller, as illustrated.



Tracing Method Invocation

You can trace method operations under Eclipse/NetBeans (Refer to the the Eclipse/NetBeans How-to article):

- **Step Over:** Treat the method call as one single step.
- **Step Into:** Step into the method, so that you can trace the operations of the method.
- **Step Out:** Complete the current method and return to the caller.
- Set "**Breakpoints**" inside the method, and "resume" running to the next breakpoint.

Method Definition Syntax

The syntax for method definition is as follows:



```
1 public static returnType methodName(arg1-type arg1, arg2-type arg2, ... ) {
   body;
3 }
```

Example



```
1  /**
   * Return circle 's area given its radius
3  */
   public static double getArea(double radius) {
5      return radius * radius * Math.PI;
   }

7
   /**
9  * Return maximum among two given integers
   */
11 public static int max(int number1, int number2) {
    if (number1 > number2) {
13     return number1;
    }

15     return number2;
17 }
```

Take note that you need to specify the type of the arguments and the return value in method definition.

Calling Methods

To call a method, simply use `methodName(arguments)`. For examples, to call the above methods:



```
1  // Calling getArea()
   double area1 = getArea(1.1); // with literal as argument
3  double r2 = 2.2;
   double area2 = getArea(r2); // with variable as argument
5  double r3 = 3.3;
   System.out.println("Area is: " + area(r3));

7
   // Calling max()
9  int result1 = max(5, 8);
   int i1 = 7;
11 int i2 = 9;
   int result2 = max(i1, i2);
13 System.out.println("Max is: " + max(15, 16));
```

Take note that you need to specify the type in the method definition, but not during invocation.

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.



```

1  /**
   * Example of Java Method definition and invocation
3  */
   public class EgMinMaxMethod {
5     public static void main(String[] args) { // The entry main() method
        int a = 6;
7         int b = 9;
        int maxValue = max(a, b); // invoke method max() with arguments
9         int minValue = min(a, b); // invoke method min() with arguments
        System.out.println(maxValue + ", " + minValue);

11
        System.out.println(max(5, 8)); // invoke method max()
13        System.out.println(min(5, 8)); // invoke method min()
    }

15
    // The max() method returns the maximum of two given int
17    public static int max(int number1, int number2) {
        if (number1 > number2) {
19        return number1;
        }
21        return number2;
    }

23
    // The min() method returns the minimum of two given int
25    public static int min(int number1, int number2) {
        return (number1 < number2) ? number1 : number2;
27    }
}

```

2.11.3 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
2 return;           // return nothing (or void)

```

2.11.4 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.

2.11.5 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 parameters* or *arguments*, whose value is then used for the actual computation. For example, when the method is invoked via `"area = getArea(radius)"`, *radius* is the actual parameter, with a value of 1.1.

2.11.6 Code Example: Magic Number

The following program contains a boolean method called `isMagic(int number)`, which returns true if the given number contains the digit 8, e.g., 18, 108, and 1288. The signature of the method is:



```
public static boolean isMagic(int number);
```

It also provides the `main()` method to test the `isMagic()`. For example,

Command window

```
1 Enter a positive integer: 1288
   1288 is a magic number
3
5 Enter a positive integer: 1234567
   1234567 is not a magic number
```



```

1 import java . util . Scanner;

3 /**
 * MagicNumber.java
5  * This program contains a boolean method called isMagic(int number),
 * which tests if the given number contains the digit 8.
7  */
public class MagicNumber {
9  public static void main(String[] args) {
    Scanner in = new Scanner(System.in);

11
    // Prompt and read input as "int "
13    System.out.print("Enter a positive integer: ");
    int number = in.nextInt();

15
    // Call isMagic() to test the input
17    if (isMagic(number)) {
        System.out.println(number + " is a magic number");
19    } else {
        System.out.println(number + " is not a magic number");
21    }
    in.close();
23 }

25 /**
 * Check if the given int contains the digit 8, e.g., 18, 82, 1688.
27 * @param number The given integer
 * @return true if number contains the digit 8
29 * @Precondition number > 0 (i.e., a positive integer)
 */
31 public static boolean isMagic(int number){
    boolean isMagic = false; // shall change to true if found a digit 8
33
    // Extract and check each digit
35 while (number > 0) {
        int digit = number % 10; // Extract the last digit
37         if (digit == 8) {
            isMagic = true;
39             break; // only need to find one digit 8
        }
41         number /= 10; // Drop the last digit and repeat
    }

43     return isMagic;
45 }
}

```

Take note of the proper *documentation comment* for the method.

2.11.7 Code Example: int Array Methods

The following program contains various method for int array with signatures as follows:



```

public static void print(int[] array);    // Print [a1, a2, ..., an]
2 public static int min(int[] array);     // Return the min of the array
public static int sum(int[] array);      // Return the sum of the array
4 public static double average(int[] array); // Return the average of the array

```

It also contains the *main()* method to test all the methods. For example,

Command window

```

Enter the number of items: 5
2 Enter the value of all items (separated by space): 8 1 3 9 4
The values are: [8, 1, 3, 9, 4]
4 The min is: 1
The sum is: 25
6 The average (rounded to 2 decimal places) is: 5.00

```



```

import java.util.Scanner;
2
/**
4  * IntArrayMethodsTest.java
  * Test various int[] methods.
6  */
public class IntArrayMethodsTest {
8  public static void main(String[] args) {
    // Declare variables
10  final int NUM_ITEMS;
    int[] items; // Declare array name, to be allocated after numItems is known
12
    // Prompt for a non-negative integer for the number of items;
14  // and read the input as "int". No input validation.
    Scanner in = new Scanner(System.in);
16  System.out.print("Enter the number of items: ");
    NUM_ITEMS = in.nextInt();
18
    // Allocate the array
20  items = new int[NUM_ITEMS];

```



```

22 // Prompt and read the items into the "int" array, if array length > 0
   if (items.length > 0) {
24     System.out.print("Enter the value of all items (separated by space): ");
       for (int i = 0; i < items.length; ++i) {
26         items[i] = in.nextInt();
       }
28   }
   in.close();

30
   // Test the methods
32   System.out.print("The values are: ");
   print(items);
34   System.out.println("The min is: " + min(items));
   System.out.println("The sum is: " + sum(items));
36   System.out.printf("The average (rounded to 2 decimal places) is: %.2f%n",
       average(items));
38 }

40 /**
   * Prints the given int array in the form of [x1, x2, ..., xn]
42   * @param array The given int array
   * @Postcondition Print output as side effect
44   */
   public static void print(int[] array) {
46     System.out.print("[");
       for (int i = 0; i < array.length; ++i) {
48         System.out.print((i == 0) ? array[i] : ", " + array[i]);
       }
50     System.out.println("]");
   }

52
   /**
54   * Get the min of the given int array
   * @param array The given int array
56   * @return The min value of the given array
   */
   public static int min(int[] array) {
58     int min = array[0];
       for (int i = 1; i < array.length; ++i) {
60         if (array[i] < min) {
62             min = array[i];
64         }
       }
       return min;
66   }

68
   /**
70   * Get the sum of the given int array
   * @param array The given int array
   * @return The sum of the given array
72   */

```




```

public static int sum(int[] array) {
74     int sum = 0;
    for (int item: array) {
76         sum += item;
    }
78     return sum;
}

80

/**
82  * Get the average of the given int array
  * @param array The given int array
84  * @return The average of the given array
  */
86  public static double average(int[] array) {
    return (double)(sum(array)) / array.length;
88  }
}

```

2.11.8 Pass-by-Value for Primitive-Type Parameters

In Java, when an argument of primitive 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  /**
   * PassByValueTest.java
3  * The PassByValueTest class
   */
5  public class PassByValueTest {
    public static void main(String[] args) {
7      int number = 8;
      System.out.println("In caller, before calling the method, number is: " + number);
9      int result = increment(number); // invoke method with primitive-type parameter
      System.out.println("In caller, after calling the method, number is: " + number);
11     System.out.println("The result is " + result); // 9
    }

13     // Return number + 1
15     public static int increment(int number) {
      System.out.println("Inside method, before operation, number is " + number);
17     ++number; // change the parameter
      System.out.println("Inside method, after operation, number is " + number);
19     return number;
    }
21 }

```

Note: Although there is a variable called `number` in both the `main()` and `increment()` method, there are two distinct copies - one available in `main()` and another available in `increment()` - happen to have the same name. You can change the name of either one, without affecting the program.

2.11.9 Pass-by-Sharing for Arrays and Objects

As mentioned, for primitive-type parameters, a cloned copy is made and passed into the method. Hence, the method **cannot modify** the values in the caller. It is known as pass-by-value.

For arrays (and objects - to be described in the later chapter), the array reference is passed into the method and the method **can modify** the contents of array's elements. It is known as pass-by-sharing. For example,



```

1 import java.util.Arrays; // for Arrays.toString ()

3 /**
4  * PassBySharingTest.java
5  * The PassBySharingTest class
6  */
7 public class PassBySharingTest {
8     public static void main(String[] args) {
9         int[] testArray = {9, 5, 6, 1, 4};
10        System.out.println("In caller, before calling the method, array is: "
11            + Arrays.toString(testArray)); // [9, 5, 6, 1, 4]

12        // Invoke method with an array parameter
13        increment(testArray);
14        System.out.println("In caller, after calling the method, array is: "
15            + Arrays.toString(testArray)); // [10, 6, 7, 2, 5]
16    }

17    // Increment each of the element of the given int array
18    public static void increment(int[] array) {
19        System.out.println("Inside method, before operation, array is "
20            + Arrays.toString(array)); // [9, 5, 6, 1, 4]
21        // Increment each elements
22        for (int i = 0; i < array.length; ++i) {
23            ++array[i];
24        }

25        System.out.println("Inside method, after operation, array is "
26            + Arrays.toString(array)); // [10, 6, 7, 2, 5]
27    }
28 }

```

2.11.10 Varargs - Method with Variable Number of Formal Arguments (JDK 5)

Before JDK 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 5 introduces variable arguments (or varargs) and a new syntax "Type...". For example,



```
1 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, because Java maintains the length of the array in an associated variable `length`.



```
/**
2  * VarargsTest.java
3  * The VarargsTest class
4  */
5  public class VarargsTest {
6      // A method which takes a variable number of arguments (varargs)
7      public static void doSomething(String ... strs) {
8          System.out.print("Arguments are: ");
9          for (String str : strs) {
10             System.out.print(str + ", ");
11         }
12         System.out.println();
13     }
14
15     // A method which takes exactly two arguments
16     public static void doSomething(String s1, String s2) {
17         System.out.println("Overloaded version with 2 args: " + s1 + ", " + s2);
18     }
19
20     // Cannot overload with this method - crash with varargs version
21     // public static void doSomething(String[] strs)
22 }
```



```
// Test main() method. Can also use String ... instead of String []
24 public static void main(String ... args) {
    doSomething("Hello", "world", "again", "and", "again");
26     doSomething("Hello", "world");

28     String [] strs = {"apple", "orange"};
    doSomething(strs); // invoke varargs version
30 }
}
```

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 5, you can also declare your *main()* method as:



```
1 public static void main(String ... args) { // JDK 5 varargs
    ....
3 }
```

2.11.11 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.

2.11.12 Method Overloading

In Java, a method (of a particular method name) can have more than one versions, each version operates on different set of parameters - known as method overloading. The versions shall be differentiated by the numbers, types, or orders of the parameters.

Example 1



```

1  /**
   * AverageMethodOverloading.java
3  * Testing Method Overloading
   */
5  public class AverageMethodOverloading {
   public static void main(String[] args) {
7      System.out.println(average(8, 6));    // invoke version 1
       System.out.println(average(8, 6, 9)); // invoke version 2
9      System.out.println(average(8.1, 6.1)); // invoke version 3
       System.out.println(average(8, 6.1));
11     // int 8 autocast to double 8.0, invoke version 3
       // average(1, 2, 3, 4) // Compilation Error – no such method
13 }

15 // Version 1 takes 2 int's
   public static int average(int n1, int n2) {
17     System.out.println("version 1");
       return (n1 + n2)/2; // int
19 }

21 // Version 2 takes 3 int's
   public static int average(int n1, int n2, int n3) {
23     System.out.println("version 2");
       return (n1 + n2 + n3)/3; // int
25 }

27 // Version 3 takes 2 doubles
   public static double average(double n1, double n2) {
29     System.out.println("version 3");
       return (n1 + n2) /2.0; // double
31 }
   }

```

The expected outputs are:

Command window

```

version 1
2 7
version 2
4 7
version 3
6 7.1
version 3
8 7.05.

```

Example 2: Arrays

Suppose you need a method to compute the sum of the elements for *int[]*, *short[]*, *float[]* and *double[]*, you need to write all overloaded versions - there is no shortcut.



```

1  /**
2   * SumArrayMethodOverloading.java
3   * Testing Array Method Overloading
4   */
5   public class SumArrayMethodOverloading {
6       public static void main(String[] args) {
7           int[] a1 = {9, 1, 2, 6, 5};
8           System.out.println(sum(a1));    // invoke version 1
9           double[] a2 = {1.1, 2.2, 3.3};
10          System.out.println(sum(a2));    // invoke version 2
11          float[] a3 = {1.1f, 2.2f, 3.3f};
12          // System.out.println(sum(a3)); // error - float[] is not casted to double[]
13      }
14
15      // Version 1 takes an int[]
16      public static int sum(int[] array) {
17          System.out.println("version 1");
18          int sum = 0;
19          for (int item : array) sum += item;
20          return sum;                    // int
21      }
22
23      // Version 2 takes a double[]
24      public static double sum(double[] array) {
25          System.out.println("version 2");
26          double sum = 0.0;
27          for (double item : array) sum += item;
28          return sum; // double
29      }
30  }

```

Notes

1. Unlike primitives, where *int* would be autocasted to *double* during method invocation, *int[]* is not casted to *double[]*.
2. To handle all the 7 primitive number type arrays, you need to write 7 overloaded versions to handle each array types!

2.11.13 "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.



```
/**
2  * BooleanMethodTest.java
  * Testing boolean method (method that returns a boolean value)
4  */
public class BooleanMethodTest {
6  // This method returns a boolean value
  public static boolean isOdd(int number) {
8    if (number % 2 == 1) {
        return true;
10   } else {
        return false;
12   }
  }
14
  public static void main(String[] args) {
16   System.out.println(isOdd(5)); // true
    System.out.println(isOdd(6)); // false
18   System.out.println(isOdd(-5)); // false
  }
20 }
```

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



```
public static boolean isOdd(int number) {
2  if (number % 2 == 0) {
    return false;
4  } else {
    return true;
6  }
}
```

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:



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



```
3 }  
  
5 public static boolean isOdd(int number) {  
    return !(number % 2 == 0);  
7 }
```

2.11.14 Mathematical Methods

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



```
1 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  
3  
double Math.random()                  // returns a random number in [0.0, 1.0)  
5  
double Math.sin(double a)  
7 double Math.cos(double a)  
  
9 double Math.max(double a, double b)  
double Math.min(double a, double b)  
11  
double toRadians(double angdeg)
```

The **Math** class also provide two constants:



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

To check all the available methods, open JDK API documentation \Rightarrow select module "java.base" \Rightarrow select package "java.lang" \Rightarrow select class "Math" \Rightarrow choose method (@ <https://docs.oracle.com/javase/17/docs/api/java/lang/Math.html> for JDK 17). For examples,



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




```

4 double radius = 5.5;
   double area = radius*radius*Math.PI;
6 area = Math.pow(radius, 2)*Math.PI;           // Not as efficient as above

8 int x1 = 1;
   int y1 = 1;
10 int x2 = 2;
   int y2 = 2;
12 double distance = Math.sqrt((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1));

14 int dx = x2 - x1;
   int dy = y2 - y1;
16 distance = Math.sqrt(dx*dx + dy*dy);         // Slightly more efficient

```

2.11.15 Exercises on Methods

1. `exponent()`

Write a method called *exponent*(*int base*, *int exp*) that returns an *int* value of base raises to the power of exp. The signature of the method is:



```
public static int exponent(int base, int exp);
```

Assume that exp is a non-negative integer and base is an integer. Do not use any Math library functions.

Also write the *main*() method that prompts user for the *base* and *exp*; and prints the result. For example,

Command window

```

1 Enter the base: 3
   Enter the exponent: 4
3 3 raises to the power of 4 is: 81

```

Hints



```

1 /**
   * Exponent.java

```



```

3  * The Exponent class
   */
5  public class Exponent {
    public static void main(String[] args) {
7      // Declare variables
        int exp;    // exponent (non-negative integer)
9      int base;    // base (integer)
        .....

11     // Prompt and read exponent and base
13     .....

15     // Print result
        System.out.println(base + " raises to the power of "
17         + exp + " is: " + exponent(base, exp));
    }

19    /**
21     * Get "base" raised to the power "exp"
22     * @param base, exp
23     * @return "base" raised to the power "exp"
24     */
25    public static int exponent(int base, int exp) {
        int product = 1;    // resulting product

27        // Multiply product and base for exp number of times
29        for (.....) {
            product *= base;
31        }

33        return product;
    }
35 }

```

2. hasEight()

Write a *boolean* method called *hasEight()*, which takes an *int* as input and returns true if the number contains the digit 8 (e.g., 18, 168, 1288). The signature of the method is as follows:



```

1  public static boolean hasEight(int number);

```

Write a program called **MagicSum**, which prompts user for integers (or -1 to end), and produce the sum of numbers containing the digit 8. Your program should use the above methods. A sample output of the program is as follows:

Command window

```

1 Enter a positive integer (or -1 to end): 1
Enter a positive integer (or -1 to end): 2
3 Enter a positive integer (or -1 to end): 3
Enter a positive integer (or -1 to end): 8
5 Enter a positive integer (or -1 to end): 88
Enter a positive integer (or -1 to end): -1
7 The magic sum is: 96

```

Hints

- (a) The *coding pattern* to repeat until input is -1 (called sentinel value) is:



```

1 final int SENTINEL = -1; // Terminating input
  int number;
3
  // Read first input to "seed" the while loop
5 System.out.print("Enter a positive integer (or -1 to end): ");
  number = in.nextInt();
7
  while (number != SENTINEL) { // Repeat until input is -1
9      .....
11
    // Read next input. Repeat if the input is not the SENTINEL
    // Take note that you need to repeat these codes!
13 System.out.print("Enter a positive integer (or -1 to end): ");
    number = in.nextInt();
15 }

```

- (b) You can either repeatedly use modulus/divide ($n \% 10$ and $n = n/10$) to extract and drop each digit in *int*; or convert the *int* to *String* and use the *String*'s *charAt()* to inspect each *char*.

3. print()

Write a method called *print()*, which takes an *int* array and print its contents in the form of $[a_1, a_2, \dots, a_n]$. Take note that there is no comma after the last element. The method's signature is as follows:



```
public static void print(int[] array);
```

Also write a test driver to test this method (you should test on empty array, one-element array, and n-element array).

How to handle *double[]* or *float[]*? You need to write an overloaded version for *double[]* and an overloaded version for *float[]*, with the following signatures:



```
1 public static void print(double[] array)
   public static void print(float[] array)
```

The above is known as *method overloading*, where the same method name can have many versions, differentiated by its parameter list.

Hints

- (a) For the first element, print its value; for subsequent elements, print commas followed by the value.

4. `arrayToString()`

Write a method called *arrayToString()*, which takes an *int* array and return a *String* in the form of $[a_1, a_2, \dots, a_n]$. Take note that this method returns a *String*, the previous exercise returns *void* but prints the output. The method's signature is as follows:



```
public static String arrayToString(int[] array);
```

Also write a test driver to test this method (you should test on empty array, one-element array, and n-element array).

Notes

- (a) This is similar to the built-in function *Arrays.toString()*. You could study its source code.

5. `contains()`

Write a boolean method called *contains()*, which takes an array of *int* and an *int*; and returns *true* if the array contains the given *int*. The method's signature is as follows:



```
1 public static boolean contains(int[] array, int key);
```

Also write a test driver to test this method.

6. **search()**

Write a method called *test search()*, which takes an array of *int* and an *int*; and returns the array index if the array contains the given *int*; or -1 otherwise. The method's signature is as follows:



```
1 public static int search(int[] array, int key);
```

Also write a test driver to test this method.

7. **equals()**

Write a boolean method called *equals()*, which takes two arrays of *int* and returns true if the two arrays are exactly the same (i.e., same length and same contents). The method's signature is as follows:



```
1 public static boolean equals(int[] array1, int[] array2)
```

Also write a test driver to test this method.

8. **copyOf()**

Write a boolean method called *copyOf()*, which takes an *int* Array and returns a copy of the given array. The method's signature is as follows:



```
1 public static int[] copyOf(int[] array)
```

Also write a test driver to test this method.

Write another version for *copyOf()* which takes a second parameter to specify the length of the new array. You should truncate or pad with zero so that the new array has the required length.



```
1 public static int[] copyOf(int[] array, int newLength)
```

Notes: This is similar to the built-in function *Arrays.copyOf()*.

9. swap()

Write a method called *swap()*, which takes two arrays of *int* and swap their contents if they have the same length. It shall return *true* if the contents are successfully swapped. The method's signature is as follows:



```
1 public static boolean swap(int[] array1, int[] array2)
```

Also write a test driver to test this method.

Notes



```
1 // Swap item1 and item2
  int item1;
3 int item2;
  int temp;
5
  temp = item1;
7 item1 = item2;
  item2 = temp;
9 // You CANNOT simply do: item1 = item2; item2 = item2;
```

10. reverse()

Write a method called *reverse()*, which takes an array of *int* and reverse its contents. For example, the reverse of *[1, 2, 3, 4]* is *[4, 3, 2, 1]*. The method's signature is as follows:



```
1 public static void reverse(int[] array)
```

Take note that the array passed into the method can be modified by the method (this is called **pass-by-sharing**). On the other hand, primitives passed into a

method cannot be modified. This is because a clone is created and passed into the method instead of the original copy (this is called **pass-by-value**).

Also write a test driver to test this method.

Hints

- (a) You might use two indexes in the loop, one moving forward and one moving backward to point to the two elements to be swapped.



```
1 for (int fIdx = 0, bIdx = array.length - 1; fIdx < bIdx; ++fIdx, --bIdx) {
    // Swap array[fIdx] and array[bIdx]
3 // Only need to transverse half of the array elements
}
```

- (b) You need to use a temporary location to swap two storage locations.



```
1 // Swap item1 and item2
  int item1;
3 int item2;
  int temp;
5
  temp = item1;
7 item1 = item2;
  item2 = item1;
9 // You CANNOT simply do: item1 = item2; item2 = item2;
```

11. GradesStatistics

Write a program called **GradesStatistics**, which reads in n grades (of int between 0 and 100, inclusive) and displays the average, minimum, maximum, median and standard deviation. Display the floating-point values upto 2 decimal places. Your output shall look like:

Command window

```
Enter the number of students: 4
2 Enter the grade for student 1: 50
Enter the grade for student 2: 51
4 Enter the grade for student 3: 56
Enter the grade for student 4: 53
```

Command window

```

1 The grades are: [50, 51, 56, 53]
  The average is: 52.50
3 The median is: 52.00
  The minimum is: 50
5 The maximum is: 56
  The standard deviation is: 2.29

```

The formula for calculating standard deviation is: $\sigma = \sqrt{\frac{1}{n} \sum_{i=0}^{n-1} x_i^2 - \mu^2}$, where μ is the mean.

Hints

```

/**
2  * GradesStatistics .java
  */
4  public class GradesStatistics {
    public static int[] grades; // Declare an int [], to be allocated later.
6    // This array is accessible by all the methods.

8    public static void main(String[] args) {
        readGrades(); // Read and save the inputs in static int[] grades
10       System.out.println("The grades are: ");
        print(grades);
12       System.out.println("The average is " + average(grades));
        System.out.println("The median is " + median(grades));
14       System.out.println("The minimum is " + min(grades));
        System.out.println("The maximum is " + max(grades));
16       System.out.println("The standard deviation is " + stdDev(grades));
    }

18    // Prompt user for the number of students and allocate
20    // the static "grades" array.
    // Then, prompt user for grade, check for valid grade, and store in "grades".
22    public static void readGrades() { ..... }

24    // Print the given int array in the form of [x1, x2, x3 ,..., xn].
    public static void print(int[] array) { ..... }

26    // Return the average value of the given int []
28    public static double average(int[] array) { ..... }

30    // Return the median value of the given int []
    // Median is the center element for odd-number array,
32    // or average of the two center elements for even-number array.
    // Use Arrays.sort(anArray) to sort anArray in place.

```




```

34 public static double median(int[] array) { ..... }

36 // Return the maximum value of the given int []
public static int max(int[] array) {
38     int max = array[0]; // Assume that max is the first element
    // From second element, if the element is more than max,
40     // set the max to this element.
    .....
42 }

44 // Return the minimum value of the given int []
public static int min(int[] array) { ..... }

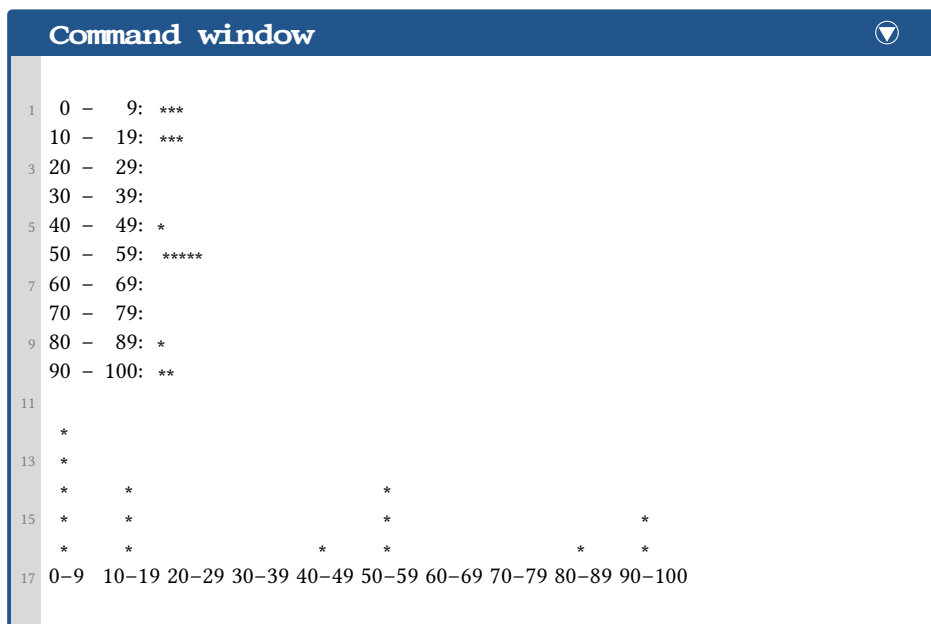
46
// Return the standard deviation of the given int []
48 public static double stdDev(int[] array) { ..... }
}

```

Take note that besides *readGrade()* that relies on class variable *grades*, all the methods are self-contained general utilities that operate on any given array.

2.11.16 GradesHistogram

Write a program called **GradesHistogram**, which reads in n grades (as in the previous exercise), and displays the horizontal and vertical histograms. For example:



2.12

Command-Line Arguments

Java's `main(String[] args)` method takes an argument: `String[] args`, i.e., a `String` array named `args`. This is known as "command-line arguments", which corresponds to the arguments 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):

Command window

```
1 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 `args`. For this example, `args` has the following properties:



```
1 args = {"12", "3456", "+"} // "args" is a String array
  args.length = 3           // length of the array args
3
4 args[0] = "12"             // Each element of the array is a String
5 args[1] = "3456"
  args[2] = "+"
7
8 args[0].length() = 2       // length of the String
9 args[1].length() = 4
  args[2].length() = 1
```

2.12.1 Code Example: Arithmetic

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

Command window

```
java Arithmetic 3 2 +
2 3 + 2 = 5
java Arithmetic 3 2 -
4 3 - 2 = 1
java Arithmetic 3 2 /
6 3/2 = 1
```



```

/**
2  * Arithmetic.java
  * The Arithmetic class
4  */
public class Arithmetic {
6  public static void main (String [] args) {
    int operand1 = Integer.parseInt (args [0]); // Convert String to int
8    int operand2 = Integer.parseInt (args [1]);
    char theOperator = args [2].charAt(0); // Consider only 1st character
10   System.out.print (args [0] + args [2] + args [1] + "=");
    switch(theOperator) {
12     case ('+'):
        System.out.println (operand1 + operand2);
14     break;
    case ('-'):
16     System.out.println (operand1 - operand2);
        break;
18     case ('*'):
        System.out.println (operand1 * operand2);
20     break;
    case ('/'):
22     System.out.println (operand1 / operand2);
        break;
24     default :
        System.out.printf ("%nError: Invalid operator!");
26   }
  }
28 }

```

2.12.2 Exercises on Command-Line Arguments

1. Arithmetic

Write a program called Arithmetic that takes three command-line arguments: two integers followed by an arithmetic operator (+, -, * or /). The program shall perform the corresponding operation on the two integers and print the result. For example,

Command window

```

java Arithmetic 3 2 +
3 + 2 = 5

java Arithmetic 3 2 -
3 - 2 = 1

```

Command window

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

Hints

The method `main(String[] args)` takes an argument: "an array of *String*", which is often (but not necessary) named `args`. This parameter captures the command-line arguments supplied by the user when the program is invoked. For example, if a user invokes:

Command window

```
java Arithmetic 12345 4567 +
```

The three command-line arguments "12345", "4567" and "+" will be captured in a *String* array "12345", "4567", "+" and passed into the `main()` method as the argument `args`. That is,

Command window

```
1 args is: {"12345", "4567", "+"} // args is a String array  
  args.length is: 3 // length of the array  
3 args[0] is: "12345" // 1st element of the String array  
  args[1] is: "4567" // 2nd element of the String array  
5 args[2] is: "+" // 3rd element of the String array  
  args[0].length() is: 5 // length of 1st String element  
7 args[1].length() is: 4 // length of the 2nd String element  
  args[2].length() is: 1 // length of the 3rd String element
```



```
/**  
2 * Arithmetic.java  
*/  
4 public class Arithmetic {  
    public static void main (String[] args) {  
6         int operand1;  
          int operand2;  
8         char theOperator;  
  
10        // Check if there are 3 command-line arguments in the
```



```

12 // String[] args by using length variable of array .
13 if (args.length != 3) {
14     System.err.println("Usage: java Arithmetic int1 int2 op");
15     return ;
16 }
17
18 // Convert the 3 Strings args [0], args [1], args [2] to int and char .
19 // Use the Integer.parseInt(aStr) to convert a String to an int .
20 operand1 = Integer.parseInt(args [0]);
21 operand2 = .....
22
23 // Get the operator, assumed to be the first character of
24 // the 3rd string . Use method charAt() of String .
25 theOperator = args [2].charAt(0);
26 System.out.print(args [0] + args [2] + args [1] + "=");
27 switch (theOperator) {
28     case ('-'):
29         System.out.println(operand1 - operand2);
30         break;
31     case ('+'):
32         .....
33     case ('*'):
34         .....
35     case ('/'):
36         .....
37     default :
38         System.err.println("Error: invalid operator!");
39 }
40 }

```

Notes

- To provide command-line arguments, use the "cmd" or "terminal" to run your program in the form "java *ClassName* *arg1 arg2*".
- To provide command-line arguments in NetBeans, right click the "Project" name → "Set Configuration" → "Customize..." → Select categories "Run" → Enter the command-line arguments, e.g., "3 2 +" in the "Arguments" box (but make sure you select the proper Main class).

Question: Try "java *Arithmetic* 2 4 *" (in CMD shell and Eclipse/NetBeans/IntelliJ IDEA) and explain the result obtained. How to resolve this problem?

In Windows' CMD shell, * is known as a wildcard character, that expands to give the list of file in the directory (called Shell Expansion). For example, "dir *.java" lists all the file with extension of ".java". You could double-quote the * to prevent shell expansion.

2.13

(Advanced) Bitwise Operations

2.13.1 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 | Mode | Usage | Description |
|----------|--------|-------|-------------------------|
| ~ | Unary | ~x | Bitwise NOT (inversion) |
| & | Binary | x & y | Bitwise AND |
| | Binary | x y | Bitwise OR |
| ^ | Binary | x ^ y | Bitwise XOR |

Example



```
/**
2  * TestBitwiseOp.java
3  * The TestBitwiseOp class
4  */
5  public class TestBitwiseOp {
6      public static void main(String[] args) {
7          int x = 0xAAAA_5555; // a negative number (sign bit (msb) = 1)
8          int y = 0x5555_1111; // a positive number (sign bit (msb) = 0)
9          System.out. printf ("%d\n", x); // -1431677611
10         System.out. printf ("%d\n", y); // 1431638289
11         System.out. printf ("%08X\n", ~x); // 5555AAAAH
12         System.out. printf ("%08X\n", x & y); // 00001111H
13         System.out. printf ("%08X\n", x | y); // FFFF5555H
14         System.out. printf ("%08X\n", x ^ y); // FFFF4444H
15     }
16 }
```

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 *booleans*. When both operands are integers, they perform bitwise operations. When both operands are *booleans*, 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 *booleans* only.



```
System.out.println (true & true);    // logical -> true
2 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
2 System.out.println (true ^ true & false); // true ^ (true & false) -> true
```

Bitwise operations are powerful and yet extremely efficient.

2.13.2 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 | Mode | Usage | Description |
|----------|--------|------------|--|
| << | Binary | x << count | Left-shift and padded with zeros |
| >> | Binary | x >> count | Right-shift and padded with sign bit (signed-extended right-shift) |
| >>> | Binary | x >>> y | 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



```

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

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

27     int i2 = -12345;
       System.out.println ("i2 divides by 2 is " + (i2 >> 1));
29     System.out.println ("i2 divides by 4 is " + (i2 >> 2));
       System.out.println ("i2 divides by 8 is " + (i2 >> 3));
31 }
}

```

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.

2.13.3 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 *booleans*, perform logical operations. Bit-shift operators are not applicable to *booleans*.

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*).

2.14

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.

2.14.1 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} .

PSEUDO-CODE

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



```
8 }  
}
```

Try: translate the above pseudocode into a Java program called **PrimeTest**.

2.14.2 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 $10 \neq 1 + 2 + 5$. Other perfect numbers are 28, 496, ...

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



```
1 // To test whether int x is a perfect number  
  int sum = 0;  
3 for (int i = 1; i < x; ++i) {  
    if (x is divisible by i) {  
5      i is a proper divisor;  
      add i into the sum;  
7    }  
  }  
9  
  if (sum == x) {  
11    x is a perfect number  
  } else {  
13    x is not a perfect number  
  }
```

Try: translate the above pseudocode into a Java program called **PerfectNumberTest**.

2.14.3 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 $a \geq b$, the Euclidean algorithm is based on these two properties:



1. $\text{GCD}(a, 0) = a$
2. $\text{GCD}(a, b) = \text{GCD}(b, a \bmod b)$, where " $a \bmod b$ " denotes the remainder of a divides by b .

For example,



- $\text{GCD}(15, 5) = \text{GCD}(5, 0) = 5$
- 2 $\text{GCD}(99, 88) = \text{GCD}(88, 11) = \text{GCD}(11, 0) = 11$
- $\text{GCD}(3456, 1233) = \text{GCD}(1233, 990) = \text{GCD}(990, 243) = \text{GCD}(243, 18)$
- 4 $\phantom{\text{GCD}(3456, 1233)} = \text{GCD}(18, 9) = \text{GCD}(9, 0) = 9$

The Euclidean algorithm is as follows:



```

GCD(a, b)  // assume that a >= b
2 while (b != 0) {
    // Change the value of a and b: a ← b, b ← a mod b, and repeat until b is 0
4   temp ← b
   b ← a mod b
6   a ← temp
}
8 // 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.



```

1 int x = 55;
  int y = 66;
3 // swap the values of x and y
  x = y;
5 y = x;
  
```

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



```
1 int x = 55;
  int y = 66;
3 int temp;
  // swap the values of x and y
5 temp = y;
  y = x;
7 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 $(a \bmod b)$, and compute $GCD(b, a \bmod b)$. 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.

2.14.4 Exercises on Algorithm - Number Theory

1. Perfect and Deficient Numbers

A positive integer is called a perfect number if the sum of all its factors (excluding the number itself, i.e., 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 $10 \neq 1 + 2 + 5$.

A positive integer is called a deficient number if the sum of all its proper divisors is less than its value. For example, 10 is a deficient number because $1 + 2 + 5 < 10$; while 12 is not because $1 + 2 + 3 + 4 + 6 > 12$.

Write a boolean method called *isPerfect(int aPosInt)* that takes a positive integer, and return true if the number is perfect. Similarly, write a boolean method called *isDeficient(int aPosInt)* to check for deficient numbers.



```
1 public static boolean isPerfect (int aPosInt);
  public static boolean isDeficient (int aPosInt);
```

Using the methods, write a program called **PerfectNumberList** that prompts user for an upper bound (a positive integer), and lists all the perfect numbers less than or equal to this upper bound. It shall also list all the numbers that are neither deficient nor perfect. The output shall look like:

```
Command window
Enter the upper bound: 1000
2 These numbers are perfect :
6 28 496
4 [3 perfect numbers found (0.30%)]

6 These numbers are neither deficient nor perfect :
12 18 20 24 30 36 40 42 48 54 56 60 66 70 72 78 80 .....
8 [246 numbers found (24.60%)]
```

2. Prime Numbers

A positive integer is a prime if it is divisible by 1 and itself only. Write a boolean method called *isPrime(int aPosInt)* that takes a positive integer and returns *true* if the number is a prime. Write a program called **PrimeList** that prompts the user for an upper bound (a positive integer), and lists all the primes less than or equal to it. Also display the percentage of prime (rounded to 2 decimal places). The output shall look like:

```
Command window
Please enter the upper bound: 10000
2 1
2 2
4 3
.....
6 .....
.....
8 9967
9973
10 [1230 primes found (12.30 %)]
```

Hints

To check if a number n is a prime, the simplest way is try dividing n by 2 to $\text{sqrt}(n)$.

3. Prime Factors

Write a boolean method called *isProductOfPrimeFactors(int aPosInt)* that takes a positive integer, and return *true* if the product of all its prime factors (excluding 1 and the number itself) is equal to its value. For example, the method returns true for 30 ($30 = 2 \times 3 \times 5$) and false for 20 ($20 \neq 2 \times 5$). You may need to use the *isPrime()* method in the previous exercise.

Write a program called **PerfectPrimeFactorList** that prompts user for an upper bound. The program shall display all the numbers (less than or equal to the upper bound) that meets the above criteria. The output shall look like:

```

Command window
Enter the upper bound: 100
2 These numbers are equal to the product of prime factors :
6 10 14 15 21 22 26 30 33 34 35 38 39 42 46 51 55 57 58 62 65 66 69 70
4 74 77 78 82 85 86 87 91 93 94 95
[36 numbers found (36.00%)]

```

4. Greatest Common Divisor (GCD)

One of the earlier known algorithms is the Euclid algorithm to find the GCD of two integers (developed by the Greek Mathematician Euclid around 300BC). By definition, $\text{GCD}(a, b)$ is the greatest factor that divides both a and b . Assume that a and b are positive integers, and $a \geq b$, the Euclid algorithm is based on these two properties:



- 1 $\text{GCD}(a, 0) = a$
 $\text{GCD}(a, b) = \text{GCD}(b, a \bmod b)$, where $(a \bmod b)$ denotes the remainder of a divides \hookrightarrow by b .

For example,



- 1 $\text{GCD}(15, 5) = \text{GCD}(5, 0) = 5$
- 2 $\text{GCD}(99, 88) = \text{GCD}(88, 11) = \text{GCD}(11, 0) = 11$
- 4 $\text{GCD}(3456, 1233) = \text{GCD}(1233, 990) = \text{GCD}(990, 243) = \text{GCD}(243, 18) = \text{GCD}(18, 9) = \text{GCD}(9, 0) = 9$

The pseudocode for the Euclid algorithm is as follows:



- ```

GCD(a, b) // assume that a >= b
2 while (b != 0) {
 // Change the value of a and b: a <- b, b <- a mod b, and repeat until b is 0

```



```

4 temp <- b
 b <- a mod b
6 a <- temp
 }
8 // after the loop completes, i.e., b is 0, we have GCD(a, 0)
 GCD is a

```

Write a method called *gcd()* with the following signature:



```

1 public static int gcd(int a, int b)

```

Your methods shall handle arbitrary values of *a* and *b*, and check for validity.

## 2.15

## Summary

This chapter covers the Java programming basics:

- Comments, Statements and Blocks.
- Variables, Literals, Expressions.
- The concept of type and Java's eight primitive types: byte, short, int, long, float, double, char, and boolean; and String.
- 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.

## 2.16 Exercises

### 2.16.1 Exercises on Algorithm - Recursion

In programming, a recursive function (or method) calls itself. The classical example is  $\text{factorial}(n)$ , which can be defined recursively as  $f(n) = n * f(n-1)$ . Nonetheless, it is important to take note that a recursive function should have a terminating condition (or base case), in the case of factorial,  $f(0) = 1$ . Hence, the full definition is:



```
1 factorial (n) = 1, for n = 0
 factorial (n) = n * factorial (n - 1), for all n > 0
```

For example, suppose  $n = 5$ :



```
// Recursive call
2 factorial (5) = 5 * factorial (4)
 factorial (4) = 4 * factorial (3)
4 factorial (3) = 3 * factorial (2)
 factorial (2) = 2 * factorial (1)
6 factorial (1) = 1 * factorial (0)
 factorial (0) = 1 // Base case
8
// Unwinding
10 factorial (1) = 1 * 1 = 1
 factorial (2) = 2 * 1 = 2
12 factorial (3) = 3 * 2 = 6
 factorial (4) = 4 * 6 = 24
14 factorial (5) = 5 * 24 = 120 (DONE)
```

#### 1. Factorial Recursive

Write a recursive method called *factorial()* to compute the factorial of the given integer.



```
public static int factorial (int n)
```

The recursive algorithm is:





```
factorial (n) = 1, for n = 0
2 factorial (n) = n * factorial (n-1), for all n > 0
```

Compare your code with the iterative version of the *factorial()*:



```
1 factorial (n) = 1 * 2 * 3 * \dots * n
```

### Hints

Writing recursive function is straight forward. You simply translate the recursive definition into code with return.



```
// Return the factorial of the given integer, recursively
2 public static int factorial (int n) {
 if (n == 0) {
4 return 1; // base case
 } else {
6 return n * factorial (n-1); // call itself
 }
8
 // or one liner
10 // return (n == 0) ? 1 : n* factorial (n-1);
}
```

or



```
// Return the factorial of the given integer, recursively
2 public static int factorial (int n) {
 if (n == 0) {
4 return 1; // base case
 }
6
 return n * factorial (n-1); // call itself
8
 // or one liner
10 // return (n == 0) ? 1 : n* factorial (n-1);
}
```

### Notes

- (a) Recursive version is often much shorter.
- (b) The recursive version uses much more computational and storage resources, and it needs to save its current states before each successive recursive call, so as to unwind later.

## 2. Fibonacci (Recursive)

Write a recursive method to compute the Fibonacci number of  $n$ , defined as follows:


$$\begin{aligned} F(0) &= 0 \\ F(1) &= 1 \\ F(n) &= F(n-1) + F(n-2) \text{ for } n \geq 2 \end{aligned}$$

Compare the recursive version with the iterative version written earlier.

### Hints



```
1 /*
2 * Translate the recursive definition into code with return statements
3 */
4 public static int fibonacci (int n) {
5 if (n == 0) {
6 return 0;
7 } else if (n == 1) {
8 return 1;
9 } else {
10 return fibonacci (n-1) + fibonacci (n-2);
11 }
12 }
```

OR



```
1 /*
2 * Translate the recursive definition into code with return statements
3 */
4 public static int fibonacci (int n) {
5 if (n == 0) {
6 return 0;
7 }
8 }
```



```

7 }

9 if (n == 1) {
 return 1;
11 }

13 return fibonacci(n-1) + fibonacci(n-2);
 }

```

### 3. Length of a Running Number Sequence (Recursive)

A special number sequence is defined as follows:



```

1 S(1) = 1
 S(2) = 12
3 S(3) = 123
 S(4) = 1234
5
 S(9) = 123456789 // length is 9
7 S(10) = 12345678910 // length is 11
 S(11) = 1234567891011 // length is 13
9 S(12) = 12345678910112 // length is 15


```

Write a recursive method to compute the length of  $S(n)$ , defined as follows:



```

1 len(1) = 1
2 len(n) = len(n-1) + numOfDigits(n)

```

### 4. GCD (Recursive)

Write a recursive method called *gcd()* to compute the greatest common divisor of two given integers.



```

1 public static void int gcd(int a, int b)

```



```
gcd(a, b) = a, if b = 0
2 gcd(a, b) = gcd(b, remainder(a,b)), if b > 0
```

## 2.16.2 Exercises on Algorithm - Searching and Sorting

### Sorting and Searching

Efficient sorting and searching are big topics, typically covered in a course called "Data Structures and Algorithms". There are many searching and sorting algorithms available, with their respective strengths and weaknesses. See Wikipedia "Sorting Algorithms" and "Searching Algorithms" for the algorithms, examples and illustrations.

JDK provides searching and sorting utilities in the Arrays class (in package *java.util*), such as *Arrays.sort()* and *Arrays.binarySearch()* - you don't have to write your searching and sorting in your production program. These exercises are for academic purpose and for you to gain some understandings and practices on these algorithms.

#### 1. Recursive Binary Search

(Reference: Wikipedia "Binary Search") Binary search is only applicable to a sorted list. For example, suppose that we want to search for the item 18 in the list {1114161820252830344045}:

Create two indexes: *firstIdx* and *lastIdx*, initially pointing at the first and last elements

{ 11 14 16 18 20 25 28 30 34 40 45 }

F                      M                      L

Compute *middleIdx* = (*firstIdx* + *lastIdx*) / 2

Compare the key (K) with the middle element (M)

If K = M, return true

else if K < M, set *firstIdx* = *middleIdx*

else if K > M, set *lastIdx* = *middleIdx*

{ 11 14 16 18 20 25 28 30 34 40 45 }

F      M              L

Recursively repeat the search between the new *firstIndex* and *lastIndex*.

Terminate with not found when *firstIndex* = *lastIndex*.

{ 11 14 16 18 20 25 28 30 34 40 45 }

F M      L

Write a recursive function called *binarySearch()* as follows:



```

1 // Return true if key is found in the array in the range of fromIdx (inclusive)
 // to toIdx (exclusive)
3 public boolean binarySearch(int [] array, int key, int fromIdx, int toIdx)

```

Use the following pseudocode implementation:



```

 If fromIdx = toIdx - 1 // Terminating one-element list
2 if key = array[fromIdx], return true
 else , return false (not found)
4 else
 middleIdx = (fromIdx + toIdx) / 2
6 if key = array[middleIdx], return true
 else if key < array[middleIdx], toIdx = middleIdx
8 else firstIdx = middleIdx + 1
 binarySearch(array, key, fromIdx, toIdx) // recursive call

```

Also write an overloaded method which uses the above to search the entire array:



```

 // Return true if key is found in the array
2 public boolean binarySearch(int [] array, int key)

```

Write a test driver to test the methods.

## 2. Linear Search

Write the following linear search methods to search for a key value in an array, by comparing each item with the search key in the linear manner. Linear search is applicable to unsorted list. (Reference: Wikipedia "Linear Search".)



```

1 // Return true if the key is found inside the array
 public static boolean linearSearch(int [] array, int key)
3
 // Return the array index, if key is found; or 0 otherwise
5 public static int linearSearchIndex(int [] array, int key)

```

Also write a test driver to test the methods.

### 3. Bubble Sort

(Reference: Wikipedia "Bubble Sort") The principle of bubble sort is to scan the elements from left-to-right, and whenever two adjacent elements are out-of-order, they are swapped. Repeat the passes until no swap are needed.

For example, given the list {9 2 4 1 5}, to sort in ascending order:

Pass 1:

9 2 4 1 5 → 2 9 4 1 5

2 9 4 1 5 → 2 4 9 1 5

2 4 9 1 5 → 2 4 1 9 5

2 4 1 9 5 → 2 4 1 5 9 (After Pass 1, the largest item sorted on the right - bubble to the right)

Pass 2:

2 4 1 5 9 → 2 4 1 5 9

2 4 1 5 9 → 2 1 4 5 9

2 1 4 5 9 → 2 1 4 5 9

2 1 4 5 9 → 2 1 4 5 9 (After Pass 2, the 2 largest items sorted on the right)

Pass 3:

2 1 4 5 9 → 1 2 4 5 9

1 2 4 5 9 → 1 2 4 5 9

1 2 4 5 9 → 1 2 4 5 9

1 2 4 5 9 → 1 2 4 5 9 (After Pass 3, the 3 largest items sorted on the right)

Pass 4:

1 2 4 5 9 → 1 2 4 5 9

1 2 4 5 9 → 1 2 4 5 9

1 2 4 5 9 → 1 2 4 5 9

1 2 4 5 9 → 1 2 4 5 9 (After Pass 4, the 4 largest items sorted on the right)

No Swap in Pass 4. Done.

See Wikipedia "Bubble Sort" for more examples and illustration.

Write a method to sort an int array (in place) with the following signature:



```
1 public static void bubbleSort(int[] array)
```

Use the following pseudocode implementation:



```

// The function to sort an array using the bubble sort algorithm
2 function bubbleSort(array)
 n = length(array)
4 boolean swapped // boolean flag to indicate swapping occurred during a
 ↪ pass
 do {
6 swapped = false // reset for each pass
 for (i = 1; i < n; ++i) {
8 // Swap if this pair is out of order
 if array[i-1] > array[i] {
10 swap(A[i-1], A[i])
 swapped = true // update flag
12 }
 }
14 n = n - 1 // One item sorted after each pass
 } while (swapped) // repeat another pass if swapping occurred, otherwise
 ↪ done

```

#### 4. Insertion Sort

(Reference: Wikipedia "Insertion Sort") Similar to the selection sort, but extract the leftmost element from the right-unsorted-sublist, and insert into the correct location of the left-sorted-sublist.

For example, given {9 6 4 1 5 2 7}, to sort in ascending order:

```

{} {9 6 4 1 5 2 7} → {9} {6 4 1 5 2 7}
{9} {6 4 1 5 2 7} → {6 9} {4 1 5 2 7}
{6 9} {4 1 5 2 7} → {4 6 9} {1 5 2 7}
{4 6 9} {1 5 2 7} → {1 4 6 9} {5 2 7}
{1 4 6 9} {5 2 7} → {1 4 5 6 9} {2 7}
{1 4 5 6 9} {2 7} → {1 2 4 5 6 9} {7}
{1 2 4 5 6 9} {7} → {1 2 4 5 6 7 9} {}
{1 2 4 5 6 7 9} {} → Done

```

Write a method to sort an int array (in place) with the following signature:



```
public static void insertionSort (int[] array)
```

## 5. Selection Sort

(Reference: Wikipedia "Selection Sort") This algorithm divides the lists into two parts: the left-sublist of items already sorted, and the right-sublist for the remaining items. Initially, the left-sorted-sublist is empty, while the right-unsorted-sublist is the entire list. The algorithm proceeds by finding the smallest (or largest) items from the right-unsorted-sublist, swapping it with the leftmost element of the right-unsorted-sublist, and increase the left-sorted-sublist by one.

For example, given the list {9 6 4 1 5}, to sort in ascending order:

```
9 6 4 1 5 → 1 6 4 9 5
1 6 4 9 5 → 1 4 6 9 5
1 4 6 9 5 → 1 4 5 9 6
1 4 5 9 6 → 1 4 5 6 9
1 4 5 6 9 → DONE
1 4 5 6 9
```

Write a method to sort an int array (in place) with the following signature:



```
public static void selectionSort (int [] array)
```

## 2.16.3 More Exercises

### 1. Matrices (2D Arrays)

Similar to **Math** class, write a **Matrix** library that supports matrix operations (such as addition, subtraction, multiplication) via 2D arrays. The operations shall support both *double* and *int*. Also write a test class to exercise all the operations programmed.

#### Hints



```
1 /**
 * Matrix.java
 */
3 public class Matrix {
5 // Method signatures
 public static void print (int [][] matrix);
 public static void print (double [][] matrix);
7
9 // Used in add(), subtract ()
```





```

public static boolean haveSameDimension(int[][] matrix1, int[][] matrix2);
11 public static boolean haveSameDimension(double[][] matrix1,
 double[][] matrix2);

13

public static int[][] add(int[][] matrix1, int[][] matrix2);
15 public static double[][] add(double[][] matrix1, double[][] matrix2);

17 public static int[][] subtract(int[][] matrix1, int[][] matrix2);
public static double[][] subtract(double[][] matrix1, double[][] matrix2);

19

public static int[][] multiply(int[][] matrix1, int[][] matrix2);
21 public static double[][] multiply(double[][] matrix1, double[][] matrix2);

.....
23 }

```

## 2. Trigonometric Series

Write a method to compute  $\sin(x)$  and  $\cos(x)$  using the following Taylor series expansion, in a class called **TrigonometricSeries**. The signatures of the methods are:



```

1 // x in radians, NOT degrees
public static double sin(double x, int numTerms);
3 public static double cos(double x, int numTerms);

```

- Taylor series expansion:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \dots$$

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \dots$$

Compare the values computed using the series with the JDK methods *Math.sin()*, *Math.cos()* at  $x = 0, \pi/6, \pi/4, \pi/3, \pi/2$  using various numbers of terms.

### Hints

Do not use int to compute the factorial; as factorial of 13 is outside the int range. Avoid generating large numerator and denominator. Use double to compute the terms as:

$$\frac{x^n}{n!} = \left(\frac{x}{n}\right) \left(\frac{x}{n-1}\right) \dots \left(\frac{x}{1}\right).$$

### 3. Exponential Series

Write a method to compute the sum of the series in a class called **SpecialSeries**. The signature of the method is:



```
1 public static double specialSeries (double x, int numTerms);
```

- Special series:

$$x + \frac{1}{2} \times \frac{x^3}{3} + \frac{1 \times 3}{2 \times 4} \times \frac{x^5}{5} + \frac{1 \times 3 \times 5}{2 \times 4 \times 6} \times \frac{x^7}{7} + \frac{1 \times 3 \times 5 \times 7}{2 \times 4 \times 6 \times 8} \times \frac{x^9}{9} + \dots;$$

$$-1 \leq x \leq 1.$$

4. **FactorialInt (Handling Overflow)** Write a program called **FactorialInt** to list all the factorials that can be expressed as an *int* (i.e., 32-bit signed integer in the range of  $[-2147483648, 2147483647]$ ). Your output shall look like:

#### Command window

```
1 The factorial of 1 is 1
 The factorial of 2 is 2
3 The factorial of 3 is 6
 ...
5 ...
 The factorial of 12 is 479001600
7 The factorial of 13 is out of range
```

### Hints

The maximum and minimum values of a 32-bit int are kept in constants *Integer.MAX\_VALUE* and *Integer.MIN\_VALUE*, respectively. Try these statements:



```
1 System.out.println (Integer.MAX_VALUE);
 System.out.println (Integer.MIN_VALUE);
3 System.out.println (Integer.MAX_VALUE + 1);
```

Take note that in the third statement, Java Runtime does not flag out an overflow error, but silently wraps the number around. Hence, you cannot use  $F(n) * (n + 1) > \text{Integer.MAX\_VALUE}$  to check for overflow. Instead, overflow occurs for

$F(n + 1)$  if  $(Integer.MAX\_VALUE / \text{Factorial}(n)) < (n + 1)$ , i.e., no more room for the next number.

### Try

Modify your program called **FactorialLong** to list all the factorial that can be expressed as a long (64-bit signed integer). The maximum value for long is kept in a constant called *Long.MAX\_VALUE*.

## 5. FibonacciInt (Handling Overflow)

Write a program called **FibonacciInt** to list all the Fibonacci numbers, which can be expressed as an int (i.e., 32-bit signed integer in the range of  $[-2147483648, 2147483647]$ ). The output shall look like:

Command window
⌵

```

1 F(0) = 1
 F(1) = 1
3 F(2) = 2
 ...
5 F(45) = 1836311903
 F(46) is out of the range of int

```

### Hints

The maximum and minimum values of a 32-bit int are kept in constants *Integer.MAX\_VALUE* and *Integer.MIN\_VALUE*, respectively. Try these statements:



```

1 System.out.println (Integer .MAX_VALUE);
2 System.out.println (Integer .MIN_VALUE);
 System.out.println (Integer .MAX_VALUE + 1);

```

Take note that in the third statement, Java Runtime does not flag out an overflow error, but silently wraps the number around. Hence, you cannot use  $F(n) = F(n - 1) + F(n - 2) > Integer.MAX\_VALUE$  to check for overflow. Instead, overflow occurs for  $F(n)$  if  $Integer.MAX\_VALUE \sim F(n - 1) < F(n - 2)$  (i.e., no more room for the next Fibonacci number).

### Try

Write a similar program called **TribonacciInt** for Tribonacci numbers.

## 6. Number System Conversion

Write a method call *toRadix()* which converts a positive integer from one radix into another. The method has the following header:



```
1 // The input and output are treated as String .
 public static String toRadix(String in, int inRadix, int outRadix)
```

Write a program called **NumberConversion**, which prompts the user for an input string, an input radix, and an output radix, and display the converted number. The output shall look like:

```
Command window
Enter a number and radix: A1B2
2 Enter the input radix: 16
Enter the output radix: 2
4 "A1B2" in radix 16 is "1010000110110010" in radix 2.
```

## 7. NumberGuess

Write a program called **NumberGuess** to play the number guessing game. The program shall generate a random number between 0 and 99. The player inputs his/her guess, and the program shall response with "Try higher", "Try lower" or "You got it in n trials" accordingly. For example,

```
Command window
java NumberGuess
2 Key in your guess:
50
4 Try higher
70
6 Try lower
65
8 Try lower
61
10 You got it in 4 trials !
```

### Hints

Use *Math.random()* to produce a random number in double between 0.0 (inclusive) and 1.0 (exclusive). To produce an int between 0 and 99, use:



```
1 final int SECRET_NUMBER = (int)(Math.random()*100); // truncate to int
```

## 8. WordGuess

Write a program called **WordGuess** to guess a word by trying to guess the individual characters. The word to be guessed shall be provided using the command-line argument. Your program shall look like:

### Command window

```
1 java WordGuess testing
Key in one character or your guess word: t
3 Trial 1: t__t__
Key in one character or your guess word: g
5 Trial 2: t__t_g
Key in one character or your guess word: e
7 Trial 3: te_t_g
Key in one character or your guess word: testing
9 Congratulations!
You got in 4 trials
```

### Hints

- Set up a *boolean* array (of the length of the word to be guessed) to indicate the positions of the word that have been guessed correctly.
- Check the length of the input *String* to determine whether the player enters a single character or a guessed word. If the player enters a single character, check it against the word to be guessed, and update the boolean array that keeping the result so far.

### Try

Try retrieving the word to be guessed from a text file (or a dictionary) randomly.

## 9. DateUtil

Complete the following methods in a class called **DateUtil**:

- boolean isLeapYear(int year)*: returns *true* if the given 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.

- *boolean isValidDate(int year, int month, int day)*: returns true if the given year, month and day constitute a given date. Assume that year is between 1 and 9999, month is between 1 (Jan) to 12 (Dec) and day shall be between 1 and 28|29|30|31 depending on the month and whether it is a leap year.
- *int getDayOfWeek(int year, int month, int day)*: returns the day of the week, where 0 for SUN, 1 for MON, ..., 6 for SAT, for the given date. Assume that the date is valid.
- *String toString(int year, int month, int day)*: prints the given date in the format "xxxday d mmm yyyy", e.g., "Tuesday 14 Feb 2012". Assume that the given date is valid.

**Hints**

To find the day of the week (Reference: Wiki "Determination of the day of the week"):

- (a) Based on the first two digit of the year, get the number from the following "century" table.

| 1700- | 1800- | 1900- | 2000- | 2100- | 2200- | 2300- | 2400- |
|-------|-------|-------|-------|-------|-------|-------|-------|
| 4     | 2     | 0     | 6     | 4     | 2     | 0     | 6     |

Take note that the entries 4, 2, 0, 6 repeat.

- (b) Add to the last two digit of the year.
- (c) Add to "the last two digit of the year divide by 4, truncate the fractional part".
- (d) Add to the number obtained from the following month table:

|               | Jan | Feb | Mar | Apr | May | Jun | Jul | Aug | Sep | Oct | Nov | Dec |
|---------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| Non-Leap Year | 0   | 3   | 3   | 6   | 1   | 4   | 6   | 2   | 5   | 0   | 3   | 5   |
| Leap Year     | 6   | 2   | 3   | 6   | 1   | 4   | 6   | 2   | 5   | 0   | 3   | 5   |

- (e) Add to the day.
- (f) The sum modulus 7 gives the day of the week, where 0 for SUN, 1 for MON, ..., 6 for SAT.

For example: 2012, Feb, 17



```
1 (6 + 12 + 12/4 + 2 + 17) % 7 = 5 (Fri)
```

The skeleton of the program is as follows:



```
1 /* *
 * DateUtil.java
 * Utilities for Date Manipulation
 */
5 public class DateUtil {
 // Month's name - for printing
6 public static String[] strMonths
 = {"Jan", "Feb", "Mar", "Apr", "May", "Jun",
7 "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};
9
11 // Number of days in each month (for non-leap years)
 public static int[] daysInMonths
13 = {31, 28, 31, 30, 31, 30, 31, 30, 31, 30, 31};
15
16 // Returns true if the given year is a leap year
 public static boolean isLeapYear(int year) { }
17
18 // Return true if the given year, month, day is a valid date
19 // year: 1-9999
20 // month: 1(Jan)-12(Dec)
21 // day: 1-28|29|30|31. The last day depends on year and month
 public static boolean isValidDate(int year, int month, int day) { }
23
24 // Return the day of the week, 0:Sun, 1:Mon, ..., 6:Sat
25 public static int getDayOfWeek(int year, int month, int day) { }
27
28 // Return String "xxxday d mmm yyyy" (e.g., Wednesday 29 Feb 2012)
 public static String printDate(int year, int month, int day) { }
29
30 // Test driver
31 public static void main(String[] args) {
 System.out.println(isLeapYear(1900)); // false
33 System.out.println(isLeapYear(2000)); // true
 System.out.println(isLeapYear(2011)); // false
35 System.out.println(isLeapYear(2012)); // true
37
 System.out.println(isValidDate(2012, 2, 29)); // true
 System.out.println(isValidDate(2011, 2, 29)); // false
39 System.out.println(isValidDate(2099, 12, 31)); // true
 System.out.println(isValidDate(2099, 12, 32)); // false
41
 System.out.println(getDayOfWeek(1982, 4, 24)); // 6: Sat
```



```

43 System.out.println (getDayOfWeek(2000, 1, 1)); // 6: Sat
 System.out.println (getDayOfWeek(2054, 6, 19)); // 5: Fri
45 System.out.println (getDayOfWeek(2012, 2, 17)); // 5: Fri

47 System.out.println (toString (2012, 2, 14)); // Tuesday 14 Feb 2012
 }
49 }

```

### Notes

You can compare the day obtained with the Java's Calendar class as follows:



```

1 // Construct a Calendar instance with the given year, month and day
 // month is 0-based
3 Calendar cal = new GregorianCalendar(year, month - 1, day);

5 // Get the day of the week number: 1 (Sunday) to 7 (Saturday)
 int dayNumber = cal.get(Calendar.DAY_OF_WEEK);
7 String [] calendarDays = { "Sunday", "Monday", "Tuesday", "Wednesday",
 "Thursday", "Friday", "Saturday" };
9 // Print result
 System.out.println ("It is " + calendarDays[dayNumber - 1]);

```

The calendar we used today is known as Gregorian calendar, which came into effect in October 15, 1582 in some countries and later in other countries. It replaces the Julian calendar. 10 days were removed from the calendar, i.e., October 4, 1582 (Julian) was followed by October 15, 1582 (Gregorian). The only difference between the Gregorian and the Julian calendar is the "leap-year rule". In Julian calendar, every four years is a leap year. In Gregorian calendar, a leap year is a year that is divisible by 4 but not divisible by 100, or it is divisible by 400, i.e., the Gregorian calendar omits century years which are not divisible by 400. Furthermore, Julian calendar considers the first day of the year as march 25th, instead of January 1st.

This above algorithm work for Gregorian dates only. It is difficult to modify the above algorithm to handle pre-Gregorian dates. A better algorithm is to find the number of days from a known date.



## **Part II    Object-Oriented Program- ming**



## **Part III   Collections Framework**



# **Part IV    Correctness, Robustness, Efficiency**



# **Part V   Design Patterns**





## **Part VI   Java Generics**



## **Part VII   Java Concurrency**



## **Part VIII   Java GUI**



## **Part IX    References**

