

MODULE I

INTRODUCTION OF JAVA

STRUCTURE

Learning Objectives

- 1.1 Introduction
- 1.2 History
- 1.3 Overview of Java
- 1.4 Object Oriented Programming
- 1.5 simple Programme, Two control statements - if statement, for loop
- 1.6 Using Blocks of codes
- 1.7 Lexical issues - White space, identifiers, Literals, comments, separators, Java Key words
- 1.8 Data types: Integers, Floating point, characters, Boolean
- 1.9 A closer look at Literals, Variables, Type conversion and casting
- 1.10 Automatic type promotion in Expressions Arrays
- 1.11 Operators
- 1.12 Arithmetic operators, The Bit wise operators, Relational Operators, Boolean Logical operators, Assignment Operator, Operator Precedence.
- 1.13 Control Statements: Selection Statements - if, Switch: Iteration Statements - While, Do-while, for Nested loops, Jump statements.
- 1.14 Summary
- 1.15 Self-Assesment Questions
- 1.16 Suggested Readings

LEARNING OBJECTIVES

- To understand the overview of Java Programming.
- To understand the use of Data Types, Variables.
- To learn Type Conversion and Operators.

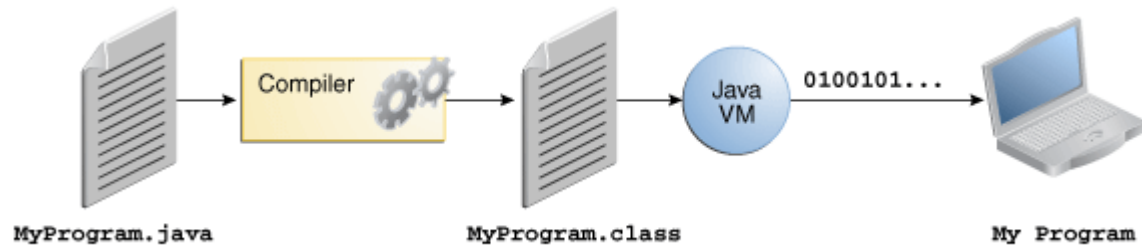
1.1 INTRODUCTION

JAVA was developed by Sun Microsystems Inc in 1991, later acquired by Oracle Corporation. It was developed by James Gosling and Patrick Naughton. It is a simple programming language. Writing, compiling and debugging a program is easy in java. It helps to create modular programs and reusable code. Java is a programming language and platform released by Sun Microsystems in 1995. Java is secure, fast, and reliable programming language. Platform means any h/w or s/w environment in which a program runs. Java has its own runtime environment i.e. JRE and API it is called platform. Java is everywhere from PCs to Mobile phone, satellites, other electronic devices. There are many applications and websites that will not run without java installation on your machine. Java is the combination of two things:

- A Programming language
- Platform

Java is one of the world's most widely used computer language. Java is a simple, general-purpose, object-oriented, interpreted, robust, secure, architecture-neutral, portable, high-performance, multithreaded computer language. It is intended to let application developers "write once, run anywhere" (WORA), meaning that code that runs on one platform does not need to be recompiled to run on another. Java technology is both a programming language and a platform. Java is a high level, robust, secured and object-oriented programming language. And any hardware or software environment in which a program runs, is known as a platform. Since Java has its own runtime environment (JRE) and API, it is called platform. In the Java programming language, all source code is first written in plain text files ending with the .java extension. Those source files are then compiled

into .class files by the javac compiler. A .class file does not contain code that is native to your processor; it instead contains bytecodes — the machine language of the Java Virtual Machine¹ (Java VM). The java launcher tool then runs your application with an instance of the Java Virtual Machine.



Because the Java VM is available on many different operating systems, the same .class files are capable of running on Microsoft Windows, the Solaris™ Operating System (Solaris OS), Linux, or Mac OS.

1.2 HISTORY

Java is a programming language created by James Gosling from Sun Microsystems (Sun) in 1991. The target of Java is to write a program once and then run this program on multiple operating systems. The first publicly available version of Java (Java 1.0) was released in 1995. Sun Microsystems was acquired by the Oracle Corporation in 2010. Oracle has now the steersmanship for Java. In 2006 Sun started to make Java available under the GNU General Public License (GPL).

Oracle continues this project called OpenJDK. Over time new enhanced versions of Java have been released. The current version of Java is Java 1.8 which is also known as Java 8. Java is defined by a specification and consists of a programming language, a compiler, core libraries and a runtime (Java virtual machine). The Java runtime allows software developers to write program code in other languages than the Java programming language which still runs on the Java virtual machine. The Java platform is usually associated with the Java virtual machine and the Java core libraries.

The Java language was designed with the following properties:

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

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

1.3 OVERVIEW OF JAVA

Java programming language was originally developed by Sun Microsystems which was initiated by James Gosling and released in 1995 as core component of Sun Microsystems' Java platform (Java 1.0 [J2SE]). The latest release of the Java Standard Edition is Java SE 8. With the advancement of Java and its widespread popularity, multiple configurations were built to suit various types of platforms. For example: J2EE for Enterprise Applications, J2ME for Mobile Applications. The new J2 versions were renamed as Java SE, Java EE, and Java ME respectively. Java is guaranteed to be Write Once, Run Anywhere.

Java is –

- **Object Oriented** – In Java, everything is an Object. Java can be easily extended since it is based on the Object model.
- **Platform Independent** – Unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code. This byte code is distributed over the web and interpreted by the Virtual Machine (JVM) on whichever platform it is being run on.
- **Simple** – Java is designed to be easy to learn. If you understand the basic concept of OOP Java, it would be easy to master.
- **Secure** – With Java's secure feature it enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.
- **Architecture-neutral** – Java compiler generates an architecture-neutral object file format, which makes the compiled code executable on many processors, with the presence of Java runtime system.
- **Portable** – Being architecture-neutral and having no implementation dependent aspects of the specification makes Java portable. Compiler in Java is written in ANSI C with a clean portability boundary, which is a POSIX subset.
- **Robust** – Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking.
- **Multithreaded** – With Java's multithreaded feature it is possible to write programs that can perform many tasks simultaneously. This design feature allows the developers to construct interactive applications that can run smoothly.
- **Interpreted** – Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light-weight process.
- **High Performance** – with the use of Just-In-Time compilers, Java enables high performance.
- **Distributed** – Java is designed for the distributed environment of the internet.
- **Dynamic** – Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

1.4 OBJECT ORIENTED PROGRAMMING

Object Oriented Programming is a method of implementation in which programs are organized as cooperative collection of objects, each of which represents an instance of a class, and whose classes are all members of a hierarchy of classes united via inheritance relationships.

OOP Concepts

Four principles of Object Oriented Programming are

- Abstraction
- Encapsulation

- Inheritance
- Polymorphism

Abstraction-Abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.

Encapsulation- Encapsulation is the process of compartmentalizing the elements of an abstraction that constitute its structure and behaviour ; encapsulation serves to separate the contractual interface of an abstraction and its implementation.

- * Hides the implementation details of a class.
- * Forces the user to use an interface to access data
- * Makes the code more maintainable.

Inheritance-Inheritance is the process by which one object acquires the properties of another object.

Polymorphism- Polymorphism is the existence of the classes or methods in different forms or single name denoting different implementations.

1.5 A SIMPLE PROGRAMME

Hello World Example-

To create a simple java program, you need to create a class that contains the main method. Let's understand the requirement first.

The requirement for Java Hello World Example

- For executing any java program, you need to
- Install the JDK if you don't have installed it, download the JDK and install it.
- Set path of the jdk/bin directory. <http://www.javatpoint.com/how-to-set-path-in-java>
- Create the java program
- Compile and run the java program

Creating Hello World Example

Let's create the hello java program:

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

save this file as Simple.java

Output: Hello Java

TWO CONTROL STATEMENTS

A program executes from top to bottom, except when we use control statements. Then, we can control the order of execution of the program, based on logic and the values.

In Java, control statements can be divided into the following three categories:

- Selection Statements
- Iteration Statements
- Jump Statements

Selection Statements-Selection statements allow you to control the flow of program execution, on the basis of the outcome of an expression or state of a variable, known during runtime.

Selection statements can be divided into the following categories:

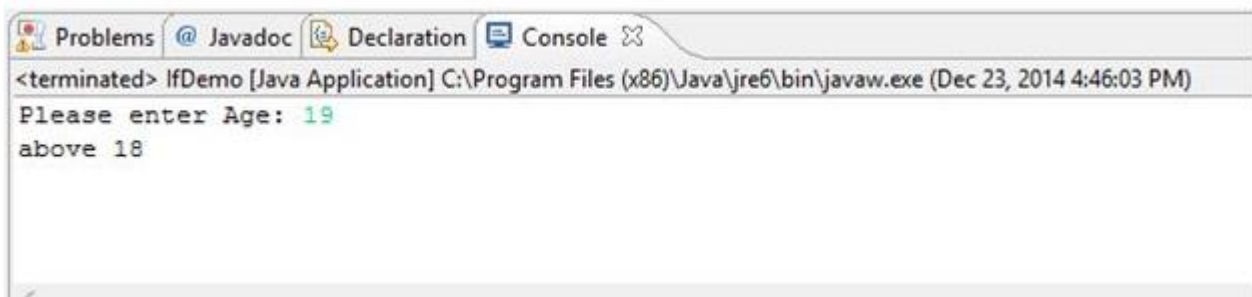
- The if and if-else statements
- The if-else statements
- The if-else-if statements
- The switch statements

The if statements-The first contained statement (that can be a block) of an if statement, only executes when the specified condition is true. If the condition is false and there is no keyword, then the first contained statement will be skipped and execution continues with the rest of the program. The condition is an expression that returns a boolean value.

Example

```
package com.deepak.main;
import java.util.Scanner;
public class IfDemo { public static void main(String[] args)
{   int age;
    Scanner inputDevice = new Scanner(System.in);
    System.out.print("Please enter Age: ");
    age = inputDevice.nextInt();
    if (age > 18)
        System.out.println("above 18 "); }
```

Output:

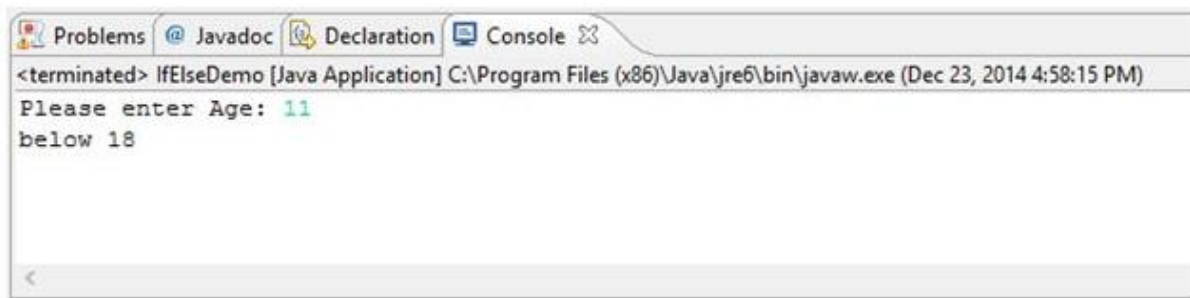


The if-else statements-In if-else statements, if the specified condition in the if the statement is false, then the statement after the else keyword (that can be a block) will execute.

Example

```
package com.deepak.main;
import java.util.Scanner;
public class IfElseDemo { public static void main(String[] args)
{   int age;
    Scanner inputDevice = new Scanner(System.in);
    System.out.print("Please enter Age: ");
    age = inputDevice.nextInt();
    if (age >= 18)
        System.out.println("above 18 ");
    else
        System.out.println("below 18");
}
```

Output



```
<terminated> IfElseDemo [Java Application] C:\Program Files (x86)\Java\jre6\bin\javaw.exe (Dec 23, 2014 4:58:15 PM)
Please enter Age: 11
below 18
```

The if-else-if statements

The statement following the else keyword can be another if or if-else statement.

That would look like this:

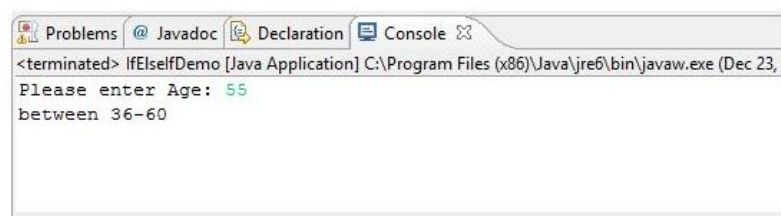
```
if (condition)
    statements;
else if (condition)
    statements;
else if (condition)
    statement;
else
    statements;
```

Whenever the condition is true, the associated statement will be executed and the remaining conditions will be bypassed. If none of the conditions are true, then the else block will execute.

Example

```
package com.deepak.main;
import java.util.Scanner;
public class IfElseIfDemo {
    public static void main(String[] args) {
        int age;
        Scanner inputDevice = new Scanner(System.in);
        System.out.print("Please enter Age: ");
        age = inputDevice.nextInt();
        if (age >= 18 && age <= 35)
            System.out.println("between 18-35 ");
        else if (age > 35 && age <= 60)
            System.out.println("between 36-60");
        else
            System.out.println("not matched");
    }
}
```

Output:



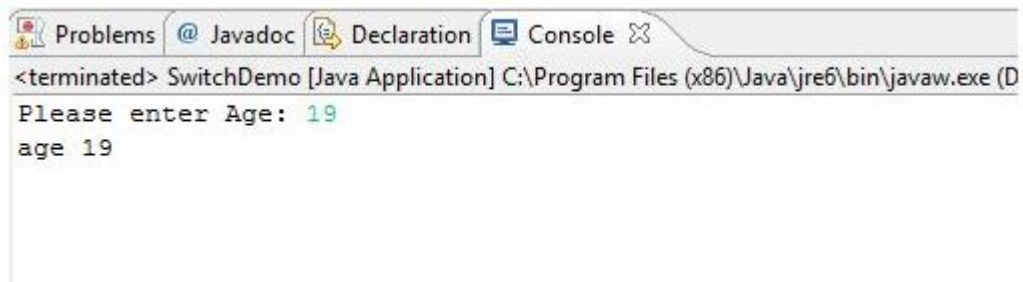
```
<terminated> IfElseIfDemo [Java Application] C:\Program Files (x86)\Java\jre6\bin\javaw.exe (Dec 23, 2014 4:58:15 PM)
Please enter Age: 55
between 36-60
```

The Switch Statements-The switch statement is a multi-way branch statement. The switch statement of Java is another selection statement that defines multiple paths of execution, of a program. It provides a better alternative than a large series of if-else-if statements.

Example

```
package com.deepak.main;
import java.util.Scanner;
public class SwitchDemo {
    public static void main(String[] args) {
        int age;
        Scanner inputDevice = new Scanner(System.in);
        System.out.print("Please enter Age: ");
        age = inputDevice.nextInt();
        switch (age) {
            case 18:
                System.out.println("age 18");
                break;
            case 19:
                System.out.println("age 19");
                break;
            default:
                System.out.println("not matched");
                break;
        }
    }
}
```

Output:



An expression must be of a type of byte, short, int, or char. Each of the values specified in the case statement must be of a type compatible with the expression. Duplicate case values are not allowed. The break statement is used inside the switch to terminate a statement sequence. The break statement is optional in the switch statement.

Iteration Statements-Repeating the same code fragment several times, until a specified condition is satisfied, is called iteration. Iteration statements execute the same set of instructions, until a termination condition is met.

Java provides the following loop for iteration statements:

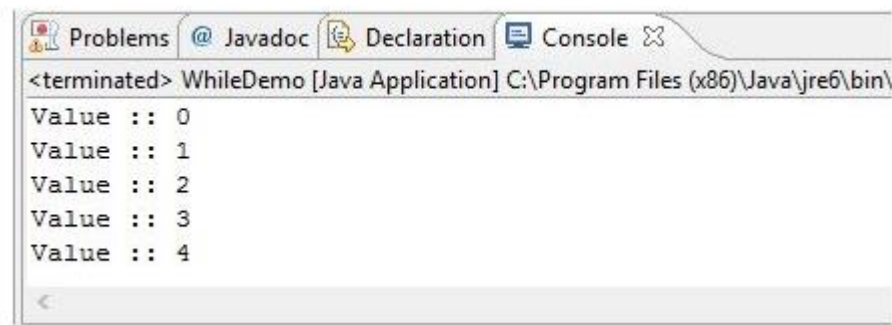
- The while loop
- The for loop
- The do-while loop
- The for-each loop

The while loop- It continually executes a statement (that is usually a block) while a condition is true. The condition must return a Boolean value.

Example

```
Package com.deepak.main;  
public class WhileDemo {  
    public static void main(String[] args) {  
        int i = 0;  
        while (i < 5) {  
            System.out.println("Value :: " + i);  
            i++;  
        }  
    }  
}
```

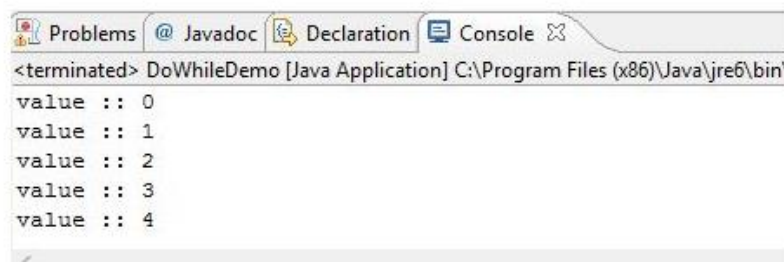
Output:



The do-while loop- The only difference between a while and a do-while loop, is that do-while evaluates its expression at the bottom of the loop, instead of the top. The do-while loop executes at least one time, then it will check the expression prior to the next iteration.

```
P ackage com.deepak.main;  
public class DoWhileDemo {  
    public static void main(String[] args) {  
        int i = 0;  
        do {  
            System.out.println("value :: " + i);  
            i++;  
        }  
        while (i < 5);  
    }  
}
```

Output



The for loop-A for loop executes a statement (that is usually a block) as long as the boolean condition evaluates to true. A for loop is a combination of the three elements initialization statement, boolean expression and increment or decrement statement.

Syntax:

```
for ( < initialization > ; < condition > ; < increment or decrement statement > ) {  
    < block of code >  
}
```

The initialization block executes first before the loop starts. It is used to initialize the loop variable.

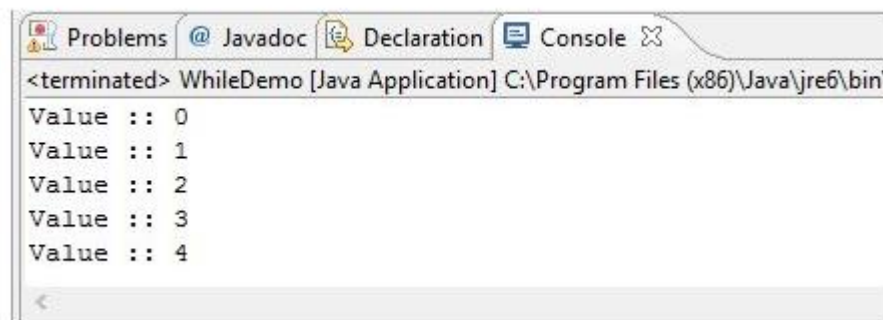
The condition statement evaluates every time prior to when the statement (that is usually a block) executes. If the condition is true then only the statement (that is usually a block) will execute.

The increment or decrement statement executes every time, after the statement (that is usually a block).

Example

```
package com.deepak.main;  
public class WhileDemo {  
    public static void main(String[] args) {  
        int i = 0;  
        while (i < 5) {  
            System.out.println("Value :: " + i);  
            i++;  
        }  
    }  
}
```

Output



1.6 USING BLOCKS OF CODES

Block of code in Java allows grouping of two or more statements. We use curly braces { } to denote a block of code in Java. Block of code in Java can be used where ever we use a statement.

- Every block of code in Java starts with a open curly brace { and ends with close curly brace }.
- There is no restriction on the number of blocks inside a block and the level of nesting the blocks. i.e. Blocks can be nested and can be included inside other blocks.
- Block of code in Java is commonly used in if, for and while statements.
- All class and method contents are also blocks e.g., the class content or the main method in the examples are blocks.
- It is advised to indent i.e. put tabs or spaces so that the inside blocks are one tab more than the containing block. Indenting the blocks will help in resolving the compilation errors faster and the programs will be easy to read.

Block of code

CODE

```
class CodeBlock
{
    public static void main(String arg[])
    {
        System.out.println("In main block");

        { // LINE A
            System.out.print("In ");
            System.out.print("inner ");
            System.out.print("block "); // LINE A1
            System.out.println("One");
        }

        { // LINE B
            System.out.print("In ");
            System.out.print("inner ");
            System.out.print("block ");
            System.out.println("Two");

            { // LINE C
                System.out.println ("Block inside inner block two");
            }
        } // LINE D
    }
}
```

Output:

In main block

In inner block One

In inner block Two

Block inside inner block two

Description-Here we have two inner blocks starting at LINE A and LINE B. Inside the second inner block we have one more block starting at LINE C. The class content and the main method content are also blocks. In total there are 5 blocks in this program.

Block of code for if

CODE

```
class IfCodeBlock
{
    public static void main(String arg[])
    {
        short marks = 95;

        if( marks > 90 )
        {
            System.out.println("Excellent"); // LINE A
            System.out.println("Scholarship Granted"); // LINE B
        }
    }
}
```

```
}  
}
```

Output

Excellent

Scholarship granted

Description-Here we want to print Excellent and Scholarship granted when marks are greater than 90. If we do not include these two statements in the code block i.e. in curly braces { }, then Scholarship granted will be printed whether or not marks are greater than 90. Since it considers only the first statement after the condition. Even indenting will not help, since java is free-form language and does not give importance to spaces, tabs etc.

1.7 LEXICAL ISSUES - WHITE SPACE, IDENTIFIERS, LITERALS, COMMENTS, SEPARATORS, JAVA KEY WORDS

Java programs is a collection of White spaces , Identifiers , comments , Literals , Operators , Separators and Keywords.

White Spaces -Java is a free form language. This means that you do not need to follow any special indentation rules. In java , white spaces is a space , tab or new line.

Identifiers-Identifiers are used for class names , method names and variable names. An identifier may be any descriptive sequence of uppercase and lowercase letters , numbers or the underscore and dollar sign design.

Literals-A constant value in java is created by using a literal representation of it. A literal can be used anywhere a value of its type is allowed.

Comments-There are 3 types of comment in java. First is single line comment and the second one is multi line comment. The third type of comment is called documentation comment. It is used to produce an HTML file that documents your program. It begins with a/** and ends with a*/.

Separators-There are few symbols in java that are used as separators.The most commonly used separator in java is the **semicolon** ';'. some other separators are **Parentheses** '()' , **Braces** '{}' , **Bracket** '[]' , **Comma** ',' , **Period** '.' .

Java Keywords-There are 49 reserved keywords currently defined in java. These keywords cannot be used as names for a variable , class or method.The **Keywords** are : abstract , assert , boolean , break , byte , case , catch , char , class , const , continue , default , do , double , else , extends , final , finally , float , for , goto , if , implements , import , instanceof , int interface , long , native , new , package , private , protected , public , return , short , static , strictfp , super , switch , synchronized , this , throw , throws , transient , try , void , volatile, while.

Java Class Libraries-The java environment relies on several built-in class libraries that contain many built-in methods that provide support for such things as I/O , string handling , networking and graphics. The standard class also provide support for windowed output. Thus java as a totality is a combination of the java language itself , plus its standard classes.

1.8 DATA TYPES

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

Primitive data types: The primitive data types include boolean, char, byte, short, int, long, float and double.

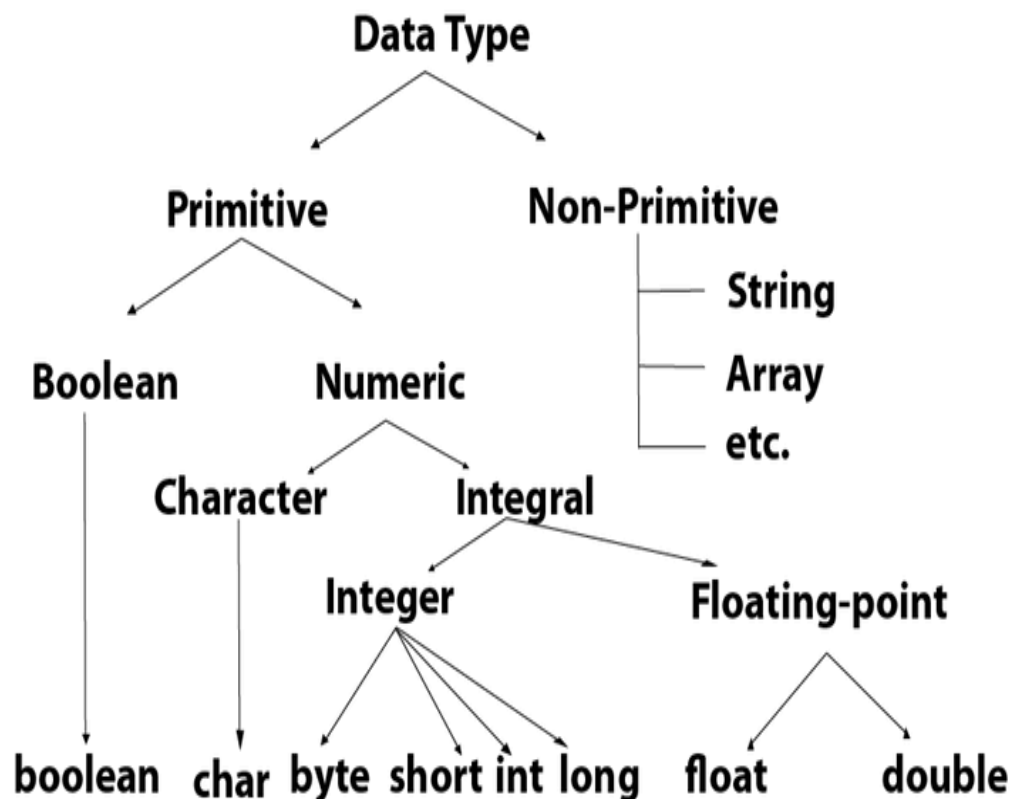
Non-primitive data types: The non-primitive data types include Classes, Interfaces, and Arrays.

Java Primitive Data Types

In Java language, primitive data types are the building blocks of data manipulation. These are the most basic data types available in Java language.

There are 8 types of primitive data types:

- boolean data type
- byte data type
- char data type
- short data type
- int data type
- long data type
- float data type
- double data type



Data Type	Default Value	Default size
boolean	false	1 bit
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

Boolean Data Type-The Boolean data type is used to store only two possible values: true and false. This data type is used for simple flags that track true/false conditions. The Boolean data type specifies one bit of information, but its "size" can't be defined precisely.

Example: Boolean one = false

Byte Data Type-The byte data type is an example of primitive data type. It is an 8-bit signed two's complement integer. Its value-range lies between -128 to 127 (inclusive). Its minimum value is -128 and maximum value is 127. Its default value is 0. The byte data type is used to save memory in large arrays where the memory savings is most required. It saves space because a byte is 4 times smaller than an integer. It can also be used in place of "int" data type.

Example: byte a = 10, byte b = -20

Short Data Type-The short data type is a 16-bit signed two's complement integer. Its value-range lies between -32,768 to 32,767 (inclusive). Its minimum value is -32,768 and maximum value is 32,767. Its default value is 0.

The short data type can also be used to save memory just like byte data type. A short data type is 2 times smaller than an integer.

Example: short s = 10000, short r = -5000

Int Data Type-The int data type is a 32-bit signed two's complement integer. Its value-range lies between - 2,147,483,648 (-2^{31}) to 2,147,483,647 ($2^{31} - 1$) (inclusive). Its minimum value is - 2,147,483,648 and maximum value is 2,147,483,647. Its default value is 0.

The int data type is generally used as a default data type for integral values unless if there is no problem about memory.

Example: int a = 100000, int b = -200000

Long Data Type

The long data type is a 64-bit two's complement integer. Its value-range lies between - 9,223,372,036,854,775,808 (-2^{63}) to 9,223,372,036,854,775,807 ($2^{63} - 1$) (inclusive). Its minimum value is - 9,223,372,036,854,775,808 and maximum value is 9,223,372,036,854,775,807. Its default value is 0. The long data type is used when you need a range of values more than those provided by int.

Example: long a = 100000L, long b = -200000L

Float Data Type-The float data type is a single-precision 32-bit IEEE 754 floating point. Its value range is unlimited. It is recommended to use a float (instead of double) if you need to save memory in large arrays of floating point numbers. The float data type should never be used for precise values, such as currency. Its default value is 0.0F.

Example: float f1 = 234.5f

Double Data Type-The double data type is a double-precision 64-bit IEEE 754 floating point. Its value range is unlimited. The double data type is generally used for decimal values just like float. The double data type also should never be used for precise values, such as currency. Its default value is 0.0d.

Example: double d1 = 12.3

Char Data Type-The char data type is a single 16-bit Unicode character. Its value-range lies between '\u0000' (or 0) to '\uffff' (or 65,535 inclusive). The char data type is used to store characters.

Example: char letterA = 'A'

1.9 A CLOSER LOOK AT LITERALS, VARIABLES, TYPE CONVERSION AND CASTING

A variable is a container that holds values that are used in a Java program. Every variable must be declared to use in program.

```
/**
 * This program demonstrates
 * how to use variables in a program
 */
public class VariableDemo
{
    public static void main(String[] args)
    {
        // Declare variables to hold data.
        int rollno;
        String name;
        double marks;
        // Assign values to variables.
        rollno = 19;
        name = "David";
        marks = 89.8;

        // Display the message
        System.out.println("Your roll number is " + rollno);
        System.out.println("Your name is " + name);
        System.out.println("Your marks is " + marks);
    }
}
```

When you compile and execute this program, the following three lines are displayed on the screen.

Your roll number is 19

Your name is David

Your marks is 89.8

To explain how this happens, let's consider following statements:

```
int rollno;
```

This line indicates the variable's name is rollno. A variable is a named storage location in the computer's memory used to hold data. Variable must be declared before they can be used. A variable declaration tells the compiler the variable's name and the type of data it will hold. In this statement int stands for integer so rollno will only be used to hold integer numbers.

Similarly, name will be used to hold text string and marks will be used to hold real numbers.

```
rollno = 19;
```

This is called an assignment statement. The equal sign is an operator that stores the value on its right into the variable named on its left. After this line executes, the rollno variable will contain the value 19.

Similarly, name will contain David and marks will contain 89.8.

```
System.out.println("Your roll number is " + rollno);
```

The println() method prints the characters between quotes to the console. You can also use the + operator to concatenate the contents of a variable to a string.

Literals-A constant value in Java is created by using a literal representation of it. A literal can be used anywhere a value of its type is allowed. In above program, we have used following types of literals

Literal	Type of Literal
19	Integer literal
"David"	String literal
89.8	Double literal (real number)
"Your roll number is "	String literal
"Your name is "	String literal
"Your marks is "	String literal

Type conversion and casting

- In Java, there are two kinds of data types – primitives and references. Primitives include int, float, long, double etc. References can be of type classes, interfaces, arrays.
- A value can change its type either implicitly or explicitly.

Implicit and explicit changes:

Example:

```
int num1 = 4;
```

```
double num2;
```

```
double d = num1 / num2;           //num1 is also converted to double
```

This kind of conversion is called automatic type conversion.

Type casting:

In some cases changes do not occur on its own, the programmer needs to specify the type explicitly. This is called **casting**. However, conversion and casting can be performed, only if they are compatible with the some rules defined.

Syntax for casting:

Identifier2 = (type) identifier1;

Example:

num 1= (int) num2;

Widening vs. Narrowing conversion: Basically, primitives are converted using either one of the two forms.

Widening Conversion: Changes a value to a type with a larger range. This will not result in any data loss.

Example, converting from **int to long**

Narrowing Conversion: Changes a value to a type with a shorter range. Some data may get lost.

Example, converting from **double to float**.

Conversion and casting of primitives: There are three situations in which conversion of primitive type takes place.

- Assignment
- Arithmetic Promotion
- Method call

Assignment conversion:

Assignment conversion occurs when a variable of one type is assigned to another type. Only widening conversions can take place through assignment.

Example:

```
double x;  
int y = 50;  
x = y;
```

1.10 AUTOMATIC TYPE PROMOTION IN EXPRESSIONS ARRAYS

Type promotion in Expressions

While evaluating expressions, the intermediate value may exceed the range of operands and hence the expression value will be promoted. Some conditions for type promotion are:

Java automatically promotes each byte, short, or char operand to int when evaluating an expression.

If one operand is a long, float or double the whole expression is promoted to long, float or double respectively.

Example:

//Java program to illustrate Type promotion in Expressions

```
class Test  
{  
    public static void main(String args[])  
    {  
        byte b = 42;  
        char c = 'a';  
        short s = 1024;  
        int i = 50000;  
        float f = 5.67f;  
        double d = .1234;
```



```
// The Expression
double result = (f * b) + (i / c) - (d * s);

//Result after all the promotions are done
System.out.println("result = " + result);
}
}
Output:
Result = 626.7784146484375
```

Explicit type casting in Expressions

While evaluating expressions, the result is automatically updated to larger data type of the operand. But if we store that result in any smaller data type it generates compile time error, due to which we need to type cast the result.

Example:

```
//Java program to illustrate type casting int to byte
class Test
{
    public static void main(String args[])
    {
        byte b = 50;

        //type casting int to byte
        b = (byte)(b * 2);
        System.out.println(b);
    }
}
Output
100
```

1.11 OPERATORS

Operators are special symbols (characters) that carry out operations on operands (variables and values). For example, `+` is an operator that performs addition. In Java variables article, you learned to declare variables and assign values to variables. Now, you will learn to use operators to manipulate variables.

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups –

- Arithmetic Operators
- Relational Operators
- Bitwise Operators
- Logical Operators
- Assignment Operators
- Misc Operators
- Introduction

1.12 ARITHMETIC OPERATORS, THE BIT WISE OPERATORS, RELATIONAL OPERATORS, BOOLEAN LOGICAL OPERATORS, ASSIGNMENT OPERATOR, OPERATOR PRECEDENCE

The Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators

Assume integer variable A holds 10 and variable B holds 20, then

Operator	Description	Example
<code>+</code> (Addition)	Adds values on either side of the operator.	<code>A + B</code> will give 30
<code>-</code> (Subtraction)	Subtracts right-hand operand from left-hand operand.	<code>A - B</code> will give -10
<code>*</code> (Multiplication)	Multiplies values on either side of the operator.	<code>A * B</code> will give 200
<code>/</code> (Division)	Divides left-hand operand by right-hand operand.	<code>B / A</code> will give 2
<code>%</code> (Modulus)	Divides left-hand operand by right-hand operand and returns remainder.	<code>B % A</code> will give 0
<code>++</code> (Increment)	Increases the value of operand by 1.	<code>B++</code> gives 21

-- (Decrement)	Decreases the value of operand by 1.	B-- gives 19
----------------	--------------------------------------	--------------

The Relational Operators-There are following relational operators supported by Java language. Assume variable A holds 10 and variable B holds 20, then

Operator	Description	Example
== (equal to)	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!= (not equal to)	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
> (greater than)	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
< (less than)	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>= (greater than or equal to)	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<= (less than or equal to)	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

The Bitwise Operators-Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte.

Bitwise operator works on bits and performs bit-by-bit operation. Assume if a = 60 and b = 13; now in binary format they will be as follows –

a = 0011 1100

b = 0000 1101

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

The following table lists the bitwise operators

Assume integer variable A holds 60 and variable B holds 13 then

Operator	Description	Example
& (bitwise and)	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
(bitwise or)	Binary OR Operator copies a bit if it exists in either operand.	(A B) will give 61 which is 0011 1101

\wedge (bitwise XOR)	Binary XOR Operator copies the bit if it is set in one operand but not both.	$(A \wedge B)$ will give 49 which is 0011 0001
\sim (bitwise compliment)	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	$(\sim A)$ will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
\ll (left shift)	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	$A \ll 2$ will give 240 which is 1111 0000
\gg (right shift)	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	$A \gg 2$ will give 15 which is 1111
\ggg (zero fill right shift)	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.	$A \ggg 2$ will give 15 which is 0000 1111

The Logical Operators-The following table lists the logical operators
Assume Boolean variables A holds true and variable B holds false, then

Operator	Description	Example
$\&\&$ (logical and)	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	$(A \&\& B)$ is false
$\ \ $ (logical or)	Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true.	$(A \ \ B)$ is true
$!$ (logical not)	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	$!(A \&\& B)$ is true

The Assignment Operators

Following are the assignment operators supported by Java language

Operator	Description	Example
$=$	Simple assignment operator. Assigns values from right side operands to left side operand.	$C = A + B$ will assign value of $A + B$ into C
$+=$	Add AND assignment operator. It adds right operand to the left operand and	$C += A$ is

	assign the result to left operand.	equivalent to $C = C + A$
<code>-=</code>	Subtract AND assignment operator. It subtracts right operand from the left operand and assign the result to left operand.	$C -= A$ is equivalent to $C = C - A$
<code>*=</code>	Multiply AND assignment operator. It multiplies right operand with the left operand and assign the result to left operand.	$C *= A$ is equivalent to $C = C * A$
<code>/=</code>	Divide AND assignment operator. It divides left operand with the right operand and assign the result to left operand.	$C /= A$ is equivalent to $C = C / A$
<code>%=</code>	Modulus AND assignment operator. It takes modulus using two operands and assign the result to left operand.	$C \% = A$ is equivalent to $C = C \% A$
<code><<=</code>	Left shift AND assignment operator.	$C <<= 2$ is same as $C = C << 2$
<code>>>=</code>	Right shift AND assignment operator.	$C >>= 2$ is same as $C = C >> 2$
<code>&=</code>	Bitwise AND assignment operator.	$C \&= 2$ is same as $C = C \& 2$
<code>^=</code>	bitwise exclusive OR and assignment operator.	$C \wedge = 2$ is same as $C = C \wedge 2$
<code> =</code>	bitwise inclusive OR and assignment operator.	$C = 2$ is same as $C = C 2$

Miscellaneous Operators-There are few other operators supported by Java Language.

Conditional Operator (? :) -conditional operator is also known as the ternary operator. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide, which value should be assigned to the variable. The operator is written as –
variable x = (expression) ? value if true : value if false

Following is an example –

Example

```
public class Test {  
    public static void main(String args[]) {  
        int a, b;  
        a = 10;  
        b = (a == 1) ? 20: 30;  
        System.out.println( "Value of b is : " + b );  
  
        b = (a == 10) ? 20: 30;  
        System.out.println( "Value of b is : " + b );  
    }  
}
```

This will produce the following result –

Output

Value of b is : 30

Value of b is : 20

Instanceof Operator-This operator is used only for object reference variables. The operator checks whether the object is of a particular type (class type or interface type). instanceof operator is written as –

(Object reference variable) instanceof (class/interface type)

If the object referred by the variable on the left side of the operator passes the IS-A check for the class/interface type on the right side, then the result will be true. Following is an example –

Example

```
public class Test {  
    public static void main(String args[]) {  
  
        String name = "James";  
  
        // following will return true since name is type of String  
        boolean result = name instanceof String;  
        System.out.println( result );  
    }  
}
```

This will produce the following result –

Output

true

This operator will still return true, if the object being compared is the assignment compatible with the type on the right. Following is one more example –

Example

```
class Vehicle { }

public class Car extends Vehicle {

    public static void main(String args[]) {

        Vehicle a = new Car();
        boolean result = a instanceof Car;
        System.out.println( result );
    }
}
```

This will produce the following result

Output

true

Operator Precedence- Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator –

For example, $x = 7 + 3 * 2$; here x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with $3 * 2$ and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity
Postfix	expression++ expression--	Left to right
Unary	++expression --expression +expression -expression ~ !	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >> >>>	Left to right
Relational	< > <= >= instanceof	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR		Left to right
Logical AND	&&	Left to right
Logical OR		Left to right

Conditional	?:	Right to left
Assignment	= += -= *= /= %= ^= = <<= >>= >>>=	Right to left

1.13 CONTROL STATEMENTS: SELECTION STATEMENTS - IF, SWITCH: ITERATION STATEMENTS - WHILE, DO-WHILE, FOR NESTED LOOPS, JUMP STATEMENTS

A control statement works as a determiner for deciding the next task of the other statements whether to execute or not. An 'If' statement decides whether to execute a statement or which statement has to execute first between the two. In Java, the control statements are divided into three categories which are selection statements, iteration statements, and jump statements. A program can execute from top to bottom but if we use a control statement. We can set order for executing a program based on values and logic.

In Java, control statements can be categorized into the following categories

1. Selection statements (if, switch)
2. Iteration statements (while, do-while, for, for-each)
3. Jump statements (break, continue, return)

Selection statements

As the name implies, selection statements in Java executes a set of statements based on the value of an expression or value of a variable. A programmer can write several blocks of code and based on the condition or expression, one block can be executed. Selection statements are also known as conditional statements or branching statements. Selection statements provided by Java are if and switch.

if statement

The if statement is used to execute a set of statements based on the boolean value returned by an expression.

Syntax of if statement is as shown below:

```
if(condition/expression)
{statements;}
if(condition/expression)
{statements;}
```

If you want to execute only one statement in the if block, you can omit the braces and write it as shown below:

```
if(condition/expression)
    statement;
```

In the above syntax, the set of statements are executed only when the condition or expression evaluates to true. Otherwise, the statement after the if block is executed.

Let's consider the following example which demonstrates the use of if statement:

```
int a = 10;
if(a < 10)
    System.out.println("a is less than 10");
int a = 10;
if(a < 10)
    System.out.println("a is less than 10");
```

The output for the above piece of code will be a blank screen i.e. no output because the condition `a < 10` returns false and the print statement will not be executed. In the above piece of code, instead of

displaying nothing when the condition returns false, we can show an appropriate message. This can be done using the else statement.

Remember that you can't use *else* without using *if* statement. By using the *else* statement our previous example now becomes as follows:

```
int a = 10;
if(a < 10)
    System.out.println("a is less than 10");
else
    System.out.println("a is greater than or equal to 10");
int a = 10;
```

Nested if statement

If you want to test for another condition after the initial condition available in the *if* statement, it is common sense to write another *if* statement. Such nesting of one *if* statement inside another *if* statement is called a nested *if* statement. Syntax of nested *if* statement is as shown below:

```
{
    if(condition)
    {
        if(condition)
        {
            ...
        }
    }
}
```

We can also combine multiple conditions into a single expression using the logical operators. As an example for nested *if*, let's look at a program for finding the largest of the three numbers a, b and c. Below we will look at a code fragment which shows logic for finding whether *a* is the greatest or not:

```
if(a > b)
{
    if(a > c)
    {
        System.out.println("a is the largest number");
    }
}
1
2
3
4
5
6
7
if(a > b)
{
    if(a > c)
    {
        System.out.println("a is the largest number");
    }
}
```

As mentioned above we can convert a nested if into a simple if by combining multiple condition into a single condition by using the logical operators as shown below:

```
if(a > b && a > c)
{
    System.out.println("a is the largest number");
}
if(a > b && a > c)
```

if-else-if Ladder- The if-else-if ladder is a multi-way decision making statement. If you want to execute one code segment among a set of code segments, based on a condition, you can use the if-else-if ladder. The syntax is as shown below:

```
if(condition)
{
    Statements;
}
else if(condition)
{
    Statements;
}
else if(condition)
if(condition)
{
    Statements;
}
else if(condition)
{
    Statements;
}
else if(condition)
{
    Statements;
}
...
else
{
    Statements;
}
```

Looking at the above syntax, you can say that only one of the blocks execute based on the condition. When all conditions fail, the else block will be executed. As an example for if-else-if ladder, let's look at a program for finding whether the given character is a vowel (a, e, i, o, u) or not:

```
char ch;
ch = 'e'; //You can also read input from the user
if(ch == 'a')
{
    System.out.println("Entered character is a vowel");
}
else if(ch=='e')
{
    System.out.println("Entered character is a vowel");
}
```

```

}
else if(ch=='i')
{
    System.out.println("Entered character is a vowel");
}
else if(ch=='o')
{
    System.out.println("Entered character is a vowel");
}
else if(ch=='u')
{
    System.out.println("Entered character is a vowel");
}
else
{
    System.out.println("Entered character is not a vowel");
}
char ch;
ch = 'e'; //You can also read input from the user
if(ch == 'a')
{
    System.out.println("Entered character is a vowel");
}
else if(ch=='e')
{
    System.out.println("Entered character is a vowel");
}
else if(ch=='i')
{
    System.out.println("Entered character is a vowel");
}
else if(ch=='o')
{
    System.out.println("Entered character is a vowel");
}
else if(ch=='u')
{
    System.out.println("Entered character is a vowel");
}
else
{
    System.out.println("Entered character is not a vowel");
}

```

In the above program, the else block serves as a default block that is to be executed in case if the entered character is not a, e, i, o or u.

switch Statement-The switch statement is another multi-way decision making statement. You can consider the *switch* as an alternative for if-else-if ladder. Every code segment written using a *switch* statement can be converted into an if-else-if ladder equivalent. Use *switch* statement only

when you want a literal or a variable or an expression to be equal to another value. The *switch* statement is simple than an equivalent *if-else-if* ladder construct.

```
{
    case label1:
        Statements;
        break;
    case label2:
        Statements;
        break;
    case label3:
        Statements;
        break;
    ...
    default:
        Statements;
        break;
}
```

Some points to remember about switch statement:

1. The expression in the above syntax can be a byte, short, int, char or an enumeration. From Java 7 onwards, we can also use Strings in a switch
2. case is a keyword use to specify a block of statements to be executed when the value of the expression matches with the corresponding label
3. The break statement at the end of each case is optional. If you don't use break at the end of a case, all the subsequent cases will be executed until a break statement is encountered or the end of the switch statement is encountered.
4. The default block is also optional. It is used in a switch statement to specify a set of statements that should be executed when none of the labels match with the value of the expression. It is a convention to place the default block at the end of the switch statement but not a rule.

As an example for switch statement, let's consider the vowel's example discussed above:

```
char ch;
ch = 'u'; //You can also read input from the user
switch(ch)
{
    case 'a':
        System.out.println("Entered character is a vowel");
        break;
    case 'e':
        System.out.println("Entered character is a vowel");
        break;
    case 'i':
        System.out.println("Entered character is a vowel");
        break;
    case 'o':
        System.out.println("Entered character is a vowel");
        break;
    case 'u':
        System.out.println("Entered character is a vowel");
        break;
}
```

```

        default:
            System.out.println("Entered character is not a vowel");
            break; //There is no need of this break. You can omit this if you want
    }
    char ch;
    ch = 'u'; //You can also read input from the user
    switch(ch)
    {
        case 'a':
            System.out.println("Entered character is a vowel");
            break;
        case 'e':
            System.out.println("Entered character is a vowel");
            break;
        case 'i':
            System.out.println("Entered character is a vowel");
            break;
        case 'o':
            System.out.println("Entered character is a vowel");
            break;
        case 'u':
            System.out.println("Entered character is a vowel");
            break;
        default:
            System.out.println("Entered character is not a vowel");
            break; //There is no need of this break. You can omit this if you want
    }

```

Iteration Statements- What is the need for iteration statements or looping statements? To answer this, let's consider an example. Suppose you want to read three numbers from user. For this you might write three statements for reading input. In another program you want to read ten numbers. For this you write ten statements for reading input. Yet in another program you want to read ten thousand numbers as input. What do you do? Even copying and pasting the statements to read input takes time. To solve this problem, Java provides us with iteration statements. Iteration statements provided by Java are: for, while, do-while and for-each

while Statement

while is the simplest of all the iteration statements in Java. Syntax of while loop is as shown below:

```

while(condition)
{
    Statements;
}
while(condition)
{
    Statements;
}

```

As long as the condition is true, the statements execute again and again. If the statement to be executed in any loop is only one, you can omit the braces. As the while loop checks the condition at the start of the loop, statements may not execute even once if the condition fails. If you want to execute the body of a loop atleast once, use do-while loop.

Let's see an example for reading 1000 numbers from the user:

While selection statements are used to select a set of statements based on a condition, iteration statements are used to repeat a set of statements again and again based on a condition for finite or infinite number of times.

```
int i;
i = 0;
while(i < 1000)
{
    //Code for reading input...
    i++;
}
1
int i;
i = 0;
while(i < 1000)
{
    //Code for reading input...
    i++;
}
```

The above loop works in this way. First *i* is initialized to zero, then condition *i* < 1000 is evaluated which is true. So the statements for reading input are executed and finally *i* is incremented by 1 (*i*++). Again the condition is evaluated and so on. When *i* value becomes 1000, condition fails and control exits the loop and goes to the next line after the while statement.

do-while Statement

do-while statement is similar to while loop except that the condition is checked after executing the body of the loop. So, the do-while loop executes the body of the loop atleast once. Syntax of do-while is as shown below:

```
do
{
    Statements;
}while(condition);
do
{
    Statements;
}while(condition);
```

This loop is generally used in cases where you want to prompt the user to ask whether he/she would like to continue or not. Then, based on user's input, the body of the loop will be executed again or else the control quits the loop.

Let's consider an example where the body of the loop will be executed based on the user's input:

```
char ch;
do
{
    //Statements;
    System.out.println("Do you want to continue? Enter Y or N);
    //Statement for reading the character from the user
}while(ch == 'Y');
```

```

char ch;
do
{
    //Statements;
    System.out.println("Do you want to continue? Enter Y or N);
    //Statement for reading the character from the user
}while(ch == 'Y');

```

In the above code segment, the body of the *do-while* loop executes again and again as long as the user inputs the character Y when prompted.

for Statement

The for statement might look busy but provides more control than other looping statements. In a for statement, initialization of the loop control variable, condition and modifying the value of the loop control variable are combined into a single line. The syntax of for statement is as shown below:

```

for(initialization; condition; iteration)
{
    Statements;
}
for(initialization; condition; iteration)
{
    Statements;
}

```

In the above syntax, the initialization is an assignment expression which initializes the loop control variable and/or other variables. The initialization expression evaluates only once. The condition is a boolean expression. If the condition evaluates to true, the body of the loop is executed. Else, the control quits the loop. Every time after the body executes, the iteration expression is evaluated. Generally, this is an increment or decrement expression which modifies the value of the control variable. All the three parts i.e. initialization, condition and iteration are optional. Let's consider an example code segment which prints the numbers from 1 to 100:

```

for(int i = 1; i <= 100; i++)
{
    System.out.println("i = " + i );
}
1
2
3
4
for(int i = 1; i <= 100; i++)
{
    System.out.println("i = " + i );
}

```

In the above code segment, `int i = 1` is the initialization expression, `i <= 100` is the condition and `i++` is the iteration expression.

We can create an infinite for loop as shown below:

```

for( ; ; )
{
    Statements;
}
1

```

```

2
3
4
for( ; ; )
{
    Statements;
}

```

for-each Statement

The for-each statement is a variation of the traditional for statement which has been introduced in JDK 5. The for-each statement is also known as enhanced for statement. It is a simplification over the for statement and is generally used when you want to perform a common operation sequentially over a collection of values like an array etc... The syntax of for-each statement is as shown below:

```

for(type var : collection)
{
    Statements;
}
for(type var : collection)
{
    Statements;
}

```

The data type of var must be same as the data type of the collection. The above syntax is can be read as, for each value in collection, execute the statements. Starting from the first value in the collection, each value is copied into var and the statements are executed. The loop executes until the values in the collection completes.

Let's consider a code segment which declares an array containing 10 values and prints the sum of those 10 values using a for-each statement:

```

int[ ] array = {1,2,3,4,5,6,7,8,9,10};
int sum = 0;
for(int x : array)
{
    sum += x;
}
System.out.println("Sum is: "+sum);
int[ ] array = {1,2,3,4,5,6,7,8,9,10};
int sum = 0;
for(int x : array)
{
    sum += x;
}
System.out.println("Sum is: "+sum);

```

The equivalent code segment for the above code using a for loop is as shown below:

```

int[ ] array = {1,2,3,4,5,6,7,8,9,10};
int sum = 0;
for(int x = 0; x < 10; x++)
{
    sum += array[x];
}

```



```
System.out.println("Sum is: "+sum);
```

```
int[ ] array = {1,2,3,4,5,6,7,8,9,10};
```

```
int sum = 0;
```

```
for(int x = 0; x < 10; x++)
```

```
{
```

```
    sum += array[x];
```

```
}
```

```
System.out.println("Sum is: "+sum);
```

Nested Loops-A loop inside another loop is called as a nested loop. For each iteration of the outer loop the inner loop also iterates. Syntax of a nested loop is shown below:

outer loop

```
{
```

```
    inner loop
```

```
{
```

```
        Statements;
```

```
}
```

```
}
```

outer loop

```
{
```

```
    inner loop
```

```
{
```

```
        Statements;
```

```
}
```

```
}
```

Let's consider an example which demonstrates a nested loop:

```
for(int i = 0; i < 5; i++)
```

```
{
```

```
    for(int j = 0; j <= i; j++)
```

```
{
```

```
        System.out.println("*");
```

```
}
```

```
}
```

```
for(int i = 0; i < 5; i++)
```

```
{
```

```
    for(int j = 0; j <= i; j++)
```

```
{
```

```
        System.out.println("*");
```

```
}
```

```
}
```

Output for the above code segment is:

```
*
```

```
**
```

```
***
```

```
****
```

```
.....
```

Jump Statements-As the name implies, jump statements are used to alter the sequential flow of the program. Jump statements supported by Java are: break, continue and return.

break Statement-The break statement has three uses in a program. First use is to terminate a case inside a switch statement, second use to terminate a loop and the third use is, break can be used as a sanitized version of goto which is available in other languages. First use is already explained above. Now we will look at the second and third use of break statement.

The break statement is used generally to terminate a loop based on a condition. For example, let's think that we have written a loop which reads data from a heat sensor continuously. When the heat level reaches to a threshold value, the loop must break and the next line after the loop must execute which might be a statement to raise an alarm. In this case the loop will be infinite as we don't know when the value from the sensor will match the threshold value. General syntax of the break statement inside a loop is shown below:

Loop

```
{
    //Statements;
    if(condition)
        break;
    //Statements after break;
}
Loop
{
    //Statements;
    if(condition)
        break;
    //Statements after break;
}
```

As an example to demonstrate the break statement inside a loop, let's look at the following code segment:

```
for(i = 1; i <= 10; i++)
{
    System.out.println(i);
    if(i == 5)
        break;
}
for(i = 1; i <= 10; i++)
{
    System.out.println(i);
    if(i == 5)
        break;
}
```

Output for the above code segment is:

```
1
2
3
4 5
```

When used inside a nested loop, break statement makes the flow of control to quit the inner loop only and not the outer loop too. Another use of the break statement is, it can be used as an alternative to

goto statement which is available in other programming languages like C and C++. General syntax of break label statement is as shown below

```
label1:
{
    Statements;
    label2:
    {
        Statements;
        if(condition)
            break label1;
    }
    Statements;
}
label1:
{
    Statements;
    label2:
    {
        Statements;
        if(condition)
            break label1;
    }
    Statements;
}
```

In the above syntax, when the condition becomes true, control transfers from that line to the line which is available after the block labeled label1. This form of break is generally used in nested loops in which the level of nesting is high and you want the control to jump from the inner most loop to the outermost loop. This form of break can be used in all loops or in any other block. Syntax for writing a label is, any valid identifier followed by a colon (:).

continue Statement

Unlike break statement which terminates the loop, continue statement is used to skip rest of the statements after it for the current iteration and continue with the next iteration. Syntax of continue is as shown below:

```
Loop
{
    //Statements;
    if(condition)
        continue;
    //Statements after continue;
}
Loop
{
    //Statements;
    if(condition)
        continue;
    //Statements after continue;
}
```

Like break statement, continue can also be used inside all loops in Java.

As an example to demonstrate the continue statement inside a loop, let's look at the following code segment:

```
for(i = 1; i <= 5; i++)
{
    if(i == 3)
        continue;
    System.out.println(i);
}
for(i = 1; i <= 5; i++)
{
    if(i == 3)
        continue;
    System.out.println(i);
}
```

Output for the above code segment is:

```
1
2
4
5
```

When the value of i becomes 3, continue is executed and the print statement gets skipped.

return Statement

The return statement can only be used inside methods which can return control or a value back to the calling method. The return statement can be written at any line inside the body of the method. Common convention is to write the return statement at the end of the method's body. Let's consider a small example to demonstrate the use of return statement:

```
void fact(int n)
{
    if(n == 0) return 1;
    else if(n == 1) return 1;
    else return fact(n)*fact(n-1);
}
void fact(int n)
{
    if(n == 0) return 1;
    else if(n == 1) return 1;
    else return fact(n)*fact(n-1);
}
```

1.14 SUMMARY

Java is one of the world's most widely used computer language. Java is a simple, general-purpose, object-oriented, interpreted, robust, secure, architecture-neutral, portable, high-performance, multithreaded computer language. It is intended to let application developers "write once, run anywhere" (WORA), meaning that code that runs on one platform does not need to be recompiled to run on another. Java technology is both a programming language and a platform. Java is a high level, robust, secured and object-oriented programming language. And any hardware or software environment in which a program runs, is known as a platform. Since Java has its own runtime environment (JRE) and API, it is called platform. In the Java programming language, all source code

is first written in plain text files ending with the .java extension. Those source files are then compiled into .class files by the javac compiler. A .class file does not contain code that is native to your processor; it instead contains bytecodes — the machine language of the Java Virtual Machine¹ (Java VM). The java launcher tool then runs your application with an instance of the Java Virtual Machine.

1.15 SELF ASSESSMENT QUESTIONS

1. Explain the bit wise operators
2. Explain Control Statements
3. Explain Operator Precedence.
4. Explain blocks of codes
5. Explain Lexical Issues - White Space, Identifiers, Literal
6. Explain data types
7. Explain control statements

1.16 SUGGESTED READINGS

- 1 The complete reference Java –2: V Edition By Herbert Schildt Pub. TMH.
2. SAMS teach yourself Java – 2: 3rd Edition by Rogers Cedenhead and Leura Lemay Pub. Pearson Education
3. Java: A Beginner's Guide by Herbert Schildt
4. Java: Programming Basics for Absolute Beginners by Nathan Clark.