Object Oriented Programming

Object Oriented Programming (OOP) is a programming concept used in several modern programming languages, like C++, Java and Python. Before Object Oriented Programming, programs were viewed as procedures that accepted data and produced an output. There was little emphasis given on the data that went into those programs. Object Oriented Programming works on the principle that objects are the most important part of your program. Manipulating these objects to get results is the goal of Object Oriented Programming.

Object Oriented Programming also addresses several other weaknesses of the other programming techniques (unstructured, procedural and modular). Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods.

For example, a person is an object which has certain properties such as height, gender, age, etc. It also has certain methods such as move, talk, and so on.

Definition: Object Oriented Programming is an approach that provides a way of modularizing programs by creating partitioned memory area for both data and functions that can be used as templates for creating copies of such modules on demand.

Basic Concepts of Object Oriented Programming

1. Object

An entity that has state and behaviour is known as an object. e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.

An object has three characteristics:

- State: represents the data (value) of an object.
- Behaviour: represents the behaviour (functionality) of an object such as deposit, withdraw, etc.

- Identity: An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

For Example, Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behaviour.

An object is an instance of a class. A class is a template or blueprint from which objects are created. So, an object is the instance (result) of a class.

Object Definitions:

- An object is a real-world entity.
- An object is a runtime entity.
- The object is an entity which has state and behaviour.
- The object is an instance of a class.

2. Class

A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

Another Definition: A class is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type.

A class in Java can contain:

- Fields
- Methods
- Constructors
- Blocks
- Nested class and interface

3. Abstraction

It refers to, providing only essential information to the outside world and hiding their background details. For example, a web server hides how it processes data it receives, the end user just hits the endpoints and gets the data back.

4. Encapsulation

Encapsulation is a process of binding data members (variables, properties) and member functions (methods) into a single unit. It is also a way of restricting access to certain properties or component. The best example for encapsulation is a class.

5. Inheritance

The ability to create a new class from an existing class is called Inheritance. Using inheritance, we can create a Child class from a Parent class such that it inherits the properties and methods of the parent class and can have its own additional properties and methods. For example, if we have a class Vehicle that has properties like Color, Price, etc, we can create 2 classes like Bike and Car from it that have those 2 properties and additional properties that are specialized for them like a car has numberOfWindows while a bike cannot. Same is applicable to methods.

6. Polymorphism

The word polymorphism means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance. Polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Benefits of Object Oriented Programming

1. Re-usability: It means reusing some facilities rather than building it again and again. This is done with the use of a class. Programmer can use it 'n' number of times as per his requirement.

2. Data Redundancy: This is a condition created at the place of data storage (you can say Databases) where the same piece of data is held in two separate places. So the data redundancy is one of the greatest advantages of OOP. If a user wants a similar functionality in multiple classes he/she can go ahead by

writing common class definitions for the similar functionalities and inherit them.

3. Code Maintenance: This feature is more of a necessity for any programming languages; it helps users from doing re-work in many ways. It is always easy and time-saving to maintain and modify the existing codes with incorporating new changes into it.

4. Security: With the use of data hiding and abstraction mechanism, we are filtering out limited data to exposure which means we are maintaining security and providing necessary data to view.

5. Design Benefits: Object Oriented Programs forces the designers to have a longer and extensive design phase, which results in better designs and fewer flaws.

6. Better productivity: The above-mentioned features of OOP enhances its users overall productivity. This leads to more work done, finish a better program, having more inbuilt features and easier to read, write and maintain. An OOP programmer cans stitch new software objects to make completely new programs. A good number of libraries with useful functions make it possible.

Features of Java

- **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. The compiler in Java is written in ANSI C with a clean portability boundary.
- **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 an extensive amount of run-time information that can be used to verify and resolve accesses to objects at run-time.

History of Java

Java was originally designed for interactive television, but it was too advanced technology for the digital cable television industry at the time. The history of Java starts with the Green Team. Java team members (also known as Green Team), initiated this project to develop a language for digital devices such as set-top boxes, televisions, etc. However, it was suited for internet programming. Later, Java technology was incorporated by Netscape.

The principles for creating Java programming were "Simple, Robust, Portable, Platform-independent, Secured, High Performance, Multithreaded, Architecture Neutral, Object-Oriented, Interpreted, and Dynamic". Java was developed by James Gosling, who is known as the father of Java, in 1995.

James Gosling, Mike Sheridan, and Patrick Naughton initiated the Java language project in June 1991. The small team of sun engineers called Green Team. Initially designed for small, embedded systems in electronic appliances like set-top boxes. Firstly, it was called "Greentalk" by James Gosling, and the file extension was .gt. After that, it was called ***Oak*** and was developed as a part of the Green project. Oak is a symbol of strength and chosen as a national tree of many countries like the U.S.A., France, Germany, Romania, etc. In 1995, Oak was renamed as "Java" because it was already a trademark by Oak Technologies.

Currently, Java is used in internet programming, mobile devices, games, e-business solutions, etc. There are given significant points that describe the history of Java.

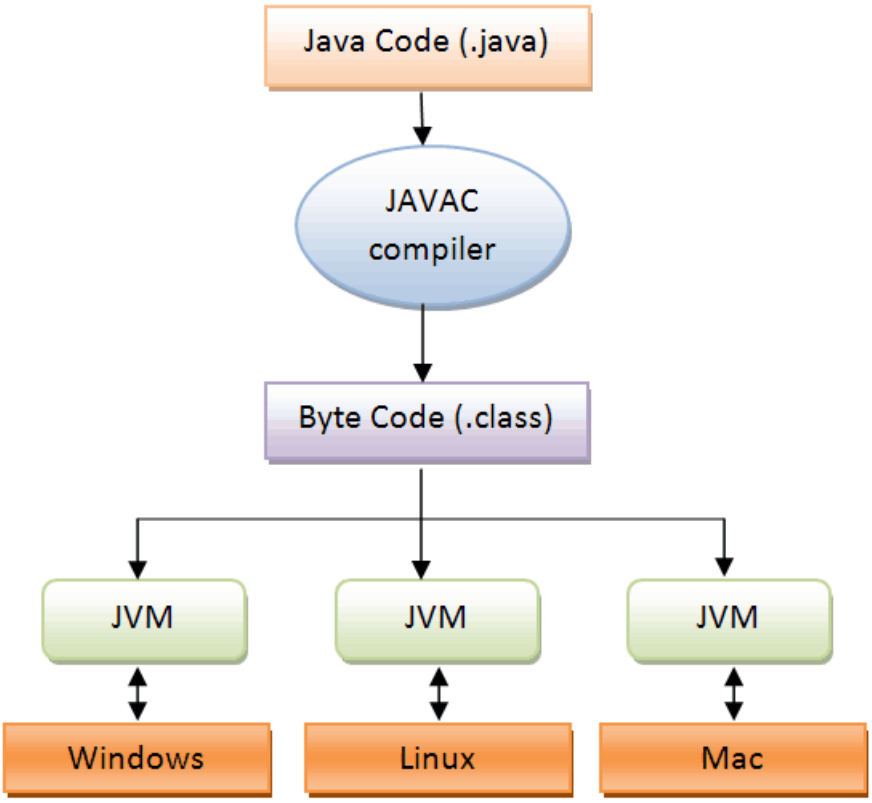
Java Environment

The Java Runtime Environment (JRE) is a set of software tools for development of Java applications. It combines the Java Virtual Machine (JVM), platform core classes and supporting libraries. JRE is part of the Java Development Kit (JDK), but can be downloaded separately.

The JVM is an abstract computing machine, having an instruction set that uses memory. Virtual machines are often used to implement a programming language. The JVM is the cornerstone of the Java programming language. It is

responsible for Java's cross-platform portability and the small size of its compiled code.

The JVM is used to execute Java applications. The Java compiler, javac, outputs bytecodes and puts them into a .class file. The JVM then interprets these bytecodes, which can then be executed by any JVM implementation, thus providing Java's cross-platform portability. The next two figures illustrate the new portable Java compile-time environment.



Java JIT compiler, an integral part of the JVM, can accelerate execution performance many times over previous levels. Long-running, compute-intensive programs show the best performance improvement.

When the JIT compiler environment variable is on (the default), the JVM reads the .class file for interpretation and passes it to the JIT compiler. The JIT compiler then compiles the bytecodes into native code for the platform on which it is running.

Java Tokens

A token is the smallest element of a program that is meaningful to the compiler. Tokens can be classified as follows:

- Keywords
- Identifiers
- Literals
- Operators
- Separators

Keywords

Keywords are pre-defined or reserved words in a programming language. Each keyword is meant to perform a specific function in a program. Since keywords are referred names for a compiler, they can't be used as variable names because by doing so, we are trying to assign a new meaning to the keyword which is not allowed. Java language supports following 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

Identifiers

Identifiers are used as the general terminology for naming of variables, functions and arrays. These are user-defined names consisting of an arbitrarily long sequence of letters and digits with either a letter or the underscore (_) as a first character. Identifier names must differ in spelling and case from any keywords. Keywords cannot be used as identifiers; they are reserved for special use. Once declared, you can use the identifier in later program statements to refer to the associated value. A special kind of identifier, called a statement label, can be used in goto statements.

Example : MyVariable , MYVARIABLE ,myvariable ,x,I,x1,i1,_myvariable, $myvariable, sum_of_array , geeks123

Examples of invalid identifiers :

My Variable // contains a space

123geeks // Begins with a digit

a+c // plus sign is not an alphanumeric character

variable-2 // hyphen is not an alphanumeric character

sum_&_difference // ampersand is not an alphanumeric character

Literals

Literals in java are sequence of characters that represent constant values to be stored in variables. Java language specifies five major types of literals. They are Integer literals, floating point literals, character literals, string literals and Boolean literals. Each of them has type associated with it. The type describe how the values behave and how they are stored.

Operators

Java provides many types of operators which can be used according to the need. They are classified based on the functionality they provide. Some of the types are Arithmetic Operators, Unary Operators, Assignment Operator, Relational Operators, Logical Operators, Ternary Operator, Bitwise Operators, Shift Operators and instance of operator.

Separators

A separator is a symbol that is used to separate a group of code from one another is called as separators in java. In java, there are few characters that are used as a separator. The most commonly used separator is a semicolon.

Symbol	**Name**	**Purpose**
()	Parentheses	Used to enclose an argument in the method definition. Also used for defining the expression in control statements.

{ }	Braces	Used to define a block of code for classes and methods. Also used to contain the values of arrays.
[]	Brackets	Used to declare an array type. Also used when dereferencing array values.
;	Semicolon	Used to separate or terminate the statement.
,	Comma	Used to separate identifiers (or) Variable declarations. Also used to chain statements together inside a for a statement.
.	Period	Used to separate package names from sub-packages and classes. Also used to separate a variable or method from a reference variable.

Variables

A variable is an identifier that denotes a piece of memory that can store a data value. A variable thus has a data type. Variables are typically used to store information which a program needs to do its job. This can be any kind of information ranging from texts, numbers, temporary results of multi step calculations etc.

Syntax for declaring a variable is ***datatype variable_name ;***

Eg: *int num;*

Rules for naming variables:

- All variable names must begin with a letter of the alphabet, an underscore, or (_), or a dollar sign ($).
- After the first initial letter, variable names may also contain letters and the digits 0 to 9. No spaces or special characters are allowed.
- The name can be of any length.

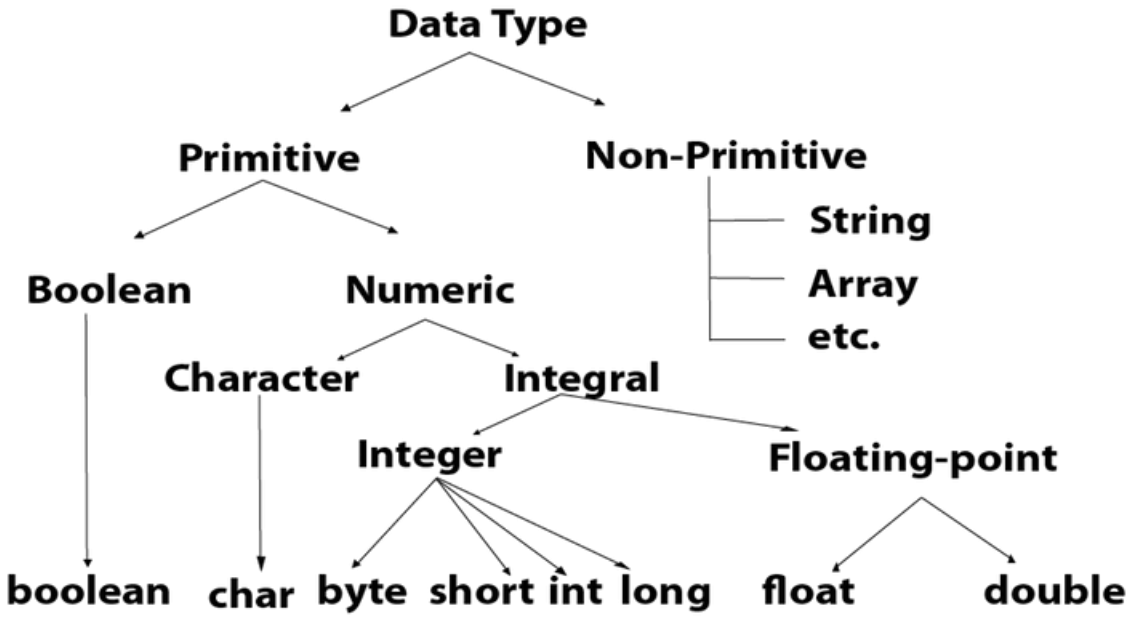
- Uppercase characters are distinct from lowercase characters. Using ALL uppercase letters are primarily used to identify constant variables. Variable names are case-sensitive.
- You cannot use a java keyword (reserved word) for a variable name.

Data Types

Data type defines the values that a variable can take, for example if a variable has ***int*** data type, it can only take integer values. In java we have two categories of data types : Primitive data types and Non-primitive data types.

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, Arrays and Strings.



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. Java is a statically-typed programming language. It means, all variables must be declared before its use. That is why we need to declare variable's type and name. Primitive data types in java with their size and default value are given in the table below.

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

Operators

Java provides a rich set of operators to manipulate variables. Operators are special symbols that carry out operations on operands (variables and values). Java operators can be classified into a number of related categories as

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. Assignment operators
5. Increment/decrement operators (Unary operators)
6. Conditional operators
7. Bitwise operators
8. Special operators

1. Arithmetic operators

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication, etc. Arithmetic operators are binary operators and are used in mathematical expressions in the same way that they are used in algebra.

Operator	Meaning
+	Addition (also used for string concatenation)
-	Subtraction Operator
*	Multiplication Operator
/	Division Operator
%	Remainder Operator

Example 1-Arithmentic operators

```
class ArithmenticOp {
  public static void main(String args[])
 {
     int a = 10, b = 20,c = 25, d = 25;
     System.out.println("a + b = " + (a + b) );
     System.out.println("a - b = " + (a - b) );
     System.out.println("a * b = " + (a * b) );
     System.out.println("b / a = " + (b / a) );
     System.out.println("b % a = " + (b % a) );
     System.out.println("c % a = " + (c % a) );
  }
}
```

Output

```
a + b = 30
a - b = -10
a * b = 200
b / a = 2
b % a = 0
c % a = 5
```

2. Relational Operators

Relational operators determine the relationship between the two operands. It checks if an operand is greater than, less than, equal to, not equal to and so on. Depending on the relationship, it is evaluated to either true or false. Following are relational operators supported by Java language.

Operator	**Description**
== (equal to)	Checks if the values of two operands are equal or not, if yes then condition becomes true.

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

Example 2-Relational operators

```
class RelationalOp
 {
        public static void main(String args[])
        {
         int a = 10, b = 20;
        System.out.println("a == b = " + (a == b) );
        System.out.println("a != b = " + (a != b) );
        System.out.println("a > b = " + (a > b) );
        System.out.println("a < b = " + (a < b) );
        System.out.println("b >= a = " + (b >= a) );
         System.out.println("b <= a = " + (b <= a) );
        }
}
```

Output

a == b = false

a != b = true

a > b = false

a < b = true

b >= a = true

b <= a = false

3. Logical Operators

These operators are used to perform “logical AND” and “logical OR” operation, i.e. the function similar to AND gate and OR gate in digital electronics. These are binary operators and the second condition is not evaluated if the first one is false. They are used extensively to test for several conditions for making a decision. "logical NOT" operator is a unary logical operator used for inverting a boolean value.

Operator	**Description**
&& (logical and)	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.
|| (logical or)	Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes 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.

Example 3-Logical operators

```
class LogicalOp
{
  public static void main(String args[])
{
    boolean a = true, b = false;
    System.out.println("a && b = " + (a&&b));
```

```
    System.out.println("a || b = " + (a||b) );
    System.out.println("!(a && b) = " + !(a && b));
  }
}
```

Output

```
a && b = false
a || b = true
!(a && b) = true
```

4. Assignment Operators

Assignment operators are used in Java to assign values to variables. For example,

```
int age;
age = 5;
```

The assignment operator assigns the value on its right to the variable on its left. Here, 5 is assigned to the variable age using = operator.

Example 4-Assignment operators

```
class AssignmentOperator
 {
   public static void main(String[] args)
 {
      int number1, number2;
      number1 = 5;   // Assigning 5 to number1
      System.out.println(number1);
      number2 = number1; // Assigning value of variable number2 to number1
      System.out.println(number2);
   }
}
```

Output

```
5
5
```

Java provides some special Compound Assignment Operators, also known as Shorthand Assignment Operators. It's called shorthand because it provides a short way to assign an expression to a variable. This operator can be used to connect Arithmetic operator with an Assignment operator.

For example, you write a statement:

a = a+6;

In Java, you can also write the above statement like this:

a += 6;

There are various compound assignment operators used in Java:

Operator	**Meaning**
+=	Increments then assigns
-=	Decrements then assigns
*=	Multiplies then assigns
/=	Divides then assigns
%=	Modulus then assigns

5. Increment/decrement Operators

Increment and decrement unary operator works as follows:

Syntax

val++;

val--;

These two operators have two forms: Postfix and Prefix. Both do increment or decrement in appropriate variables. These two operators can be placed before or after of variables. When it is placed before the variable, it is called prefix. And when it is placed after, it is called postfix.

Following example table, demonstrates the work of Increment and decrement operators with postfix and prefix:

Example	**Description**
val = a++;	Store the value of "a" in "val" then increments.

val = a--;	Store the value of "a" in "val" then decrements.
val = ++a;	Increments "a" then store the new value of "a" in "val".
val = --a;	Decrements "a" then store the new value of "a" in "val".

Example 5-Increment/Decrement operators

```
class OperatorExample
{
public static void main(String args[])
{
int x=10;
System.out.println(x++);//10 (11)
System.out.println(++x);//12
System.out.println(x--);//12 (11)
System.out.println(--x);//10
}
}
```

Output:

```
10
12
12
10
```

Example 6-Increment/Decrement operators

```
class OperatorExample1
{
public static void main(String args[])
{
int a=10;
int b=10;
System.out.println(a++ + ++a);//10+12=22
System.out.println(b++ + b++);//10+11=21
```

```
  }
}
```

Output:

22
21

6. Conditional Operators

The Java Conditional Operator selects one of two expressions for evaluation, which is based on the value of the first operands. It is also called ternary operator because it takes three arguments. The conditional operator is used to handling simple situations in a line.

Syntax

expression1 ? expression2:expression3;

The above syntax means that if the value given in Expression1 is true, then Expression2 will be evaluated; otherwise, expression3 will be evaluated.

Example 7-Ternary operator

```
class TernaryOperator
{
public static void main(String args[])
{
int a=10;
int b=5;
int min=(a<b)?a:b;
System.out.println(min);
}
}
```

Output:

5

7. Bitwise operators

The Java Bitwise Operators allow access and modification of a particular bit inside a section of the data. It can be applied to integer types and bytes, and cannot be applied to float and double. 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 −

Operator	**Meaning**	**Work**
&	Binary AND Operator	Binary & operator work very much the same as logical && operators works, except it works with two bits instead of two expressions. The "Binary AND operator" returns 1 if both operands are equal to 1.
|	Binary OR Operator	Binary | Operator work similar to logical || operators works, except it, works with two bits instead of two expressions. The "Binary OR operator" returns 1 if one of its operands evaluates as 1. if either or both operands evaluate to 1, the result is 1.

^	Binary XOR Operator	It stands for "exclusive OR" and means "one or the other", but not both. The "Binary XOR operator" returns 1 if and only if exactly one of its operands is 1. If both operands are 1, or both are 0, then the result is 0.
~	BinaryComplement Operator	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.
<<	Binary Left Shift Operator	The left operands value is moved left by the number of bits specified by the right operand.
>>	Binary Right Shift Operator	The left operands value is moved right by the number of bits specified by the right operand.
>>>	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.

Example 8-Bitwise operators

```
class bitwiseop
 {
 public static void main(String[] args)
{
  //Variables Definition and Initialization
  int num1 = 30, num2 = 6, num3 =0;
  //Bitwise AND
  System.out.println("num1 & num2 = " + (num1 & num2));
```

```
    //Bitwise OR
   System.out.println("num1 | num2 = " + (num1 | num2) );
    //Bitwise XOR
   System.out.println("num1 ^ num2 = " + (num1 ^ num2) );
    //Binary Complement Operator
   System.out.println("~num1 = " + ~num1 );
   //Binary Left Shift Operator
   num3 = num1 <<  2;
   System.out.println("num1 << 1 = " + num3 );
   //Binary Right Shift Operator
   num3 = num1 >>  2;
   System.out.println("num1 >> 1  = " + num3 );
   //Shift right zero fill operator
   num3 = num1 >>> 2;
   System.out.println("num1 >>> 1 = " + num3 );
    }
}
```

Output

```
num1 & num2 = 6
num1 | num2 = 30
num1 ^ num2 = 24
~num1 = -31
num1 << 1 = 120
num1 >> 1  = 7
num1 >>> 1 = 7
```

8. Special operators

instanceof operator

The Java instanceof Operator is used to determining whether this object belongs to this particular class or not. This operator gives the boolean values such as true or false. If the object referred by the variable on the left side of the operator passes the IS-A check for the class type on the right side, then the result will be true. Otherwise, it returns false as output.

Syntax

object-reference instanceof type;

Example 9-Special operators

```
class SpecialOp
{
  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 );
  }
}
```

Output

```
true
```

dot operator

The dot operator, also known as separator or period used to separate a variable or method from a reference variable.

Control Statements

A control statement in java is a statement that determines whether the other statements will be executed or not. It controls the flow of a program. Statements inside a program are usually executed sequentially. Sometimes a programmer wants to break the normal flow and jump to another statement or execute a set of statements repeatedly. Statements in java which breaks the normal sequential flow of the program are called control statements. Control statements in java enables ***decision making, looping and branching***. Statements that determine which statement to execute and when are known as **decision-making statements**.

Decision making and Branching

Decision making structures have one or more conditions to be evaluated or tested by the program, along with a statement or statements that are to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false. Java programming language provides following types of decision making statements.

- if statements
- switch statement
- conditional operator

if statements

Java if statement is used to test the condition. It checks boolean condition: true or false. There are various types of if statement in Java.

1. Simple if statement

Simple if statement tests the condition. It executes if block if condition is true.

Syntax is

```
if(condition)
{
Statement –block;
}
statement-x;
```

The **Statement –block** may be a single statement or a group of statements. If the condition is ***true***, **Statement –block** will be executed; otherwise **Statement –block** will be skipped and execution will jump to **statement-x**. When the condition is true both **Statement –block and statement –x** are executed in sequence.

Example 10: Simple if statement

```
class IfExample
{
public static void main(String[] args)
{
   int age=20;  //defining an 'age' variable
   if(age>18)            //checking the age
  {
      System.out.print("Age is greater than 18");
   }
}
}
```

Output

Age is greater than 18

2. if-else statement

The Java if-else statement also tests the condition. It executes if block if condition is true otherwise else block is executed.

Syntax is

```
if(condition)
{
True-block Statements;
}
else
{
 False-block Statements;
}
statement-x;
```

If the condition is true, then **True-block Statements** immediately following if statement, are executed; otherwise **False-block Statements** are executed. In either case, either **True-block Statements** or **False-block Statements** will be executed, not both. In both cases control is transferred subsequently to the **statement-x**.

Example 11: if… else statement

```
class IfElseExample
{
	public static void main(String[] args)
	{
		int number=13;	//defining a variable
		if(number%2==0)	//Check if the number is divisible by 2 or not
		{
			System.out.println("even number");
		}
		else
		{
			System.out.println("odd number");
		}
	}
}
```

Output

```
odd number
```

Example 12: if… else statement

```
class LeapYearExample
{
	public static void main(String[] args)
	{
		int year=2020;
		if(((year % 4 ==0) && (year % 100 !=0)) || (year % 400==0))
			System.out.println("LEAP YEAR");
		else
			System.out.println("COMMON YEAR");
	}
}
```

Output

```
LEAP YEAR
```

3. if-else-if ladder statement

if-else-if ladder provides a way of putting ifs together when multipath decisions are involved. A multipath decision is a chain of ifs in which the statements associated with each ***else*** is an ***if***. General form of if-else-if ladder is

```
if(condition1)
        statement-1;
else if(condition2)
        statement-2;
else if(condition3)
        statement-3;
.....................
else if(condition n)
        statement-n;
else
        default-statement;
statement-x;
```

This construct is known as **else if** ladder. The conditions are evaluated from top downwards. As soon as the true condition is found, the statement associated with it is executed and the control is transferred to **statement-x** (skipping rest of the ladder)**.** When all **n** conditions become false, then the final **else** containing **default-statement** the will be executed.

Example 13: if-else-if statement

```
public class PositiveNegativeExample
{
        public static void main(String[] args)
        {
                int number= -13;
                if(number>0)
                        System.out.println("POSITIVE");
                else if(number<0)
                        System.out.println("NEGATIVE");
                else
                        System.out.println("ZERO");
        }
}
```

Output

NEGATIVE

Example 14: if-else-if statement

```
class IfElseIfExample
{
        public static void main(String[] args)
        {
           int marks=65;
           if(marks<50)
           {
                System.out.println("fail");
           }
           else if(marks>=50 && marks<60)
           {
                System.out.println("D grade");
           }
           else if(marks>=60 && marks<70)
           {
                System.out.println("C grade");
           }
           else if(marks>=70 && marks<80)
           {
                System.out.println("B grade");
           }
           else if(marks>=80 && marks<90)
           {
                System.out.println("A grade");
           }
           else if(marks>=90 && marks<100)
           {
                System.out.println("A+ grade");
           }
           else
           {
        System.out.println("Invalid!");
           }
        }
}
```

Output

C grade

4. nested if statement

The nested if statement represents the **if** block within another **if** block. Here, the inner if block condition executes only when outer if block condition is true. Nested **if** is used when a series of decisions are involved. General form is

```
if(condition1)
{
        if(condition2)
        {
                statement-1;
        }
        else
        {
                statement-2;
        }
}
else
{
        Statement-3;
}
statement-x;
```

Here if **condition1** is false, **statement-3** will be executed; otherwise it continues to perform second test. If is **condition2** is true, the **statement-1** will be executed; otherwise **statement-2** will be executed and control will then transferred to **statement-x.**

Example 15: nested-if statement

```
class JavaNestedIfExample
{
                public static void main(String[] args)
                {
                        int age=20;
                        int weight=80;        //Creating two variables for age and weight
                        if(age>=18)     //applying condition on age and weight
                        {
                                if(weight>50)
                                {
                                  System.out.println("You are eligible to donate blood");
                                }
                        }
                }
}
```

Output

```
You are eligible to donate blood
```

switch statement

The switch is a built-in multiway decision statement in java. The Java switch statement executes one statement from multiple conditions. It is like if-else-if ladder statement. In other words, the switch statement tests the equality of a variable against multiple values. The switch statement works with byte, long, int and character. General form of switch statement is

```
switch(expression)
{
        case value-1:
              block-1
              break;  //optional
        case value-2:
              block-2
              break;  //optional
        ..........……
        ...... ...... ......
        default:
              default-block
              break;
}
statement-x;
```

The **expression** is an integer expression or characters. **value-1,value-2**…are constants or constant expressions (evaluate to an integral constant). And are known as case labels. Each of these values should be unique within a switch statement**. block-1, block-2** … are statement lists and may contain zero or more statements. The **break** statement at the end of each block signals the end of a particular cases and causes an exit from the switch statement, transferring control to the **statement-x** following switch.

Example 16: switch statement

```
class SwitchExample
{
      public static void main(String[] args)
     {
                    int number=20;//Declaring a variable for switch expression
                    switch(number)  //Switch expression
                    {
                              //Case statements
                              case 10:
                                        System.out.println("10");
                                        break;
                              case 20:
                                        System.out.println("20");
                                        break;
```

```
                        case 30:
                                System.out.println("30");
                                break;
                        //Default case statement
                        default:
                                System.out.println("Not in 10, 20 or 30");
                }
        }
}
```

Output

```
20
```

Example 17: switch statement

```
class SwitchVowelExample
{
                public static void main(String[] args)
                {
                    char ch='O';
                    switch(ch)
                    {
                        case 'a':
                            System.out.println("Vowel");
                            break;
                        case 'e':
                            System.out.println("Vowel");
                            break;
                        case 'i':
                            System.out.println("Vowel");
                            break;
                        case 'o':
                            System.out.println("Vowel");
                            break;
                        case 'u':
                            System.out.println("Vowel");
                            break;
                        case 'A':
                            System.out.println("Vowel");
                            break;
                        case 'E':
                            System.out.println("Vowel");
                            break;
                        case 'I':
```

```
                                System.out.println("Vowel");
                                break;
                        case 'O':
                                System.out.println("Vowel");
                                break;
                        case 'U':
                                System.out.println("Vowel");
                                break;
                        default:
                                System.out.println("Consonant");
                }
        }
}
```

Output

Vowel

Example 18: switch statement without break statement

```
class SwitchExample2
{
        public static void main(String[] args)
        {
                int number=20;
                switch(number)          //switch expression with int value
                {
                        //switch cases without break statements
                    case 10: System.out.println("10");
                    case 20: System.out.println("20");
                    case 30: System.out.println("30");
                    default:System.out.println("Not in 10, 20 or 30");
                }
        }
}
```

Output

20
30
Not in 10, 20 or 30

Conditional Operator

Refer Operators.

Decision making and Looping

Another set of control statements in java are ***iteration statements***. Iteration statements enable program execution to repeat one or more statements. That is iteration statements forms loops.

A loop is a way of repeating lines of code more than once. The block of code contained within the loop will be executed again and again until the condition required by the loop is met. The process of repeatedly executing a block of statements is known as ***looping***. Statements in the block may be executed any number of times from zero to infinite number. If a loop continues forever, it is called an ***infinite loop***.

A looping process, in general, includes the following steps.

1. Setting and initializing a control variable.
2. Execution of statements in the loop.
3. Test for a specified condition for execution of loop
4. Incrementing/decrementing control variable

Java language provides three types of iterative statements:- **for, while and do-while**.

1. while

The while loop is java's most fundamental looping statement. It repeats a statement or block of statements while its controlling expression (condition) is true. Its general form is

```
Initialisation;
while (condition)
 {
    body of the loop
}
```

Condition can be any Boolean expression. The body of loop will be executed as long as the conditional expression is true. When condition becomes false, control passes to the next line of code immediately following the loop.

Example 19 : while loop

```
class WhileExample
{
        public static void main(String[] args)
        {
                int i=1;
                while(i<=5)
                {
                        System.out.println(i);
                        i++;
                }
        }
}
```

Output

```
1
2
3
4
5
```

2. do-while

The Java do-while loop is used to iterate a part of the program several times. If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use do-while loop. The Java do-while loop is executed at least once because condition is checked after loop body. Its general form is

```
Initialisation;
do
{
    body of the loop
 } while (condition);
```

Each iteration of the do-while loop first executes the body of the loop and then evaluates the condition. If this expression is true, the loop will repeat. Otherwise

the loop terminates. Condition must be a boolean expression.

Example 20 :do- while loop

```
class DoWhileExample
 {
        public static void main(String args[])
        {
                int x = 10;
                do
               {
                        System.out.print("value of x : " + x );
                        x++;
                        System.out.print("\n");
                }while( x < 20 );
        }
}
```

Output

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

3. for

A for loop is a repetition control structure that allows programmer to efficiently write a loop that needs to be executed a specific number of times. A for loop is useful when the number of times a task is to be repeated is previously known. The General form of for loop is

```
for(initialisation; test condition; increment/decrement)
 {
    body of the loop
}
```

The execution of for loop statement is as follows.

1. **Initialisation** of control variable is done first, using assignment statements such as i=1 or count =0.
2. The value of control variable is tested using **condition**. Condition should be a relative expression. If the condition is true, the body of the loop is executed; otherwise loop is terminated and execution continues with the statement that immediately follows loop.
3. When the body of the loop is executed, the control is transferred back to the **for** statement after evaluating the last statement in the loop. Now the control variable is again tested to see whether it satisfies the loop condition. If the condition is satisfied, the body of the loop is again executed.
4. This process continues till the value of the control variable fails to satisfy the **test condition**.

Example 21 : for loop

```
class ForExample
{
        public static void main(String[] args)
        {
                int sum=0;
                for(int i=1;i<=10;i++)
               {
                   System.out.println("Sum of first 10 natural numbers is  "+sum);
                }
        }
}
```

Output

```
Sum of first 10 natural numbers is  55
```

Example 22 : Nested for loop

```
class PyramidExample2
{
	public static void main(String[] args)
	{
		int term=6;
		for(int i=1;i<=term;i++)
		{
			for(int j=term;j>=i;j--)
			{
				System.out.print("* ");
			 }
			 System.out.println();//new line
		 }
	}
}
```

Output

```
* * * * * *
* * * * *
* * * *
* * *
* *
*
```

Jump statements

The Java jumping statements are the control statements which transfer the program execution control to a specific statement. Java has three types of jumping statements ***break, continue and return***. These statements transfer execution control to another part of the program.

1. break

When a break statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop. The Java break statement is used to break loop or switch statement. It breaks the current flow of the program at specified condition. In case of inner loop, it breaks only inner loop. Java break statement can be used with all types of loops such as for loop, while loop and do-while loop.

Example 23 : break statement

```
class BreakExample
{
        public static void main(String[] args)
        {
                for(int i=1;i<=10;i++)
               {
                        if(i==5)
                       {
                                //breaking the loop
                                break;
                         }
                        System.out.println(i);
                 }
        }
}
```

Output

```
1
2
3
4
```

2. continue

The continue statement is used in loop control structure when you need to jump to the next iteration of the loop immediately. The Java continue statement is used to continue the loop. It continues the current flow of the program and skips

the remaining code at the specified condition. In case of an inner loop, it continues the inner loop only. Java continue statement can be used in all types of loops such as for loop, while loop and do-while loop.

Example 24: continue statement

```
class ContinueExample
{
        public static void main(String[] args)
        {
                for(int i=1;i<=10;i++)
               {
                        if(i==5)
                        {
                                continue;//it will skip the rest statement
                        }
                        System.out.println(i);
                }
        }
}
```

Output

```
1
2
3
4
6
7
8
9
10
```

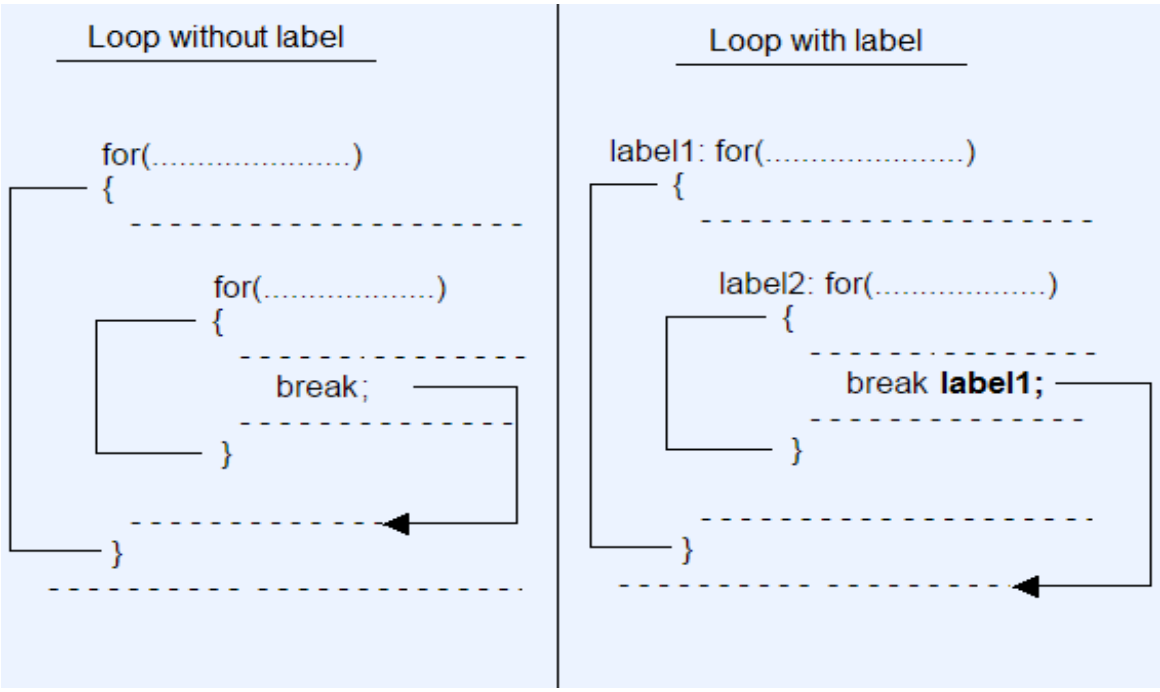
3. return

The return statement is mainly used in methods in order to terminate a method in between and return back to the caller method.

Note: return statement will be explain in second module

Labelled loops

The Java allows you to stick a label to a loop. It is like you name a loop, which is useful when you use multiple nested loops in a program.



Example 25: labelled loop

```
class WithLabelledLoop
{
	public static void main(String args[])
	{
		int i,j;
		loop1:  for(i=1;i<=10;i++)
		{
			System.out.println();
			loop2:  for(j=1;j<=10;j++)
			{
				System.out.print(j + " ");
				if(j==5)
					break loop1;    //Statement 1
			}
		}
	}
}
```

Output

```
 1 2 3 4 5
```