

INTRODUCTION

Brief History of Java

James Gosling et al. of Sun Microsystems created the Oak programming language in honor of the tree found in Gosling's window. Since there is already a programming language named Oak, he later changed it to Java.

Java was created as a platform independent language which could be embedded in various consumer electronic products like toasters and refrigerators. Star 7, a personal hand-held remote control, was one of the first projects developed using Java.

With the popularity of the World Wide Web and the Internet, the creators of Java realized that the programming language could be used to design transactional web applications.

The Java Technology

Java is a combination of independent but inter-related technologies working as one in order to achieve a specific purpose. The technologies embedded in Java are the following:

1. Programming Language
 - Java can generate all kinds of application that can be created using conventional programming languages.
2. Development Environment
 - Java provides the developer tools such as a compiler, an interpreter, a document generator, and a class file packaging tool to name a few.
3. Application Environment
 - Java applications are general-purpose programs running on a machine where the Java runtime environment (JRE) is installed.
4. Deployment Environment
 - JRE supplies a Software Development Kit (SDK) that contains a complete set of class files for all types Java technology packages.
 - Most commercial web browsers supply a Java technology interpreter and runtime environment.

Features of Java

Enumerated are some features of Java Technology.

1. Java Virtual Machine (JVM)
 - An imaginary machine that is utilized to emulate software on a real machine providing hardware platform specifications which enables the Java software to be platform-independent since compilation is done for a generic machine, the JVM.
2. Garbage Collection
 - The garbage collection thread is responsible for deallocating memory previously allocated by the programmer. In earlier programming languages, freeing memory that was used is the responsibility of the programmer.
3. Code Security
 - JRE runs the code compiled for a JVM and performs class loading, code verification, and code execution.
 - The Class Loader is responsible for loading all classes needed for the Java Program
 - The byte code verifier tests the format of the code fragments and checks the code fragments for illegal code that can violate access rights to objects.
 - Lastly, code is executed.

Tools Used in Java Programming

Table 1. Summary of Phases of a Java Program

Activity	Tool to use
Writing the Program	NetBeans, JCreator or any Text Editor
Compiling the Program	Java Compiler
Executing the Program	Java Interpreter

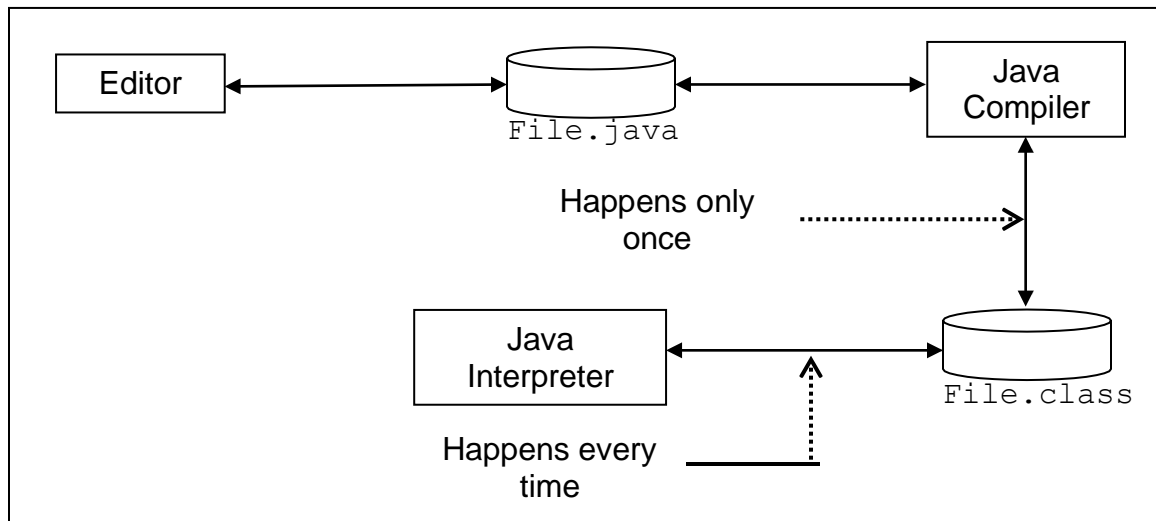
Phases of a Java Program

First phase in creating a Java program is writing the program in a text editor and store it with an extension filename of `.java`.

After creating and saving the Java program, compile the program by using the Java compiler resulting in Java bytecodes with extension filename `.class`.

The `.class` file is then interpreted by the Java interpreter that converts the bytecodes into a machine language of the computer being used.

Figure 1. Phases of a Java Program



UNIT 1

The Basics of a Java Program

1. MY FIRST JAVA PROGRAM

Typically, when learning a new programming language, the first program created is how to display on the computer screen the text **"Hello World!"**

We can write the codes in Java using any text editor by typing the following lines:

```
/* My first Java Program */

public class Hello{
    public static void main(String[] args){

        //prints the string hello world on the screen
        System.out.println("Hello World!");
    }
}
```

The program shown in the example above is called a *program source code*. The source code is made of letters, numbers, and other symbols available in the ASCII format. Or in a more technical term, it is an aggregate of characters.

After writing the source code, the next step to do is to COMPILE and RUN it in order to see whether we have achieved our objective. If there are errors reported by the compiler (also known as syntax errors), then there is a need to debug (fix) the source code before it is recompiled and tested again.

The output of the program above should show on the screen:

```
Hello World!
—
```

1.1. Setting the Paths

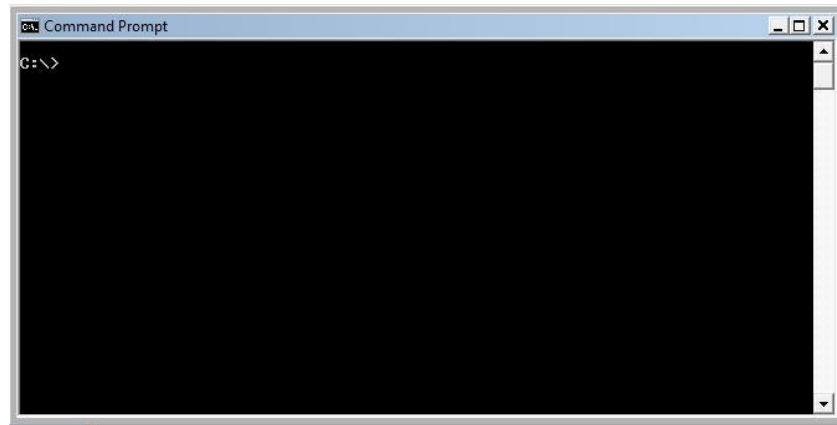
Paths for the Java compiler and Java classes must be set first before compiling and executing a Java program from the command line.

To set the paths in a MS Windows environment, the following steps are done:

Step 1: Open the command prompt by clicking Programs → Run

Step 2: Type `cmd` on the Run window and click Ok. The command prompt window should appear.

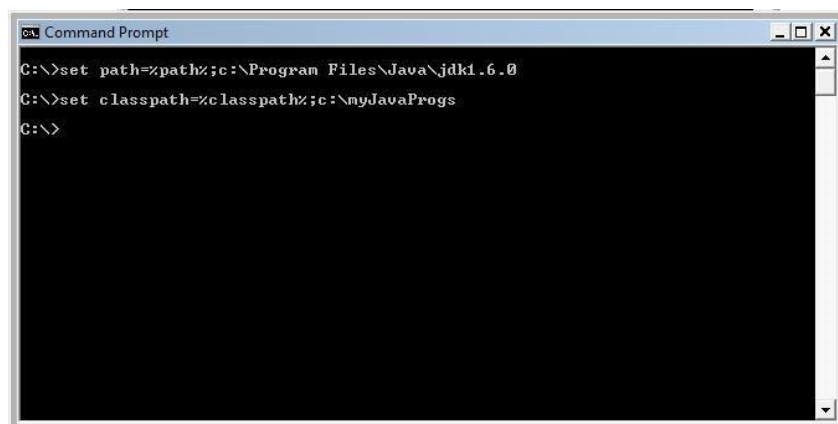
Figure 2. Command Line Interface



Step 3: Set the path for the Java Development Kit (JDK) to be able to use the functions needed to compile and execute Java programs by typing `set path=%path%;c:\Program Files\jdk1.6.0\bin` on the command line.

Step 4: A class path is where all generated classes of a Java program is placed whenever a Java program is compiled. To set the class path type `set classpath=%classpath%;classpath1;classpath2;...`, where *classpath* is your working folder.

Figure 3. Setting Java Path and Class Path



Note: The Java and class paths for Ubuntu Linux is already set and no further modification to the paths is necessary.

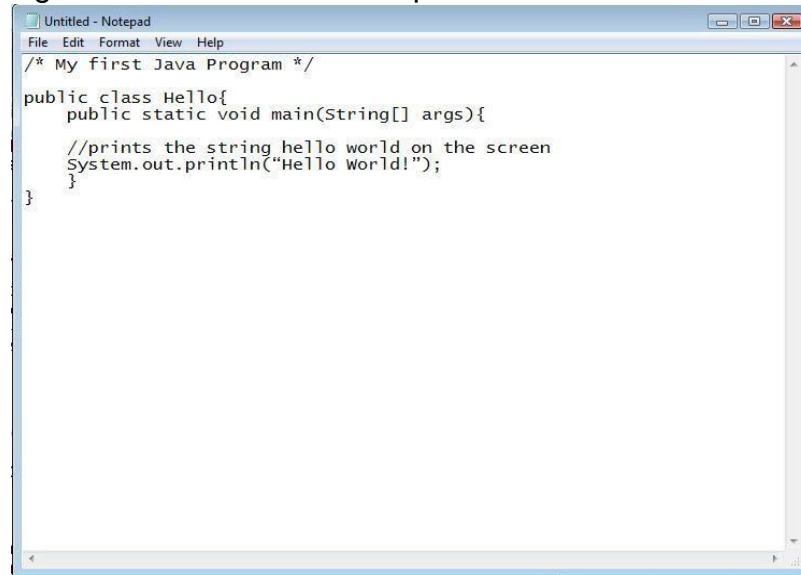
1.2. Using a Text Editor

A Java source code can be encoded in any text editor (notepad for MS Windows and gEdit for Ubuntu Linux among others). Below are simple steps on how to encode source code in a text editor.

Step 1: Open MS Windows Notepad, or gEdit for Ubuntu Linux

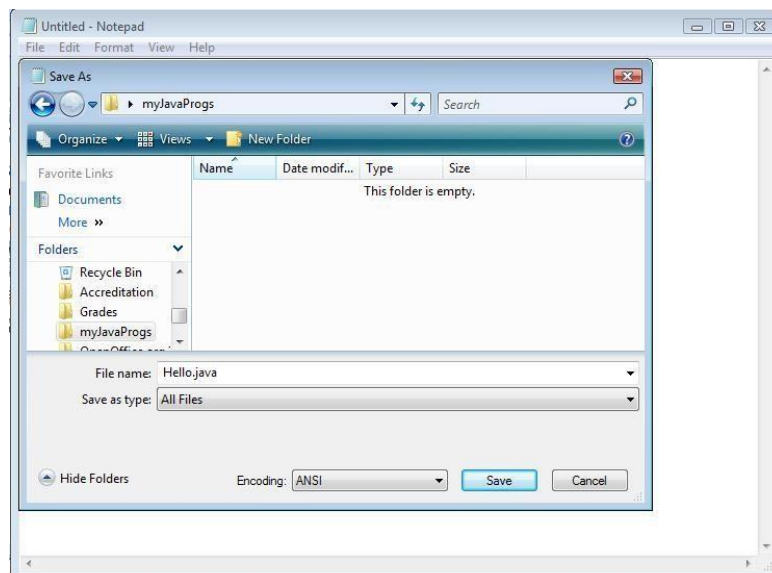
Step 2: Type the Java source code on the text editor.

Figure 4. MS Windows Notepad



Step 3: Save the file as `Hello.java` (note: the class name should be the same as the file name when saving Java source code).

Figure 5. Save As Hello.java



1.3. Compiling and Executing a Java Source Code

After encoding the source code and saving it to its proper filename, it is now ready for compilation (generation of the bytecodes) and execution (running the program to view the output).

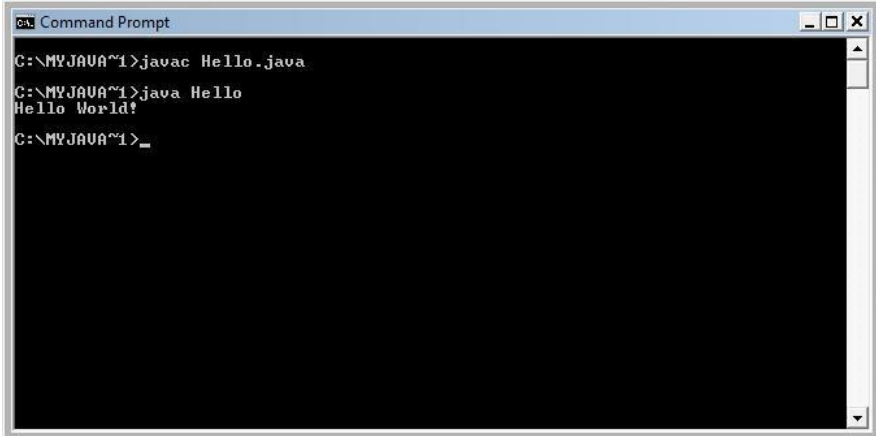
To compile and execute a Java program, follow these steps:

Step 1: Go to the Command prompt.

Step 2: Type `javac Hello.java` to compile

Step 3: Execute the program by typing `java Hello`

Figure 6. Compiling and Executing the Java program



```
Command Prompt
C:\MYJAVAA~1>javac Hello.java
C:\MYJAVAA~1>java Hello
Hello World!
C:\MYJAVAA~1>_
```

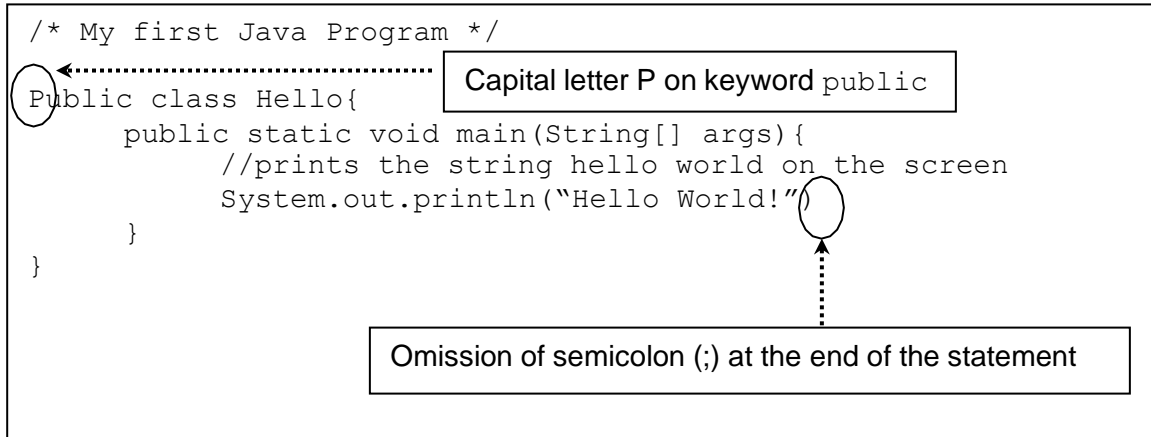
1.4. Errors

Errors in programming may be encountered along the way. There are two (2) types of errors encountered during programming, syntax errors and run-time errors.

1.4.1. Syntax Errors

Syntax errors are errors in form. These are encountered if a programmer inadvertently committed an error in typing the source code.

Syntax errors are checked and reported by the compiler. Common mistakes include capitalization, wrong spelling, the use of incorrect special characters, and omission of correct punctuation.



1.4.2. Run-Time Errors

Run-time errors are errors in meaning (semantic). These are sometimes referred to as logical errors.

Run-time errors are not detected and reported by the compiler making it difficult to debug. A program might be syntactically correct, but semantically wrong. Even if the programmer followed the syntax correctly, the program may display wrong answers if the logical processes and structures of the program is not carefully thought through.

2. THE JAVA PROGRAM STRUCTURE

Consider the `Hello.java` program:

```

/* My first Java Program */

public class Hello
{
    public static void main(String[] args){
        //prints the string hello world on the screen
        System.out.println("Hello World!");
    }
}

```

`/* My first Java Program */` The first line of code indicates a Java comment or remark. A comment is used to document a part of the code. It is not part of the program itself and is only used for documentation purposes to make the program more readable.

A comment is indicated by delimiters “/*” and “*/” for multi-line comments. Anything written in between these delimiters are considered comments. A double forward slash, //, is also used if only one (1) line is to be commented.

```
/* My first Java Program */
```

or

```
/* My  
   First  
   Java  
   Program */
```

or

```
// My first Java Program
```

`public class Hello` indicates the name of the class which is `Hello`. All codes should be written inside the class declaration. A class declaration is done by using the keyword `class` preceded by an access modifier that indicates the accessibility of the class to other classes. On this example, the access modifier `public` was used indicating that the class is accessible to other classes from other packages.

{ an open curly brace indicates the start of a block. On the `Hello.java` example, the open curly brace is placed after the class declaration but it can also be placed immediately after the class declaration as follows:
`public class Hello {`

`public static void main(String[] args){`, can also be written as, `public static void main (String args[]){`, indicates the name of a method, particularly, the `main` method. The `main` method is where a Java program starts.

`//prints the string Hello World! on the screen,` is also a Java comment.

`System.out.println("Hello World!");` displays the text enclosed in double quotes on the screen and the cursor is shown in the next line.

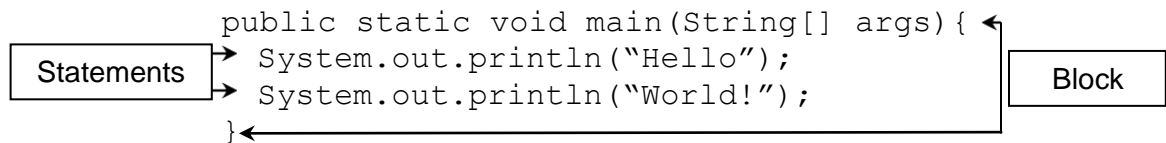
} The first close curly brace indicates the end of the `main` method.

} The last close curly brace indicates the end of the `Hello` class.

2.1. Java Statements and Blocks

A *statement* or an instruction is one or more lines of code terminated by a semicolon.

A *block* is one or more statements bounded by an open and close curly braces that groups the statements as one unit.



2.2. Java Identifiers

Identifiers are tokens. These are usually user-defined names that represent labels of variables, methods, and classes to name a few. Identifiers in Java are *case-sensitive*, which means that it should be used the way it was written (capital letters are not the same with small letters).

Example: age is not the same as Age

Identifiers cannot be any of the reserved words or key words used in Java.

2.2.1. Java Conventions in Declaring Identifiers

When declaring identifiers, some rules or conventions must be observed in order to avoid syntax and run-time errors. Such conventions are the following:

1. Names are made up of letters and numbers.
2. The first character in a name should always start with a letter.
3. Java is case sensitive. Meaning 'A' is not the same as 'a'.
4. The underscore symbol `_` and dollar sign `$` are considered to be letters in Java, but are not recommended to be used as the first character in a name; rather it is used as a connector instead.

2.2.2. Java Coding Convention

1. For names of classes, capitalize the first letter of the class name.
2. For method and variable names, the first letter of the word should start with a small letter.

Example: `ThisIsAnExampleOfClassName`
`thisIsAnExampleOfMethodName`
`thisIsAnExampleOfVariableName`

3. For multi-word identifiers, capitalize the first letter of each word except the first word.

Example: `charArray`
`fileNumber`

Examples of Valid Identifiers:

`a`
`A`
`age`
`num1`
`xyz`
`final_grade`
`employee33`
`id_Number`
`firstName`
`sum`
`aReA5b3h1`

Examples of Invalid Identifiers:

<code>6</code>	Must start with a letter
<code>1st_Name</code>	Must start with a letter
<code>u&me</code>	& symbol is not a letter or number
<code>percent%</code>	% symbol is not a letter or number
<code>last name</code>	Space is not a letter or number

2.2.3. Java Keywords

Keywords are predefined identifiers reserved by Java for specific purposes. Further, Java keywords cannot be used as identifiers.

Table 2. List of Java Keywords

Java Keywords				
abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

2.3. Java Literals and Constants

Literals and constants are tokens whose values do not change. In Java, there are 5 kinds of literals.

2.3.1. Integer Literals

Integer literals can be represented in *decimal* (base 10), *hexadecimal* (base 16) and *octal* (base 8) number systems. Certain notations should be followed when using integer literals in a program.

For decimal numbers, there is no special notation, the number is written as it is. For hexadecimal numbers, it should be preceded by 0x, while octal numbers are preceded by 0.

Example:

```
12 (decimal)
0xC (hexadecimal)
014 (octal)
```

2.3.2. Floating Point Literals

Represent decimals with fractional parts that can be expressed in standard or scientific notations.

Example:

```
3.1416
54.567
5.8234e2
10.2000e4
```

2.3.3. Boolean Literals

Boolean literals have only two (2) values. It is either *“True”* or *“False”*.

2.3.4. Character Literals

Characters represent a single Unicode character. It is a 16-bit character set which allows the inclusion of symbols and special characters from other languages. Character should be enclosed in single quote delimiters.

Example:

<code>'a'</code>	Letter a
<code>'Z'</code>	Letter Z
<code>'\n'</code>	New line character
<code>'\b'</code>	Carriage return character

2.3.5. String Literals

Represents multiple characters and are enclosed by double quotes.

Example:

```
"Hello World"  
"Java Programming"
```

2.4. Primitive Data Types

In Java, data is stored by setting aside space for it in memory by giving it a name. The space allotted for the data depends on the kind of data being stored. Data types determine the range of values the variable can have and set of operations that are defined for values of this type.

A data type specifies:

- The kind of values that can be assumed by a variable of that type
- The range of values that can be assumed by a variable of that type
- The amount of memory (in bytes) needed by a variable to store a value of that type

2.4.1. Logical – Boolean

A boolean data type stores only one of two values, either `“true”` or `“false”`.

Example:

```
boolean state = true;
boolean value;
```

The first example above declares a variable named `state` having a `boolean` data type assigned a default value of `true`. On the second example, the name of the variable is `value` of type `boolean` without any default value.

2.4.2. Textual – char

The `char` data type represents a single Unicode character. When assigning literals to a `char` data type, the values must be enclosed in single quotes.

Example:

```
char letter = 'A';
char choice;
```

2.4.3. Integral – byte, short, int, and long

Integral data type uses three forms – decimal, octal or hexadecimal. The default type is `int`, but it can also be defined as `byte`, `long` or `short`. The difference lies on the length of values that can be stored on the variable (refer to Table 3).

Example:

```
byte no = 10;
int number = 500;
short x;
long z;
```

Table 3. Integral types and ranges

Integer Length	Name or Type	Range	Value
8 bits	Byte	-2^7 to 2^7-1	-127 to 126
16 bits	Short	-2^{15} to $2^{15}-1$	-32,768 to 32,767
32 bits	Int	-2^{31} to $2^{31}-1$	-2,147,483,648 to 2,147,483,647
64 bits	Long	-2^{63} to $2^{63}-1$	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

2.4.4. Floating Point – float and double

Uses `double` as default data type which includes either a decimal point or one of the following:

```
E or e    //(add exponential value)
F or f    //(float)
D or d    //(double)
```

Example:

```
float pi = 3.14.16f;
double decimal = 123456.73e10D;
```

Table 4. Floating point types and ranges

Integer Length	Name or Type	Range	Value
32 bits	Float	-2^{31} to $2^{31}-1$	3.4E-38 to 3.4E+38
64 bits	Double	-2^{63} to $2^{63}-1$	1.7E-308 to 1.7E+308

2.5. String Class

A string contain series or multiple characters whose literal is enclosed in double quotes (" "). In Java, the class `String` is not a primitive data type, instead, it is referred to as a *reference type*. It is used to declare a variable which can hold multiple characters.

Example:

```
String subject = "ICS2";
String college = "CITCS";
String name;
```

2.6. Program Variables

Variables are entities where data can be stored into it. Values stored in the variable can be changed anytime. It is an abstraction of the computer memory cell or collection of cells

A variable can be characterized as a sextuple (six parts) of attributes:

1. Name - more specifically a symbolic name and often referred to as identifiers.

2. Address – the memory address with which it is associated (portion in the RAM)
3. Value – the contents of the memory cell or cells associated with it.
4. Data Type - determines the range of values the variable can have and set of operations that are defined for values of this type.
5. Lifetime – the time during which the variable is bound to a specific memory location.
6. Scope – the scope of such a variable is from its declaration to the **end** reserved word of its procedure.

Naming convention used in variables is the same as what was discussed in section 2.2

2.6.1. Declaring and Initializing Variables

Syntax for declaring a variable:

```
<data type> <name> [=initial value];
```

- Values enclosed in < > are required values
- Values enclosed in [] are optional values

As a general coding guidelines:

1. It is always good to initialize variables as they are declared.
2. Use descriptive names and not just some random letters you choose.
3. Declare one variable per line.

Sample Program that declares and initializes variables:

```
public class VariableSamples{
    public static void main (String[] args){
        /*declaration of a variable with
        boolean as data type and ask as
        name*/
        Boolean ask;

        /*declaration of a variable with char
        as data type and letter as name*/
```



```
char letter;
letter = 'A'; // assign A to letter

/*declaration of variable with double
data type and grade as name
initialized to 90.0*/
double grade = 90.0;

/*declaration of variable with String
type and collegeName as name
initialized to CITCS*/
String college = "CITCS";
}
```

2.6.2. Outputting Variable Data

To produce output on the screen, we can use the following commands:

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

Sample program:

```
public class OutputVariables{
    public static void main(String[] args){
        int number = 100;
        char letter;
        letter = 'A';

        System.out.println(number);
        System.out.println("The letter is = " + letter);
    }
}
```

When compiled and executed, the program will produce the following output on the screen:

```
100
The letter is = A
```

2.6.3. System.out.println() and System.out.print()

The difference between `System.out.println()` and `System.out.print()` lies on how the output will be printed on screen.

The first command appends a newline at the end of the data, while the latter does not.

Consider the statements:

```
System.out.println("Hello");
System.out.println("World");
```

The output on the screen would be:

```
Hello
World_
```

Now, take a look at the following statements:

```
System.out.print("Hello ");
System.out.print("World");
```

The output generated would be:

```
Hello World_
```

2.6.4. Primitive and Reference Variables

Primitive variables are variables with primitive data types which stores data in the actual memory location of where the variable is found.

Reference variables are variables that store the address of the variable and points to another memory location where the data is found.

Figure 7. Representation of Primitive and Reference variables in memory

Variable Declaration	Memory Address	Data
int number = 100;	1001	100

double pi = 3.1416;	1347	3.1416

char letter = 'A';	1567	A

String text = "Hello World"	1871	Address (2000)

	2000	"Hello World"

3. Operators

Operators are symbols representing operations that can be performed on constants and variables.

3.1. Assignment Operator

The operator used in assignment operation is the equal symbol (=). This is use to store or assign a value from the right-hand side (RHS) of an expression to the left-hand side (LHS).

Example:

LHS = RHS

This is interpreted as the value on the RHS is stored or assigned to the LHS, or it can also be interpreted as the LHS gets the value of the RHS.

It is good programming practice that once you use the assignment operator, it is best to read it as “GETS THE VALUE OF” and not “EQUAL TO”. “Equal to” is not an assignment operator, but a relational operator.

The assignment operation is also called an assignment statement; thus should be terminated by a semicolon.

The syntax or proper way of writing assignment operation is:

`<variable name> = <expression>;`

Example:

```
age = 5;           /* age GETS THE VALUE OF 5 */
x = x + 1;         /* x GETS THE VELUE OF x + 1 */
```

Example of a complete program using assignment operation:

```
public class AssignmentStatement{
    public static void main(String[] args){
        char ch;
        int i;
        double d;
        String s;

        ch = 'A';
        i = 6;
        d = 3.14159212345876;
        s = "Hello";
    }
}
```

Note that it is also possible to assign a value of a variable to another variable of compatible data type.

Example:

```
X = 5;
Y = X; Z = Y;
```

3.1.1. Storing Values with Different Types

The Java programming language is a *strongly typed* language, which means that every variable and every expression has a type that is known at compile time. Types limit the values that a variable can hold or that an expression can produce, limit the operations supported on those values, and determine the meaning of the operations. Strong typing helps detect errors at compile time.

Even though Java is a *strongly typed* language, there are still automatic conversions that can take place among different data types. Such conversions are: from `int` to `double/float` and vice versa; `char` to `int` and vice versa; `char` to `double/float` and vice versa.

Example:

```
int i = 65;
double d = 14.50987;
char c = 'B';
```

Based on the following statements, assuming that the statement `i = d` was written after the declarations, it would not produce a syntax error, rather, it would simply convert the double value into an integer value stripping it from any decimal place. `i` now would have a value of 14.

Table 5. Value interpretation based on example above

Assignment Statements	Interpretation
<code>i = d</code>	<code>i</code> Gets the Value 14
<code>d = i</code>	<code>d</code> Gets the Value 65.0
<code>i = c</code>	<code>i</code> Gets the Value 66 (66 is the Unicode integer equivalent of letter B)
<code>d = c</code>	<code>d</code> Gets the Value 66.0 (66.0 is the Unicode double equivalent of letter B)
<code>c = i</code>	Syntax Error (Possible Loss of Precision)
<code>c = d</code>	Syntax Error (Possible Loss of Precision)

3.2. Arithmetic Operations

Arithmetic operators are also known as mathematical operators. The basic arithmetic operations available in Java are:

+	addition
-	subtraction
*	multiplication
/	division
%	modulo (yields the remainder)

The +, -, *, / can be used for operands of type `int`, `float`, and `double` data types. The % operator can be used only with integer operands. All of these operators are called binary operators because they require two operands.

Example: Arithmetic operation on integer data type:

```
// Assignment and Arithmetic operations

public class AssignArithOperators{
    public static void main(String[] args)
    {
        int a, b, c, d, e;
        a = 3 + 5;
        b = 45 - 23;
        c = 6 * 3;
        d = 32 / 4;
        e = 33 % 5;

        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
        System.out.println("d = " + d);
        System.out.println("e = " + e);
    }
}
```

The / operator when used with operand whose data types are of integer values will result to integer division which truncates or discards any fractional part. If the fractional part is needed, then at least one of the operand should be `float` or `double` data type.

3.2.1. Precedence and Associativity Rule

When evaluating multiple arithmetic operation, the *, /, and % operators are evaluated first before + and -. The usual evaluation order for operators of the same precedence is from left to right.

Table 7. Arithmetic Operators and their Precedence

Operator	Associativity
*, /, %	Left to Right
+, -	Left to Right

To override precedence, use a pair of parentheses to enclose expression that needs to be prioritized first. For multiple parenthesized expression, innermost parenthesized expressions are to be evaluated first.

Example:

$$\begin{aligned}
 X &= 3 + 6 * 2 - 5 + 10 / 2 * 8 / 2 - 3 \\
 X &= 3 + 12 - 5 + 10 / 2 * 8 / 2 - 3 \\
 X &= 3 + 12 - 5 + 5 * 8 / 2 - 3 \\
 X &= 3 + 12 - 5 + 40 / 2 - 3 \\
 X &= 3 + 12 - 5 + 20 - 3 \\
 X &= 15 - 5 + 20 - 3 \\
 X &= 10 + 20 - 3 \\
 X &= 30 - 3 \\
 X &= 27
 \end{aligned}$$

Example:

$$\begin{aligned}
 X &= (3 + 6) * (2 - (5 + (10 / 2))) * 8 / 2 - 3 \\
 X &= (3 + 6) * (2 - (5 + 5) * 8) / 2 - 3 \\
 X &= (3 + 6) * (2 - 10 * 8) / 2 - 3 \\
 X &= (3 + 6) * (2 - 80) / 2 - 3 \\
 X &= 9 * (2 - 80) / 2 - 3 \\
 X &= 9 * -78 / 2 - 3 \\
 X &= -702 / 2 - 3 \\
 X &= -351 - 3 \\
 X &= -354
 \end{aligned}$$

3.3. Increment and Decrement Operators

Aside from basic arithmetic operators, Java also includes a unary increment (++) and unary decrement (--) operators that can be written before or after a variable.

Example:

```
count = count + 1; //increment value of count by 1
count++;          //increment value of count by 1
count = count - 1; //decrement value of count by 1
count--;          //decrement value of count by 1
```

Table 8. Increment and Decrement usage and description

Usage	Description
var++	Increments var by 1; evaluates to the value of var prior to incrementing
++var	Increments var by 1; evaluates to the value of var after it was incremented
var--	Decrements var by 1; evaluates to the value of var prior to decrementing
--var	decrements var by 1; evaluates to the value of var after it was decremented

When the increment or decrement operator is placed before an operand, it causes the variable to be incremented or decremented first and the new value will be used in the expression.

Example:

```
int i = 10;
int j = 3;
int k = 0;

k = ++j + i;    //expression is k = 4 + 10
```

When the increment or decrement operator is placed after an operand, the old value will be used in the expression.

Example:

```
int i = 10;
int j = 3;
int k = 0;

k = j++ + i;    //expression is k = 3 + 10
```

3.4. Relational Operators

Relational operations are used to check association of the LHS of the expression to the RHS. The result of the operation will yield whether the condition being tested is either TRUE or FALSE. The basic relational operators available in Java are as follows:

<code>==</code>	Equal to
<code>!=</code>	Not equal to
<code>></code>	Greater than
<code><</code>	Less than
<code>>=</code>	Greater than or equal to
<code><=</code>	Less than or equal to

Relational operators can be performed on all the basic data types available in Java. It is very important to remember that relational operations should always be enclosed in a pair of parenthesis.

As discussed earlier, the result of a relational operation will yield to either `true` or `false` only.

It is important to remember that the test for equality uses two equal signs. Forgetting one of the equal sign is a very common logical (not syntactical) error which can be very difficult to find (debug) in a fairly large program.

Example:

<code>x = 8</code>	
<code>y = 13</code>	
<code>a = (x == y)</code>	→ result is False
<code>b = (x != y)</code>	→ result is True
<code>c = (x > y)</code>	→ result is False
<code>d = (x < y)</code>	→ result is True
<code>e = (x >= y)</code>	→ result is False
<code>f = (x <= y)</code>	→ result is True

Example: Relational Operators

```
// Relational Operations

public class RelationalOperators{
    public static void main(String[] args)
    {
        boolean a, b, c, d, e, f;
        int x, y;

        x = 8;
        y = 13;

        a = (x == y);
        b = (x != y);
        c = (x > y);
        d = (x < y);
        e = (x >= y);
        f = (x <= y);

        System.out.println(a);
        System.out.println(b);
        System.out.println(c);
        System.out.println(d);
        System.out.println(e);
        System.out.println(f);
    }
}
```

3.5. Logical Operators

The logical operators are used to test multiple conditions and are normally used in conjunction with relational operators. Logical operators available in Java are the following:

!	Logical NOT
&&	Logical AND
&	boolean Logical AND
	Logical OR
	boolean Logical Inclusive OR
^	boolean Logical Exclusive OR

The general statement for a logical operation is:

```
<expr1> <Logical Operator> <expr1>
```

3.5.1. && (logical AND) and & (boolean logical AND)

The general idea behind the AND operation is that all expressions evaluated should be `true` in order to conclude that the statement is `true`; otherwise, the statement will be `false`.

Table 9. Truth table for the AND operator

expr1	expr2	Result
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

The basic difference between `&&` and `&` operators is that `&&` supports *short-circuit evaluations* (partial evaluation). If `&&` will evaluate `expr1` as `false`; the operator will not anymore evaluate `expr2` because the result will be `false` regardless of the value of `expr2`. In contrast, the `&` operator evaluates both expressions before returning the result.

Example:

```

/*Sample source code for logical AND and
boolean logical AND */

public class ANDOperator{
    public static void main(String[] args){
        int i = 0;
        int j = 10;
        boolean result;

        //&& operator
        result = (i>10) && (j>9);
        System.out.println(i);
        System.out.println(j);
        System.out.println(result);

        //& operator
        result = (i>10) & (j>9);
        System.out.println(i);
        System.out.println(j);
        System.out.println(result);
    }
}

```

The output of the given program would be:

```
0
10
false
0
10
false
```

3.5.2. || (logical OR) and | (boolean logical inclusive OR)

The general idea behind the OR operation is that at least one (1) expression evaluated should be `true` in order to conclude that the statement is `true`; otherwise, the statement will be `false`.

Table 10. Truth table for the OR operator

expr1	expr2	Result
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

The basic difference between `||` and `|` operators is that `||` supports *short-circuit evaluations* (partial evaluation). If `||` will evaluate `expr1` as `true`; the operator will not anymore evaluate `expr2` because the result will be `true` regardless of the value of `expr2`. In contrast, the `|` operator evaluates both expressions before returning the result.

Example:

```
/*Sample source code for logical OR and
boolean logical inclusive OR */

public class OROperator{
    public static void main(String[] args){
        int i = 0;
        int j = 10;
        boolean result;

        //|| operator
        result = (i<10) || (j<9);
        System.out.println(i);
        System.out.println(j);
        System.out.println(result);
    }
}
```

```

        //| operator
        result = (i<10) | (j<9);
        System.out.println(i);
        System.out.println(j);
        System.out.println(result);
    }
}

```

The output of the given program would be:

```

0
10
true
0
10
True

```

3.5.3. ^ (boolean logical exclusive OR)

The general idea behind the *exclusive OR* operation is that the statement is true if and only if one (1) operand is true while the other is false.

Table 11. Truth table for the exclusive OR operator

expr1	expr2	Result
TRUE	TRUE	FALSE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

Example:

```

/*Sample source code for boolean logical
exclusive OR */

public class ExclusiveOROperator{
    public static void main(String[] args){

        boolean val1 = true;
        boolean val2 = true;
        System.out.println(val1 ^ val2);

        val1 = false;
        System.out.println(val1 ^ val2);

        val2 = false;
        System.out.println(val1 ^ val2);
    }
}

```

```

        val1 = true;
        System.out.println(val1 ^ val2);
    }
}

```

The output of the given program would be:

```

false
true
false
true

```

3.5.4. !(logical NOT)

The logical NOT is a unary operator; it is used only on one operand. Logical NOT operator is used to negate or get the opposite of a certain result in a relational operation.

Table 12. Truth table for logical NOT operator

expr	Result
TRUE	FALSE
FALSE	TRUE

Example:

```

//Sample source code for logical NOT

public class NOTOperator{
    public static void main(String[] args){

        boolean val1 = true;
        boolean val2 = false;
        System.out.println(!val1);
        System.out.println(!val2);

    }
}

```

The output of the given program would be:

```

false
true

```

3.6. Conditional Operator (?:)

The conditional operator `?:` is a *ternary operator* since it takes in three (3) arguments that together form a conditional expression.

The general statement for a logical operation is:

`expr1 ? expr2 : expr3`

Where `expr1` is a boolean expression whose result must be either `true` or `false`.

The general idea in using the operator is that when `expr1` is `true`, value of `expr2` is returned, otherwise, `expr3` is returned.

Example:

```
//Sample source code for Conditional Operator ?:  
  
public class ConditionalOperator{  
    public static void main(String[] args){  
  
        String status;  
        int grade = 80;  
  
        //get status of student  
        Status = (grade >= 60)?"Pass":"Fail"  
  
        //print value of status  
        System.out.println(status);  
    }  
}
```

The output of the given program would be:

Pass

3.7. Operator Precedence

The precedence of operators defines the compiler's order of evaluation of operators so as to come up with an unambiguous result.

Table 13. Operator Precedence

Java Operators Precedence from Highest to Lowest			
.	[]	()	
++	--	!	~
*	/	%	
+	-		
<<	>>	<<<	>>>
<	>	<=	>=
==	!=		
&			
^			
&&			
? :			
=			

4. Getting Input from the Keyboard

Java programming language makes use of the `BufferedReader` class found in `java.io` package to obtain user input from the keyboard.

Steps in getting input from keyboard:

1. Add the statement `import java.io.*` on top of the code.
2. Add the statement `BufferedReader dataIn = new BufferedReader(new InputStreamReader(System.in));` in the `main()` block.
3. Declare a temporary `String` variable to hold the input value then invoke the `readLine()` method to get the input from the keyboard. The `readLine()` method must be type inside a try-catch block.

```
try{
    String input = dataIn.readLine();
}catch(IOException e){
    System.out.println("Error in getting input");
}
```

Example:

```
//Sample source code for getting input from keyboard

import java.io.*;

public class GetUserInput{
    public static void main(String[] args){

        String text;

        BufferedReader dataIn=new BufferedReader(new
            InputStreamReader(System.in));

        System.out.println("Enter Your Name: ");

        try{
            text=dataIn.readLine();
        }catch(IOException e){
            System.out.println("Error!");
        }

        System.out.println("Hello " + text + "!");
    }
}
```

To better understand the sample code written, let us explain the statements per line.

The statement `import java.io.*` indicate that we want to use all the classes found inside `java.io` package like `BufferedReader`, `InputStreamReader`, and `IOException` in the program. *Packages* contain classes that have related purposes.

The statements,

```
public class GetUserInput{
    public static void main(String[] args){
```

means we are declaring a class named `GetUserInput` and also declaring the `main()` method.

The next statement,

```
BufferedReader dataIn = new BufferedReader(new
    InputStreamReader(System.in));
```

is used to declare a variable named `dataIn` with the class type `BufferedReader`.

The next statement, `String text;`, is used to declare an identifier name `text` of type `String`.

The statement `System.out.println("Enter Your Name: ");` will output the `String` on the screen asking for the user's name.

The try-catch block,

```
try{
    text=dataIn.readLine();
}catch(IOException e){
    System.out.println("Error!");
}
```

assures that the possible exceptions that could occur in the statement `text=dataIn.readLine();` will be caught. Note that when using the `readLine()` method, it should be inside a try-catch block

The statement, `text=dataIn.readLine();` gets input from the user and will return a `String` value to be stored to the `text` variable.

The user input, whose value is stored on variable `text`, will be displayed in place of the text on the statement `System.out.println("Hello " + text + "!");`.

4.1. Inputting Other Data Types

The use of `BufferedReader` and `InputStreamReader`, together with the method `readLine()`, will obtain user input from keyboard and store it on a variable whose type is of `String` only.

When user wants to input values other than that of a `String` type, the user input must be *parsed* (converted) first to be used for its intended purpose.

4.1.1. Converting Input to Integer

Immediately after the `try-catch` block, include the statement `Integer.parseInt()` and assign it to an integer variable.

Example:

```
try{
    string=dataIn.readLine();
}catch(IOException e){
    System.out.println("Error!");
}
int number=Integer.parseInt(string);
```

4.1.2. Converting Input to Double

Immediately after the `try-catch` block, include the statement `Double.parseDouble()` and assign it to an integer variable.

Example:

```
try{
    string=dataIn.readLine();
}catch(IOException e){
    System.out.println("Error!");
}
double number=Double.parseDouble(string);
```

UNIT 2

Control Structures

1. Control Flows

Basically, control structures specify the sequence of execution of a group of statements. Some statement may need to be executed in sequential fashion, some may have to be repeated several times while others may or may not be executed depending on some condition.

1.1. Different types of control structures in JAVA

There are three available control structures in Java. These are as follows:

1. Sequential Control Structure
2. Decision Control Structures
3. Repetition Control Structures

2. Sequential Control Structures

A sequential control structure is organized such that statements are executed one after the other in the order of their appearance in the source code.

Example: Assuming that variables a, b, c, were declared as integer data types

```
a = 1;           // First Statement
b = 2;           // Second Statement
c = a + b;       // Third Statement
```

In the above example, the order by which execution occurs will be from the first statement, followed by the second statement, and finally the third statement.

3. Decision Control Structures

A decision control structure is organized in such a way that there should be a condition to be evaluated first. The behavior of the program will depend on the truthfulness of a condition being tested. The result of the evaluation will yield to either true or false condition. The result of the condition will then dictate the course of action to be taken.

3.1 if Statement

The syntax of an if-Statement:

```
if(boolean expression)
    statement;
```

or

```
if(boolean expression){
    statement1;
    statement2;
    . . .
    statementN;
}
```

Note that for multiple statements under an if-statement, it requires a pair of curly braces. But for a single statement, the curly brace becomes optional.

Statements under an if-statement will only be executed if and only if the condition being tested results to a true condition.

Example: Write a program that will accept an integer value and determine whether this is a POSITIVE number.

```
// if statement sample program
import java.io.*;

public class PositiveNumber{
    public static void main(String[] args){
        int num;
        String in;
        BufferedReader input = new BufferedReader(new
            InputStreamReader(System.in));
        System.out.print("Input a Number: ");
        try{
            in=input.readLine();
        }catch(IOException e){
            System.out.println("Error");
        }
        num = Integer.parseInt(in);
        if(num>=0)
            System.out.println("Number is POSITIVE");
    }
}
```

The sample program will evaluate in an integer input if it is POSITIVE. The boolean condition being tested must yield to a TRUE value in order for the statement under the if statement to be executed.

```
// if statement sample program
import java.io.*;

public class PositiveNumber{
    public static void main(String[] args){
        int num;
        String in;
        BufferedReader input = new BufferedReader(new
            InputStreamReader(System.in));
        System.out.print("Input a Number: ");
        try{
            in=input.readLine();
        }catch(IOException e){
            System.out.println("Error");
        }
        num = Integer.parseInt(in);
        if(num>=0)
            System.out.println("Number is POSITIVE");
        if(num<0)
            System.out.println("Number is NEGATIVE");
    }
}
```

The modified program above uses two (2) `if()` blocks, one to evaluate whether an input number is greater than or equal to zero hence a positive number, and one to evaluate whether an input number is lesser than zero which is a negative number. The program is not the best implementation of an `if()` statement because if you will trace it, both conditions are tested. Meaning, the program will test for the second condition still, even though the first condition has already been met. But first let us try to trace and simulate the program.

Simulation:

```
Input a number = 5
n = 5 is POSITIVE
```

—

Analysis:

When the user inputs 5, the value is then stored in the variable `in` prior to conversion to an integer value to be stored in the variable `num`. The program performs the first conditional statement `if (num>=0)`. Since it

results to TRUE, the `System.out.println()` under it will be executed next. It doesn't stop there (that's the problem)! The program will evaluate the second conditional statement `if(num<0)` even though it will always result to FALSE. The extra test results to inefficiency of the program.

Example: Write a program that will accept a number representing the day of the week. Thereafter, the program will display the equivalent day.

```
// Day program using if-statement only
import java.io.*;
public class DaysOfWeek{
    public static void main(String[] args){
        int day;
        String in;
        BufferedReader input = new BufferedReader(new
            InputStreamReader(System.in));
        System.out.print("Input a Number from 1-7: ");
        try{
            in=input.readLine();
        }catch(IOException e){
            System.out.println("Error");
        }
        day = Integer.parseInt(in);

        if(day == 1){
            System.out.println("That day is a Monday!");
            System.out.println("Have a nice day.");
        }
        if(day == 2){
            System.out.println("That day is a Tuesday!");
            System.out.println("Have a nice day.");
        }
        if(day == 3){
            System.out.println("That day is a Wednesday!");
            System.out.println("Have a nice day.");
        }
        if(day == 4){
            System.out.println("That day is a Thursday!");
            System.out.println("Have a nice day.");
        }
        if(day == 5){
            System.out.println("That day is a Friday!");
            System.out.println("Have a nice day.");
        }
        if(day == 6){
            System.out.println("That day is a Saturday!");
            System.out.println("Have a nice day.");
        }
        if(day == 7){
            System.out.println("That day is a Sunday!");
            System.out.println("Have a nice day.");
        }
    }
}
```

Analysis:

The day program is just one possible solution, and it is not really a good solution like the number program! If you'll notice, the statement **System.out.println("Have a nice day.")** is common to all if-statement and it is repeated seven times! This common instruction can actually be factored out. A better solution is given below (but not yet the best!).

```
/* Modified Day program using if-statement only */
import java.io.*;
public class DaysOfWeek{
    public static void main(String[] args){
        int day;
        String in;
        BufferedReader input = new BufferedReader(new
            InputStreamReader(System.in));
        System.out.print("Input a Number from 1-7: ");
        try{
            in=input.readLine();
        }catch(IOException e){
            System.out.println("Error");
        }
        day = Integer.parseInt(in);

        if(day == 1)
            System.out.println("That day is a Monday!");
        if(day == 2)
            System.out.println("That day is a Tuesday!");
        if(day == 3)
            System.out.println("That day is a Wednesday!");
        if(day == 4)
            System.out.println("That day is a Thursday!");
        if(day == 5)
            System.out.println("That day is a Friday!");
        if(day == 6){
            System.out.println("That day is a Saturday!");
        }
        if(day == 7){
            System.out.println("That day is a Sunday!");
        }

        System.out.println("Have a nice day.");
    }
}
```

Note that this program is better than the previous one. One of the desirable qualities that an aspiring programmer should have is the ability to factor out common instructions to reduce the perceived length and complexity of a program.

However, please note that this is still not the best solution to the problem. Why? Assume that day is equivalent to a value of 1. If that is the case, then the first if condition will be evaluated as true and the program will output "That day is a Monday!". So far there's no problem, however, please note that after executing the `System.out.println()`, the program will still execute the remaining six if statement. Logic tells us that we need not do them. As much as possible, we don't want the program to execute unnecessary instructions since it will be a waste of computation time. There must be a much better solution to the problem.

Example:

In temperate countries like Japan, there are four seasons. April to June is spring, July to September is summer, October to December is autumn and January to March is winter. We wish to write a program that allows the user to input month in its numeric equivalent (i.e., January is 1, February is 2 and so on...). The output of the program is the season associated with that month.

```
// Season program using if-statement
import java.io.*;
public class Season{
    public static void main(String[] args){
        int month;
        String in;
        BufferedReader input = new BufferedReader(new
            InputStreamReader(System.in));
        System.out.print("Input a Number from 1-12: ");
        try{
            in=input.readLine();
        }catch(IOException e){
            System.out.println("Error");
        }
        month = Integer.parseInt(in);
        if(month == 1)
            System.out.println("It's winter.");
        if(month == 2)
            System.out.println("It's winter.");
        if(month == 3)
            System.out.println("It's winter.");
        if(month == 4)
            System.out.println("It's spring.");
        if(month == 5)
            System.out.println("It's spring.");
        if(month == 6)
            System.out.println("It's spring.");
        if(month == 7)
            System.out.println("It's summer.");
```



```
        if(month == 8)
            System.out.println("It's summer.");
        if(month == 9)
            System.out.println("It's summer.");
        if(month == 10)
            System.out.println("It's autumn.");
        if(month == 11)
            System.out.println("It's autumn.");
        if(month == 12)
            System.out.println("It's autumn.");
    }
}
```

Note: The program above will give the required result, however, it is not really the best possible solution to the problem.

A better solution to the Season program is shown below using two different approaches:

1. Using Logical OR

```
// Season program using if-statement and logical OR
import java.io.*;
public class Season{
    public static void main(String[] args){
        int month;
        String in;
        BufferedReader input = new BufferedReader(new
            InputStreamReader(System.in));
        System.out.print("Input a Number from 1-12: ");
        try{
            in=input.readLine();
        }catch(IOException e){
            System.out.println("Error");
        }

        month = Integer.parseInt(in);

        if(month == 1 || month == 2 || month == 3)
            System.out.println("It's winter.");
        if(month == 4 || month == 5 || month == 6)
            System.out.println("It's spring.");
        if(month == 7 || month == 8 || month == 9)
            System.out.println("It's summer.");
        if(month == 10 || month == 11 || month == 12)
            System.out.println("It's autumn.");
    }
}
```

```
// Season program using if-statement and logical AND
import java.io.*;
public class Season{
    public static void main(String[] args){
        int month;
        String in;
        BufferedReader input = new BufferedReader(new
            InputStreamReader(System.in));
        System.out.print("Input a Number from 1-12: ");
        try{
            in=input.readLine();
        }catch(IOException e){
            System.out.println("Error");
        }

        month = Integer.parseInt(in);

        if(month >= 1 && month <= 3)
            System.out.println("It's winter.");
        if(month >= 4 && month <=6)
            System.out.println("It's spring.");
        if(month >= 7 && month <= 9)
            System.out.println("It's summer.");
        if(month >= 10 && month <= 12)
            System.out.println("It's autumn.");

    }
}
```

3.2. if-else Statement

A better approach in multiple choices type of problem solving is the if-else statement. Recall that example programs implementing if-statement is not the best solution since it tests for all conditions even though the result has already been achieved. if-else works in such a way that if one condition is evaluated to be TRUE, then it executes statements belonging to that condition and disregards the rest of the conditions not yet evaluated. This in short is the principle of what we call short circuit evaluation. Once a condition is met, no need to check the rest.

Syntax of an if-else statement is as follows:

```
if(<condition>)
{
    statement1;
    statement2;
    . . .
    statementN;
}
else
{
    statement1;
    statement2;
    . . .
    statementN;
}
```

The **<condition>** is first tested. If the **<condition>** results a **TRUE** value then the statement/s under the **if** block will be executed and the statements under the **else** block will no longer be tested and executed. Otherwise, if the **<condition>** results to a **FALSE** the statement under the **if** block will not be executed but statements under the **else** block will be the one to be executed.

```
// Number program using if-else
import java.io.*;
public class NumberProgram{
    public static void main(String[] args){
        int num;
        String in;
        BufferedReader input = new BufferedReader(new
            InputStreamReader(System.in));
        System.out.print("Input a Number: ");
        try{
            in=input.readLine();
        }catch(IOException e){
            System.out.println("Error");
        }

        num = Integer.parseInt(in);

        if(num>=0)
            System.out.println("Number is POSITIVE");
        else
            System.out.println("Number is NEGATIVE");
    }
}
```

Exercises: Using If-Else statement, write a program for the following

1. Accept an integer value and thereafter display whether the number is ODD or EVEN number.
2. Ask the user to input either 1 or 0. If the input value is 1 then the program should display TRUE, otherwise displays FALSE.
3. Ask the user to input an integer value representing the day in the week. Just like in the previous program problem, assume that 1 represents Monday, 2 is Tuesday and so on. The program should display whether the input day belongs to a WEEKEND or a WEEKDAY.
4. Ask the user to input a letter. If the user entered a lower-case letter, display the words LOWER CASE, otherwise displays the word UPPER CASE.
5. Ask the user to input the last digit in his car plate and thereafter the program will display what day is prohibited for him to use his car. This is also known as the number coding scheme.
6. You've been hired as a programmer by SFT (Send Fail Telecom) – a mobile telecommunications company. You are to write a program that will ask the user to input the number of text messages involved by SFT cellphone subscriber. The text messaging charge is then computed based on the number of test messages. The charge is zero for the first 200 messages. Text messages beyond that would be charged by 0.5 peso per message. The program should output the charge incurred.

3.3. Nested if statements

Since if and if-else are statement by themselves, it can actually be used as statement(s) inside and if or if-else. The term used to refer to such a construction is nested if statement.

Example:

You were hired as a programmer by TNP (Telepono ng Pilipino) a phone company. TNP employs a metering scheme in computing the telephone bill. The metering scheme is as follows: Calls made on a weekday between 6:00 AM to 6:00 PM are charged at 2.50 pesos per minute. Charges made at other times during a weekday are charged a discounted rate of 2.00 per minute. Calls made anytime on a weekend are charged a weekend rate of 1.50 pesos per minute. Your job is to write a program that will ask the user to enter the following information: (a) an integer representing the day the call was made, let 1 represent Monday, 2 represent Tuesday and so on, (b) an integer representing the time (in 24 hours format) the call started (c) and integer representing the length of time or duration of the call in minutes (assume that all calls are rounded to the next minute, i.e. a call lasting 2 minutes and 35 seconds

is billed as a 3 minute call). The rate applied depends on the day the call was made and on the time the call was started (not when it ended). Based on the information entered, the program should print the bill corresponding to the call.

```
// TNP program using nested if/if-else
import java.io.*;
public class TNPPProgram{
    public static void main(String[] args){
        double charge, rate;
        int day=0, sTime=0, duration=0;
        String in;
        BufferedReader input = new BufferedReader(new
            InputStreamReader(System.in));

        try{
            /* input the day, start time and duration
            of the call */
            System.out.print("Day:");
            in=input.readLine();
            day = Integer.parseInt(in);
            System.out.print("Start Time:");
            in=input.readLine();
            sTime = Integer.parseInt(in);
            System.out.print("Duration of Call:");
            in=input.readLine();
            duration = Integer.parseInt(in);

        }catch(IOException e){
            System.out.println("Error");
        }

        // Sample of nested if statement.
        if(day >= 1 && day <= 5){ //check for weekday

            if(sTime >= 0600 && sTime <=1800)
                rate = 2.5;    //regular rate
            else
                rate = 2.0;    // discounted rate
        }

        else // assume it's a weekend
            rate = 1.5;    // weekend rate
        charge = rate * duration;
        System.out.println("The call charge is: " +
            charge);
    }
}
```

Example:

Write a program that will determine the income of an employee. An employee is either a part-time employee or a full-time employee. A part-time employee's gross income is computed as the product of his/her hourly rate and the number of hours worked. The gross income of a full-time employee is computed as regular pay plus overtime pay. The overtime pay is computed as overtime rate multiplied by the number of overtime hours. The overtime pay should be computed if and only if the overtime hours rendered is not zero.

```
// Salary Program using nested if/if-else */
import java.io.*;
public class SalaryProgram{
    public static void main(String[] args){
        int eType=0;
        double gIncome=0, hRate=0, hWorked=0;
        double rPay=0, otHours=0, otRate=0, otPay=0;
        int day=0, sTime=0, duration=0;
        String in;
        BufferedReader input = new BufferedReader(new
            InputStreamReader(System.in));

        try{
            // input employee type
            System.out.print("Employee is [1]Part-time
                               [2] Full-time");
            in=input.readLine();
            eType = Integer.parseInt(in);
        }catch(IOException e){
            System.out.println("Error");
        }

        if(eType == 1){ // part-time
            try{
                // input hourly rate and hours worked
                System.out.print("Input hourly rate: ");
                in=input.readLine();
                hRate = Double.parseDouble(in);
                System.out.print("Input number of hours
                                   worked: ");
                in=input.readLine();
                hWorked = Double.parseDouble(in);
            }catch(IOException e){
                System.out.println("Error");
            }
            gIncome = hRate * hWorked;
        }
    }
}
```

```
else{ // employee_type == 2
    try{
        /* input regular pay and overtime hours
           and overtime rate */
        System.out.print("Input regular pay: ");
        in=input.readLine();
        rPay = Double.parseDouble(in);
        System.out.print("Input number of
                           overtime hours: ");
        in=input.readLine();
        otHours = Double.parseDouble(in);
        System.out.print("Input overtime rate");
        in=input.readLine();
        otRate = Double.parseDouble(in);

    }catch(IOException e){
        System.out.println("Error");
    }

    if(otHours > 0.0)
        otPay = otRate * otHours;
    else
        otPay = 0.0;

    gincome = rPay + otPay;
}

System.out.println("Gross income= " + gIncome);
}
```

3.4. Cascading if-else statement

There are many situations wherein we will need to cascade if-else statements. Improved versions of the program example given in the simple if statement are rewritten below using cascaded if-else. Cascaded if-else are also nested if-else.

```
//Program Days of the Week
import java.io.*;
public class DayOfWeek{
    public static void main(String[] args){
        int day = 0;
        String in = " ";
        BufferedReader input = new BufferedReader(new
                                                    InputStreamReader(System.in));
        System.out.print("Input number from 1 - 7: ");
        try{
            in=input.readLine();
            day = Integer.parseInt(in);
        }catch(IOException e){
            System.out.println("Error");
        }

        if(day == 1)
            System.out.println("That day is a Monday!");
        else if(day == 2)
            System.out.println("That day is a Tuesday!");
        else if(day == 3)
            System.out.println("That day is a Wednesday!");
        else if(day == 4)
            System.out.println("That day is a Thursday!");
        else if(day == 5)
            System.out.println("That day is a Friday!");
        else if(day == 6)
            System.out.println("That day is a Saturday!");
        else if(day == 7)
            System.out.println("That day is a Sunday!");

        System.out.println("Have a nice day.\n");
    }
}
```

NOTE: What will happen if the user entered 1 as the value of day? The program will print "That day is a Monday!" and following it below is the text "Have a nice day.". It will not test the other condition unlike in the simple if solution presented before. Thus, there are no unnecessary computations.

Example: Season Program

```
// Season program using Logical AND
import java.io.*;
public class SeasonProgram{
    public static void main(String[] args){
        String in = " ";
        int month = 0;
        BufferedReader input = new BufferedReader(new
                                                    InputStreamReader(System.in));
        System.out.print("Input month: ");
        try{
            in = input.readLine();
            month = Integer.parseInt(in);
        }catch(IOException e){
            System.out.println("Error!");
        }
        if(month >= 1 && month <= 3)
            System.out.println("It's winter.");
        else if(month >= 4 && month <= 6)
            System.out.println("It's spring.");
        else if(month >= 7 && month <= 9)
            System.out.println("It's summer.");
        else if(month >= 10 && month <= 12)
            System.out.println("It's autumn.");
    }
}
```

Another implementation of the Season Program, assuming that input values are from 1 to 12 only is presented below:

```
import java.io.*;
public class SeasonProgram{
    public static void main(String[] args){
        String in = " ";
        int month = 0;
        BufferedReader input = new BufferedReader(new
                                                    InputStreamReader(System.in));
        System.out.print("Input month: ");
        try{
            in = input.readLine();
            month = Integer.parseInt(in);
        }catch(IOException e){
            System.out.println("Error!");
        }

        if(month <= 3)
            System.out.println("It's winter.");
    }
}
```

```

else if(month <= 6)
    System.out.println("It's spring.");
else if(month <= 9)
    System.out.println("It's summer.");
else
    System.out.println("It's autumn.");
}
}

```

3.5. The switch Statement

The switch statement is a good alternative to cascading if-else statement. The syntax for the switch-case statement is as follow:

Syntax:

```

switch(<expression>)
{
    case <value1>: <statement1.1>;
                  <statement1.2>;
                  <statement1.N>;
                  [break];

    case <value2>: <statement2.1>;
                  <statement2.2>;
                  <statement2.N>;
                  [break];

    case <valueN>: <statementN.1>;
                  <statementN.2>;
                  <statementN.N>;
                  [break];

    [default : <statement> ];
}

```

Note:

- The expression must always be enclosed within a pair of parenthesis.
- The expression must evaluate to a whole number (i.e., an integer); the use of single precision and double precision floating point values will result into an error.
- The <expression> is tested if it is equivalent to any of <value1>, <value2> and so on. If it is equivalent to <value1> for example, then <statement1.1> until <statement1.N> will be executed. The same applies to other cases.

- The `break` statement is optional. If it is present, it will cause the program to “break” or “jump” out of the switch statement, and to execute the next statement following switch statement. If the break is not present, it will cause the program to execute the statement in the following case, i.e., <statement2.1> above, causing a waterfall effect. The same applies to the other cases.
- If the <expression> is not equivalent to any of the values, then it will execute the <statement> in the default case if it is present.

Example: Day Program

```
//Program Days of the Week using switch statement
import java.io.*;
public class DayOfWeek{
    public static void main(String[] args){
        int day = 0;
        String in = " ";
        BufferedReader input = new BufferedReader(new
                                                    InputStreamReader(System.in));
        System.out.print("Input number from 1 - 7: ");
        try{
            in=input.readLine();
            day = Integer.parseInt(in);
        }catch(IOException e){
            System.out.println("Error");
        }

        switch(day)
        {
            case 1: System.out.println("That day is a Monday!");
                    break;
            case 2: System.out.println("That day is a Tuesday!");
                    break;
            case 3: System.out.println("That day is a Wednesday!");
                    break;
            case 4: System.out.println("That day is a Thursday!");
                    break;
            case 5: System.out.println("That day is a Friday!");
                    break;
            case 6: System.out.println("That day is a Saturday!");
                    break;
            case 7: System.out.println("That day is a Sunday!");
                    break
            default: System.out.println("Invalid input try again");
        }
        System.out.println("Have a nice day.");
    }
}
```

Example: Season Program

```
//Season Program using switch statement
import java.io.*;
public class SeasonProgram{
    public static void main(String[] args){
        String in = " ";
        int month = 0;
        BufferedReader input = new BufferedReader(new
                                                    InputStreamReader(System.in));
        System.out.print("Input month: ");
        try{
            in = input.readLine();
            month = Integer.parseInt(in);
        }catch(IOException e){
            System.out.println("Error!");
        }

        switch(month) {
            case 1: System.out.println("It is winter.");
                    break;
            case 2: System.out.println("It is winter.");
                    break;
            case 3: System.out.println("It is winter.");
                    break;
            case 4: System.out.println("It is spring.");
                    break;
            case 5: System.out.println("It is spring.");
                    break;
            case 6: System.out.println("It is spring.");
                    break;
            case 7: System.out.println("It is summer.");
                    break;
            case 8: System.out.println("It is summer.");
                    break;
            case 9: System.out.println("It is summer.");
                    break;
            case 10: System.out.println("It is autumn.");
                    break;
            case 11: System.out.println("It is autumn.");
                    break;
            case 12: System.out.println("It is autumn.");
                    break;
        }
    }
}
```

NOTE: This program will run but it is not the best implementation. It is possible to have several cases respond to the same statement. A better way of implementing the previous program is shown below:

```
//Modified Season Program using switch statement
import java.io.*;
public class SeasonProgram{
    public static void main(String[] args){
        String in = " ";
        int month = 0;
        BufferedReader input = new BufferedReader(new
                                                    InputStreamReader(System.in));
        System.out.print("Input month: ");
        try{
            in = input.readLine();
            month = Integer.parseInt(in);
        }catch(IOException e){
            System.out.println("Error!");
        }

        switch(month){
            case 1:
            case 2:
            case 3: System.out.println("It is winter.");
                    break;

            case 4:
            case 5:
            case 6: System.out.println("It is spring.");
                    break;

            case 7:
            case 8:
            case 9: System.out.println("It is summer.");
                    break;

            case 10:
            case 11:
            case 12: System.out.println("It is autumn.");
                     break;

            default: System.out.println("Invalid Input!");
        }
    }
}
```

4. Iterative (Looping) Control Structures

One of a computer's most important attributes is the ability to perform repeatedly certain actions quickly, and loops are at the very heart of that. A loop is a repetition of a certain code segment/s in the program while the condition being evaluated remains to be true.

Loops usually have the following components:

1. *Initialization* of a variable or of several variables
2. *Condition* (that would evaluate to either true or false); the condition check is usually made on the current value of the variable initialized in (1) above.
3. *Body of the loop* which maybe a single statement or a group of statements
4. *A change of state* which is usually a statement inside the body of the loop that changes the contents of the variable(s)

Example: Printing the word "HELLO" fifty times.

Initialization:	counter = 1
Condition:	While counter <= 50
Body of the loop:	Print "HELLO"
Change of state:	counter = counter + 1

There are three available loop control structures in Java. These are:

1. `while` Loop
2. `for` Loop
3. `do-while` Loop

4.1. while Loop

The syntax of for the while loop is as follows:

```
<initialization>
while(<condition>)
{
    <statement1>;
    . . .
    <statementN>;
    <change of state>;
}
```

The body of the loop or statements under the while-loop will be repeatedly executed while the condition is `TRUE`. Otherwise, the loop will terminate and the next statement after the end of the while loop will be executed next.

Example: Considering the example given above, we have to write a program that will print the word “HELLO” fifty times.

```
//Printing HELLO 50 times

public class PrintHello{
    public static void main(String[] args){
        int counter;
        counter = 1;                // Initialization
        while(counter <= 50){       // Condition

            System.out.println("HELLO"); // Body
            counter = counter + 1;    // Change of State
        }
    }
}
```

The program above can be modified easily to print the word “HELLO” any number of times. It is simply a matter of changing the value inside the condition.

Exercises: Evaluating the above program answer the following questions:

1. What will happen if we remove the initialization `counter = 1` in the program? This means that the value of **counter** is undefined or garbage.
2. What will happen if we remove the curly brackets enclosing the body of the loop?
3. What will happen if the programmer committed a typographical error, such that instead of pressing the less than symbol, the greater than symbol was pressed, i.e. the condition becomes `counter >= 50`?
4. What will happen if the programmer forgot changing the value of counter, i.e., `counter = counter + 1` was omitted.
5. Write a new program using a while loop where it will print the numbers 10 down to 1 in vertical way.

Note that the change of state need not always be an increment or decrement by 1. It can be of any value (also known as the step value) that is appropriate in the problem. Consider the next example:

Example: Program that will compute the sum of all numbers that are divisible by 5 from 0 to 100.

```
// by 5's program using while loop

public class ByFive{
    public static void main(String[] args){
        int num, sum;
        sum = 0;
        num = 0;

        while(num <= 100){
            sum = sum + num;
            num = num + 5;
        }
        System.out.println("sum = " + sum);
    }
}
```

Note that the change of state in the above program is an increment of 5. Once the value of `num` reaches 105, the body of the loop will terminate and the statement below it will be executed next, which is `System.out.println("sum = " + sum);`.

Variable used for the change of state are not limited to integers only and need not always be incremental. Consider the following example:

```
// Step down using while loop
public class StepDown{
    public static void main(String[] args){
        double n = 10;
        while(n >= 1){
            System.out.println(n);
            n = n - 0.5;
        }
    }
}
```

Try simulating the output of the step down program written above.

4.2. for Loop

The syntax for a for-loop is as follows:

```
for([initialization]; [condition]; [change of state]){
    <statement1>;
    . . .
    <statementN>;
}
```


The for-loop is actually a “more compact” form of the while loop. The loop is executed as follows:

1. Perform the initialization
2. Check the condition
3. If the condition results to a `TRUE` value, the statement/s inside the for-loop (body) will be executed. Else proceed to #6
4. Change of state
5. Goes back to #2.
6. Exit the loop and the statement after the end of the for-loop will be executed.

Example: Print the word “HELLO” fifty times using for-loop:

```
//HELLO 50 times using for-loop example

public class PrintHello{
    public static void main(String[] args){
        int counter;

        for(counter = 1; counter <= 50; counter++){
            System.out.println("HELLO"); // Body
        }
    }
}
```

Note that the initialization, condition, and the change of state are only found in one line. The body of the for-loop like the while-loop is enclosed in a pair of curly braces. If there is only one statement representing the body of the loop, then the pair of curly brackets can be optional.

`counter++` is also the same as `counter = counter + 1`

Example: Print all EVEN numbers from 100 down to 0

```
// Even from 100 down to 0 using for-loop

public class EvenNumbers{
    public static void main(String[] args){
        int i;

        for(i = 100; i >= 0; i = i - 2)
            System.out.println(i);
    }
}
```

Example: A program that display the Fibonacci series.

```
// Fibonacci series using for-loops
import java.io.*;
public class FibonacciNumbers{
    public static void main(String[] args){
        int prev = 0, next = 1, ans = 0;
        int i, num = 0;
        String input = " ";

        BufferedReader in = new BufferedReader(new
            InputStreamReader(System.in));

        System.out.print("Input a number: ");
        try{
            input = in.readLine();
        }catch(IOException e){
            System.out.println("Error!");
        }
        num = Integer.parseInt(input);
        System.out.println("Fibonacci series is: " + num);

        for(i = num; i >=0; i--){
            System.out.print(ans);
            prev = next;
            next = ans;
            ans = prev + next;
        }
    }
}
```

Sample Output: The Fibonacci of 10

```
Input a number: 10
The Fibonacci series of 10 is:
0 1 1 2 3 5 8 13 21 34 55 _
```

4.3. do-while Loop

The syntax for a Do-While-loop is as follows:

```
<initialization>;

do{
    <statement1>;
    . . .
    <statementN>;
    <change of state>;
}while(<condition>;
```

The Do-While is executed as follows:

1. Performs the initialization.
2. Performs the body of the loop
3. Change of state
4. Checks the condition. If the condition results to a TRUE value, the statement/s inside the do-while Loop (body) will be executed until condition will be FALSE.
5. If condition is FALSE, exit the loop and the statement after the end of the do-while Loop will be executed.

Example: Print the word “HELLO” fifty times using do-while-Loop:

```
//HELLO x 50 using do-while-loop example

public class PrintHello{
    public static void main(String[] args){
        int counter;
        counter = 1;                // Initialization
        do{
            System.out.println("HELLO");    // Body
            counter = counter + 1;    // Change of State
        }while(counter <= 50);        // Condition
    }
}
```

Note that in a do-while Loop, the body is executed at least once before the condition is checked. Unlike in the `while` and `for` loops, the condition is first checked before the body of the loop can be executed.

Example: Print all EVEN numbers from 100 down to 0

```
// Even from 100 down to 0 using do-while-loop

public class EvenNumbers{
    public static void main(String[] args){
        int i;
        i = 100;
        do{
            System.out.println(i);
            i = i - 2;
        }while(i >= 0);
    }
}
```

4.4. Other implementation of loops

Loops are not confined on processing numeric data, the program will depend on the problem that is being solved. Let us consider the problem below.

```
// Interactive program using loops
import java.io.*;
public class InteractiveProgram{
    public static void main(String[] args){
        int a, b = 0, sum;
        char again;
        String input = " ";
        BufferedReader in = new BufferedReader(new
            InputStreamReader(System.in));

        do{
            sum = 0;
            for(a = 0; a < 10; a++){
                System.out.println("Input number " + a + " of 9: ");
                try{
                    input = in.readLine();
                }catch(IOException e){
                    System.out.println("Error!");
                }
                b = Integer.parseInt(input);
                sum = sum + b;
            }
            System.out.println("The sum = " + sum);
            System.out.println("Do you want to try again? (Y/N)");
            try{
                input = in.readLine();
            }catch(IOException e){}
            again = input.charAt(0);
        }while(again == 'Y' || again == 'y');
        System.out.println("Have a nice day!");
    }
}
```

4.5. Counters

A counter is a variable that is used to keep track of the count (frequency) of a certain group of items. Usually,

- Its data is integer
- It is initialized to a value of 0
- Incremented by 1 inside a loop

Example: Write a program that will ask the user to input 10 integers. The program should output how many of the data are positive. Assume that zero is a positive integer.

```
// Counter program
import java.io.*;
public class CounterProgram
{
    public static void main(String[] args){
        int i=0, num, ctr_positive;
        String input = "";
        ctr_positive = 0;

        BufferedReader in = new BufferedReader(new
            InputStreamReader(System.in));

        for(i = 0; i < 10; i++){
            System.out.println("Input number " + i + " of 9:", i);
            try{
                input = in.readLine();
            }catch(IOException e){
                System.out.println("Error !");
            }
            num = Integer.parseInt(input);
            if(num >= 0){
                ctr_positive = ctr_positive + 1;
            }
        }
        System.out.println("Positive integers= " + ctr_positive);
    }
}
```

Exercise:

1. Modify the previous program such that it will also count and print the number of negative integers that were inputted by the user.
2. Create a new program that will ask the user to input **n** integer values then print the number of ODD and EVEN numbers.
3. Create a program the will display all numbers that are divisible by 5 and count how many of them are there in the range of a signed integer data type.

4.6. Accumulators

An accumulator is a variable that is used to keep track of the accumulated value of a certain group of items. An accumulator

- May have a data type of `int`, `float` or `double`
- It is usually initialized to a value of 0
- Changes by assuming the sum of the current value of the accumulator and the value of another variable

Example: Write a program that will ask the user to input **n** integers. The program should output the sum of all the input data.

```
// Accumulator Sample
import java.io.*;
public class AccumulatorSample{
    public static void main(String[] args){
        int n, i, num, sum;
        String input = " ";
        sum = 0;

        BufferedReader in = new BufferedReader(new
                                                InputStreamReader(System.in));
        System.out.println("How many numbers to process? ");
        try{
            input = in.readLine();
        }catch(IOException e){
            System.out.println("Error!");
        }
        n = Integer.parseInt(input);
        for(i = 1; i <= n; i++)
        {
            System.out.println("Input number " + i + " of " + n);
            try{
                input = in.readLine();
            }catch(IOException e){
                System.out.println("Error!");
            }
            num = Integer.parseInt(input);
            sum = sum + num;
        }
        System.out.println("Sum of all input data is = " + sum);
    }
}
```

Exercises:

1. Modify the previous program such that it will compute and output the sum of positive numbers and the sum of negative numbers.
2. Create a new program that will ask the user to input how many students are there in a class. Thereafter, the user will be asked to input the final grade for each student. The program should determine the average of all the grades of the student in that class. How many of them passed, and how many of them failed.

Assume that the passing grades are from 75 to 99, and failing grades are from 70 to 74.

UNIT 3

Arrays and Introduction to Methods

1. Java Arrays

An array stores multiple data items of the same type, in a contiguous block of memory, divided into a number of slots. Furthermore, array is a capability of programming languages wherein one variable can store a list of data and manipulate these data more efficiently.

In simple words:

- An array is a group of elements with the same data type (homogeneous).
- An array is characterized by its name, dimension, size, and element data type.
- An array has a dimension:
 - If dimension is 1, we say that the array is a one-dimensional; Also known as a list.
 - If dimension is 2, we say that the array is a two-dimensional; Also known as a table.
 - Array having more than one dimension is also known as multi-dimensional array.

In our course, we will just discuss one and two dimensional arrays. But in the Java language, you can have more than two dimensions for an array.

1.1. Declaring a One-Dimensional Array

The syntax for a one-dimensional array:

```
<data_type> <array_name>[];  
           or  
<data_type>[] <array_name>;
```

After declaring, you must create the array and specify its length using a **constructor**. The process of using a **constructor** to create an array is called **instantiation**.

Sample Declaration:

```
//declaration  
int number[];  
//instantiation  
number = new int[10];
```


It can also be written as:

```
//declare and instantiate  
int number[] = new int[10];
```

The declaration tells the Java Compiler that the identifier `number` will be used as the name of the array containing integers and creates or instantiate a new array containing 10 elements.

Examples:

```
char name[] = new char[35];  
int grades[] = new int[50];  
float area[] = new float[5];  
double data[] = new double[10];
```

Note: array elements not explicitly initialized will have garbage values.

To reference an element in a one-dimensional array:

```
<array_name>[<index>]
```

Key Points when referencing elements in an array in Java:

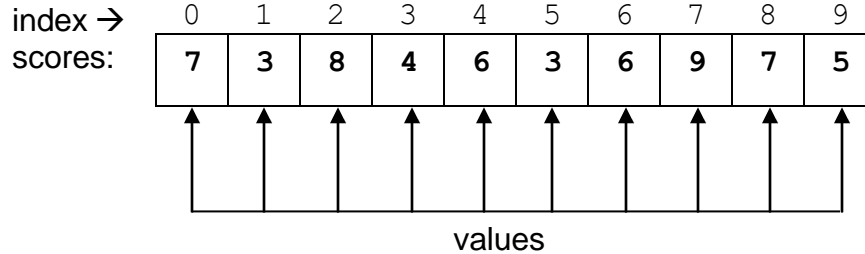
- The range of values that you can use as array index is from 0 to `<size-1>`
- The first element is always at index 0
- The last element is always at index `<size -1>`
- Index cannot be negative
- Index cannot be a float or double (real) number, only integer values

Example: A list of 10 scores in a quiz:

```
int scores[] = new int[10];  
• The range of index values is from 0 to 9  
• The first element is at scores[0]  
• The last element is at scores[9]
```

```
scores[0] = 7  
scores[1] = 3  
scores[2] = 8  
scores[3] = 4  
scores[4] = 6  
scores[5] = 3  
scores[6] = 6  
scores[7] = 9  
scores[8] = 7  
scores[9] = 5
```

Graphical representation of scores array having 10 as its size:



You can also initialize values of an array during declaration. Examples are:

```
/*creates an array of boolean variable with identifier
values initialized to True and False */
```

```
boolean values[] = {true, false};
```

```
/*creates an array of 4 double values (100, 90, 80,
75) with identifier name grades*/
```

```
double[] grades = {100, 90, 80, 75};
```

```
//creates an array of Strings with identifier days
```

```
String days[] = {"Mon", "Tues", "Wed", "Thurs", "Fri",
                 "Sat", "Sun"};
```

Example: Create a program that will declare a one-dimensional array with a size of 10. Initialize the array and let the user input 10 quiz scores as given above, thereafter compute the average of these 10 scores. Assume that the perfect score per quiz is 10 points.

```
// 10 quiz scores using 1-D array
import java.io.*;
public class TenQuizScores{
    public static void main(String[] args){
        int scores[] = new int[10];
        int i, sum = 0, num = 0;
        double ave = 0;
        String input = " ";

        BufferedReader in = new BufferedReader(new
            InputStreamReader(System.in));

        // Initialize list to zero
        for(i = 0; i < 10; i++){
            scores[i] = 0;
        }
        // Input 10 scores
        for(i = 0; i < 10; i++){
            System.out.print("Input score " + i + " = ");
            try{
                input = in.readLine();
            }catch(IOException e){
                System.out.println("Error!");
            }
            num = Integer.parseInt(input);
            scores[i] = num;
        }
        // Print inputted scores
        for(i = 0; i < 10; i++){
            System.out.println("score " + i + " = " + scores[i]);
        }
        // Compute sum of all 10 scores
        for(i = 0; i < 10; i++){
            sum = sum + scores[i];
        }
        // Compute the average
        ave = (double)sum / 10; // type casts (double)sum

        System.out.println("Sum of all scores is = " + sum
            + " over 100");
        System.out.println("The average is = " + ave);
    }
}
```

Note: `ave = (double)sum/10`. If we remove the cast `(double)` before the variable `sum`, the result will be an integer division, since `sum` is an integer and `10` is also an integer. The fraction part will be dropped. In order to retain the fraction part for the `ave` variable, we need to preface the variable with a new type in parenthesis. This is called type casting in Java.

1.2. Declaring Two-Dimensional Array

The syntax in declaring a two-dimensional array is as follows:

```
<data_type> <array_name>[<rowsize>][<colsize>;
Or
<data_type>[<rowsize>][<colsize>] <array_name>;
```

The **instantiation** of a two-dimensional array is the same as one-dimensional array except that the size would contain two value, size for the row and size for the column.

Sample Declaration:

```
//declaration
int number[][];
//instantiation
number = new int[5][10];
```

It can also be written as:

```
//declare and instantiate
int number[][] = new int[5][10];
```

Example:

```
int table[][] = new int[3][4];
```

Graphical representation of `table[][]` of `int[3][4]`:

	Col 0	Col 1	Col 2	Col 3
Row 0	[0][0]	[0][1]	[0][2]	[0][3]
Row 1	[1][0]	[1][1]	[1][2]	[1][3]
Row 2	[2][0]	[2][1]	[2][2]	[2][3]

Graphical representation of `table[][]` of `int[3][4]` using row major implementation:

[0] [0]	}	Row 0
[0] [1]		
[0] [2]		
[0] [3]		
[1] [0]	}	Row 1
[1] [1]		
[1] [2]		
[1] [3]		
[2] [0]	}	Row 2
[2] [1]		
[2] [2]		
[2] [3]		

To reference an element in a two-dimensional array:

`<array_name>[<row_index>][<column_index>]`

Key Points when referencing elements in a two-dimensional array in Java:

- The range of row index is from 0 to `<rowsize> - 1`
- The range of the column index is from 0 to `<columnsize> - 1`
- The first element is always at index `<array_name>[0][0]`
- The last element is at index `<array_name>[rowsize-1][colsize-1]`
- Index cannot be negative
- Index cannot be a float or double (real) number, only integer values
- Normally, a double loop is used as control structure for processing two-dimensional array.
- Normally, the processing is done row-by-row, and column-by-column within the same row.

Example: A table 3 x 4 of integer data :

```
int table[][] = new int[3][4];
table[0][0] = 2;
table[0][1] = 4;
table[0][2] = 5;
table[0][3] = 7;
table[1][0] = 1;
table[1][1] = 6;
table[1][2] = 9;
table[1][3] = 3;
table[2][0] = 4;
table[2][1] = 6;
table[2][2] = 2;
table[2][3] = 8;
```

Graphical representation of `table[][]` of `int[3][4]`:

	Col 0	Col 1	Col 2	Col 3
Row 0	2	4	5	7
Row 1	1	6	9	3
Row 2	4	6	2	8

```
//Table 3 x 4 program
public class Table3x4{
    public static void main(String[] args){
        int table[][] = new int[3][4];
        int i, j;
        table[0][0] = 2;
        table[0][1] = 4;
        table[0][2] = 5;
        table[0][3] = 7;
        table[1][0] = 1;
        table[1][1] = 6;
        table[1][2] = 9;
        table[1][3] = 3;
        table[2][0] = 4;
        table[2][1] = 6;
        table[2][2] = 2;
        table[2][3] = 8;
        // print table in matrix format
        for(i = 0; i < 3; i++){
            for(j = 0; j < 4; j++){
                System.out.print(" " + table[i][j]);
            }
            System.out.println();
        }
    }
}
```

Using the same table, create a program that will initialize the table to zero and ask the user to input values. Thereafter, the program will print the inputted values in matrix form. The program should also display the sum of all the values inputted.

```
//Table 3 x 4 modified program
import java.io.*;
public class Table3x4Modified{
    public static void main(String[] args){
        int table[][] = new int[3][4];
        int i, j, sum = 0, num;
        String input = " ";

        BufferedReader in = new BufferedReader(new
                                                    InputStreamReader(System.in));
        // Initialize table to zero
        for(i = 0; i < 3; i++){
            for(j = 0; j < 4; j++){
                table[i][j] = 0;
            }
        }
        // Input values
        for(i = 0; i < 3; i++){
            for(j = 0; j < 4; j++) {
                System.out.print("Input table [" + i + "][" +
                                   j + "] = ");

                try{
                    input = in.readLine();
                }catch(IOException e){
                    System.out.println("Error!");
                }
                num = Integer.parseInt(input);
                table[i][j] = num;
            }
        }
        // computing the sum
        for(i = 0; i < 3; i++){
            for(j = 0; j < 4; j++){
                sum = sum + table[i][j]);
            }
        }
        // print table in matrix format
        for(i = 0; i < 3; i++){
            for(j = 0; j < 4; j++){
                System.out.print(" " + table[i][j]);
            }
            System.out.println();
        }
        System.out.println("Sum = " + sum);
    }
}
```


Exercises:

Write a program that will create a 2-dimensional array with a row size = 10 and a column size = 10 also. Or simply a 10 x 10 table, thereafter the program should:

1. Initialize the array to zero.
2. Ask the user to input integer values.
3. Display the table in matrix form.
4. Display the sum of all elements.
5. Display the average of all elements.
6. Display the number of negative elements in the table.
7. Print only the elements on the main diagonals of the table.
8. Modify the program and display the sum of the elements on a specified row.
9. Modify the program and display the sum of the elements on a specified column.
10. Assume three matrices of the same size, say matrix A, B and C. Write a program that will add the two matrices A and B and store the sum to C. Matrix addition is done component-wise, i.e., $C[i][j] = A[i][j] + B[i][j]$ where i, j are the row and column index respectively.

1.3 Array Lengths

In order to get the number of elements in an array, the `length` field of an array can be used. The `length` field returns the size of the array and is written as follows:

`arrayName.length`

Sample Program: Using length field

```
public class ArrayLength{
    public static void main(String[] args){
        int[] scores = new int[50];
        int i;

        for(i = 0; i < scores.length; i++)
            System.out.print(scores);
    }
}
```

Note: It is better to use the `length` field as a condition statement of a loop to allow the loop to adjust automatically to different - sized arrays.

2. Java Methods

To know more of what a method is, let us first consider the following scenario. Let us assume that you were a programmer and you were asked to write a program that will be used to handle Automated Teller Machine transactions. Such transactions includes: (1) balance inquiry, (2) deposit and (3) withdraw. Aside from these, the user should be presented a menu showing there transaction as possible options. For a non-trivial program like these, it is advise to write the codes under one method which the `main()` method.

It is better to subdivide the programs into smaller portions – which we call methods. Each method should be able to solve part of the bigger problem. In the scenario above, one method will handle the generation of the menu, another method will be in-charge of handling the balance inquiry, another method will be used for deposit transaction, and a method that will handle withdrawal transactions. Notice that each method does a specified job of other methods.

In a more concrete term, a method is basically a program by itself – it has inputs, output, and will perform some kind of processing steps.

In computer programming, methods are also called subprograms. An aspiring computer programmer should be able to determine based from the problem statements, what methods are needed and how to relate these methods in order to solve the problem.

Characteristics of Methods

- It can return one or no values
- It may accept as many parameters it needs or no parameter at all
 - Parameters are also called function arguments
- After the method has finished execution, it goes back to the method that called it

2.1 Declaring Static Methods

Static methods are methods that can be invoked without instantiating a class. These methods belong to the class as a whole and not to a certain instance of a class. Static method is defined by the keyword `static` and can only be invoked inside another static method.

To declare methods, we follow the syntax:

```
<access_modifier> static <return_type>
    <method name>([parameters]) {
        method body
    }
```

Where,

- `access_modifier` can either be `public`, `private`, `protected` (if no access is present, the default modifier is in effect).
- `static` is a keyword used to indicate that the method is a static method
- `return_type` can be the different data types.
- `method_name` is any name given by the programmer to identify the method.
- `parameters` are arguments that is passed from one-method to another (optional if no arguments are to be passed).

Sample Method Declarations:

```
public static int withParameter(int n1, int n2){
    //body of the method
}

public static void withOutParameters(){
    //body of method
}
```

2.2 Calling Static Methods

To call a static method, the syntax to follow is,

```
ClassName.staticMethodName(params);
Or
staticMethodName(params);
```

Sample Program: Program that print the text “Hello World!” where “Hello” is to be printed by one method and “World!” is to be printed by another method.

```
//Declaring and Calling Methods
public class UsingMethods{
    public static void printHello(){
        System.out.print("Hello ");
    }
    public static void printWorld(){
        System.out.println("World!");
    }
    public static void main(String[] args){
        printHello();//same as UsingMethods.printHello
        printWorld();//same as UsingMethods.printWorld
    }
}
```

The previous program is interpreted as,

- `//Declaring and Calling Methods`
 - A program comment
- `public class UsingMethods{`
 - The name of the class (and the file)
- `public static void printHello(){`
 - A user – defined static method without any return value whose method name is `printHello` without any parameters
- `System.out.print("Hello ");`
 - Prints the text Hello on screen
- `}`
 - Ends the method body
- `public static void printWorld(){`
 - A user – defined static method without any return value whose method name is `printWorld` without any parameters
- `System.out.println("World!");`
 - Prints the text World! on screen
- `}`
 - Ends the method body
- `public static void main(String[] args){`
 - the main method (pre-defined method) with 1 parameter
- `printHello();`
 - Calling the method `printHello` and performing statements found therein
- `printWorld();`
 - Calling the method `printWorld` and performing statement found therein
- `}`
 - Ends the main method body
- `}`
 - Ends the class

Example: Calling `hello()` function three times inside `main()`

```
// calling hello() three times in main()
public class Hello3x{
    public static void hello(){
        System.out.println("Hello World!");
    }
    public static void main(String[] args){
        hello(); // 1st method call
        hello(); // 2nd method call
        hello(); // 3rd method call
    }
}
```

Multiple invocation of the same method must be placed inside a loop.

```
// calling hello() 1000 times in main()

public class Hello1000x{
    public static void hello(){
        System.out.println("Hello World!");
    }
    public static void main(String[] args){
        int i;

        for(i = 0; i < 1000; i++){
            hello();
        }
    }
}
```

Exercise: What will happen if we write the `hello()` function after the `main()` function? Will the program behave as what is expected? If not, how will you remedy the situation where functions are written after the `main()` method?

2.2.1. Calling a Method Inside another Method

A method can be invoked inside another method aside from `main()`.

Example:

```
// function call beside main()
public class FunctionCallBesideMain{
    public static void printHello(){
        System.out.print("Hello");
    }
    public static void printSpace(){
        System.out.print(" ");
    }
    public static void printWorld(){
        System.out.println("world!");
    }
    public static void allTogetherNow(){
        printHello();
        printSpace();
        printWorld();
    }
    public static void main(String[] args){
        allTogetherNow();
    }
}
```

2.3. Passing Variables in Methods

There are two ways of passing data to methods, one is to pass-by-value and the other is to pass-by-reference.

2.3.1. Pass-by-value

When pass-by-value occurs, the method makes a copy of the value of the variable passed to the method.

Example: Passing a value to a method

```
//Pass by value
public class PassByValue{
    //Method with parameter
    public static void methodWithParam(int j){
        //print value of j
        System.out.println("Value of j = " + j);
    }
    public static void main(String[] args){
        int i;
        i = 50;
        //call methodWithParam passing value of i
        methodWithParam(i);
        //print value of i
        System.out.println(i);
    }
}
```

Return to the caller and perform the statement after //call methodWithParam the method call

Pass i as a parameter which is copied to j

The output of the above program would be:

```
Value of j = 50;
Value of i = 50;
```

Supposed that you add the statement `j = 100;` before the `System.out.println("Value of j = " + j);` inside `methodWithParam(int j)`, the output would be:

```
Value of j = 100;
Value of i = 50;
```

Note: The method cannot modify the original value of the variable being passed. By default, all primitive data types when passed to a method are pass-by-value.

Sample program passing two parameters.

```
//Pass by value
public class PassByValue{
    //Method with parameter
    public static void methodWithParam(int j, int k){
        //print value of j
        System.out.println("Value of j = " + j);
        System.out.println("Value ok k = " + k);
    }
    public static void main(String[] args){
        int i;
        i = 50;

        /*call methodWithParam passing value of i
        copied to both parameters of
        methodWithParam*/
        methodWithParam(i, i);
        //print value of i
        System.out.println(i);
    }
}
```

The output of the above program would be

```
Value of j = 50
Value of k = 50
Value of i = 50
```

2.3.2. Pass-by-reference

When pass-by-reference occurs, the reference to an object is passed to the calling method. But unlike pass-by-value, the method can modify the actual object that the reference is pointing to since the location of the data pointed to is the same.

Example: Passing reference to a method

```

public class PassByReference{
    public static void methodWithParam(int[] arr){
        //change values of array
        for(int i = 0; i<arr.length; i++)
            arr[i] = i + 50;
    }

    public static void main(String[] args){
        //create array of integers
        int numbers[] = {10, 20, 30};

        //print array values
        for(int i = 0; i < numbers.length; i++)
            System.out.println(numbers[i]);

        /*call methodWithParam ans pass reference to
           array */
        methodWithParam(numbers);

        //print array values again
        for(int i = 0; i < numbers.length; i++)
            System.out.println(numbers[i]);
    }
}

```

Pass numbers as parameter which whose memory reference is pointed to by variable arr

Return to the caller and perform the statement after the method call

The output of the above program is:

```

10
20
30
60
70
80

```

Note: Instead of copying the values passed by the method, the pointer to the memory reference is the one being passes since it does not involve a primitive data type.

2.4. Methods with return types

So far, example given regarding methods makes use of `void` as a return type. The type `void` should be used as the return type only when the method does not return anything. However, there are cases

wherein the method will have to return a character, an integer, a float, or a double data type depending on the problem being solved.

Example: A program that will return integer type.

```
// method with return type int
public class MethodWithReturnType{

    //declaring method with int as return type
    public static int computeSum(int x, int y){
        int z;
        z = x + y;
        return z;
    }

    public static void main(String[] args){
        int a, b, c;

        /*call method computeSum passing 8 and 13 as
        parameters*/
        System.out.println("Sum = " + computeSum(8, 13));

        /*calling method computeSum passing 100 and 200
        as parameters and storing the result to variable
        c*/
        c = computeSum(100, 200);
        System.out.println("Sum = " + c);

        a = 22;
        b = 14;
        /*call method computeSum passing values of variables
        A and b as parameters*/
        System.out.println("Sum = " + computeSum(a, b));

    }
}
```

The output of the above program would be:

```
Sum = 21
Sum = 300
Sum = 36
```

Exercise:

1. What will happen if the statement **return z** was omitted?
2. Write a method named **Difference()** that accepts two integer parameters named **x** and **y**. The method should compute and return the value of the difference of **x** and **y**. Call this method inside **main()**.
3. Write a method named **Product()** that accepts two floating point parameters named **x** and **y**. The method should compute and return the value of the product for **x** and **y**. Use **double** as the return type. Call this method inside **main()**.

Example: Write a method that will accept an integer variable as parameters, thereafter, the method should return 1 if the integer is positive, otherwise it should return 0.

```
// function to return 1 if positive otherwise return 0

public static int numberPositive(int n){
    int result;

    if (n >= 0)
        result = 1;

    else
        result = 0;

    return result;
}
```

Another way to implement this method is as follows:

```
// Another implementation

public static int numberPositive(int n)
{
    if (n >= 0)
        return 1;
    else
        return 0;
}
```

2.5 Important things to remember about methods

1. If the method will not return anything, use `void` as the return type.
2. If the method will return a value, the appropriate data type should be specified.
3. If the method will return a value, don't forget to use the `return` statement inside the method.
4. The data type of the value that appears after the `return` statement must be compatible with the return data type.
5. You can use any name for the method as long as you follow the naming convention and do not use Java keywords as method names.
6. A method may have zero, one or more parameters.
7. If the method does not have any parameter, leave the area in between the parentheses blank.
8. If the method has parameters, specify their data types and names.
9. Parameters should be separated by comma.
10. The name of the parameters does not matter. Always remember that the *number of parameters, their respective data types, and their sequence* are the things that actually matter!
11. When calling a method, constants, expressions and variables maybe used as actual parameters.
12. A method can return at the most only one value at any given point in time.

3. Scope of Variable

The **scope** determines where in the program the variable is accessible. The scope also determines the lifetime or how long the variable can exist in memory. The placement of the variable, or where the variable is declared, determines the scope for that variable.

For example:

```

    public class ScopeExample{
        public static void main(String[] args){
            int i = 0;
            int j = 0;

            for(int k = 0 ; k<10; k++){
                //some code
            }
        }
    }

```

The sample code snippet written above represents two scopes indicated by the lines and letter representing the scope.

Scope A – the scope of variables `i` and `j` is true to the whole `main()`, meaning, both variables `i` and `j` can be used inside the `main()` block

Scope B - variable `k` can only be used inside the `for` loop block.

Given a complete sample program with method,

```
public class PassByReference{
    public static void main(String[] args){
        //create an array of integers
        int numbers[] = {10, 20, 30};

        //print array values
        B ----- for(int i = 0; i < number.length; i++){
            System.out.println(numbers[i]);
        }

        A ----- //call method and pass reference to array
        methodWithParam(numbers);

        //print array values again
        C ----- for(int i = 0; i < numbers.length; i++){
            System.out.println(numbers[i]);
        }
    }

    public static methodWithParam(int[] arr){
        //change value of array
        E ----- for(int i = 0; i < arr.length; i++){
            arr[i] = i + 50;
        }
    }
}
```

D -----

Inside the `main()` method:

`numbers[]` – scope A

`i` in B – scope B

`i` in C – scope C

Inside the `methodWithParam()` method:

`arr[]` – scope D

`i` in E – scope E

REFERENCES

A. Books

Cadenhead, R., Lemay, L. (2007), **Sams Teach yourself Java 6 in 21 Days, 5th Edition**, Sams Publishing.

Deitel, P.J., Deitel, H.M. (2007), **Java: How to Program, 7th Edition**, Prentice Hall.

Bates, B., Sierra, K. (2005), **Head First Java, 2nd Edition**, O' Reilly.

Schildt, H. (2005), **Java: The Complete Reference, 5th Edition**, McGraw-Hill.

Williams, A. (2002), **Java 2. Core Language. Little Black Book**, Paraglyph.

B. Websites

The Java Tutorial: A Practical Guide for Programmers.

Available at <http://java.sun.com/docs/books/tutorial>

Java in a Nutshell. Chapter 4: The Java Platform.

Available at http://www.unix.org.ua/oreilly/java-ent/jnut/ch04_10.htm

Diagnosing Java Code: Java Generics Without the Pain, Part 1 by Eric Allen.

Available at

<http://www-106.ibm.com/developerworks/java/library/j-djc02113.html>

Java Programming (with Passion!)

Available at

http://www.javapassion.com/portal/index.php?option=com_content&view=article&id=66

