

Board of Technical Education (BTE) Uttar Pradesh

# GRAVITY

  
Prakash

Series

for



# POLYTECHNIC

Written by

**Prabhat Gupta**

**Based on  
Latest Syllabus**

**Object Oriented  
Programming Using Java**

Published by

  
Prakash

**Publishers & Distributors**

Meerut-250 002 (UP)



● *Published by :*

**Prakash Publishers & Distributors**

196, Gandhi Nagar, Garh Road

Meerut-250 002 (UP)

Phones : 09358196756, 57

● © *Publisher*

- The publishers have taken all possible precautions in publishing this book, yet if any mistake has crept in, the publishers shall not be responsible for the same.
  - This book or any part thereof may not be reproduced in any form by Photographic Mechanical or any other method for any use, without written permission of the publishers.
- All dispute shall be subjected to the jurisdiction of court at Meerut.

● *Laser Type & Setting by :*

*Khusbhoo* Computers (Meerut)

● *Printed by :*

Vimal Offset Printers, Meerut

---

## Contents

---

|                                 |         |
|---------------------------------|---------|
| 1. Introduction and Features    | 3-10    |
| 2. Language Constructs          | 11-50   |
| 3. Classes and Objects          | 51-68   |
| 4. Inheritance                  | 70-81   |
| 5. Abstract Class and Interface | 82-88   |
| 6. Polymorphism                 | 89-97   |
| 7. Exception Handling           | 97-105  |
| 8. Multithreading               | 105-111 |



# Introduction and Features

**Q.1. Explain Fundamentals of Object Oriented Programming.**

**Ans.** Object-Oriented Programming is a paradigm that provides many concepts, such as **inheritance, data binding, polymorphism**, etc.

Simula is considered the first object-oriented programming language. The programming paradigm where everything is represented as an object is known as a truly object-oriented programming language.

Smalltalk is considered the first truly object-oriented programming language.

The popular object-oriented languages are Java, C#, PHP, Python, C++, etc.

The main aim of object-oriented programming is to implement real-world entities, for example, object, classes, abstraction, inheritance, polymorphism, etc.

OOPs (Object-Oriented Programming System)

Object means a real-world entity such as a pen, chair, table, computer, watch, etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies software development and maintenance by providing some concepts:

- ☐ Object
- ☐ Inheritance
- ☐ Abstraction
- ☐ Class
- ☐ Polymorphism
- ☐ Encapsulation.

Apart from these concepts, there are some other terms which are used in Object-Oriented design:

- ☐ Coupling
- ☐ Association
- ☐ Composition.
- ☐ Cohesion
- ☐ Aggregation.

## Syllabus

Fundamentals of object oriented programming-procedure oriented programming Vs. object oriented programming (OOP), Object oriented programming concepts - Classes, object, object reference, abstraction, encapsulation, inheritance, polymorphism, Introduction of eclipse (IDE) for developing programs in Java.

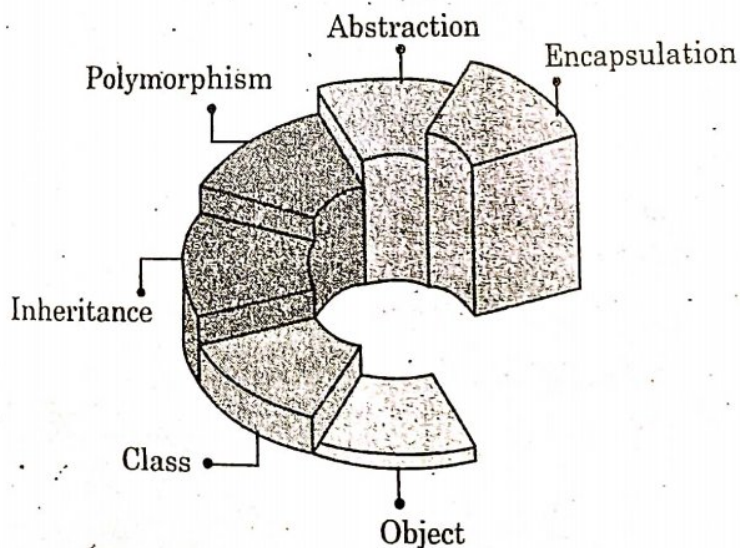


Fig.



**Q.2.** Write the difference between procedure oriented programming and object oriented programming.

**Ans. Procedural-oriented Programming :** Traditional languages like Fortran, Pascal, C, and Cobol, are procedural-oriented languages and they have some drawbacks when it comes to reusing the components they built from.

Procedural programming decomposes a program into various different functional units, each of which can gather and manipulate data as needed.

Procedural-oriented languages built from functions and functions are not as easy to reuse as an object, or a class is. Copying a function from a program and then using it in another is not very easy because that function is more than likely going to reference other functions and global variables. In basic terms, a function is not encapsulated correctly as a unit that is self-contained and reusable.

Also, procedural-oriented programming languages are not suited to solving real-world problems because they don't have a high enough level of abstraction. For example, in C programming we use constructs, like the for loop, if-else statements, methods, arrays, and pointers. These are all low-level and very hard to abstract a real-world problem from, like a computer football game or a CRM (Customer Relationship Management) system. In short, traditional languages, the procedural-oriented languages, separate the variables (the data structures) and the functions (the algorithms).

**Object-Oriented Programming :** Object-oriented programming, on the other hand, decomposes a program into various different data-oriented units or other conceptual units; each unit contains data and various operations that may be performed on that data. Procedural programming forced developers to write highly interdependent code.

With object-oriented programming or OOP as it has become known, are designed to get around these kinds of problems. The basic OOP unit is a class, and a class will encapsulate both the static properties and the dynamic operations inside a container or 'box.' It specifies what the public interface is for making use of these boxes too.

Because the class is well-encapsulated, it is much easier to reuse. In short, OOP combines the algorithms and the data structures that make up a software entity inside one box.

With OOP languages we can use a much higher abstraction level, allowing us to solve realworld problems. While a traditional language, the procedural languages like Pascal and C force you to do your thinking in terms of the computer structure, like bytes, memory, decisions loops, and arrays, instead of in terms of the problem that needs to be solved, the OOP languages, like Java, allow you to do your thinking in the problem space. They make use of software objects that represent the problem space entities, and allow you to abstract them to find a solution to the problem.

Let's say, as an example that you want to write a computer game based on football. Now this is quite a complex type of game to build and using a procedural-oriented language would prove quite difficult. With an OOP language, it is much easier because the game can mode according to real-life things that happen in the game. For example, your classes could be:

- Player: the attribute for a player would be name, location on the playing field, number and etc. while the operations would include running, jumping, kicking the ball and so on
- Field
- Reference



- ❑ Weather
- ❑ Audience Perhaps more importantly, some classes, like Audience, or Ball, could easily be reused in a different application, perhaps a basketball game, without the need for too much, if any, modification.

We can summarize the differences as follows:

#### **Procedural Programming**

- ❑ Top down design
- ❑ Create functions to do small tasks
- ❑ Communicate by parameters and return values.

#### **Object Oriented Programming**

- ❑ Design and represent objects
- ❑ Determine relationships between objects
- ❑ Determine attributes each object has
- ❑ Determine behaviors each object will respond to
- ❑ Create objects and send messages to them to use or manipulate their attributes.

#### **Q.3. Define Object oriented programming.**

**Ans.** All the previous languages are structured or we can say that they were procedural programming means in them processing is to be done in sequence manner and These are also called the Top down or either they were bottom up languages. Most important things those must be in the languages are Reliability, Maintainability and Reusability and user friendly so for achieving these things they developed java.

OOP concentrates more on data rather the procedure to be followed or the structure of the program. It considers data as a crucial element and work towards its security. It ties data and functions together under a single entity called class and prevents any modification to the data by providing access specifies to the class. Some of the features of OOP are explained below.

Java is a purely OOP language that is developed by sun Microsoft in 1991 and this is very popular in 1992 with named called as Oak and in 1992 java was very popular in electrical things at that time java was used for making VCR's and television and handheld devices but in 1993 java was so popular and that time Scientists think that this will be the very popular in internet for transferring files from remote computer to local. Java is called as purely OOP language because every program of java is written into in classes

#### **Q.4. Explain the Features of the Object Oriented Programming.**

**Ans.** A language called as OOP language if it supports all the features of the OOP language for understanding OOPs first you have to understand few concepts as-

**1. Object :** An object is a real word thing which performs a specific task and which has a set of properties and methods. Properties of object are also called as attributes of an object and method is also known as the function of the object like, what an object can do. An object may represent a person, place, thing, or even a bank account. For example a car is an object which has a certain number of properties like color, model no, average etc and a function known as run when a user gives some race then the car will be moved. For example, the objects to the class employee can be created as :

**Employee empl, emp2;**



Where empl, emp2 are the objects of class employee.

**2. Class :** A class is that which contains the set of properties and methods of an object in a single unit like animals is name of class which contains all the properties and methods of an object of another animals so if we want to access any data from the Class then first we have to create the Object of a class then we can use any data and method from class. A class can be defined using the keyword 'class'. For example, a class named employee can be defined as class employee

**3. Class Employee Abstraction :** It is one of the chief features of OOP that hides the implementation details from their specifications. In simple words, it is the blend of encapsulation and information hiding. It encapsulates all the essential features of the objects of a class. The attributes of the objects are called data members and the functions that operate on that data are called member functions or methods. Since Java works with classes, that feature abstraction, so, the classes are also called abstract data types.

**4. Encapsulation :** It is the wrapping up of data and functions under single unit called class. It is one of the most important features of OOP. The data is not accessible to the outside world. It provides a most striking feature called information hiding which means hiding data from direct access by the outside world. The example below shows encapsulation:

```
class Employee {
    public static void main(String args[]) {
        int emp_id, salary;
        char emp_name[10];
        void emp_detail() {
            Code
        }
    }
}
```

**5. Inheritance :** Another feature of OOP is inheritance. Inheritance allows programmers to create new classes from existing ones. The main class is called base class or parent class and the class derived from the base class is called derived class or child class. A child class inherits its properties and attributes from its parents. There are various methods of achieving inheritance such as single inheritance, multilevel inheritance and hybrid inheritance.

**6. Polymorphism :** It provides the ability to take more than one form. Functions as well as operators can take more than one form. When an operator exhibits different forms or behavior, it is called operator overloading. When a function exhibits more than one form or behavior, it is called function overloading. Some complexity is generated through operator overloading, so this feature has been removed in Java programming.

**Q.5. Explain the Other Concepts of the OOPs.**

**Ans.** There are five concepts in OOPs are Following :

**1. Data Abstraction :** Data abstraction is that in which a user can use any of the data and method from the class without knowing about how this is created so in other words we can say that a user use all the functions without knowing about its detail. For example when a user gives race to car the car will be moved but a user doesn't know how its engine will work.



**2. Inheritance :** Inheritance is very popular concept in OOP this provides the capability to a user to use the predefined code or the code that is not created by the user himself but if he may wants to use that code then he can use that code. This is called inheritance but always remember in inheritance a user only using the code but he will not be able to change the code that is previously created he can only use that code.

**3. Data Encapsulation :** Data encapsulation is also known as data hiding as we know with the inheritance concept of opps user can use any code that is previously created but if a user wants to use that code then it is must that previously code must be public as the name suggests public means for other peoples but if a code is private then it will be known as encapsulate and user will not be able to use that code so with the help of OPPS we can alter or change the code means we can make the code as private or public this allows us to make our code either as public or private

**4. Polymorphism :** Poly means many and morphism means many function the concepts introduces in the form of many behaviors of an object like an operator + is used for addition of two numbers and + is also used for joining two names the polymorphism in java introduces in the form of functions overloading and in the form of constructor overloading

**5. Dynamic Binding :** Binding is used when we call the code of the procedure in binding all the code that is linked with the single procedure is called when a call is made to that procedure then the compiler will found the entire code of the single procedure if a compiler will fond all the code of single procedure in compile time then it is called as the early binding because compiler knows about the code at the time of compilation but in the late binding compiler will understand all the code at run time or at the time of the execution.

**6. Message Communication :** Message communication is occurred when an object passes the call to method of class for execution we know for executing any method from the class first we have to create the object of class when an object passes references to function of class then in message communication first of all we have to create the object of the class the we make communication between the object and the methods of the class.

#### **Q.6. Write down the Benefits of OOPS.**

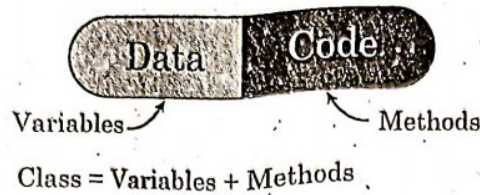
**Ans.** Because procedural-oriented languages concentrate on procedures, using a basic unit of a function, you need to spend time first workings out what all of the functions are going to be and then thinking about how they should represent. With OOP languages, we focus our attention on the components perceived by the user using a basic unit of an object. You work out what all the objects are by combining data with the operations that are used to describe the way a user interacts with them.

Much easier to use in terms of designing software. Rather than thinking in terms of bytes and bits, you can think in terms of the problem space instead. You are using abstraction and higher level concepts and, because the design is more comfortable, your application development is much more productive.

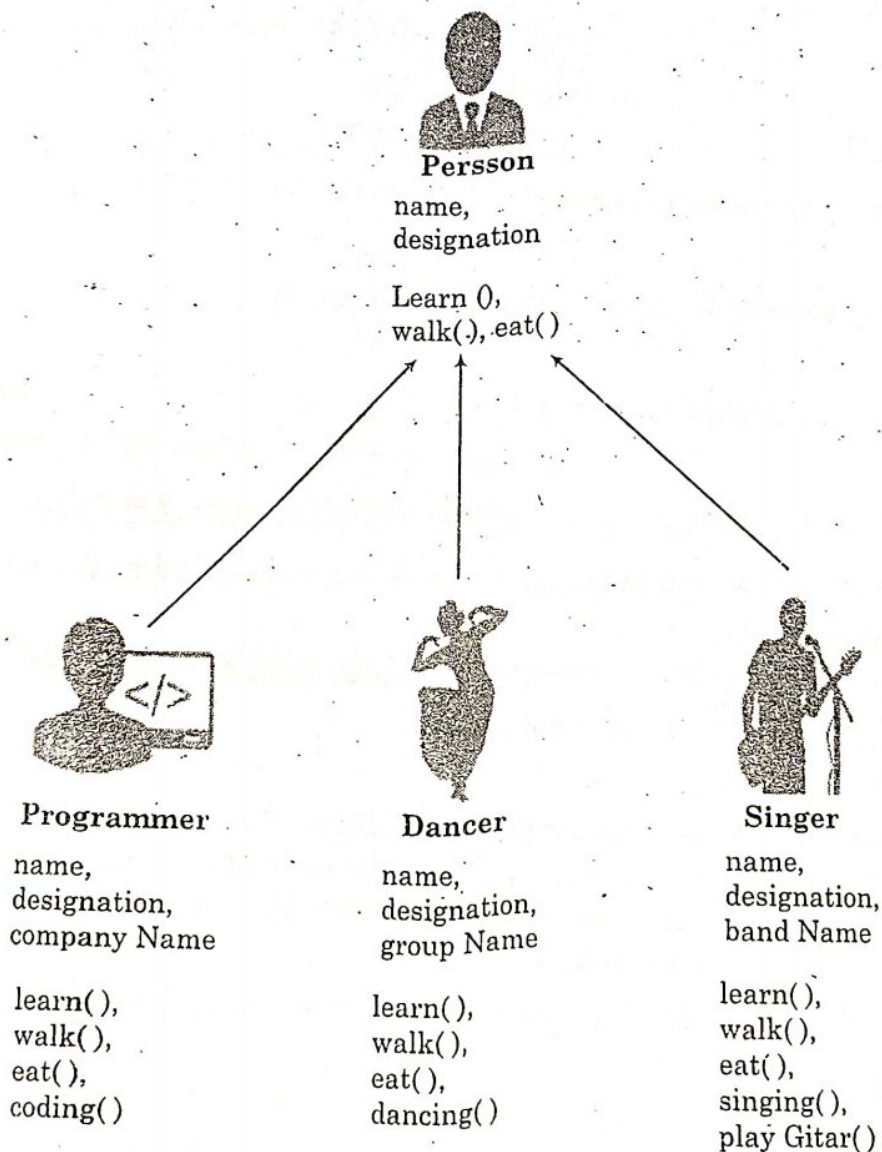
The software is easier to maintain because it is much easier to understand. In turn, that makes it easier to test and to debug.

The software is reusable. There is no longer any need to keep writing the same functions repeatedly for different scenarios. Being able to reuse the same code is fast and safe because you are using code that has been thoroughly tested already and is proven to work.



**Q.7. Define Encapsulation.****Ans.**

Encapsulation is the process of combining data and code into a single unit (object / class). In OOP, every object is associated with its data and code. In programming, data is defined as variables and code is defined as methods. The java programming language uses the class concept to implement encapsulation.

**Q.8. Define Inheritance.****Ans.**

Inheritance is the process of acquiring properties and behaviors from one object to another object or one class to another class. In inheritance, we derive a new class from the existing class. Here, the new class acquires the properties and behaviors from the existing class. In the inheritance concept, the class which provides properties is called as parent class and the class which receives the properties is called as child class. The parent class is also known as base class or super class. The child class is also known as derived class or sub class.

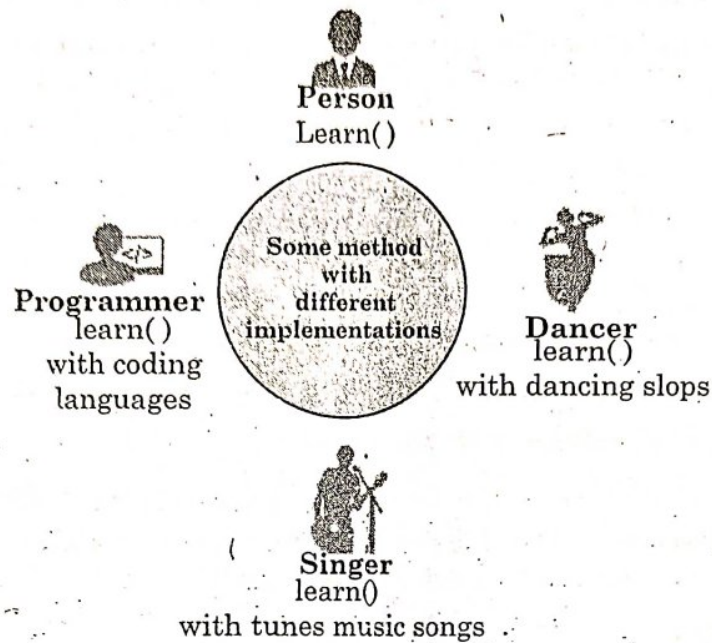


In the inheritance, the properties and behaviors of base class extended to its derived class, but the base class never receive properties or behaviors from its derived class.

In java programming language the keyword extends is used to implement inheritance.

### Q.9. Define Polymorphism.

Ans.

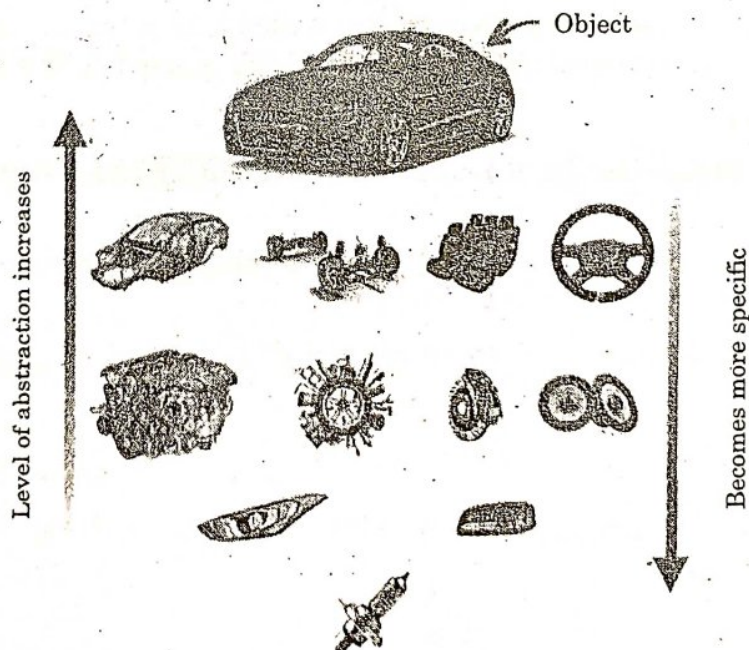


Polymorphism is the process of defining same method with different implementation. That means creating multiple methods with different behaviors.

The java uses method overloading and method overriding to implement polymorphism. Method overloading - multiple methods with same name but different parameters. Method overriding - multiple methods with same name and same parameters.

### Q.10. Define Abstraction.

Ans.



Abstraction is hiding the internal details and showing only essential functionality. In the abstraction concept, we do not show the actual implementation to the end user, instead we



provide only essential things. For example, if we want to drive a car, we do not need to know about the internal functionality like how wheel system works? How brake system works? How music system works? Etc.

**Q.11.** Write about introduction of eclipse (IDE) for developing program in Java.

**Ans.** Eclipse is one of the most popular open source Java IDEs available today.

Most Java developers use Eclipse more than any other Java tool available on the internet. The Eclipse Foundation allows development by providing the infrastructure, and a structured process of development. The Eclipse Foundation has built its open source community and ecosystem of products and services since 2001.

**Eclipse for Java :** Eclipse IDE (integrated development environment) for Java has been the leading development environment with a market share of 65% as of today. It can be extended with an additional software component. Eclipse calls these software components as plug-ins, which can grouped into features. Many companies have extended Eclipse IDE on top of the Eclipse framework. It is also available as IDE for other languages.

**Q.12.** How to install Eclipse IDE for Java?

**Ans.** The Eclipse IDE for Java is designed for standard Java development. It contains typically required packages like support for Maven and Gradle. It also supports Git version controlling system. You can download this IDE from the official Eclipse website.

**Basic concepts :**

- ❑ **Workspace:** This is at a physical location of file path where you store certain meta data of development artefacts. Projects, source files, images, and other artefacts are stored either inside or outside of your workspace.
- ❑ **Views and editors:** Eclipse offers views and editors to navigate and change content. A view is used to work on a set of data, which is in a hierarchical structure. If data is changed via view, the underlying data is directly changed.
- ❑ **Eclipse project:** This is an open source, configuration and binary file that are related to a certain task. Eclipse project groups these files into a buildable and reusable unit. These projects can have natures assigned to it which describes the purpose of this project.

**Steps to install :**

1. You need to install the Java Development Kit (JDK) to setup the environment for Java programming.
2. Download the latest version of Eclipse from the official Eclipse website. You can find the latest version under 'Get Eclipse IDE 2018-12'.
3. To install the Eclipse, simply unzip the downloaded file and execute the installable package.





**Q.1. Define Variables, with variable types and type declarations.**

**Ans.** A variable is a named memory location used to store a data value. A variable can be defined as a container that holds a data value.

In java, we use the following syntax to create variables.

**Syntax :**

data\_type variable\_name;

(or)

data\_type variable\_name\_1, variable\_name\_2,...;

(or)

data\_type variable\_name = value;

(or)

data\_type variable\_name\_1 = value, variable\_name\_2 = value,...

In java programming language variables are classified as follows-

- ☐ Local variables
- ☐ Instance variables or Member variables or Global variables
- ☐ Static variables or Class variables
- ☐ Final variables.

**1. Local variables :** The variables declared inside a method or blocks are known as local variables. A local variable is visible within the method in which it is declared. The local variable is created when execution control enters into the method or block and destroyed after the method or block execution completed.

Let's look at the following example java program to illustrate local variable in java.

**Example :**

```
public class LocalVariables {
    public void show() {
        int a = 10;
        //static int x = 100;
        System.out.println("Inside show method, a = " + a);
    }
    public void display() {
        int b = 20;
```

### Syllabus

Variables, types and type declarations, data types : Integer, floating point type, character, boolean, all Operators, iteration and jump statement, if then else clause; conditional expressions, Input using scanner class and output statement, loops, switch case, arrays, methods.



```

        System.out.println("Inside display method, b = " + b);
        // trying to access variable 'a' - generates an ERROR
        System.out.println("Inside display method, a = " + a);
    }
    public static void main(String args[]) {
        LocalVariables obj = new LocalVariables();
        obj.show();
        obj.display();
    }
}

```

**2. Instance variables or member variables or global variables :** The variables declared inside a class and outside any method, constructor or blocks are known as instance variables or member variables. These variables are visible to all the methods of the class. The changes made to these variables by method affects all the methods in the class. These variables are created separate copy for every object of that class.

Let's look at the following example java program to illustrate instance variable in java.

**Example :**

```

public class ClassVariables {
    int x = 100;
    public void show() {
        System.out.println("Inside show method, x = " + x);
        x = x + 100;
    }
    public void display() {
        System.out.println("Inside display method, x = " + x);
    }
    public static void main(String[] args) {
        ClassVariables obj = new ClassVariables();
        obj.show();
        obj.display();
    }
}

```

**3. Static variables or Class variables :** A static variable is a variable that declared using static keyword. The instance variables can be static variables but local variables cannot. Static variables are initialized only once, at the start of the program execution. The static variable only has one copy per class irrespective of how many objects we create.

The static variable is access by using class name.

Let's look at the following example java program to illustrate static variable in java.



**Example :**

```
public class StaticVariablesExample {
    int x, y; // Instance variables
    static int z; // Static variable
    StaticVariablesExample(int x, int y){
        this.x = x;
        this.y = y;
    }
    public void show() {
        int a; // Local variables
        System.out.println("Inside show method,");
        System.out.println("x = " + x + ", y = " + y + ", z = " + z);
    }
    public static void main(String[] args) {
        StaticVariablesExample obj_1 = new StaticVariablesExample(10, 20);
        StaticVariablesExample obj_2 = new StaticVariablesExample(100, 200);
        obj_1.show();
        StaticVariablesExample.z = 1000;
        obj_2.show();
    }
}
```

**4. Final Variables :** A final variable is a variable that declared using final keyword. The final variable is initialized only once, and does not allow any method to change its value again. The variable created using final keyword acts as constant. All variables like local, instance, and static variables can be final variables.

Let's look at the following example java program to illustrate final variable in java.

**Example :**

```
public class FinalVariableExample {
    final int a = 10;
    void show() {
        System.out.println("a = " + a);
        a = 20; //Error due to final variable can't be modified
    }
    public static void main(String[] args) {
        FinalVariableExample obj = new FinalVariableExample();
        obj.show();
    }
}
```

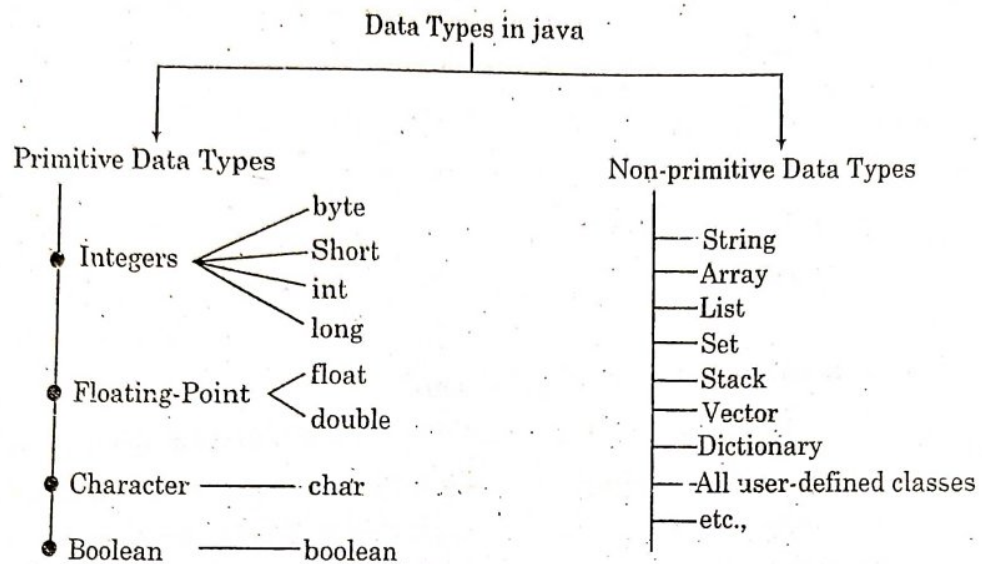


**Q.2. Explain Data types (Integer, floating point type, Character, Boolean or more) in brief.**

**Ans.** Java programming language has a rich set of data types. The data type is a category of data stored in variables.

In java, data types are classified into two types and they are as follows-

- Primitive Data Types
- Non-primitive Data Types.



**1. Primitive Data Types :** The primitive data types are built-in data types and they specify the type of value stored in a variable and the memory size. The primitive data types do not have any additional methods.

In java, primitive data types includes byte, short, int, long, float, double, char, and Boolean.

The following table provides more description of each primitive data type.

| Data Type | Meaning            | Memory size | Range  | Default Value |
|-----------|--------------------|-------------|--|---------------|
| byte      | Whole numbers      | 1 byte      | -128 to +127   | 0             |
| short     | Whole numbers      | 2 bytes     | -32768 to +32767   | 0             |
| int       | Whole numbers      | 4 bytes     | -2,147,483,648 to +2,147,483,647                         | 0             |
| long      | Whole numbers      | 8 bytes     | -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807 | 0 L           |
| float     | Fractional numbers | 4 bytes     | —  | 0.0f          |
| double    | Fractional numbers | 8 bytes     | —  | 0.0d          |
| char      | Single character   | 2 bytes     | 0 to 65535   | \u0000        |
| Boolean   | unsigned char      | 1 bit       | 0 or 1   | 0 (false)     |

Let's look at the following example java program to illustrate primitive data types in java and their default values.



**Example :**

```

public class PrimitiveDataTypes {
    byte i;
    short j;
    int k;
    long l;
    float m;
    double n;
    char ch;
    boolean p;
    public static void main(String[] args) {
        PrimitiveDataTypes obj = new PrimitiveDataTypes();
        System.out.println("i = " + obj.i + ", j = " + obj.j + ", k = " + obj.k + ", l = " + obj.l);
        System.out.println("m = " + obj.m + ", n = " + obj.n);
        System.out.println("ch = " + obj.ch);
        System.out.println("p = " + obj.p);
    }
}

```

**a. Integer :** This group includes byte, short, int, long

**Byte :** It is 1 byte (8-bits) integer data type. Value range from -128 to 127. Default value zero. Example: **byte b=10;**

**Short :** It is 2 bytes (16-bits) integer data type. Value range from -32768 to 32767. Default value zero. Example: **short s=11;**

**Int :** It is 4 bytes (32-bits) integer data type. Value range from -2147483648 to 2147483647. Default value zero. Example: **int i = 10;**

**Long :** It is 8 bytes (64-bits) integer data type. Value range from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. Default value zero. Example: **long l=100012;**

**Example :** Let's create an example in which we work with **integer type** data and we can get idea how to use datatype in the java program.

```

package corejava;
public class Demo{
    public static void main(String[] args) {
        // byte type
        byte b = 20;
        System.out.println("b= "+b);
        // short type
        short s = 20;
        System.out.println("s= "+s);
        // int type
    }
}

```



```

    int i = 20;
    System.out.println("i= "+i);
    // long type
    long l = 20;
    System.out.println("l= "+l);
}
}

```

Output:-

b= 20

s= 20

i= 20

l= 20

**b. Floating-Point Number :** This group includes **float**, **double**

**Float :** It is 4 bytes (32-bits) float data type. Default value 0.0f. Example: float ff=10.3f;

**Double :** It is 8 bytes (64-bits) float data type. Default value 0.0d. Example: double db=11.12d;

**Example :** In this example, we used floating point type and declared variables to hold floating values. Floating type is useful to store decimal point values.

```

public class Demo{
    public static void main(String[] args) {
        // float type
        float f = 20.25f;
        System.out.println("f= "+f);
        // double type
        double d = 20.25;
        System.out.println("d= "+d);
    }
}

```

Output:-

f= 20.25

d= 20.25

**c. Characters :** This group represents char, which represent symbols in a character set, like letters and numbers.

**Char :** It is 2 bytes (16-bits) unsigned unicode character. Range 0 to 65,535. Example: char c='a';

**Char Type Example :** Char type in Java uses 2 bytes to unicode characters. Since it works with unicode then we can store alphabet character, currency character or other characters that are comes under the unicode set.



```
public class Demo {  
    public static void main(String[] args) {  
        char ch = 'S';  
        System.out.println(ch);  
        char ch2 = '&';  
        System.out.println(ch2);  
        char ch3 = '$';  
        System.out.println(ch3);  
    }  
}
```

Output:-

S  
&  
\$

d. **Boolean** : This group represent Boolean, which is a special type for representing true/false values. They are defined constant of the language. Example: **Boolean b=true;**

**Boolean Type Example** : Boolean type in Java works with two values only either true or false. It is mostly use in conditional expressions to perform conditional based programming.

```
public class Demo {  
    public static void main(String[] args) {  
        boolean t = true;  
        System.out.println(t);  
        boolean f = false;  
        System.out.println(f);  
    }  
}
```

Output:-

true  
false

**3. Non-primitive Data Types** : In java, non-primitive data types are the reference data types or user-created data types. All nonprimitive data types are implemented using object concepts. Every variable of the non-primitive data type is an object. The non-primitive data types may use additional methods to perform certain operations. The default value of non-primitive data type variable is null.

In java, examples of non-primitive data types are String, Array, List, Queue, Stack, Class, Interface, etc.



```

public class NonPrimitiveDataTypes {
    String str;
    public static void main(String[] args) {
        String name = "BTech Smart Class!";
        String wish = "Hello, ";
        NonPrimitiveDataTypes obj = new NonPrimitiveDataTypes();
        System.out.println("str = " + obj.str);
        //using addition method
        System.out.println(wish.concat(name));
    }
}

```

### Q.3. Difference between Primitive Vs Non-primitive Data Types.

Ans.

| Primitive Data Type                 | Non-primitive Data Type        |
|-------------------------------------|--------------------------------|
| These are built-in data types       | These are created by the users |
| Does not support additional methods | Support additional methods     |
| Always has a value                  | It can be null                 |
| Starts with lower-case letter       | Starts with upper-case letter  |
| Size depends on the data type       | Same size for all              |

### Q.4. Describe all java Operators in brief.

Ans. Operator is a symbol which tells to the compiler to perform some operation. Java provides a rich set of operators to deal with various types of operations. Sometimes we need to perform arithmetic operations then we use plus (+) operator for addition, multiply (\*) for multiplication etc.

Operators are always essential part of any programming language.

Java operators can be divided into following categories:

- ☐ Arithmetic operators
- ☐ Relation operators
- ☐ Logical operators
- ☐ Bitwise operators
- ☐ Assignment operators
- ☐ Conditional operators
- ☐ Misc operators.

### Q.4. Define Arithmetic Operators with example.

Ans. Arithmetic operators are used to perform arithmetic operations like: addition, subtraction etc and helpful to solve mathematical expressions. The below table contains Arithmetic operators.

| Operator | Description                         |
|----------|-------------------------------------|
| +        | adds two operands                   |
| -        | subtract second operands from first |



|    |   |
|----|---|
| *  | multiply two operand                              |
| /  | divide numerator by denominator                   |
| %  | remainder of division                             |
| ++ | Increment operator increases integer value by one |
| -- | Decrement operator decreases integer value by one |

**Example :** Let's create an example to understand arithmetic operators and their operations.

```
class Arithmetic_operators1{
    public static void main(String as[])
    {
        int a, b, c;
        a=10;
        b=2;
        c=a+b;
        System.out.println("Addition: "+c);
        c=a-b;
        System.out.println("Substraction: "+c);
        c=a*b;
        System.out.println("Multiplication: "+c);
        c=a/b;
        System.out.println("Division: "+c);
        b=3;
        c=a%b;
        System.out.println("Remainder: "+c);
        a=++a;
        System.out.println("Increment Operator: "+a);
        a=--a;
        System.out.println("decrement Operator: "+a);
    }
}
```

#### Command Prompt

```
0: \stud_tonight> javac Arithmetic_operators1.java
0: \stud_tonight> java Arithmetic_operators1
Addition: 12
Substraction: 8
Multiplication: 20
Division: 5
Remainder: 1
Increment Operator: 11
decrement Operator: 10
```



**Q.5. Explain Relation Operators with example.**

**Ans.** Relational operators are used to test comparison between operands or values. It can be used to test whether two values are equal or not equal or less than or greater than etc.

The following table shows all relation operators supported by Java.

| Operator | Description  |
|----------|--|
| ==       | Check if two operand are equal                                     |
| !=       | Check if two operand are not equal.                                |
| >        | Check if operand on the left is greater than operand on the right  |
| <        | Check operand on the left is smaller than right operand            |
| >=       | check left operand is greater than or equal to right operand       |
| <=       | Check if operand on left is smaller than or equal to right operand |

**Example :** In this example, we are using relational operators to test comparison like less than, greater than etc.

```
class Relational_operators1{
    public static void main(String as[])
    {
        int a, b;
        a=40;
        b=30;
        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) );
    }
}
```

```

D:\Studytonight> javac Relational_operators1.java
D:\Studytonight> java Relational_operators1
a == b = false
a != b = true
a > b = true
a < b = false
b >= a = false
b <= a = true
D:\Studytonight>

```



**Q.6. Define Logical Operators with example.**

**Ans.** Logical Operators are used to check conditional expression. For example, we can use logical operators in if statement to evaluate conditional based expression. We can use them into loop as well to evaluate a condition.

Java supports following 3 logical operators. Suppose we have two variables whose values are: a=true and b=false.

| Operator | Description | Example           |
|----------|-------------|-------------------|
| &&       | Logical AND | (a && b) is false |
|          | Logical OR  | (a    b) is true  |
| !        | Logical NOT | (!a) is false     |

```

class Logical_operators1{
    public static void main(String as[])
    {
        boolean a = true;
        boolean b = false;
        System.out.println("a && b = " + (a&&b));
        System.out.println("a || b = " + (a || b));
        System.out.println("!(a && b) = " + !(a && b));
    }
}

```

```

D:\Studytonight>java Logical_operators1.java
a && b = false
a || b = true
!(a && b) = true
D:\Studytonight>

```

**Q.7. Define Bitwise Operators with example.**

**Ans.** Bitwise operators are used to perform operations bit by bit.

Java defines several bitwise operators that can be applied to the integer types long, int, short, char and byte.

The following table shows all bitwise operators supported by Java :

| Operator | Description |
|----------|-------------|
| &        | Bitwise AND |



|    |                      |
|----|----------------------|
|    | Bitwise OR           |
| ^  | Bitwise exclusive OR |
| << | left shift           |
| >> | right shift          |

Now let's see truth table for bitwise &, | and ^

| A | a | a & b | a   b | a ^ b |
|---|---|-------|-------|-------|
| 0 | 0 | 0     | 0     | 0     |
| 0 | 1 | 0     | 1     | 1     |
| 1 | 0 | 0     | 1     | 1     |
| 1 | 1 | 1     | 1     | 0     |

The bitwise shift operators shift the bit value. The left operand specifies the value to be shifted and the right operand specifies the number of positions that the bits in the value are to be shifted. Both operands have the same precedence.

**Example :** Let's create an example that shows working of bitwise operators.

```

a = 0001000
b = 2
a << b = 0100000
a >> b = 0000010
class Bitwise_operators1{
public static void main(String as[])
{
    int a = 50;
    int b = 25;
    int c = 0;
    c = a & b;
    System.out.println("a & b = " + c);
    c = a | b;
    System.out.println("a | b = " + c);
    c = a ^ b;
    System.out.println("a ^ b = " + c);
    c = ~a;
    System.out.println("~a = " + c);
    c = a << 2;
    System.out.println("a << 2 = " + c);
    c = a >> 2;
    System.out.println("a >> 2 = " + c);
    c = a >>> 2;
    System.out.println("a >>> 2 = " + c);
}
}

```



## Command Prompt

```
D:\Studytonight> javac Bitwise_operators1.java
```

```
D:\Studytonight> java Bitwise_operators1
```

```
a & b = 16
```

```
a | b = 59
```

```
a ^ b = 43
```

```
~a = -51
```

```
a << 2 = 200
```

```
a >> 2 = 12
```

```
a >>> 2 = 12
```

```
a >>> 2 = 12
```

```
D:\Studytonight>
```

## Q.8. Explain Assignment Operators with example.

Ans. Assignment operators are used to assign a value to a variable. It can also be used combine with arithmetic operators to perform arithmetic operations and then assign the result to the variable. Assignment operators supported by Java are as follows:

| Operator | Description   | Example               |
|----------|---|-----------------------|
| =        | assigns values from right side operands to left side operand                        | a = b                 |
| +=       | adds right operand to the left operand and assign the result to left                | a+=b is same as a=a+b |
| -=       | subtracts right operand from the left operand and assign the result to left operand | a-=b is same as a=a-b |
| *=       | multiply left operand with the right operand and assign the result to left operand  | a*=b is same as a=a*b |
| /=       | divides left operand with the right operand and assign the result to left operand   | a/=b is same as a=a/b |
| %=       | calculate modulus using two operands and assign the result to left operand          | a%=b is same as a=a%b |

**Example :** Let's create an example to understand use of assignment operators. All assignment operators have right to left associativity.

```
class Assignment_operators1{
    public static void main(String as[])
    {
        int a = 30;
        int b = 10;
        int c = 0;
        c = a + b;
```



```

System.out.println("c = a + b = " + c);
    c += a;
System.out.println("c += a = " + c);
    c -= a;
System.out.println("c -= a = " + c);
    c *= a;
System.out.println("c *= a = " + c);
    a = 20;
    c = 25;
    c /= a;
System.out.println("c /= a = " + c);
    a = 20;
    c = 25;
    c %= a;
System.out.println("c %= a = " + c);
    c <<= 2;
System.out.println("c <<= 2 = " + c);
    c >>= 2;
System.out.println("c >>= 2 = " + c);
    c >>= 2;
System.out.println("c >>= 2 = " + c);
    c &= a;
System.out.println("c &= a = " + c);
    c ^= a;
System.out.println("c ^= a = " + c);
    c |= a;
System.out.println("c |= a = " + c);
}
}

```

```

D:\Studytonight> javac Assignment_operators1.java
D:\Studytonight> java Assignment_operators1
c = a + b = 40
c += a = 70
c -= a = 40
c *= a = 1200
c /= a = 1
c %= a = 5
c <<= 2 = 20
c >>= 2 = 5
c >>= 2 = 1
c &= a = 0
c ^= a = 20
c |= a = 20
D:\Studytonight>

```



**Q.9. Explain Misc. Operator with example.**

**Ans.** There are few other operators supported by java language.

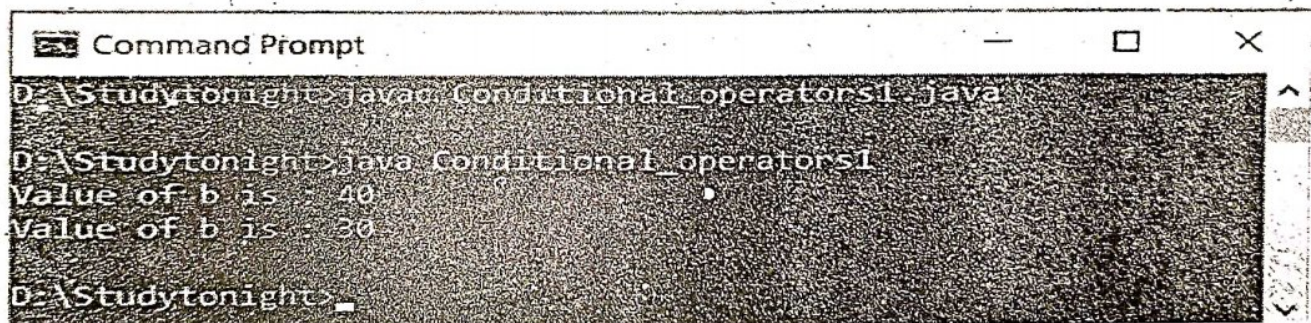
**Conditional operator**

It is also known as ternary operator because it works with three operands. It is short alternate of **if-else statement**. It can be used to evaluate Boolean expression and return either true or false value

`expr1 ? expr2 : expr3`

**Example :** In ternary operator, if `expr1` is true then expression evaluates after question mark (?) else evaluates after colon (:). See the below example.

```
class Conditional_operators1{
public static void main(String as[])
{
int a, b;
a = 20;
b = (a == 1) ? 30: 40;
System.out.println("Value of b is : " + b);
b = (a == 20) ? 30: 40;
System.out.println("Value of b is : " + b);
}
}
```



```
Command Prompt
D:\Studytonight> java Conditional_operators1.java
D:\Studytonight> java Conditional_operators1
Value of b is : 40
Value of b is : 30
D:\Studytonight>
```

**Q.10. Define Instanceof Operator with example.**

**Ans.** It is a java keyword and used to test whether the given reference belongs to provided type or not. Type can be a class or interface. It returns either true or false.

**Example :** Here, we created a string reference variable that stores "studytonight". Since it stores string value so we test it using is instance operator to check whether it belongs to string class or not. See the below example.

```
class instanceof_operators1{
public static void main(String as[])
{
String a = "Studytonight";
boolean b = a instanceof String;
System.out.println( b );
}
}
```



```

D:\Studytonight>javac instanceof_operator$1.java
D:\Studytonight>java instanceof_operator$1
true
D:\Studytonight>
  
```

### Q.11. Explain iteration statement in java.

**Ans.** The java programming language provides a set of iterative statements that are used to execute a statement or a block of statements repeatedly as long as the given condition is true. The iterative statements are also known as looping statements or repetitive statements.

Java provides the following iterative statements :

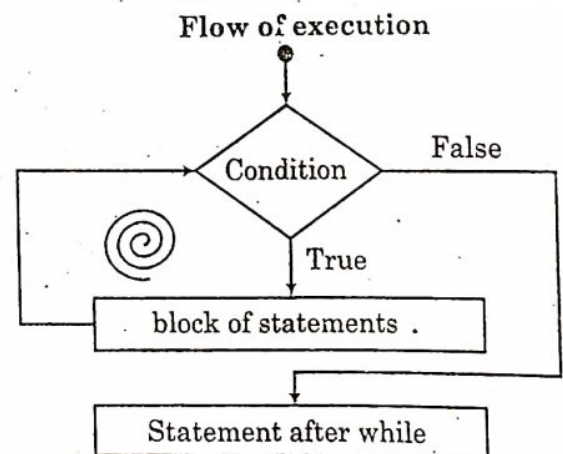
- ☐ while statement
- ☐ do-while statement
- ☐ for statement.

**1. While statement in java :** The while statement is used to execute a single statement or block of statements repeatedly as long as the given condition is TRUE. The while statement is also known as Entry control looping statement. The syntax and execution flow of while statement is as follows.

#### Syntax

```

while (boolean-expression){
    block of statements;
    1.
    staement after while;
    ""
  
```



Let's look at the following example java code.

#### Java Program

```

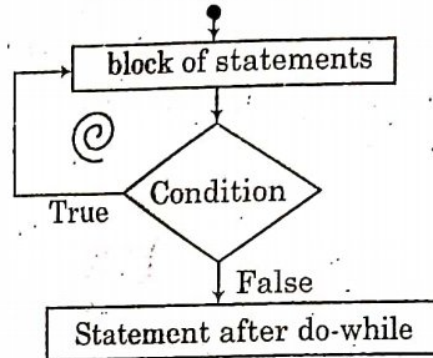
ipublic class WhileTest {
    public static void main(String[] args) {
        int num = 1;
        while(num <= 10) {
            System.out.println(num);
            num++;
        }
        System.out.println("Statement after while!");
    }
}
  
```



**2. Do-while Statement in Java :** The do-while statement is used to execute a single statement or block of statements repeatedly as long as given the condition is TRUE. The do-while statement is also known as the Exit control looping statement. The do-while statement has the following syntax.

**Syntax**

```
do[
    block of statements;
]while(boolean-expression);
statement after do-while
...
```

**Flow of execution**

Let's look at the following example java code.

**Java Program**

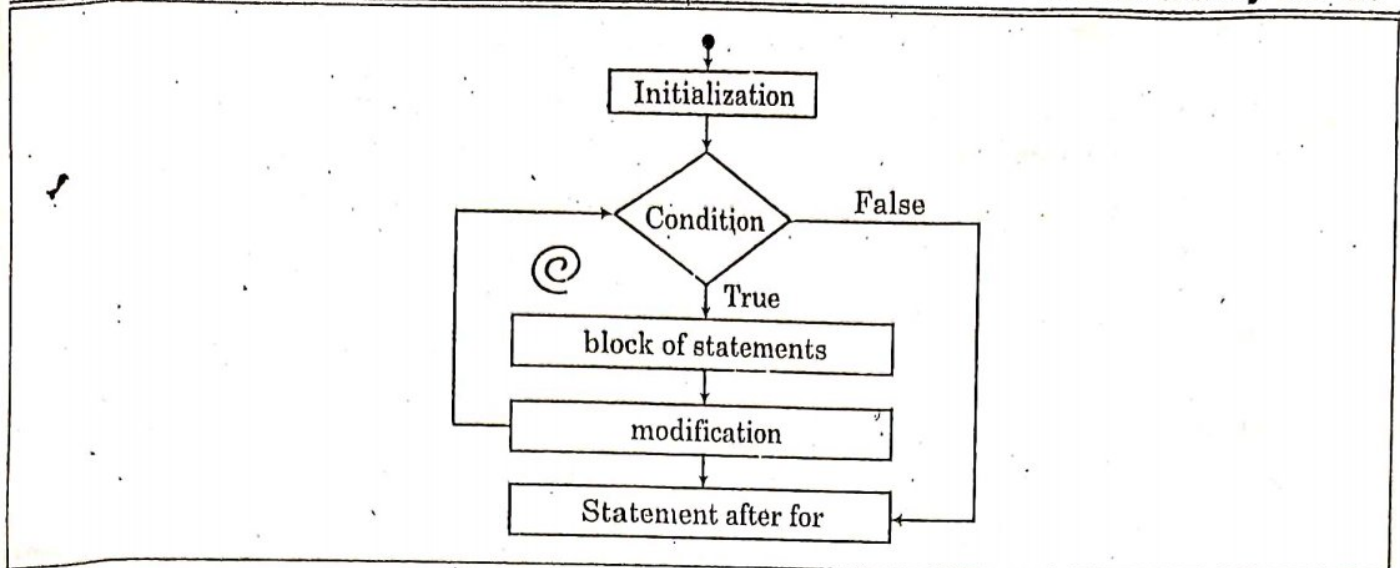
```
public class DoWhileTest {
    public static void main(String[] args) {
        int num = 1;
        do {
            System.out.println(num);
            num++;
        }while(num <= 10);
        System.out.println("Statement after do-while!");
    }
}
```

**3. For Statement in Java :** The for statement is used to execute a single statement or a block of statements repeatedly as long as the given condition is TRUE. The for statement has the following syntax and execution flow diagram.

**Syntax**

```
for(initialization; boolean-expression; modification)[
    block statements;
    ...
    :
statement after for;
...
]
```





In for-statement, the execution begins with the initialization statement. After the initialization statement, it executes Condition. If the condition is evaluated to true, then the block of statements executed otherwise it terminates the for-statement. After the block of statements execution, the modification statement gets executed, followed by condition again.

Let's look at the following example java code.

#### Java Program :

```

public class ForTest {
    public static void main(String[] args) {
        for(int i = 0; i < 10; i++) {
            System.out.println("i = " + i);
        }
        System.out.println("Statement after for!");
    }
}

```

#### Q.12. Explain jump statement with example.

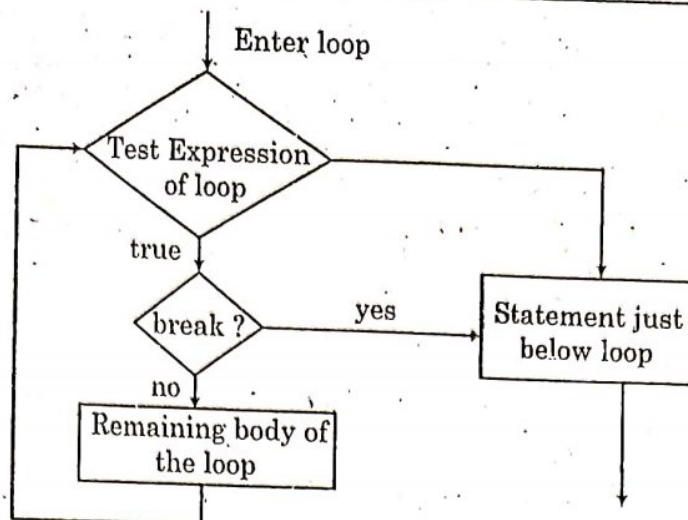
**Ans.** Java supports three jump statements: break, continue and return. These three statements transfer control to other part of the program.

**Break:** In Java, break is majorly used for:

- ☐ Terminate a sequence in a switch statement (discussed above).
- ☐ To exit a loop.
- ☐ Used as a "civilized" form of goto.

**a. Using break to exit a Loop :** Using break, we can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop. Note: Break, when used inside a set of nested loops, will only break out of the innermost loop.



**Example:**

// Java program to illustrate using

// break to exit a loop

class BreakLoopDemo

{

public static void main(String args[])

{

// Initially loop is set to run from 0-9

for (int i = 0; i < 10; i++)

{

// terminate loop when i is 5.

if (i == 5)

break;

System.out.println("i: " + i);

}

System.out.println("Loop complete.");

}

}

**Output:**

i: 0

i: 1

i: 2

i: 3

i: 4

Loop complete.



**b. Using break as a Form of goto :** Java does not have a goto statement because it provides a way to branch in an arbitrary and unstructured manner. Java uses label. A Label is use to identifies a block of code.

**Syntax :**

```
label:
{
    statement1;
    statement2;
    statement3;
```

Now, break statement can be use to jump out of target block.

Note: You cannot break to any label which is not defined for an enclosing block.

**Syntax:**

```
break label;
```

**Example :**

// Java program to illustrate using break with goto

```
class BreakLabelDemo
```

```
{
    public static void main(String args[])
    {
        boolean t = true;
        // label first
        first:
        {
            // Illegal statement here as label second is not
            // introduced yet break second;
            second:
            {
                third:
                {
                    // Before break
                    System.out.println("Before the break statement");
                    // break will take the control out of
                    // second label
                    if (t)
                        break second;
                    System.out.println("This won't execute.");
```



```

    }
    System.out.println("This won't execute.");
}
// First block
System.out.println("This is after second block.");
}
}

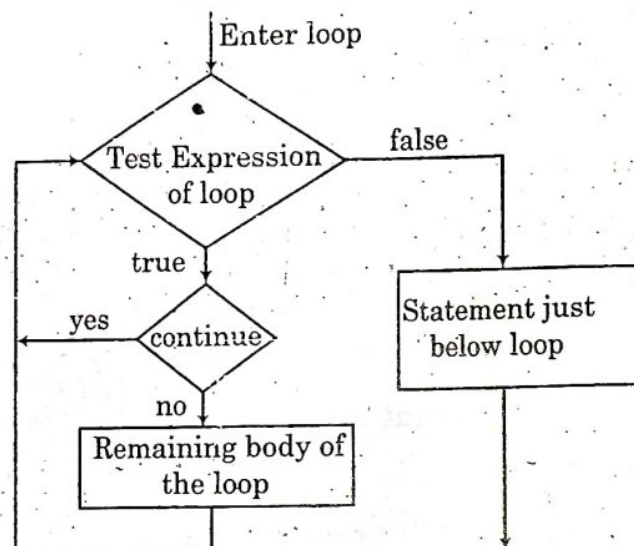
```

**Output :**

Before the break.

This is after second block.

**Continue :** Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration. This is, in effect, a goto just past the body of the loop, to the loop's end. The continue statement performs such an action.

**Example:**

// Java program to illustrate using

// continue in an if statement

```
class ContinueDemo
```

```
{
```

```
    public static void main(String args[])
```

```
    {
```

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

```
        {
```

```
            // If the number is even
```

```
            // skip and continue
```

```
            if (i%2 == 0)
```

```
                continue;
```



```
// If number is odd, print it
System.out.print(i + " ");
```

```
}
```

**Output:**

1 3 5 7 9

**Return:** The return statement is used to explicitly return from a method. That is, it causes a program control to transfer back to the caller of the method.

**Example:**

// Java program to illustrate using return

class Return

```
{
```

```
    public static void main(String args[])
```

```
{
```

```
    boolean t = true;
```

```
    System.out.println("Before the return.");
```

```
    if (t)
```

```
        return;
```

```
    // Compiler will bypass every statement
```

```
    // after return
```

```
    System.out.println("This won't execute.");
```

```
}
```

```
}
```

**Output:**

Before the return.

**Q.13. Explain if then else statement in java.**

**Ans.** In Java, if statement is used for testing the conditions. The condition matches the statement it returns true else it returns false.

There are four types of If statement they are:

For example, if we want to create a program to test positive integers then we have to test the

integer whether it is greater than zero or not.

In this scenario, if statement is helpful.

There are four types of if statement in Java:

(i) if statement

(ii) if-else statement

(iii) if-else-if ladder

(iv) nested if statement



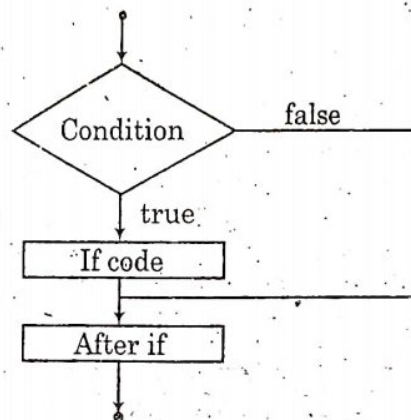
**Q.14. Define if Statement with example.**

Ans. The if statement is a single conditional based statement that executes only if the provided condition is true.

**If Statement Syntax :**

```
if(condition)
{
    //code
}
```

We can understand flow of if statement by using the below diagram. It shows that code written inside the if will execute only if the condition is true.

**Data-flow-diagram of If Block**

**Example:** In this example, we are testing student's marks. If the marks are greater than 65 then student will get first division.

```
public class IfDemo1 {
    public static void main(String[] args)
    {
        int marks=70;
        if(marks > 65)
        {
            System.out.print("First division");
        }
    }
}
```

C:\Windows\System32\cmd.exe

```
D:\Studytonight> javac IfDemo1.java
D:\Studytonight> java IfDemo1
First division
D:\Studytonight>
```



**Q.15. Define if-else Statement with xample.**

**Ans.** The if-else statement is **used** for testing condition. If the condition is true, if block executes otherwise else block executes.

It is useful in the scenario when we want to perform some operation based on the false result.

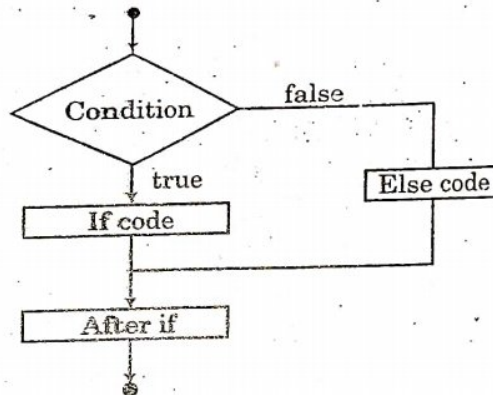
The else block execute only when condition is false.

**Syntax:**

```
if(condition)
{
    //code for true
}
else
{
    //code for false
}
```

In this block diagram, we can see that when condition is true, if block executes otherwise else block executes.

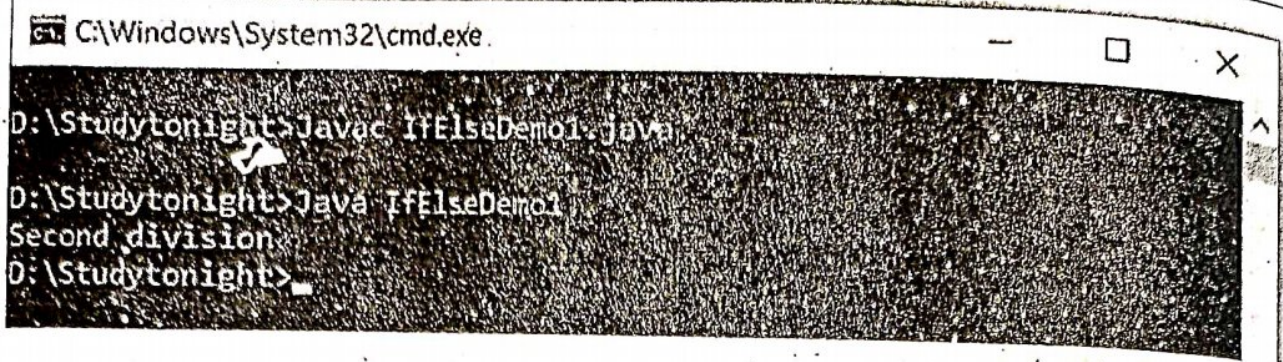
**Data-flow-diagram of If Else Block**



**if else Example:** In this example, we are testing student marks, if marks is greater than 65 then if block executes otherwise else block executes.

```
public class IfElseDemo1 {
    public static void main(String[] args)
    {
        int marks=50;
        if (marks > 65)
        {
            System.out.print("First division");
        }
        else
        {
            System.out.print("Second division");
        }
    }
}
```





```
C:\Windows\System32\cmd.exe

D:\Studytonight>Javac IfElseDemo1.java

D:\Studytonight>Java IfElseDemo1
Second division
D:\Studytonight>
```

**Q.16. Define if-else-if ladder Statement with example.**

**Ans.** In Java, the if-else-if ladder statement is used for testing conditions. It is used for testing one condition from multiple statements.

When we have multiple conditions to execute then it is recommend to use if -else-if ladder.

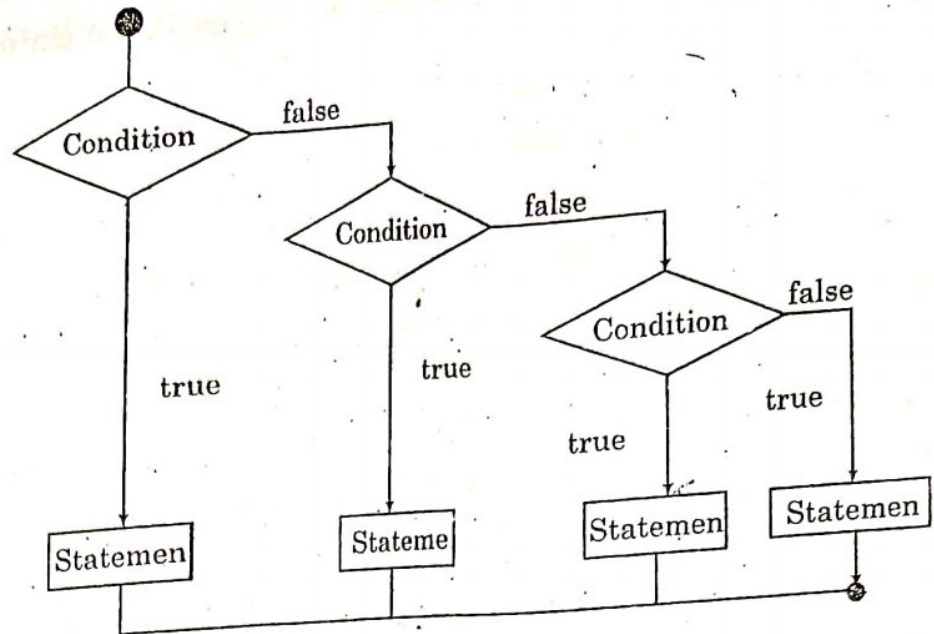
**Syntax:**

```
if(condition1)
{
    //code for if condition1 is true
}
else if(condition2)
{
    //code for if condition2 is true
}
else if(condition3)
{
    //code for if condition3 is true
}
...
else
{
    //code for all the false conditions
}
```

It contains multiple conditions and execute if any condition is true otherwise executes else block.

**Data-flow-diagram of If Else If Block**





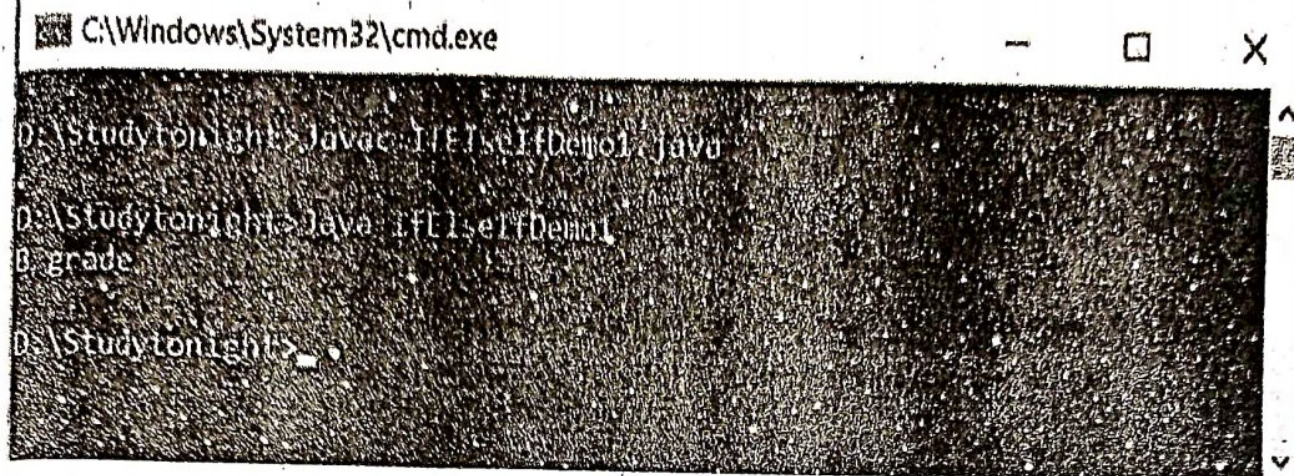
**Example:** Here, we are testing student marks and displaying result based on the obtained marks. If marks are greater than 50 students gets his grades.

```

public class IfElseIfDemo1 {
    public static void main(String[] args) {
        int marks=75;
        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!");
        }
    }
}
  
```





```
C:\Windows\System32\cmd.exe

D:\Studytonight> javac IfElseIfDemo1.java
D:\Studytonight> java IfElseIfDemo1
B grade
D:\Studytonight>
```

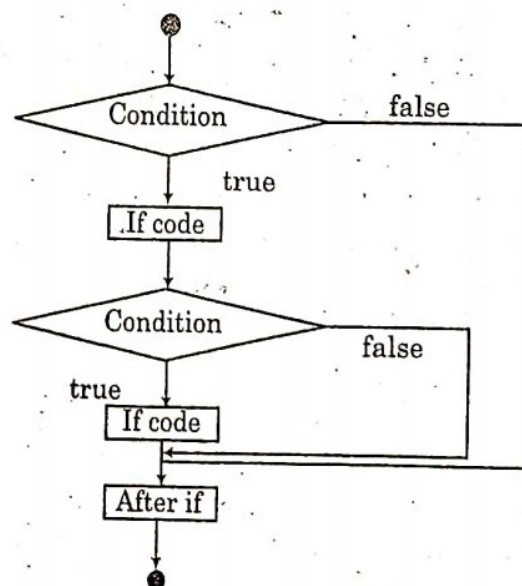
**Q. 17. Define Nested if statement with example.**

**Ans.** In Java, the Nested if statement is a if inside another if. In this, one if block is created inside another if blocks when the outer block is true then only the inner block is executed.

**Syntax:**

```
if(condition)
{
    //statement
    if(condition)
    {
        //statement
    }
}
```

**Data-flow-diagram of Nested If Block :**





**Example:**

```

public class NestedIfDemo1 {
    public static void main(String[] args)
    {
        int age=25;
        int weight=70;
        if(age>=18)
        {
            if(weight>50)
            {
                System.out.println("You are eligible");
            }
        }
    }
}

```

The screenshot shows a Windows command prompt window titled "C:\Windows\System32\cmd.exe". The user has entered the following commands and received the output:

```

D:\Studytonight> javac NestedIfDemo1.java
D:\Studytonight> java NestedIfDemo1
You are eligible

```

**Q.18. Explain Conditional expressions in java.**

**Ans.** Java is equipped with a selection operator that allows us to construct a conditional expression. The use of a conditional expression can in some cases simplify the code with respect to the use of an if-else statement.

**Syntax:**

condition ? expression-1 : expression-2

- condition is a boolean expression
- expression-1 and expression-2 are two arbitrary expressions, which must be of the same type

**Semantics :** Evaluate condition. If the result is true, then evaluate expression-1 and return its value, otherwise evaluate expression-2 and return its value.

**Example:** `System.out.println("bigger value = " + (a > b)? a : b);`

The statement in the example, which makes use of a conditional expression, is equivalent to:

```

if (a > b)
    System.out.println("bigger value = " + a);
else
    System.out.println("bigger value = " + b);

```



Note that the selection operator is similar to the if-else statement, but it works at a different syntactic level:

- The selection operator combines expressions and returns another expression. Hence it can be used wherever an expression can be used.
- The if-else statement groups statements, and the result is a composite statement.

Write a program input using scanner class and print the output statement in java?

Let's see a simple example of Java Scanner where we are getting a single input from the user.

```
import java.util.*;

public class ScannerClassExample1
{
    public static void main(String args[])
    {
        String s = "Hello, This is Java.";
        //Create scanner Object and pass string in it
        Scanner scan = new Scanner(s);
        //Check if the scanner has a token
        System.out.println("Boolean Result: " + scan.hasNext());
        //Print the string
        System.out.println("String: " + scan.nextLine());
        scan.close();
        System.out.println("-----Enter Your Details----- ");
        Scanner in = new Scanner(System.in);
        System.out.print("Enter your name: ");
        String name = in.next();
        System.out.println("Name: " + name);
        System.out.print("Enter your age: ");
        int i = in.nextInt();
        System.out.println("Age: " + i);
        System.out.print("Enter your salary: ");
        double d = in.nextDouble();
        System.out.println("Salary: " + d);
        in.close();
    }
}
```



**Output:**

```

Boolean Result: true
String: Hello, This is Java.
-----Enter Your Details-----
Enter your name: Abhishek
Name: Abhishek
Enter your age: 23
Age: 23
Enter your salary: 25000
Salary: 25000.0

```

**Q.19. Explain Loops in java.**

**Ans.** Loop is an important concept of a programming that allows iterating over the sequence of statements. Loop is designed to execute particular code block till the specified condition is true or all the elements of a collection (array, list etc) are completely traversed. The most common use of loop is to perform repetitive tasks. For example if we want to print table of a number then we need to write print statement 10 times. However, we can do the same with a single print statement by using loop. Loop is designed to execute its block till the specified condition is true.

Java provides mainly three loop based on the loop structure.

1. for loop
2. while loop
3. do while loop

We will explain each loop individually in details in the below.

**Q.20. Explain For Loop with example.**

**Ans.** The for loop is used for executing a part of the program repeatedly. When the number of execution is fixed then it is suggested to use for loop.

**For Loop Syntax :** Following is the syntax for declaring for loop in Java.

```

for(initialization;condition;increment/decrement)
{
    //statement
}

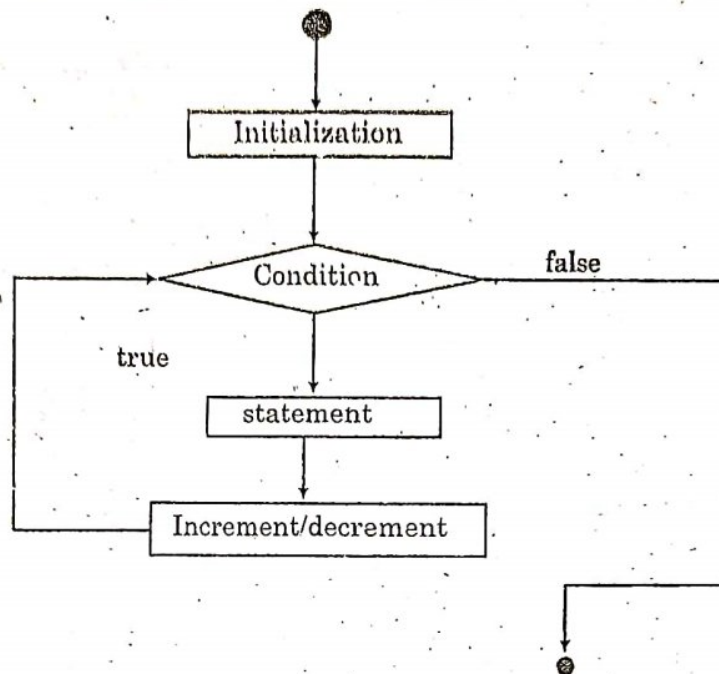
```

**For loop Parameters:** To create a for loop, we need to set the following parameters.

1. **Initialization :** It is the initial part, where we set initial value for the loop. It is executed only once at the starting of loop. It is optional, if we don't want to set initial value.
2. **Condition :** It is used to test a condition each time while executing. The execution continues until the condition is false. It is optional and if we don't specify, loop will be infinite.
3. **Statement :** It is loop body and executed every time until the condition is false.
4. **Increment/Decrement :** It is used for set increment or decrement value for the loop.

**Data Flow Diagram of for loop :** This flow diagram shows flow of the loop. Here we can understand flow of the loop.





**For loop Example :** In this example, initial value of loop is set to 1 and incrementing it by 1 till the condition is true and executes 10 times.

```
public class ForDemo1
{
    public static void main(String[] args)
    {
        int n, i;
        n=2;
        for(i=1;i<=10;i++)
        {
            System.out.println(n+"*"+i+"="+n*i);
        }
    }
}
```

The screenshot shows a Windows command prompt window titled 'C:\Windows\System32\cmd.exe'. The user has entered the following commands and received the following output:

```
D:\StudyTonight> javac ForDemo1.java
D:\StudyTonight> java ForDemo1
2*1=2
2*2=4
2*3=6
2*4=8
2*5=10
2*6=12
2*7=14
2*8=16
2*9=18
2*10=20
```

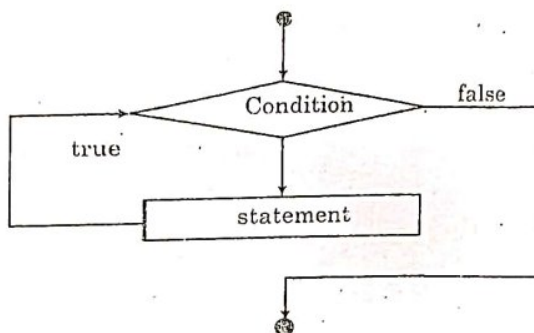


**Q.21. Explain While Loop with example.**

**Ans.** Like for loop, while loop is also used to execute code repeatedly. a control statement. It is used for iterating a part of the program several times. When the number of iteration is not fixed then while loop is used.

**Syntax:**

```
while(condition)
{
    //code for execution
}
```

**Data-flow-diagram of While Block :**

**Example:** In this example, we are using while loop to print 1 to 10 values. In first step, we set conditional variable then test the condition and if condition is true execute the loop body and increment the variable by 1.

```
public class WhileDemo1
{
    public static void main(String[] args)
    {
        inti=1;
        while(i<=10)
        {
            System.out.println(i);
            i++;
        }
    }
}
```

```

C:\Windows\System32\cmd.exe
D:\Studytonight\Java\ac\WhileDemo1>java WhileDemo1
1
2
3
4
5
6
7
8
9
10
  
```



**Q.22. Describe Do-while loop with example.**

**Ans.** In Java, the do-while loop is used to execute statements again and again. This loop executes at least once because the loop is executed before the condition is checked. It means loop condition evaluates after executing of loop body.

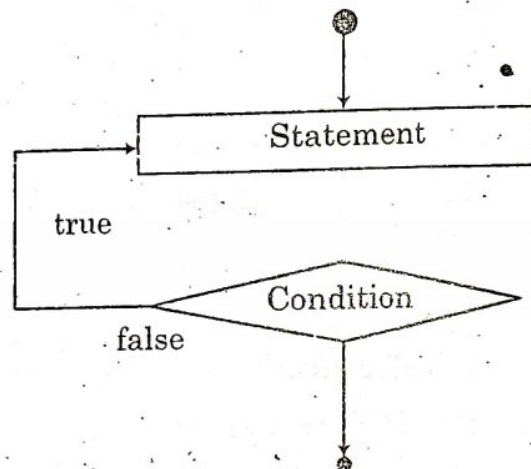
The main difference between while and do-while loop is, in do while loop condition evaluates after executing the loop.

**Syntax:**

Following is the syntax to declare do-while loop in Java.

```
do
{
//code for execution
}
while(condition);
```

**Data Flow Diagram of do-while Block**



**Example:** In this example, we are printing values from 1 to 10 by using the do while loop.

```
public class DoWhileDemo1
{
public static void main(String[] args)
{
inti=1;
do
{
System.out.println(i);
i++;
}while(i<=10);
}
```





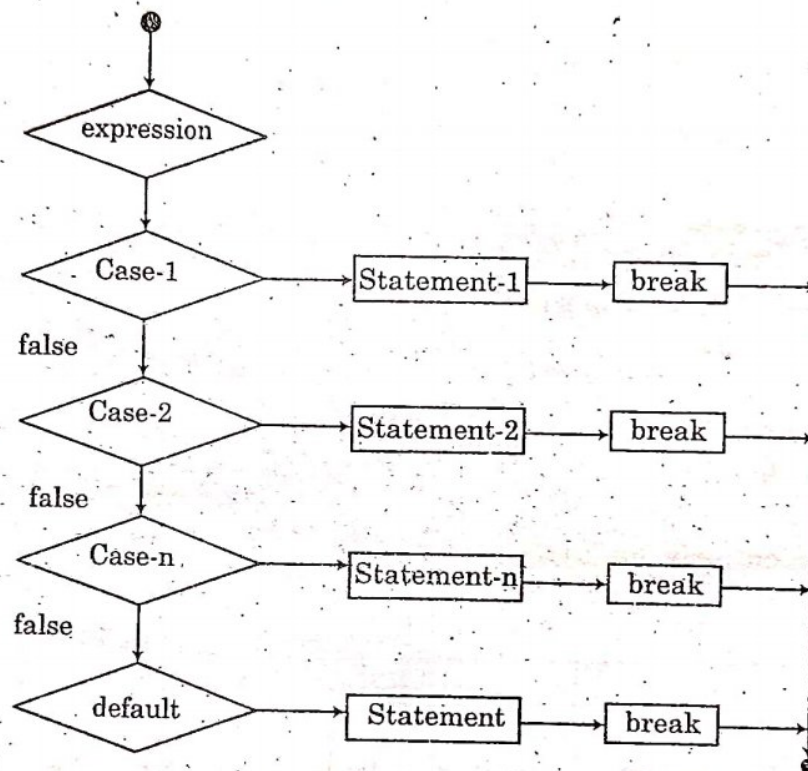


```

Case value n:
// code for execution
break; //optional
default:
code for execution when none of the case is true;
}

```

### Data Flow Diagram of switch Block



### Example: Using integer value

In this example, we are using int type value to match cases. This example returns day based on the numeric value.

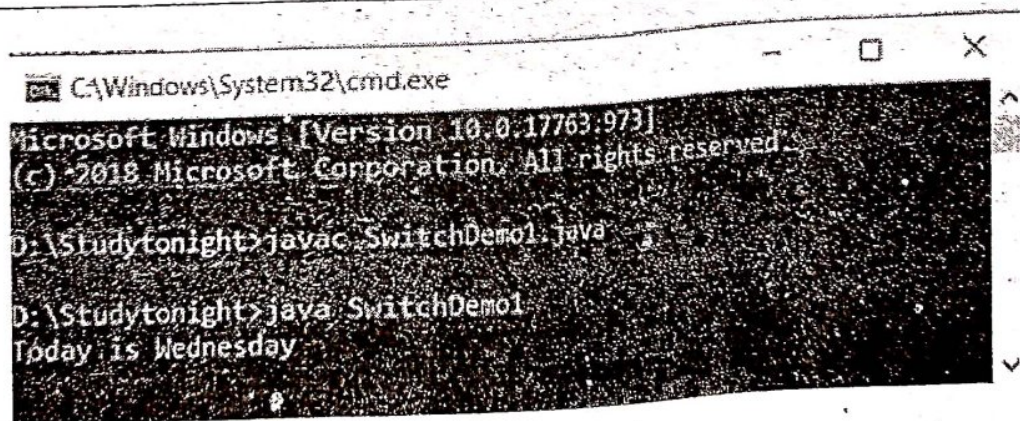
```

public class SwitchDemo1{
    public static void main(String[] args)
    {
        int day = 3;
        String dayName;
        switch (day) {
            case 1:
                dayName = "Today is Monday";
                break;
            case 2:
                dayName = "Today is Tuesday";

```



```
        break;
    case 3:
        dayName = "Today is Wednesday";
        break;
    case 4:
        dayName = "Today is Thursday";
        break;
    case 5:
        dayName = "Today is Friday";
        break;
    case 6:
        dayName = "Today is Saturday";
        break;
    case 7:
        dayName = "Today is Sunday";
        break;
    default:
        dayName = "Invalid day";
        break;
    }
    System.out.println(dayName);
}
```



The screenshot shows a Windows command prompt window titled "C:\Windows\System32\cmd.exe". The text inside the window is as follows:

```
Microsoft Windows [Version 10.0.17763.973]
(c) 2018 Microsoft Corporation. All rights reserved.

D:\Studytonight>javac SwitchDemo1.java

D:\Studytonight>java SwitchDemo1
Today is Wednesday
```

**Q.24.** Explain arrays in java.

**Ans.** An array is a collection of similar data values with a single name. An array can also be defined as, a special type of variable that holds multiple values of the same data type at a time.

In java, arrays are objects and they are created dynamically using new operator. Every array in java is organized using index values. The index value of an array starts with '0' and ends with 'size-1'. We use the index value to access individual elements of an array.

In java, there are two types of arrays and they are as follows :

- One Dimensional Array
- Multi Dimensional Array.



**Creating an array/One Dimensional array :** In the java programming language, an array must be created using new operator and with a specific size. The size must be an integer value but not a byte, short, or long. We use the following syntax to create an array.

### Syntax

```
data_type array_name[ ] = new data_type[size];  
(or)  
data_type[ ] array_name = new data_type[size];
```

Let's look at the following example program.

### Example

```
public class ArrayExample {  
    public static void main(String[] args) {  
        int list[] = new int[5];  
        list[0] = 10;  
        System.out.println("Value at index 0 - " + list[0]);  
        System.out.println("Length of the array - " + list.length);  
    }  
}
```

In java, an array can also be initialized at the time of its declaration. When an array is initialized at the time of its declaration, it need not specify the size of the array and use of the new operator. Here, the size is automatically decided based on the number of values that are initialized.

### Example

```
int list[] = {10, 20, 30, 40, 50};
```

**Multidimensional Array :** In java, we can create an array with multiple dimensions. We can create 2-dimensional, 3-dimensional, or any dimensional array.

In Java, multidimensional arrays are arrays of arrays. To create a multidimensional array variable, specify each additional index using another set of square brackets. We use the following syntax to create a two-dimensional array.

### Syntax

```
data_type array_name[ ][ ] = new data_type[rows][columns];  
(or)  
data_type[ ][ ] array_name = new data_type[rows][columns];
```

When we create a two-dimensional array, it is created with a separate index for rows and columns. The individual element is accessed using the respective row index followed by the column index. A multidimensional array can be initialized while it is created using the following syntax.

### Syntax

```
data_type array_name[ ][ ] = {{value1, value2}, {value3, value4}, {value5, value6},...};
```



When an array is initialized at the time of declaration, it need not specify the size of the array and use of the new operator. Here, the size is automatically decided based on the number of values that are initialized.

### Example

```
int matrix_a[ ][ ] = {{1, 2},{3, 4},{5, 6}};
```

The above statement creates a two-dimensional array of three rows and two columns.

### Q.25. Define methods in java.

**Ans.** A method is a block of statements under a name that gets executes only when it is called. Every method is used to perform a specific task. The major advantage of methods is code re-usability (define the code once, and use it many times).

In a java programming language, a method defined as a behavior of an object. That means, every method in java must belong to a class. Every method in java must be declared inside a class.

Every method declaration has the following characteristics.

- **Return Type** - Specifies the data type of a return value.
- **Name** - Specifies a unique name to identify it.
- **Parameters** - The data values it may accept or receive.
- **{ }** - Defines the block belongs to the method.

**1. Creating a Method:** A method is created inside the class and it may be created with any access specifier. However, specifying access specifier is optional.

Following is the syntax for creating methods in java.

### Syntax

```
class <ClassName>{
    <accessSpecifier> <returnType> <methodName>( parameters ){
        ...
        block of statements;
        ...
    }
}
```

The methodName must begin with an alphabet, and the Lower-case letter is preferred. The methodName must follow all naming rules. If you don't want to pass parameters, we ignore it. If a method defined with return type other than void, it must contain the return statement; otherwise, it may be ignored.

**2. Calling a Method:** In java, a method call precedes with the object name of the class to which it belongs and a dot operator. It may call directly if the method defined with the static modifier. Every method call must be made, as to the method name with parentheses (), and it must terminate with a semicolon;

### Syntax

```
<objectName>.<methodName>( actualArguments );
```



The method call must pass the values to parameters if it has. If the method has a return type, we must provide the receiver.

Let's look at the following example java code.

**Example :**

```
import java.util.Scanner;

public class JavaMethodsExample
{
    int sNo;
    String name;
    Scanner read = new Scanner(System.in);
    void readData()
    {
        System.out.print("Enter Serial Number: ");
        sNo = read.nextInt();
        System.out.print("Enter the Name: ");
        name = read.next();
    }
    static void showData(int sNo, String name)
    {
        System.out.println("Hello, " + name + "! your serial number is " + sNo);
    }
    public static void main(String[] args)
    {
        JavaMethodsExample obj = new JavaMethodsExample();
        obj.readData(); // method call using object
        showData(obj.sNo, obj.name); // method call without using object
    }
}
```

The objectName must begin with an alphabet, and a Lower-case letter is preferred. The objectName must follow all naming rules.

**3. Variable Arguments of a Method :** In java, a method can be defined with a variable number of arguments. That means creating a method that receives any number of arguments of the same data type.

### Syntax

```
<returnType> <methodName>(<dataType...parameterName>);
```



Let's look at the following example java code :

### Example

```
public class JavaMethodWithVariableArgs
{
    void diaplay(int...list)
    {
        System.out.println("\nNumber of arguments: " + list.length);
        for(int i : list) {
            System.out.print(i + "\t");
        }
    }
    public static void main(String[] args)
    {
        JavaMethodWithVariableArgs obj = new JavaMethodWithVariableArgs();
        obj.diaplay(1, 2);
        obj.diaplay(10, 20, 30, 40, 50);
    }
}
```

When a method has the normal parameter and variable-argument, then the variable argument must be specified at the end in the parameters list.





## Classes and Objects

**Q.1. Define fundamentals of Class.**

**Ans.** In Java everything is encapsulated under classes. Class is the core of Java language. It can be defined as a template that describes the behaviors and states of a particular entity.

A class defines new data type. Once defined this new type can be used to create object of that type.

Object is an instance of class. You may also call it as physical existence of a logical template class.

In Java, to declare a class **class** keyword is used. A class contains both **data** and **methods** that operate on that data. The data or variables defined within a class are called **instance variables** and the code that operates on this data is known as **methods**.

Thus, the instance variables and methods are known as **class members**.

### Rules for Java Class

- ❑ A class can have only **public** or **default (no modifier)** access specifier.
- ❑ It can be either **abstract**, **final** or **concrete** (normal class).
- ❑ It must have the **class** keyword, and class must be followed by a legal identifier.
- ❑ It may optionally extend only one parent class. By default, it extends **Object** class.
- ❑ The variables and methods are declared within a set of curly braces.

A Java class can contain fields, methods, constructors, and blocks. Lets see a general structure of a class.

### Java class Syntax

```
class class_name
```

```
{
```

```
    field;
```

```
    method;
```

```
}
```

**A simple class example :** Suppose, Student is a class and student's name, roll-number, age are its **fields** and **info()** is a method. Then class will look like below :

### Syllabus

Class fundamentals, constructors, declaring objects (Object & Object Reference), creating and accessing variables and methods, static and non static variables/methods defining packages, Creating and accessing a package, Importing packages, Understanding CLASSPATH, auto boxing, String, String Buffer.



```

class Student.
{
String name;
int rollno;
int age;
void info(){
// some code
}
}

```

## Q.2. Define constructors.

**Ans.** A constructor is a special method that is used to initialize an object. Every class has a constructor either implicitly or explicitly.

If we don't declare a constructor in the class then JVM builds a default constructor for that class. This is known as **default constructor**.

A constructor has same name as the class name in which it is declared. Constructor must have no explicit return type. Constructor in Java cannot be abstract, static, final or synchronized. These modifiers are not allowed for constructor.

### Syntax to declare constructor

```

className (parameter-list)
{
code-statements
}

```

className is the name of class, as constructor name is same as class name. parameter-list is optional, because constructors can be parameterize and non-parameterize as well.

**Constructor Example :** In Java, constructor structurally looks like given in below program. A Car class has a constructor that provides values to instance variables.

```

class Car
{
String name ;
String model;
Car() //Constructor
{
name = "";
model = "";
}
}

```

**Types of Constructor :** Java Supports two types of constructors:

- ☐ Default Constructor
- ☐ Parameterized constructor

Each time a new object is created at least one constructor will be invoked.



```
Car c = new Car() //Default constructor invoked  
Car c = new Car(name); //Parameterized constructor invoked
```

### Q.3. Explain declaring objects (Object and Object Reference).

**Ans.** Object is an instance of a class while class is a blueprint of an object. An object represents the class and consists of properties and behavior.

Properties refer to the fields declared with in class and behavior represents to the methods available in the class.

In real world, we can understand object as a cell phone that has its properties like: name, cost, color etc and behavior like calling, chatting etc.

So we can say that object is a real world entity. Some real world objects are: ball, fan, car etc.

There is syntax to create an object in the Java.

#### Java Object Syntax

```
className variable_name = new className();
```

Here, className is the name of class that can be anything like: Student that we declared in the above example.

variable\_name is name of reference variable that is used to hold the reference of created object.

The new is a keyword which is used to allocate memory for the object.

Let's see an example to create an object of class Student that we created in the above class section.

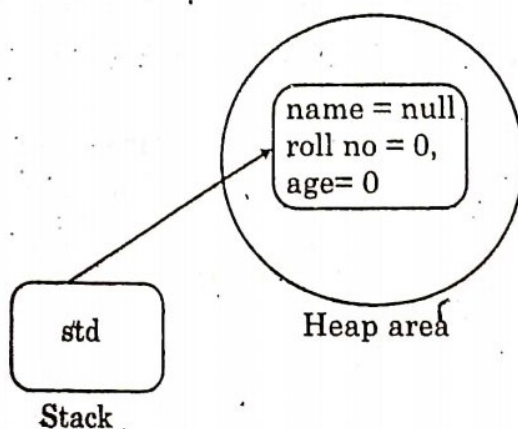
Although there are many other ways by which we can create object of the class. We have covered this section in details in a separate topic.

#### Example: Object creation

```
Student std = new Student();
```

Here, std is an object that represents the class Student during runtime.

The new keyword creates an actual physical copy of the object and assigns it to the std variable. It will have physical existence and get memory in heap area. The new operator dynamically allocates memory for an object.



In this image, we can get idea how the object refer to the memory area. Now let's understand object and class combine by using a real example.



**Example: Creating a Class and its object**

```

public class Student{
    String name;
    int rollno;
    int age;
    void info(){
        System.out.println("Name: "+name);
        System.out.println("Roll Number: "+rollno);
        System.out.println("Age: "+age);
    }
    public static void main(String[] args) {
        Student student = new Student();
        // Accessing and property value
        student.name = "Ramesh";
        student.rollno = 253;
        student.age = 25;
        // Calling method
        student.info();
    }
}

```

Output:

Name: Ramesh

Roll Number: 253

Age: 25

In this example, we created a class Student and an object. Here you may be surprised of seeing main () method but don't worry it is just an entry point of the program by which JVM starts execution.

Here, we used main method to create object of Student class and access its fields and methods.

**Q.4. How can Create and access methods.**

**Ans.** Method in Java is similar to a function defined in other programming languages. Method describes behavior of an object. A method is a collection of statements that are grouped together to perform an operation.

For example, if we have a class Human, then this class should have methods like eating(), walking(), talking() etc, which describes the behavior of the object.

Declaring method is similar to function. See the syntax to declare the method in Java.

```

return-type methodName(parameter-list)
{
    //body of method
}

```

Return-type refers to the type of value returned by the method.



**MethodName** is a valid meaningful name that represent name of a method.

**Parameter-list** represents list of parameters accepted by this method.

Method may have an optional return statement that is used to return value to the caller function.

**Example of a Method:** Lets understand the method by simple example that takes a parameter and returns a string value.

```
public String getName(String st)
{
    String name="StudyTonight";
    name=name+st;
    return name;
}
```

Public string get Name (string st)

modifier   return-type   method-name   parameter

**Modifier:** Modifier are access type of method. We will discuss it in detail later.

**Return Type :** A method may return value. Data type of value return by a method is declare in method heading.

**Method name :** Actual name of the method.

**Parameter:** Value passed to a method.

**Method body:** collection of statement that defines what method does.

**Calling a Method :** Methods are called to perform the functionality implemented in it. We can call method by its name and store the returned value into a variable.

```
String val = GetName(".com")
```

It will return a value studytonight.com after appending the argument passed during method call.

**Returning multiple values :** In Java, we can return multiple values from a method by using array. We store all the values into an array that want to return and then return back it to the caller method. We must specify return-type as an array while creating an array. Let's see an example.

**Example:** Below is an example in which we return an array that holds multiple values.

```
class MethodDemo2{
    static int[] total(int a, int b)
    {
        int[] s = new int[2];
```



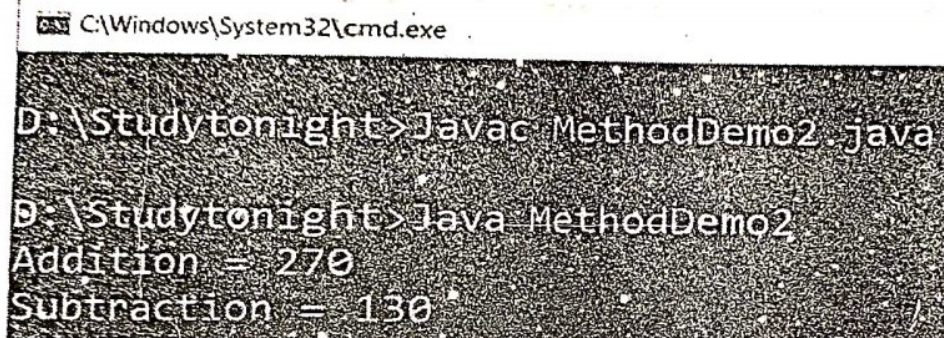
```

    s[0] = a + b;
    s[1] = a - b;
    return s;
}

public static void main(String[] args)
{
    int[] s = total(200, 70);
    System.out.println("Addition = " + s[0]);
    System.out.println("Subtraction = " + s[1]);
}
}

```

**Output:**



```

C:\Windows\System32\cmd.exe
D:\Studytonight>javac MethodDemo2.java
D:\Studytonight>java MethodDemo2
Addition = 270
Subtraction = 130

```

**Return Object from Method :** In some scenario there can be need to return object of a class to the caller function. In this case, we must specify class name in the method definition.

Below is an example in which we are getting an object from the method call. It can also be used to return collection of data.

**Example:** In this example, we created a method `get()` that returns object of Demo class.

```

class Demo{
    int a;
    double b;
    int c;
    Demo(int m, double d, int a)
    {
        a = m;
        b = d;
        c = a;
    }
}

class MethodDemo4{

```



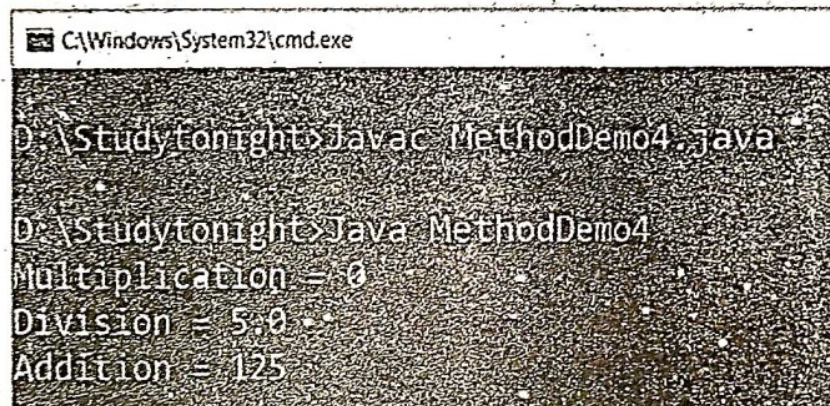
```

static Demo get(int x, int y)
{
    return new Demo(x * y, (double)x / y, (x + y));
}

public static void main(String[] args)
{
    Demo ans = get(25, 5);
    System.out.println("Multiplication = " + ans.a);
    System.out.println("Division = " + ans.b);
    System.out.println("Addition = " + ans.c);
}
}

```

Output:



```

C:\Windows\System32\cmd.exe
D:\Studytonight>javac MethodDemo4.java
D:\Studytonight>Java MethodDemo4
Multiplication = 0
Division = 5.0
Addition = 125

```

**Parameter vs. Argument in a Method :** While talking about method, it is important to know the difference between two terms parameter and argument.

Parameter is variable defined by a method that receives value when the method is called.

Parameter are always local to the method they don't have scope outside the method.

While argument is a value that is passed to a method when it is called.

You can understand it by the below image that explain parameter and argument using a program example.

```

public void sum( int x, int y )
{
    System.out.println(x+y);
}

public static void main( String[] args )
{
    Test b=new Test( );
    b.sum( 10, 20 );
}

```

parameter

argument



**Call-by-value and call-by-reference :** There are two ways to pass an argument to a method.

1. **Call-by-value :** In this approach copy of an argument value is passing to a method. Changes made to the argument value inside the method will have no effect on the arguments.

2. **Call-by-reference :** In this reference of an argument is passing to a method. Any changes made inside the method will affect the argument value.

**Note :** However there is no concept of call-by-reference in Java. Java supports only call by value.

**Example of call-by-value :** Let's see an example in which we are passing argument to a method and modifying its value.

```
public class Test
{
    public void callByValue(int x)
    {
        x=100;
    }

    public static void main(String[] args)
    {
        int x=50;
        Test t = new Test();
        t.callByValue(x); //function call
        System.out.println(x);
    }
}
```

Output: 50

### Q.5. How can Create and access variables?

**Ans.** When we want to store any information, we store it in an address of the computer. Instead of remembering the complex address where we have stored our information, we name that address. The naming of an address is known as variable. Variable is the name of memory location.

In other words, variable is a name which is used to store a value of any type during program execution.

To declare the variable in Java, we can use following syntax

```
datatype variableName;
```

Here, datatype refers to type of variable which can any like: int, float etc.

and variableName can be any like: empId, amount, price etc.

Java Programming language defines mainly three kinds of variables.

#### 1. Instance Variables



## 2. Static Variables (Class Variables)

### 3. Local Variables.

**Instance variables in Java :** Instance variables are variables that are declared inside a class but outside any method, constructor or block. Instance variables are also variables of objects commonly known as field or property. They are referred to as object variables. Each object has its own copy of each variable and thus, it doesn't affect the instance variable if one object changes the value of the variable.

```
class Student
```

```
{
```

```
    String name;
```

```
    int age;
```

```
}
```

Here name and age are instance variables of Student class.

**Static variables in Java :** Static are class variables declared with static keyword. Static variables are initialized only once. Static variables are also used in declaring constants along with final keyword.

```
class Student
```

```
{
```

```
    String name;
```

```
    int age;
```

```
    static int instituteCode=1101;
```

```
}
```

Here instituteCode is a static variable. Each object of Student class will share instituteCode property.

#### Additional points on static variable:

- ☐ static variables are also known as class variables.
- ☐ static means to remain constant.
- ☐ In Java, it means that it will be constant for all the instances created for that class.
- ☐ static variables need not be called from objects.
- ☐ It is called by classname.static\_variable\_name

**Note:** A static variable can never be defined inside a method i.e., it can never be a local variable.

**Example:** Suppose you make 2 objects of class Student and you change the value of static variable from one object. Now when you print it from another object, it will display the changed value. This is because it was declared static i.e., it is constant for every object created.



```

package studytonight;
class Student{
    int a;
    static int id = 35;
    void change(){
        System.out.println(id);
    }
}

public class StudyTonight {
    public static void main(String[] args) {
        Student o1 = new Student();
        Student o2 = new Student();
        o1.change();
        Student.id = 1;
        o2.change();
    }
}

```

**Output:**

35

1

**Local variables in Java :** Local variables are declared in method, constructor or block. Local variables are initialized when method, constructor or block start and will be destroyed once its end. Local variable reside in stack. Access modifiers are not used for local variable.

```

float getDiscount(int price)
{
    float discount;
    discount=price*(20/100);
    return discount;
}

```

Here discount is a local variable.

#### Q.6. Define static Variables/methods with example.

**Ans. Static Variables:** When a variable is declared as static, then a single copy of the variable is created and shared among all objects at a class level. Static variables are, essentially, global variables. All instances of the class share the same static variable.

Important points for static variables :

- ❑ We can create static variables at class-level only.
- ❑ Static block and static variables are executed in order they are present in a program.

Below is the Java program to demonstrate that static block and static variables are executed in order they are present in a program.

```
// Java program to demonstrate execution
```

```
// of static blocks and variables
```



```
class Test {  
    // static variable  
    static int a = m1();  
    // static block  
    static  
    {  
        System.out.println("Inside static block");  
    }  
    // static method  
    static int m1()  
    {  
        System.out.println("from m1");  
        return 20;  
    }  
    // static method(main !!)  
    public static void main(String[] args)  
    {  
        System.out.println("Value of a : " + a);  
        System.out.println("from main");  
    }  
}
```

**Output:**

from m1

Inside static block

Value of a : 20

from main

**Q.7. Define Non-Static Variable with example:**

**Ans. Local Variables :** A variable defined within a block or method or constructor is called local variable. These variables are created when the block is entered or the function is called and destroyed after exiting from the block or when the call returns from the function. The scope of these variables exists only within the block in which the variable is declared. I.e. we can access these variables only within that block. Initialization of Local Variable is Mandatory.

**Instance Variables :** Instance variables are non-static variables and are declared in a class outside any method, constructor or block. As instance variables are declared in a class, these variables are created when an object of the class is created and destroyed when the object is destroyed. Unlike local variables, we may use access specifier for instance variables. If we do



not specify any access specifier then the default access specifier will be used. Initialization of Instance Variable is not Mandatory. Its default value is 0 Instance Variable can be accessed only by creating objects.

**Example:**

```
// Java program to demonstrate
// non-static variables
class GfG {
    // non-static variable
    int rk = 10;
    public static void main(String[] args)
    {
        // Instance created inorder to access
        // a non static variable.
        Gfg g = new Gfg();
        System.out.println("Non static variable"
            + " accessed using instance"
            + " of a class");
        System.out.println("Non Static variable "
            + f.rk);
    }
}
```

**Output:**

Non static variable accessed using instance of a class.

Non Static variable 10

**Q.8. Write the differences between static and non static variables.**

**Ans.**

| Static Variable   | Non Static Variable   |
|---|---|
| Static variables can be accessed using class name                 | Non-static variables can be accessed using instance of a class            |
| Static variables can be accessed by static and non static methods | Non-static variables cannot be accessed inside a static method.           |
| Static variables reduce the amount of memory used by a program.   | Non-static variables do not reduce the amount of memory used by a program |
| Static variables are shared among all instances of a class.       | Non-static variables are specific to that instance of a class.            |



Static variable is like a global variable and is available to all methods.

Non-static variable is like a local variable and they can be accessed through only instance of a class.

### Q.9. Define packages? How can Create and access a package.

**Ans.** A package as the name suggests is a pack (group) of classes, interfaces and other packages. In java we use packages to organize our classes and interfaces. We have two types of packages in Java: built-in packages and the packages we can create (also known as user defined package). In this guide we will learn what packages are, what are user-defined packages in java and how to use them.

In java we have several built-in packages, for example when we need user input, we import a package like this:

```
import java.util.Scanner
```

Here:

java is a top level package

util is a sub package

and Scanner is a class which is present in the sub package util.

Before we see how to create a user-defined package in java, let's see the advantages of using a package.

**Example : Java Packages :** I have created a class Calculator inside a package named letmecalculate. To create a class inside a package, declare the package name in the first statement in your program. A class can have only one package declaration.

Calculator.java file created inside a package letmecalculate

```
package letmecalculate;
public class Calculator {
    public int add(int a, int b){
        return a+b;
    }
    public static void main(String args[]){
        Calculator obj = new Calculator();
        System.out.println(obj.add(10, 20));
    }
}
```

Now let's see how to use this package in another program.

```
import letmecalculate.Calculator;
public class Demo{
    public static void main(String args[]){
        Calculator obj = new Calculator();
        System.out.println(obj.add(100, 200));
    }
}
```



To use the class Calculator, I have imported the package letmecalculate. In the above program I have imported the package as letmecalculate. Calculator, this only imports the Calculator class. However if you have several classes inside package letmecalculate then you can import the package like this, to use all the classes of this package.

```
import letmecalculate.*;
```

**Q.10. Write down the Advantages of using a package in Java.**

**Ans.** These are the reasons why you should use packages in Java:

- **Reusability** : While developing a project in java, we often feel that there are few things that we are writing again and again in our code. Using packages, you can create such things in form of classes inside a package and whenever you need to perform that same task, just import that package and use the class.
- **Better Organization** : Again, in large java projects where we have several hundreds of classes, it is always required to group the similar types of classes in a meaningful package name so that you can organize your project better and when you need something you can quickly locate it and use it, which improves the efficiency.
- **Name Conflicts** : We can define two classes with the same name in different packages so to avoid name collision, we can use packages

**Q.11. Write down the types of Packages in Java.**

**Ans.** As mentioned in the beginning of this guide that we have two types of packages in java.

**1. User Defined Package** : The package we create is called user-defined package.

**2. Built-in Package** : The already defined package like java.io.\*, java.lang.\* etc are known as built-in packages.

We have already discussed built-in packages, lets discuss user-defined packages with the help of examples.

**Q.12. How can Import Packages? Explain with Example.**

**Ans.** As we have seen that both package declaration and package import should be the first statement in your java program. Let's see what should be the order when we are creating a class inside a package while importing another package.

```
//Declaring a package
package anotherpackage;
//importing a package
import letmecalculate.Calculator;
public class Example{
    public static void main(String args[]){
        Calculator obj = new Calculator();
        System.out.println(obj.add(100, 200));
    }
}
```

So the order in this case should be:

package declaration

package import



**Example :** Using fully qualified name while importing a class

You can use fully qualified name to avoid the import statement. Lets see an example to understand this:

Calculator.java

```
package letmecalculate;
public class Calculator {
    public int add(int a, int b){
        return a+b;
    }
    public static void main(String args[]){
        Calculator obj = new Calculator();
        System.out.println(obj.add(10, 20));
    }
}
```

Example.java

```
//Declaring a package
package anotherpackage;
public class Example{
    public static void main(String args[]){
        //Using fully qualified name instead of import
        letmecalculate.Calculator obj =
            new letmecalculate.Calculator();
        System.out.println(obj.add(100, 200));
    }
}
```

In the Example class, instead of importing the package, I have used the full qualified name such as package\_name.class\_name to create the object of it.

### Q.13. How to Set CLASSPATH in Java?

**Ans. CLASSPATH :** CLASSPATH is an environment variable which is used by Application ClassLoader to locate and load the .class files. The CLASSPATH defines the path, to find thirdparty and user-defined classes that are not extensions or part of Java platform. Include all the directories which contain .class files and JAR files when setting the CLASSPATH.

You need to set the CLASSPATH if :

- ☐ You need to load a class that is not present in the current directory or any sub-directories.
- ☐ You need to load a class that is not in a location specified by the extensions mechanism.

The CLASSPATH depends on what you are setting the CLASSPATH. The CLASSPATH has a directory name or file name at the end. The following points describe what should be the end of the CLASSPATH.



- ❑ If a JAR or zip, the file contains class files, the CLASSPATH end with the name of the zip or JAR file.
- ❑ If class files placed in an unnamed package, the CLASSPATH ends with the directory that contains the class files.
- ❑ If class files placed in a named package, the CLASSPATH ends with the directory that contains the root package in the full package name, that is the first package in the full package name.

The default value of CLASSPATH is a dot (.). It means the only current directory searched. The default value of CLASSPATH overrides when you set the CLASSPATH variable or using the -classpath command (for short -cp). Put a dot (.) in the new setting if you want to include the current directory in the search path.

If CLASSPATH finds a class file which is present in the current directory, then it will load the class and use it, irrespective of the same name class presents in another directory which is also included in the CLASSPATH.

If you want to set multiple classpaths, then you need to separate each CLASSPATH by a semicolon (;).

The third-party applications (MySQL and Oracle) that use the JVM can modify the CLASSPATH environment variable to include the libraries they use. The classes can be stored in directories or archives files. The classes of the Java platform are stored in rt.jar.

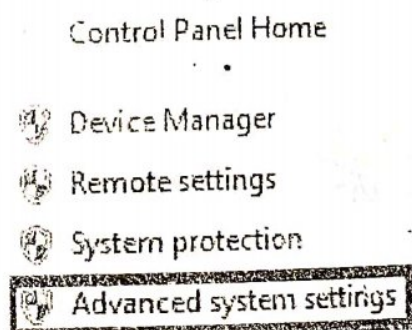
There are two ways to ways to set CLASSPATH: through Command Prompt or by setting Environment Variable.

Let's see how to set CLASSPATH of MySQL database:

**Step 1:** Click on the Windows button and choose Control Panel. Select System.

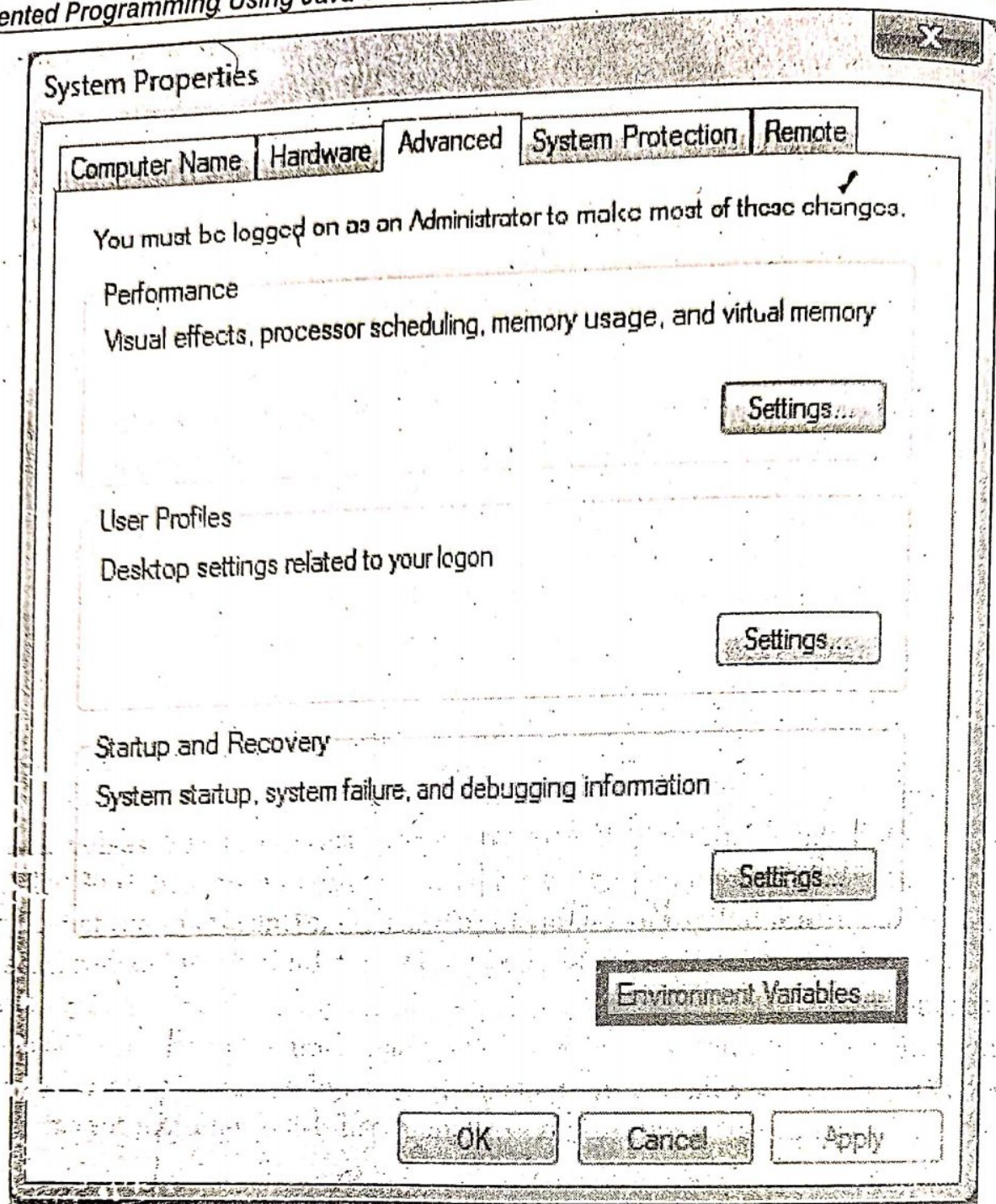


**Step 2:** Click on Advanced System Settings.



**Step 3:** A dialog box will open. Click on Environment Variables.

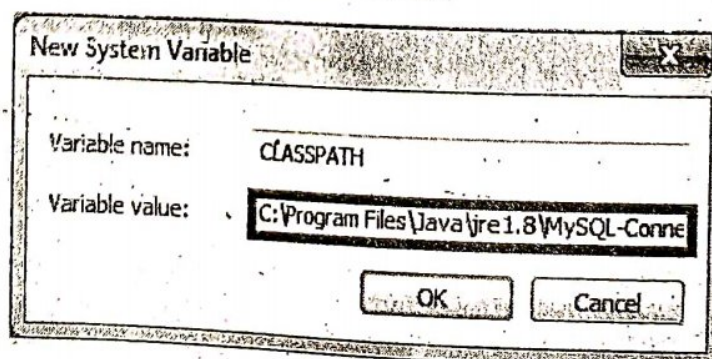




**Step 4:** If the CLASSPATH already exists in System Variables, click on the Edit button then put a semicolon (;) at the end. Paste the Path of MySQL-Connector Java.jar file.

If the CLASSPATH doesn't exist in System Variables, then click on the New button and type Variable name as CLASSPATH and Variable value as C:\Program Files\Java\jre1.8\MySQLConnector Java.jar;;

Remember: Put ;; at the end of the CLASSPATH.





**Q.14. Define auto boxing and Unboxing.**

**Ans. Auto boxing :** Converting a primitive value into an object of the corresponding wrapper class is called auto boxing. For example, converting int to Integer class. The Java compiler applies auto boxing when a primitive value is:

- ☐ Passed as a parameter to a method that expects an object of the corresponding wrapper class.
- ☐ Assigned to a variable of the corresponding wrapper class.

**Unboxing :** Converting an object of a wrapper type to its corresponding primitive value is called unboxing. For example conversion of Integer to int. The Java compiler applies unboxing when an object of a wrapper class is:

- ☐ Passed as a parameter to a method that expects a value of the corresponding primitive type.
- ☐ Assigned to a variable of the corresponding primitive type.

The following table lists the primitive types and their corresponding wrapper classes, which are used by the Java compiler for auto boxing and unboxing:

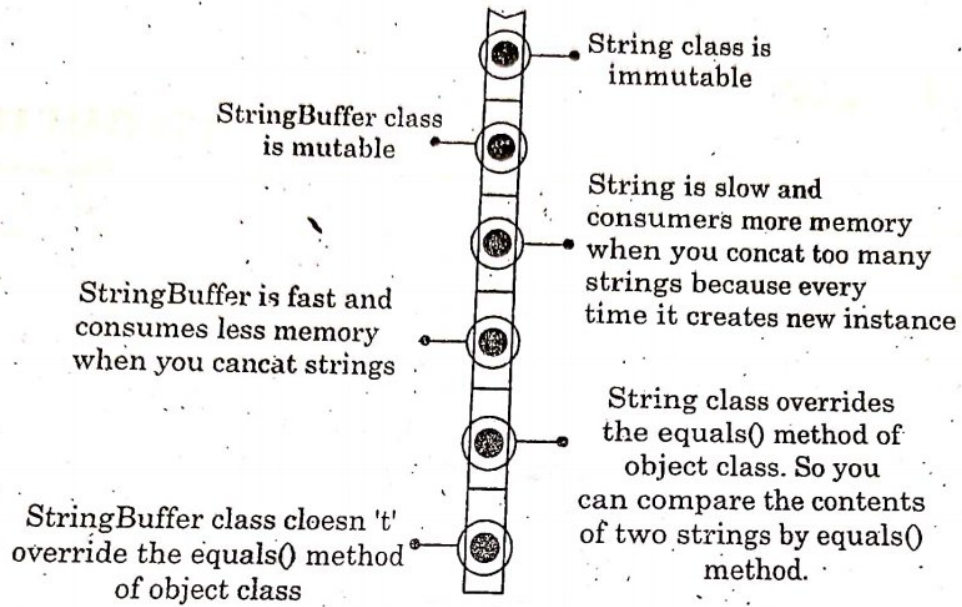
| Primitive type | Wrapper Class |
|----------------|---------------|
| boolean        | Boolean       |
| byte           | Byte          |
| char           | Character     |
| float          | Float         |
| int            | Integer       |
| long           | Long          |
| short          | Short         |
| double         | Double        |

There are many differences between String and StringBuffer. A list of differences between String and StringBuffer are given below:

| No. | String   | String Buffer  |
|-----|--|--|
| 1.  | String class is immutable.   | StringBuffer class is mutable.   |
| 2.  | String is slow and consumes more memory when you concat too many strings because every time it creates new instance.           | StringBuffer is fast and consumes less memory when you concat strings.   |
| 3.  | String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method. | StringBuffer class doesn't override the equals() method of Object class. |



## String Buffer vs String





## Inheritance

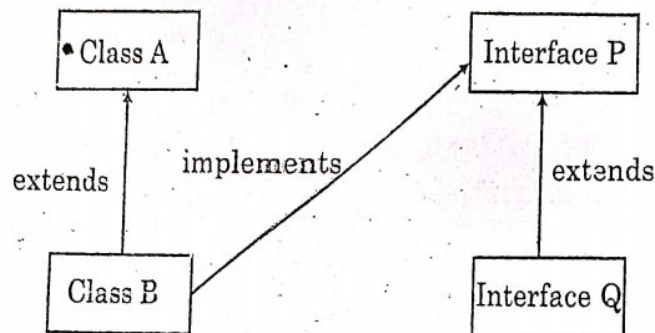
## Q.1. Define inheritance.

**Ans.** Inheritance is one of the key features of Object Oriented Programming. Inheritance provided mechanism that allowed a class to inherit property of another class. When a Class extends another class it inherits all non-private members including fields and methods. Inheritance in Java can be best understood in terms of Parent and Child relationship, also known as Super class (Parent) and Sub class (child) in Java language.

Inheritance defines is-a relationship between a Super class and its Sub class. Extends and implements keywords are used to describe inheritance in Java.

## Syllabus

Definition of inheritance, protected data, private data, public data, constructor changing, order of invocation, types of inheritance, single inheritance, multilevel inheritance, hierarchical inheritance, hybrid inheritance, access control (Private Vs Public Vs Protected Vs Default).



Let us see how extends keyword is used to achieve Inheritance. It shows super class and subclass relationship.

```

class Vehicle
{
    .....
}
class Car extends Vehicle
{
    ..... //extends the property of vehicle class
}
  
```

Now based on above example. In OOPs term we can say that,

- Vehicle is super class of Car.
- Car is sub class of Vehicle.
- Car IS-A Vehicle.

## Q.2. Write the Purpose of Inheritance with example.

**Ans.** 1. It promotes the code reusability i.e., the same methods and variables which are defined in a parent/super/base class can be used in the child/sub/derived class.

2. It promotes polymorphism by allowing method overriding.



## Object Oriented Programming Using Java

**Disadvantages of Inheritance :** Main disadvantage of using inheritance is that the two classes (parent and child class) get tightly coupled.

This means that if we change code of parent class, it will affect to all the child classes which is inheriting/deriving the parent class, and hence, it cannot be independent of each other.

**Simple example of Inheritance :** Before moving ahead let's take a quick example and try to understand the concept of Inheritance better,

```
class Parent
{
    public void p1()
    {
        System.out.println("Parent method");
    }
}

public class Child extends Parent {
    public void c1()
    {
        System.out.println("Child method");
    }

    public static void main(String[] args)
    {
        Child cobj = new Child();
        cobj.c1(); //method of Child class
        cobj.p1(); //method of Parent class
    }
}
```

Output:

Child method

**Parent method :** In the code above we have a class Parent which has a method p1(). We then create a new class Child which inherits the class Parent using the extends keyword and defines its own method c1(). Now by virtue of inheritance the class Child can also access the public method p1() of the class Parent.

**Inheriting variables of super class**

All the members of super class implicitly inherits to the child class. Member consists of instance variable and methods of the class.

**Example :** In this example the sub-class will be accessing the variable defined in the super class.

```
class Vehicle
{
    // variable defined
    String vehicleType;
```



```

}
public class Car extends Vehicle {
    String modelType;
    public void showDetail()
    {
        vehicleType = "Car";           //accessing Vehicle class member variable
        modelType = "Sports";
        System.out.println(modelType + " " + vehicleType);
    }
    public static void main(String[] args)
    {
        Car car = new Car();
        car.showDetail();
    }
}

```

Output: sports Car

**Q.3. Define protected data, private data and public data in inheritance.**

**Ans. Private Data :** The scope of private modifier is limited to the class only.

1. Private Data members and methods are only accessible within the class
2. Class and Interface cannot be declared as private
3. If a class has private constructor then you cannot create the object of that class from outside of the class.

Let's see an example to understand this:

**Private Data Example in Java :** This example throws compilation error because we are trying to access the private data member and method of class ABC in the class Example. The private data member and method are only accessible within the class.

```

class ABC{
    private double num = 100;
    private int square(int a){
        return a*a;
    }
}

public class Example{
    public static void main(String args[]){
        ABC obj = new ABC();
        System.out.println(obj.num);
        System.out.println(obj.square(10));
    }
}

```

**Output:**

Compile - time error



**2. Protected Data :** Protected data member and method are only accessible by the classes of the same package and the subclasses present in any package. You can also say that the protected access modifier is similar to default access modifier with one exception that it has visibility in sub classes. Classes cannot be declared protected. This access modifier is generally used in a parent child relationship.

**Protected Data Example in Java :** In this example the class Test which is present in another package is able to call the addTwoNumbers() method, which is declared protected. This is because the Test class extends class Addition and the protected modifier allows the access of protected members in subclasses (in any packages).

Addition.java

```
package abcpackage;
public class Addition {
    protected int addTwoNumbers(int a, int b){
        return a+b;
    }
}
```

Test.java

```
package xyzpackage;
import abcpackage.*;
class Test extends Addition{
    public static void main(String args[]){
        Test obj = new Test();
        System.out.println(obj.addTwoNumbers(11, 22));
    }
}
```

**Output:**

33

**3. Public Data :** The members, methods and classes that are declared public can be accessed from anywhere. This modifier doesn't put any restriction on the access.

**Public Data Example in Java :** Let's take the same example that we have seen above but this time the method addTwoNumbers() has public modifier and class Test is able to access this method without even extending the Addition class. This is because public modifier has visibility everywhere.

Addition.java

```
package abcpackage;
public class Addition {
    public int addTwoNumbers(int a, int b){
        return a+b;
    }
}
```

Test.java



```
package xyzpackage;
import abcpackage.*;
class Test{
    public static void main(String args[]){
        Addition obj = new Addition();
        System.out.println(obj.addTwoNumbers(100, 1));
    }
}
```

Output:

101

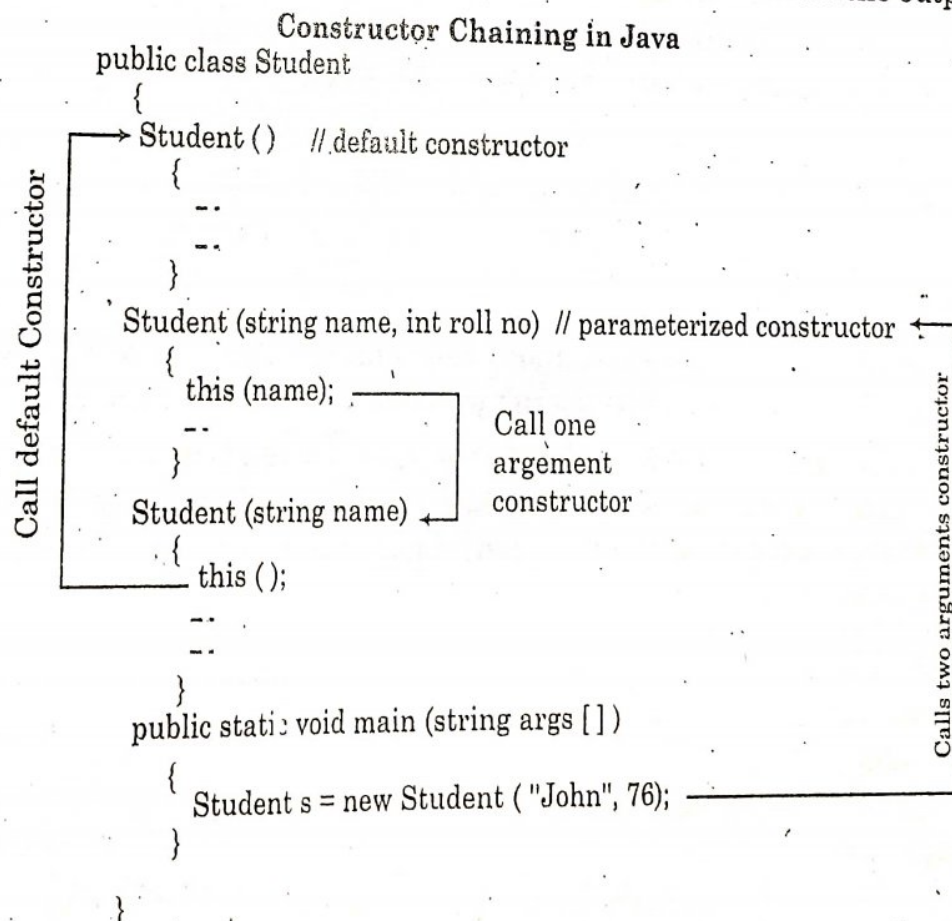
**Q.4. Explain Constructor chaining in java with example.**

**Ans.** In constructor chain, a constructor is called from another constructor in the same class this process is known as constructor chaining. It occurs through inheritance. When we create an instance of a derived class, all the constructors of the inherited class (base class) are first invoked, after that the constructor of the calling class (derived class) is invoked.

We can achieve constructor chaining in two ways:

- ❑ Within the same class: If the constructors belong to the same class, we use this
- ❑ From the base class: If the constructor belongs to different classes (parent and child classes), we use the super keyword to call the constructor from the base class.

Remember that changing the order of the constructor does not affect the output.



**The Need of Constructor Chaining :** Suppose, there are five tasks to perform. There are two ways to perform these tasks, either implement all the tasks in a single constructor or create separate tasks in a single constructor.



By using the constructor chaining mechanism, we can implement multiple tasks in a single constructor. So, whenever we face such types of problems, we should use constructor chaining. We can make the program more readable and understandable by using constructor chaining.

### Rules of Constructor Chaining

- ❑ An expression that uses this keyword must be the first line of the constructor.
- ❑ Order does not matter in constructor chaining.
- ❑ There must exist at least one constructor that does not use this

### Constructor calling from another Constructor

The calling of the constructor can be done in two ways:

- ❑ By using this() keyword: It is used when we want to call the current class constructor within the same class.
- ❑ By using super() keyword: It is used when we want to call the superclass constructor from the base class.

**Constructor Chaining Example :** We use this () keyword if we want to call the current class constructor within the same class. The use of this () is mandatory because JVM never puts it automatically like the super () keyword. Note that this () must be the first line of the constructor. There must exist at least one constructor without this () keyword.

#### Syntax:

this(); or this(parameters list);

#### For example:

this();

this("Javatpoint");

Let's create a Java program and call the current class constructor.

#### ConstructorChain.java

```
public class ConstructorChain
{
    //default constructor
    ConstructorChain()
    {
        this("Javatpoint");
        System.out.println("Default constructor called.");
    }
    //parameterized constructor
    ConstructorChain(String str)
    {
        System.out.println("Parameterized constructor called");
    }
    //main method
    public static void main(String args[])
    {
        //initializes the instance of example class
        ConstructorChain cc= new ConstructorChain();
    }
}
```



Output:

Parameterized constructor called  
Default constructor called

In the above example, we have created an instance of the class without passing any parameter. It first calls the default constructor and the default constructor redirects the call to the parameterized one because of this (). The statements inside the parameterized constructor are executed and return back to the default constructor. After that, the rest of the statements in the default constructor is executed and the object is successfully initialized. The following is the calling sequence of the constructor:

```
ConstructorChain cc = new ConstructorChain(); -> ConstructorChain() -
> ConstructorChain(String str) -> System.out.println() -> ConstructorChain() -
> System.out.println()
```

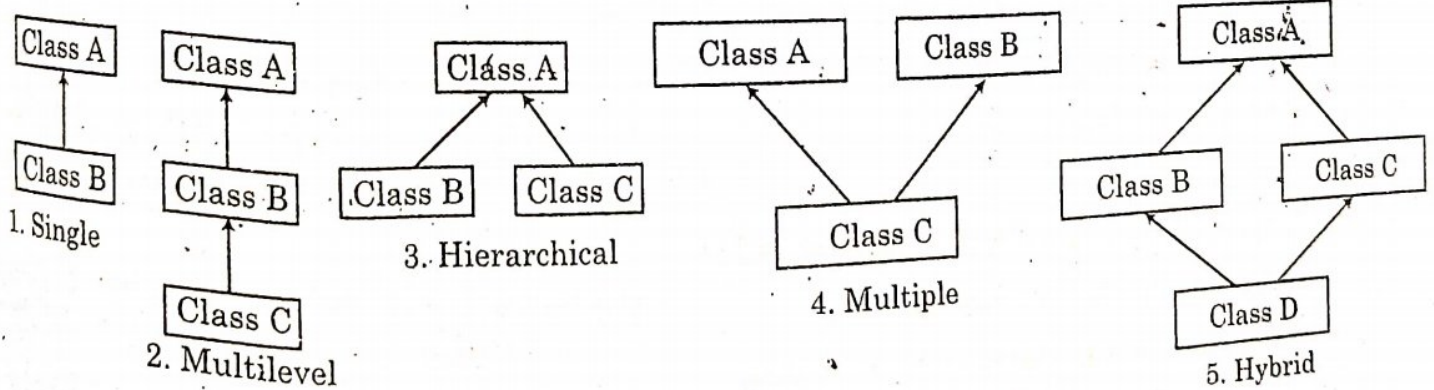
**Q.5. Explain how many types of inheritance.**

**Ans.** Java mainly supports only three types of inheritance that are listed below.

1. Single Inheritance
2. Multilevel Inheritance
3. Hierarchical Inheritance
4. Multiple Inheritance
5. Hybrid Inheritance.

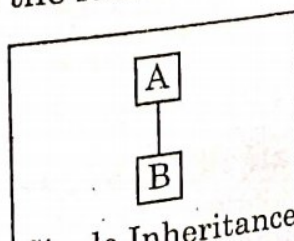
**Note:** Multiple inheritance is not supported in java.

We can get a quick view of type of inheritance from the below image :



**Q.6. Define single inheritance with example.**

**Ans.** When a class inherits another class, it is known as a single inheritance. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.





File: TestInheritance.java

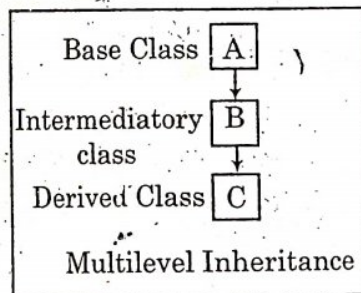
```
class Animal{
    void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
    void bark(){System.out.println("barking...");}
}
class TestInheritance{
    public static void main(String args[]){
        Dog d=new Dog();
        d.bark();
        d.eat();
    }
}
```

Output:

barking...  
eating...

#### Q.7. Explain multilevel Inheritance with example.

**Ans.** When there is a chain of inheritance, it is known as multilevel inheritance. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.



File: TestInheritance2.java

```
class Animal{
    void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
    void bark(){System.out.println("barking...");}
}
class BabyDog extends Dog{
    void weep(){System.out.println("weeping...");}
}
class TestInheritance2{
    public static void main(String args[]){
```



```

BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}

```

**Output:**

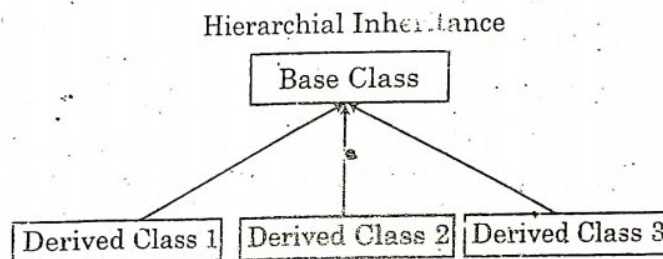
```

weeping...
barking...
eating...

```

**Q.8.** Define hierarchical inheritance with example.

**Ans.** When two or more classes inherits a single class, it is known as hierarchical inheritance. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.



File: TestInheritance3.java

```

class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class Cat extends Animal{
void meow(){System.out.println("meowing...");}
}
class TestInheritance3{
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();
//c.bark();//C.T.Error
}
}

```

**Output:**

```

meowing...

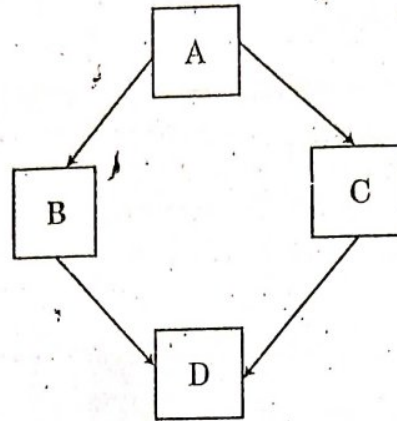
```



eating...

### Q.9. Define hybrid inheritance with example.

**Ans.** It is a mix of two or more of the above types of inheritance. Since java doesn't support multiple inheritance with classes, the hybrid inheritance is also not possible with classes. In java, we can achieve hybrid inheritance only through Interfaces.



Hybrid Inheritance

File: Solarsystem.java

```

class SolarSystem {
}
class Earth extends SolarSystem {
}
class Mars extends SolarSystem {
}
public class Moon extends Earth {
}
public static void main(String args[]) {
    SolarSystem s = new SolarSystem();
    Earth e = new Earth();
    Mars m = new Mars();
    System.out.println(s instanceof SolarSystem);
    System.out.println(e instanceof Earth);
    System.out.println(m instanceof SolarSystem);
}
}
  
```

**Output:**

```

true
true
true
  
```

### Q.10. Explain Access control (Private Vs Public Vs Protected Vs Default) Inheritance in Java.

**Ans.** Java provides a number of access modifiers to set access levels for classes, variables, methods, and constructors. The four access levels are -



- Visible to the package, the default. No modifiers are needed.
- Visible to the class only (private).
- Visible to the world (public).
- Visible to the package and all subclasses (protected).

**1. Default Access Modifier - No Keyword (Default) :** Default access modifier means we do not explicitly declare an access modifier for a class, field, method, etc.

A variable or method declared without any access control modifier is available to any other class in the same package. The fields in an interface are implicitly public static final and the methods in an interface are by default public.

**Example :** Variables and methods can be declared without any modifiers, as in the following examples -

```
String version = "1.5.1";
boolean processOrder() {
    return true;
}
```

**2. Private Access Modifier-Private :** Methods, variables, and constructors that are declared private can only be accessed within the declared class itself.

Private access modifier is the most restrictive access level. Class and interfaces cannot be private.

Variables that are declared private can be accessed outside the class, if public getter methods are present in the class.

Using the private modifier is the main way that an object encapsulates itself and hides data from the outside world.

**Example :** The following class uses private access control -

```
public class Logger {
    private String format;
    public String getFormat() {
        return this.format;
    }
    public void setFormat(String format) {
        this.format = format;
    }
}
```

Here, the format variable of the Logger class is private, so there's no way for other classes to retrieve or set its value directly.

So, to make this variable available to the outside world, we defined two public methods: getFormat(), which returns the value of format, and setFormat(String), which sets its value.

**3. Public Access Modifier-Public :** A class, method, constructor, interface, etc. declared public can be accessed from any other class. Therefore, fields, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe.



However, if the public class we are trying to access is in a different package, then the public class still needs to be imported. Because of class inheritance, all public methods and variables of a class are inherited by its subclasses.

**Example :** The following function uses public access control -

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

The main () method of an application has to be public. Otherwise, it could not be called by a Java interpreter (such as java) to run the class.

#### 4. Protected Access Modifier-Protected :

Variables, methods, and constructors, which are declared protected in a super class can be accessed only by the subclasses in other package or any class within the package of the protected members' class.

The protected access modifier cannot be applied to class and interfaces. Methods, fields can be declared protected, however methods and fields in a interface cannot be declared protected.

Protected access gives the subclass a chance to use the helper method or variable, while preventing a nonrelated class from trying to use it.

**Example :** The following parent class uses protected access control, to allow its child class override openSpeaker() method -

```
class AudioPlayer {
    protected boolean openSpeaker(Speaker sp) {
        // implementation details
    }
}

class StreamingAudioPlayer extends AudioPlayer {
    boolean openSpeaker(Speaker sp) {
        // implementation details
    }
}
```

Here, if we define openSpeaker() method as private, then it would not be accessible from any other class other than AudioPlayer. If we define it as public, then it would become accessible to the entire outside world. But our intention is to expose this method to its subclass only, that's why we have used protected modifier.

**Access Control and Inheritance :** The following rules for inherited methods are enforced :

- ☐ Methods declared public in a super class also must be public in all subclasses.
- ☐ Methods declared protected in a super class must either be protected or public in subclasses; they cannot be private.
- ☐ Methods declared private are not inherited at all, so there is no rule for them.





# Abstract Class and Interface

## Q.1. Define interface.

**Ans.** Interface is a concept which is used to achieve abstraction in Java. This is the only way by which we can achieve full abstraction. Interfaces are syntactically similar to classes, but you cannot create instance of an Interface and their methods are declared without any body. It can have when you create an interface it defines

what a class can do without saying anything about how the class will do it.

It can have only abstract methods and static fields. However, from **Java 8**, interface can have default and static methods and from **Java 9**, it can have private methods as well.

When an interface inherits another interface `extends` keyword is used whereas class use `implements` keyword to inherit an interface.

### Advantages of Interface

- ☐ It Support multiple inheritance
- ☐ It helps to achieve abstraction
- ☐ It can be used to achieve loose coupling.

### Syntax:

```
interface interface_name {
    // fields
    // abstract/private/default methods
}
```

### Interface Key Points

- ☐ Methods inside interface must not be static, final, native or strictfp.
- ☐ All variables declared inside interface are implicitly public, static and final.
- ☐ All methods declared inside interfaces are implicitly public and abstract, even if you don't use public or abstract keyword.
- ☐ Interface can extend one or more other interface.
- ☐ Interface cannot implement a class.
- ☐ Interface can be nested inside another interface.

Time for an Example!

Let's take a simple code example and understand what interfaces are:

```
interface Moveable
{
    int AVERAGE-SPEED = 40;
    void move();
}
```

### Syllabus

Defining an interface, difference between classes and interface, Key points of Abstract class & interface, difference between and abstract class & interface, implementation of multiple inheritance through interface.



```

interface Moveable
{
    int AVERAGE-SPEED=40;
    void move();
}

```

what you declare

```

interface Moveable
{
    public static final int AVERAGE-SPEED=40;
    public abstract void move();
}

```

what the compiler sees

**Note:** Compiler automatically converts methods of Interface as public and abstract, and the data members as public, static and final by default.

### Example of Interface implementation

In this example, we created an interface and implemented using a class. lets see how to implement the interface.

```

interface Moveable
{
    int AVG-SPEED = 40;
    void move();
}

class Vehicle implements Moveable
{
    public void move()
    {
        System.out.println("Average speed is"+AVG-SPEED);
    }

    public static void main (String[] arg)
    {
        Vehicle vc = new Vehicle();
        vc.move();
    }
}

```

Output: Average speed is 40.

### Q.2. Write the difference between classes and interface.

**Ans.** Class, interface, abstract class, final classes are the important component of the Java language. Before discussing about the differences among them first let's get little intro about all these. So that we can get some idea what these terms refer to.



**Concrete Class :** A class that has all its methods implemented, no method is present without body is known as concrete class.

In other words, a class that contains only non-abstract method will be called concrete class.

**Abstract Class :** A class which is declared as abstract using abstract keyword is known as abstract class. Abstract contains abstract methods and used to achieve abstraction, an important feature of OOP programming. Abstract class cannot be instantiated.

For more details, you can refer our detailed tutorial.

**Interface :** Interface is a blueprint of an class and used to achieve abstraction in Java. Interface contains abstract methods and default, private methods. We cannot create object of the interface. Interface can be used to implement multiple inheritance in Java.

For more details, you can refer our detailed tutorial.

**Final class :** Final class is a class, which is declared using final keyword. Final class is used to prevent inheritance, since we cannot inherit final class. We can create its object and can create static and non-static methods as well.

For more details, you can refer our detailed tutorial. [Click here](#)

Here in this table we are differentiating class, abstract class, interface etc based on some properties like; access modifier, static, non-static etc.

|                              | Concrete Class      | Abstract Class    | Final Class       | Interface      |
|------------------------------|---------------------|-------------------|-------------------|----------------|
| Constructor                  | Yes                 | Yes               | Yes               | No             |
| Non-static (method)          | Yes                 | Yes               | Yes               | Yes            |
| Non-static (variable)        | Yes                 | Yes               | Yes               | No             |
| Access Modifier (by default) | Default             | Default           | Default           | Public         |
| Object Declaration           | Yes                 | Yes               | Yes               | Yes            |
| Instantiation                | Yes                 | No                | Yes               | No             |
| Relation                     | Both (IS-A & HAS-A) | IS-A              | HAS-A             | IS-A           |
| Final Declarations           | May or May not be   | May or May not be | May or May not be | Only Final     |
| Abstract Declarations        | No                  | May or May not be | No                | Fully Abstract |
| Inheritance Keyword          | Extends             | Extends           | No Inheritance    | Implements     |
| Overloading                  | Yes                 | Yes               | Yes               | Yes            |
| Overriding                   | No                  | No                | No                | No             |
| Super keyword                | Yes                 | Yes               | Yes               | No             |



|                               |            |                  |                |                   |
|-------------------------------|------------|------------------|----------------|-------------------|
| This keyword                  | Yes        | Yes              | Yes            | Yes               |
| Byte Code                     | .class     | .class           | .class         | .class            |
| Anonymous Class               | No         | Yes              | No             | Yes               |
| Keywords used for declaration | No keyword | Abstract keyword | Final keyword  | Interface keyword |
| Inheritance                   | Single     | Single           | No Inheritance | Multiple          |
| Static Variables              | Yes        | Yes              | Yes            | Yes               |

**Q.3. Write down the Key points of Abstract class and interface.**

**Ans. Important Reasons for Using Interfaces**

- ☐ Interfaces are used to achieve abstraction.
- ☐ Designed to support dynamic method resolution at run time.
- ☐ It helps you to achieve loose coupling.
- ☐ Allows you to separate the definition of a method from the inheritance hierarchy.

**Important Reasons for Using Abstract Class**

- ☐ Abstract classes offer default functionality for the subclasses.
- ☐ Provides a template for future specific classes.
- ☐ Helps you to define a common interface for its subclasses.
- ☐ Abstract class allows code reusability.

**Q.4. Write the difference between an abstract class and interface.**

**Ans. Abstract class vs. Interface**

**1. Type of methods:** Interface can have only abstract methods. Abstract class can have abstract and non-abstract methods. From Java 8, it can have default and static methods also.

**2. Final Variables:** Variables declared in a Java interface are by default final. An abstract class may contain non-final variables.

**3. Type of variables:** Abstract class can have final, non-final, static and non-static variables. Interface has only static and final variables.

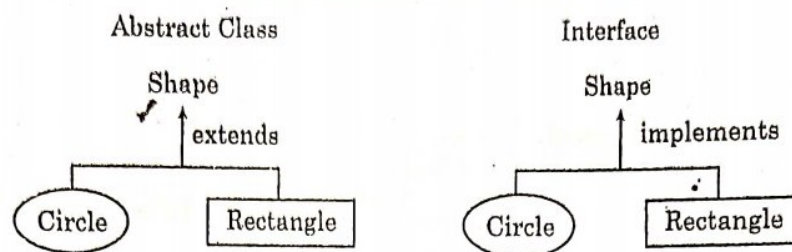
**4. Implementation:** Abstract class can provide the implementation of interface. Interface can't provide the implementation of abstract class.

**5. Inheritance vs. Abstraction:** A Java interface can be implemented using keyword "implements" and abstract class can be extended using keyword "extends".

**6. Multiple implementations:** An interface can extend another Java interface only, an abstract class can extend another Java class and implement multiple Java interfaces.

**7. Accessibility of Data Members:** Members of a Java interface are public by default. A Java abstract class can have class members like private, protected, etc.





**Q.5. Write down code for Interface and Abstract Class in Java.**

**Ans.** Following is sample code to create an interface and abstract class in Java

#### Interface Syntax

```

interface name
{
    //methods
}
  
```

#### Java Interface Example:

```

interface Pet {
    public void test();
}

class Dog implements Pet {
    public void test() {
        System.out.println("Interface Method Implemented");
    }

    public static void main(String args[]) {
        Pet p = new Dog();
        p.test();
    }
}
  
```

#### Abstract Class Syntax

```

abstract class name
{
    // code
}
  
```

#### Abstract class example:



```
abstract class Shape {
    int b = 20;
    abstract public void calculateArea();
}

public class Rectangle extends Shape {
    public static void main(String args[]) {
        Rectangle obj = new Rectangle();
        obj.b = 200;
        obj.calculateArea();
    }

    public void calculateArea() {
        System.out.println("Area is " + (obj.b * obj.b));
    }
}
```

**Q.6. Explain the Implementation of multiple inheritance through interface.**

**Ans.** An interface contains variables and methods like a class but the methods in an interface are abstract by default unlike a class. Multiple inheritance by interface occurs if a class implements multiple interfaces or also if an interface itself extends multiple interfaces.

A program that demonstrates multiple inheritance by interface in Java is given as follows:

**Example**

```
interface AnimalEat {
    void eat();
}

interface AnimalTravel {
    void travel();
}

class Animal implements AnimalEat, AnimalTravel {
    public void eat() {
        System.out.println("Animal is eating");
    }

    public void travel() {
        System.out.println("Animal is travelling");
    }
}

public class Demo {
    public static void main(String args[]) {
        Animal a = new Animal();
        a.eat();
        a.travel();
    }
}
```



**Output**

Animal is eating  
Animal is travelling

Now let us understand the above program.

The interface `AnimalEat` and `AnimalTravel` have one abstract method each i.e., `eat()` and `travel()`. The class `Animal` implements the interfaces `AnimalEat` and `AnimalTravel`. A code snippet which demonstrates this is as follows:

```
interface AnimalEat {  
    void eat();  
}  
interface AnimalTravel {  
    void travel();  
}  
class Animal implements AnimalEat, AnimalTravel {  
    public void eat() {  
        System.out.println("Animal is eating");  
    }  
    public void travel() {  
        System.out.println("Animal is travelling");  
    }  
}
```

In the method `main()` in class `Demo`, an object `a` of class `Animal` is created. Then the methods `eat()` and `travel()` are called. A code snippet which demonstrates this is as follows:

```
public class Demo {  
    public static void main(String args[]) {  
        Animal a = new Animal();  
        a.eat();  
        a.travel();  
    }  
}
```





## UNIT

# 6

## Polymorphism

**Q.1. Explain Method overloading in polymorphism with example.**

**Ans.** Method Overloading is a feature that allows a class to have more than one method having the same name, if their argument lists are different. It is similar to constructor overloading in Java that allows a class to have more than one constructor having different argument lists.

Let's get back to the point, when I say argument list it means the parameters that a method has: For example the argument list of a method `add (int a, int b)` having two parameters is different from the argument list of the method `add(int a, int b, int c)` having three parameters.

**Three ways to overload a method :** In order to overload a method, the argument lists of the methods must differ in either of these:

**1. Number of Parameters.** For example: This is a valid case of overloading :

```
add(int, int)
add(int, int, int)
```

**2. Data type of Parameters.** For example:

```
add(int, int)
add(int, float)
```

**3. Sequence of Data type of Parameters.** For example:

```
add(int, float)
add(float, int)
```

**Invalid Case of Method Overloading :** When I say argument list, I am not talking about return type of the method, for example if two methods have same name, same parameters and have different return type, then this is not a valid method overloading example. This will throw compilation error.

```
int add(int, int)
float add(int, int)
```

Method overloading is an example of Static Polymorphism. We will discuss polymorphism and types of it in a separate tutorial.

### Points to Note:

1. Static Polymorphism is also known as compiled time binding or early binding.
2. Static binding happens at compile time. Method overloading is an example of static binding where binding of method call to its definition happens at Compile time.

### Syllabus

Method and constructor overloading, method overriding, up-casting and down-casting.



**Method Overloading Examples :** As discussed in the beginning of this guide, method overloading is done by declaring same method with different parameters. The parameters must be different in either of these: number, sequence or types of parameters (or arguments). Let's see examples of each of these cases. ✓

Argument list is also known as parameter list.

**Example : Overloading – Different Number of parameters in argument list**

This example shows how method overloading is done by having different number of parameters :

```
class DisplayOverloading
{
    public void disp(char c)
    {
        System.out.println(c);
    }
    public void disp(char c, int num)
    {
        System.out.println(c + " "+num);
    }
}
class Sample
{
    public static void main(String args[])
    {
        DisplayOverloading obj = new DisplayOverloading();
        obj.disp('a');
        obj.disp('a',10);
    }
}
```

Output:

a

a 10

In the above example – method disp() is overloaded based on the number of parameters – We have two methods with the name disp but the parameters they have are different. Both are having different number of parameters.

## Q.2. Explain constructor overloading in polymorphism with example.

**Ans.** Like methods, constructors can also be overloaded. In this guide we will see Constructor overloading with the help of examples. Before we proceed further let's understand what is constructor overloading and why we do it.

Constructor overloading is a concept of having more than one constructor with different parameters list, in such a way so that each constructor performs a different task. For e.g., Vector class has 4 types of constructors. If you do not want to specify the initial capacity and capacity increment then you can simply use default constructor of Vector class like this `Vector v = new Vector();` however if you need to specify the capacity and increment then you call the



parameterized constructor of Vector class with two int arguments like this: Vector v= new Vector(10, 5);

```
public class Demo{
    Demo () {
    ...
    }
    Demo (String s){
    ...
    }
    Demo(Int i){
    ...
    }
    .....
}
```

Three overloaded  
constructors  
They must have  
different  
Parameters list

You must have understood the purpose of constructor overloading. Let's see how to overload a constructor with the help of following java program.

**Constructor Overloading Example :** Here we are creating two objects of class StudentData. One is with default constructor and another one using parameterized constructor. Both the constructors have different initialization code; similarly you can create any number of constructors with different-2 initialization codes for different-2 purposes.

#### StudentData.java

```
class StudentData
{
    private int stuID;
    private String stuName;
    private int stuAge;
    StudentData()
    {
        //Default constructor
        stuID = 100;
        stuName = "New Student";
        stuAge = 18;
    }
    StudentData(int num1, String str, int num2)
    {
        //Parameterized constructor
        stuID = num1;
        stuName = str;
        stuAge = num2;
    }
    //Getter and setter methods
    public int getStuID() {
```



```

return stuID;
}
public void setStuID(int stuID) {
    this.stuID = stuID;
}
public String getStuName() {
    return stuName;
}
public void setStuName(String stuName) {
    this.stuName = stuName;
}
public int getStuAge() {
    return stuAge;
}
public void setStuAge(int stuAge) {
    this.stuAge = stuAge;
}
public static void main(String args[])
{
    //This object creation would call the default constructor
    StudentData myobj = new StudentData();
    System.out.println("Student Name is: "+myobj.getStuName());
    System.out.println("Student Age is: "+myobj.getStuAge());
    System.out.println("Student ID is: "+myobj.getStuID());
    /*This object creation would call the parameterized
    * constructor StudentData(int, String, int)*/
    StudentData myobj2 = new StudentData(555, "Chaitanya", 25);
    System.out.println("Student Name is: "+myobj2.getStuName());
    System.out.println("Student Age is: "+myobj2.getStuAge());
    System.out.println("Student ID is: "+myobj2.getStuID());
}
}

```

### Output:

Student Name is: New Student  
 Student Age is: 18  
 Student ID is: 100  
 Student Name is: Chaitanya  
 Student Age is: 25  
 Student ID is: 555

### Q.3. Define method overriding in polymorphism with example.

**Ans.** Declaring a method in sub class which is already present in parent class is known as method overriding. Overriding is done so that a child class can give its own implementation to a method which is already provided by the parent class. In this case the method in parent class is

### Object Ori

called over  
guide, we

'Meth  
 two classe  
 Both the c  
 to the eat

The p  
 implemen  
 eating.

class

//

" p

{

}

}

class

/

{

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}



```

return stuID;
}
public void setStuID(int stuID) {
    this.stuID = stuID;
}
public String getStuName() {
    return stuName;
}
public void setStuName(String stuName) {
    this.stuName = stuName;
}
public int getStuAge() {
    return stuAge;
}
public void setStuAge(int stuAge) {
    this.stuAge = stuAge;
}
public static void main(String args[])
{
    //This object creation would call the default constructor
    StudentData myobj = new StudentData();
    System.out.println("Student Name is: "+myobj.getStuName());
    System.out.println("Student Age is: "+myobj.getStuAge());
    System.out.println("Student ID is: "+myobj.getStuID());
    /*This object creation would call the parameterized
    * constructor StudentData(int, String, int)*/
    StudentData myobj2 = new StudentData(555, "Chaitanya", 25);
    System.out.println("Student Name is: "+myobj2.getStuName());
    System.out.println("Student Age is: "+myobj2.getStuAge());
    System.out.println("Student ID is: "+myobj2.getStuID());
}
}

```

**Output:**

```

Student Name is: New Student
Student Age is: 18
Student ID is: 100
Student Name is: Chaitanya
Student Age is: 25
Student ID is: 555

```

**Q.3. Define method overriding in polymorphism with example.**

**Ans.** Declaring a method in sub class which is already present in parent class is known as method overriding. Overriding is done so that a child class can give its own implementation to a method which is already provided by the parent class. In this case the method in parent class is



called overridden method and the method in child class is called overriding method. In this guide, we will see what is method overriding in Java and why we use it.

**Method Overriding Example :** Let's take a simple example to understand this. We have two classes: A child class Boy and a parent class Human. The Boy class extends Human class. Both the classes have a common method void eat (). Boy class is giving its own implementation to the eat () method or in other words it is overriding the eat() method.

The purpose of Method Overriding is clear here. Child class wants to give its own implementation so that when it calls this method, it prints Boy is eating instead of Human is eating.

```
class Human{
    //Overridden method
    public void eat()
    {
        System.out.println("Human is eating");
    }
}
class Boy extends Human{
    //Overriding method
    public void eat(){
        System.out.println("Boy is eating");
    }
    public static void main( String args[]) {
        Boy obj = new Boy();
        //This will call the child class version of eat()
        obj.eat();
    }
}
```

Output: Boy is eating

**Advantage of Method Overriding :** The main advantage of method overriding is that the class can give its own specific implementation to a inherited method without even modifying the parent class code.

This is helpful when a class has several child classes, so if a child class needs to use the parent class method, it can use it and the other classes that want to have different implementation can use overriding feature to make changes without touching the parent class code.

#### Q.4. Write down the Rules of Method Overriding in Java.

**Ans. 1. Argument List :** The argument list of overriding method (method of child class) must match the Overridden method (the method of parent class). The data types of the arguments and their sequence should exactly match.

**2. Access Modifier of the overriding method (method of subclass) cannot be more restrictive than the overridden method of parent class.** For e.g. if the Access Modifier of parent class method is public then the overriding method (child class method) cannot have private, protected and default Access modifier, because all of these three access modifiers are more restrictive than public.



For e.g., This is not allowed as child class disp method is more restrictive(protected) than base class(public)

```
class MyBaseClass{
    public void disp()
    {
        System.out.println("Parent class method");
    }
}
class MyChildClass extends MyBaseClass{
    protected void disp(){
        System.out.println("Child class method");
    }
    public static void main( String args[] ) {
        MyChildClass obj = new MyChildClass();
        obj.disp();
    }
}
```

#### Output:

Exception in thread "main" java.lang.Error: Unresolved compilation problem: Cannot reduce the visibility of the inherited method from MyBaseClass

However this is perfectly valid scenario as public is less restrictive than protected. Same access modifier is also a valid one.

```
class MyBaseClass{
    protected void disp()
    {
        System.out.println("Parent class method");
    }
}
class MyChildClass extends MyBaseClass{
    public void disp(){
        System.out.println("Child class method");
    }
    public static void main( String args[] ) {
        MyChildClass obj = new MyChildClass();
        obj.disp();
    }
}
```

#### Output : Child class method

3. Private, static and final methods cannot be overridden as they are local to the class. However static methods can be re-declared in the sub class, in this case the sub-class method would act differently and will have nothing to do with the same static method of parent class.



4. Overriding method (method of child class) can throw unchecked exceptions, regardless of whether the overridden method (method of parent class) throws any exception or not. However the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. We will discuss this in detail with example in the upcoming tutorial.

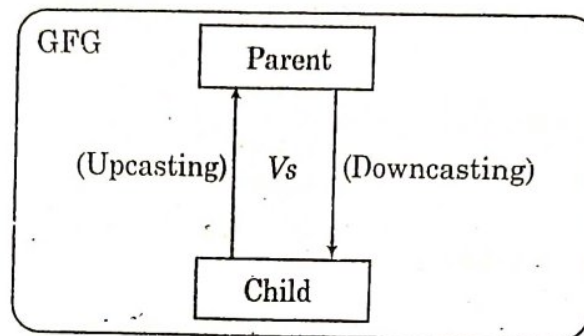
5. Binding of overridden methods happen at runtime which is known as dynamic binding.

6. If a class is extending an abstract class or implementing an interface then it has to override all the abstract methods unless the class itself is a abstract class.

### Q.5. Describe up-casting and down casting.

**Ans.** Typecasting is converting one data type to another.

The following image illustrates the concept of up casting and down casting:



**Up-casting** - Converting a subclass type to a super class type is known as up casting.

Up casting is the typecasting of a child object to a parent object. Up casting can be done implicitly. Up casting gives us the flexibility to access the parent class members but it is not possible to access all the child class members using this feature. Instead of all the members, we can access some specified members of the child class. For instance, we can access the overridden methods.

### Example

```

class Super {
    void Sample() {
        System.out.println("method of super class");
    }
}

public class Sub extends Super {
    void Sample() {
        System.out.println("method of sub class");
    }

    public static void main(String args[]) {
        Super obj =(Super) new Sub(); obj.Sample();
    }
}

```

**Down-casting** - Converting a super class type to a subclass type is known as down casting. Similarly, down casting means the typecasting of a parent object to a child object. Down casting cannot be implicitly.



### Example

```
class Super {
    void Sample() {
        System.out.println("method of super class");
    }
}

public class Sub extends Super {
    void Sample() {
        System.out.println("method of sub class");
    }

    public static void main(String args[]) {
        Super obj = new Sub();
        Sub sub = (Sub) obj; sub.Sample();
    }
}
```



# UNIT

## 7

# Exception Handling

**Q.1. Define exception handling.**

**Ans.** The Exception Handling in Java is one of the powerful mechanisms to handle the runtime errors so that normal flow of the application can be maintained.

**Dictionary Meaning :** Exception is an abnormal condition.

In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

Exception Handling is a mechanism to handle runtime errors such as Class Not Found Exception, IOException, SQLException, RemoteException, etc.

**Advantage of Exception Handling :** The core advantage of exception handling is to maintain the normal flow of the application. An exception normally disrupts the normal flow of the application that is why we use exception handling. Let's take a scenario:

Statement 1;

Statement 2;

Statement 3;

Statement 4;

Statement 5;//exception occurs

Statement 6;

Statement 7;

Statement 8;

Statement 9;

Statement 10;

Suppose there are 10 statements in your program and there occurs an exception at statement 5, the rest of the code will not be executed i.e., statement 6 to 10 will not be executed. If we perform exception handling, the rest of the statement will be executed. That is why we use exception handling in Java.

**Q.2. Write the implementation of keywords (like try, catches, finally, throw and throws) in exceptions?**

**Ans.** There are 5 keywords which are used in handling exceptions in Java are follows :

| Keyword | Description   |
|---------|---|
| try     | The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone. |

### Syllabus

Definition of exception handling, implementation of keywords like try, catches, finally, throw & throws, built in exceptions, creating own exception sub classes importance of exception handling in practical implementation of live projects.



|         |   |
|---------|---|
| catch   | The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.                |
| finally | The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.  |
| throw   | The "throw" keyword is used to throw an exception.  |
| throws  | The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature. |

### Q.3. Explain Java Try Block handling.

Ans. Java try block is used to enclose the code that might throw an exception. It must be used within the method. If an exception occurs at the particular statement of try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception. Java try block must be followed by either catch or finally block.

#### Syntax of Java try-catch.

```
try
{
    //code that may throw an exception
}
catch(Exception_class_Name ref){}
```

#### Syntax of try-finally block

```
try
{
    //code that may throw an exception
}
finally{}
```

### Q.4. Explain Java Catch Block handling?

Ans. Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception. The catch block must be used after the try block only. You can use multiple catch block with a single try block.

**Problem without exception handling :** Let's try to understand the problem if we don't use a try-catch block.

#### Example 1

```
public class TryCatchExample1 {
    public static void main(String[] args) {
```



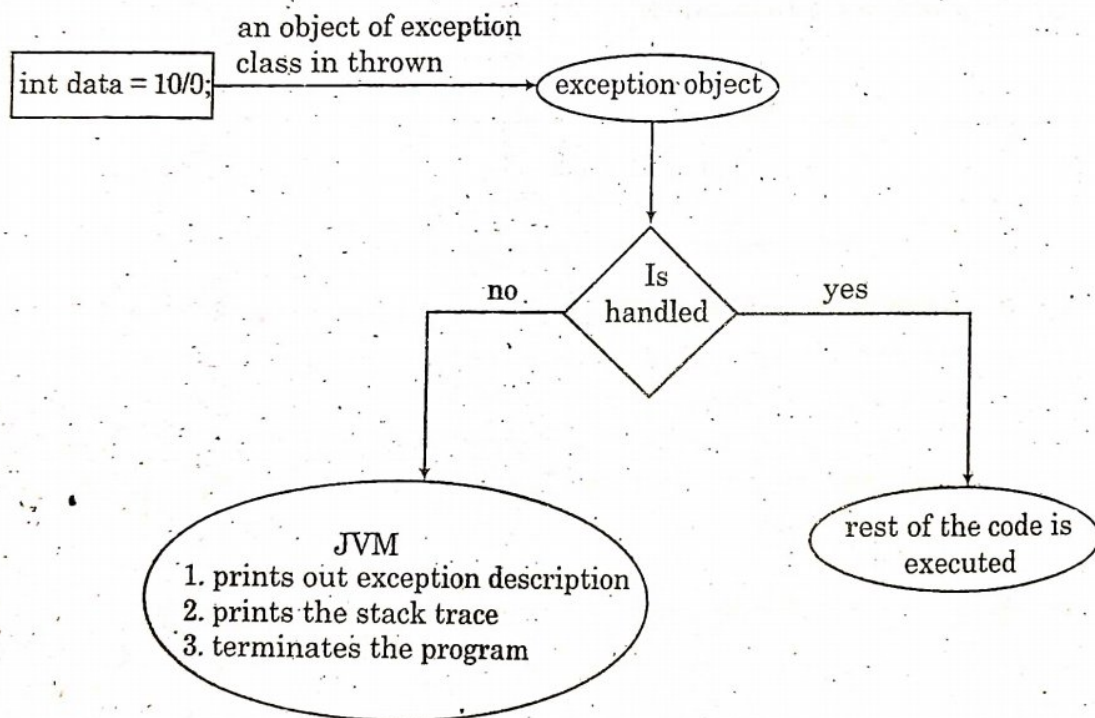
```
int data=50/0; //may throw exception
System.out.println("rest of the code");
```

**Output:**

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

As displayed in the above example, the rest of the code is not executed (in such case, the rest of the code statement is not printed).

There can be 100 lines of code after exception. So all the code after exception will not be executed.

**Internal working of java try-catch block**

The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

- ❑ Prints out exception description.
- ❑ Prints the stack trace (Hierarchy of methods where the exception occurred).
- ❑ Causes the program to terminate.

But if exception is handled by the application programmer, normal flow of the application is maintained i.e., rest of the code is executed.

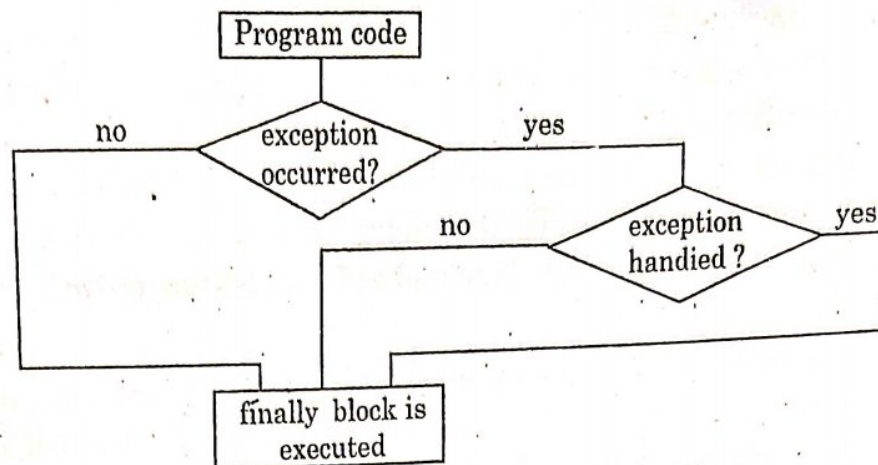
**Q.5. Explain Java Finally Block handling.**

**Ans.** Java finally block is a block that is used to execute important code such as closing connection, stream etc.

Java finally block is always executed whether exception is handled or not.

Java finally block follows try or catch block.





**Usage of Java finally :** Let's see the different cases where java finally block can be used.

**Case 1 :** Let's see the java finally example where exception doesn't occur.

class TestFinallyBlock

```

{
    public static void main(String args[])
    {
        Try
        {
            int data=25/5;
            System.out.println(data);
        }
        catch(NullPointerException e){System.out.println(e);
        }
        finally{System.out.println("finally block is always executed");
        }
        System.out.println("rest of the code...");
    }
}
  
```

**Output:5**

finally block is always executed  
rest of the code...

**Case 2**

Let's see the java finally example where **exception occurs and not handled.**

class TestFinallyBlock1

```

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



```

int data=25/0;
System.out.println(data);
}
catch(NullPointerException e){System.out.println(e);
}
finally{System.out.println("finally block is always executed");
}
System.out.println("rest of the code...");
}
}

```

**Output:**finally block is always executed

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

Case 3 : Let's see the java finally example where exception occurs and handled.

```

public class TestFinallyBlock2
{
public static void main(String args[])
{
try
{
int data=25/0;
System.out.println(data);
}
catch(ArithmeticException e){System.out.println(e);
}
finally{System.out.println("finally block is always executed");
}
System.out.println("rest of the code...");
}
}

```

**Output:** Exception in thread main java.lang.ArithmeticException:/ by zero  
finally block is always executed  
rest of the code...

**Q.6. Difference between Throw and Throws handling in Java.**

**Ans.** There are many differences between throw and throws keywords. A list of differences between throw and throws are given below:

| Throw  | Throws   |
|--|--|
| Java throw keyword is used to explicitly throw an exception. | Java throws keyword is used to declare an exception. |



|    |  |   |
|----|--|---|
| 2. | Checked exception cannot be propagated using throw only. | Checked exception can be propagated with throws.  |
| 3. | Throw is followed by an instance.                        | Throws is followed by class.  |
| 4. | Throw is used within the method.                         | Throws is used with the method signature.   |
| 5. | You cannot throw multiple exceptions.                    | You can declare multiple exceptions e.g.<br>public void method()throws<br>IOException,SQLException. |

**Java throw example**

```
void m(){
    throw new ArithmeticException("sorry");
}
```

**Java throws example**

```
void m()throws ArithmeticException
{
    //method code
}
```

**Java throw and throws example**

```
void m()throws ArithmeticException
{
    throw new ArithmeticException("sorry");
}
```

**Q.7. How to create own exception sub classes?**

**Ans.** Although Java's built-in exceptions handle most common errors, you will probably want to create your own exception types to handle situations specific to your applications. This is quite easy to do: just define a subclass of **Exception** (which is, of course, a subclass of **Throwable**).

Your subclasses don't need to actually implement anything— it is their existence in the type system that allows you to use them as exceptions.

The exception class does not define any methods of its own. It does, of course, inherit those methods provided by **Throwable**. Thus, all exceptions, including those that you create, have the methods defined by **Throwable** available to them. They are shown in Table 10-3. You may also wish to override one or more of these methods in exception classes that you create.

**Exception** defines four public constructors. Two support chained exceptions, described in the next section. The other two are shown here:

```
Exception()
```

```
Exception(String msg)
```

The first form creates an exception that has no description. The second form lets you specify a description of the exception.



Although specifying a description when an exception is created is often useful, sometimes it is better to override `toString()`. Here's why: The version of `toString()` defined by `Throwable` (and inherited by `Exception`) first displays the name of the exception followed by a colon, which is then followed by your description. By overriding `toString()`, you can prevent the exception name and colon from being displayed. This makes for a cleaner output, which is desirable in some cases.

The following example declares a new subclass of `Exception` and then uses that subclass to signal an error condition in a method. It overrides the `toString()` method, allowing a carefully tailored description of the exception to be displayed.

// This program creates a custom exception type.

```
class MyException extends Exception
{
    private int detail;
    MyException(int a) { detail = a;
    }
    public String toString() {
        return "MyException[" + detail + "]";
    }
}

class ExceptionDemo {
    static void compute(int a) throws MyException {
        System.out.println("Called compute(" + a + ")"); if(a > 10)
        throw new MyException(a); System.out.println("Normal exit");
    }
    public static void main(String args[]) { try {
        compute(1);
        compute(20);
    } catch (MyException e) {
        System.out.println("Caught " + e);
    }
}
```

This example defines a subclass of `Exception` called `MyException`. This subclass is quite simple: It has only a constructor plus an overridden `toString()` method that displays the value of the exception. The `ExceptionDemo` class defines a method named `compute()` that throws a `MyException` object. The exception is thrown when `compute()`'s integer parameter is greater than 10. The `main()` method sets up an exception handler for `MyException`, then calls `compute()` with a legal value (less than 10) and an illegal one (greater than 10).



Here is the result:

Called compute(1)

Normal exit

Called compute(20)

Caught MyException[20]

**Q.8. Write down the Importance of exception handling in practical implementation.**

**Ans.** Exception handling is important because it helps maintain the normal, desired flow of the program even when unexpected events occur. If exceptions are not handled, programs may crash or requests may fail. These can be very frustrating for customers and if it happens repeatedly, you could lose those customers.

The worst situation is if your application crashes while the user is doing any important work, especially if their data is lost. To make the user interface robust, it is important to handle exceptions to prevent the application from unexpectedly crashing and losing data. There can be many causes for a sudden crash of the system, such as incorrect or unexpected data input. For example, if we try to add two users with duplicate IDs to the database, we should throw an exception since the action would affect database integrity.

Developers can predict many of the exceptions that a piece of code is capable of throwing. The best course of action is to explicitly handle those exceptions to recover from them gracefully. As we will see ahead, programming languages provide ways to handle exceptions starting from specific ones and moving toward the more generic ones. Exceptions that you cannot easily predict ahead of time are called unhandled exceptions. It's good to capture these in order to gracefully recover. Tracking these exceptions centrally offers visibility to your development team on the quality of the code and what causes these errors so they can fix them.





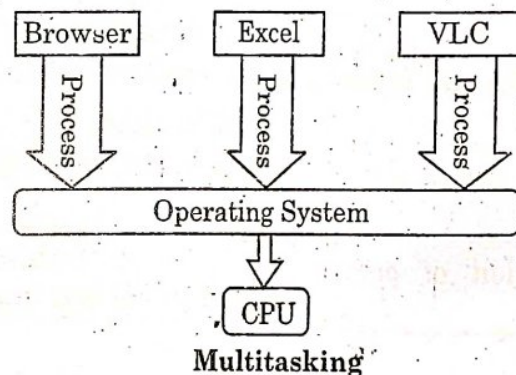
# Multithreading

**Q.1.** Write the Difference between multi threading and multi tasking.

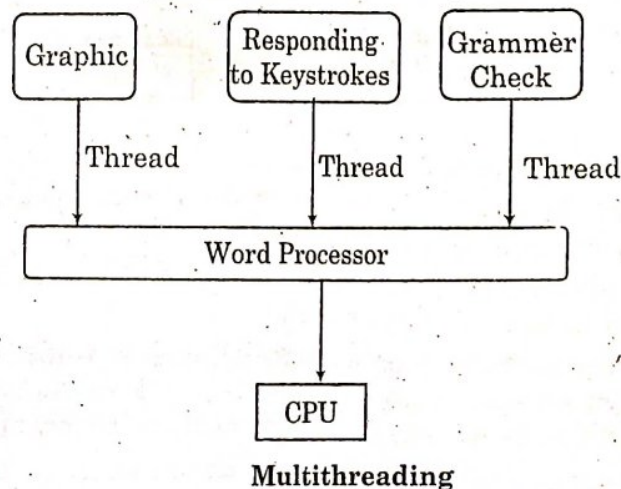
**Ans. Multitasking :** Multitasking is when a CPU is provided to execute multiple tasks at a time. Multitasking involves often CPU switching between the tasks, so that users can collaborate with each program together. Unlike multithreading, in multitasking, the processes share separate memory and resources. As multitasking involves CPU switching between the tasks rapidly, So the little time is needed in order to switch from the one user to next.

## Syllabus

Difference between multi threading and multi tasking, thread life cycle, creating threads, thread priorities, synchronizing threads.



**Multithreading :** Multithreading is a system in which many threads are created from a process through which the computer power is increased. In multithreading, CPU is provided in order to execute many threads from a process at a time, and in multithreading, process creation is performed according to cost. Unlike multitasking, multithreading provides the same memory and resources to the processes for execution.



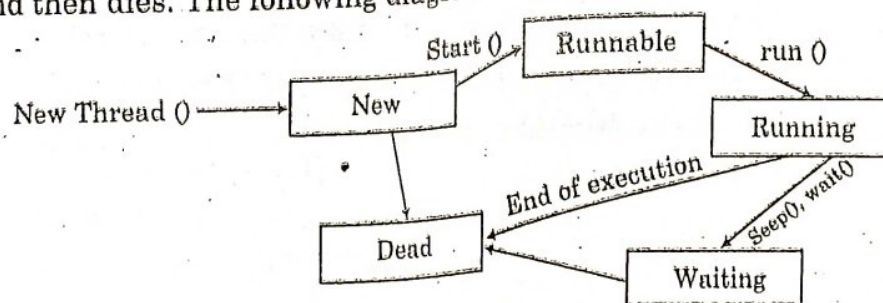


Let's see the difference between multitasking and multithreading:

| S.No. | Multitasking   | Multithreading  |
|-------|--|---|
| 1.    | In multitasking, users are allowed to perform many tasks by CPU.                                     | While in multithreading, many threads are created from a process through which computer power is increased. |
| 2.    | Multitasking involves often CPU switching between the tasks.   | While in multithreading also, CPU switching is often involved between the threads.                          |
| 3.    | In multitasking, the processes share separate memory.  | While in multithreading, processes are allocated same memory.   |
| 4.    | Multitasking component involves multiprocessing.   | While multithreading component does not involve multiprocessing.  |
| 5.    | In multitasking, CPU is provided in order to execute many tasks at a time.                           | While in multithreading also, CPU is provided in order to execute many threads from a process at a time.    |
| 6.    | In multitasking, processes don't share same resources; each process is allocated separate resources. | While in multithreading, each process share same resources.   |
| 7.    | Multitasking is slow compared to multithreading.   | While multithreading is faster.   |
| 8.    | In multitasking, termination of process takes more time.   | While in multithreading, termination of thread takes less time.   |

## Q.2. Explain thread life cycle.

**Ans.** A thread goes through various stages in its lifecycle. For example, a thread is born, started, runs, and then dies. The following diagram shows the complete life cycle of a thread.



Following are the stages of the life cycle :

- ❑ **New** : A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.
- ❑ **Runnable** : after a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
- ❑ **Waiting** : Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. Thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- ❑ **Timed Waiting** : A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transition back to the runnable state when that time interval expires or when the event it is waiting for occurs.



- ❑ **Terminated (Dead) :** A runnable thread enters the terminated state when it completes its task or otherwise terminates.

### Q.3. How to create threads?

**Ans.** There are two ways to create a thread:

- ❑ By extending Thread class
- ❑ By implementing Runnable interface.

**1. Thread class :** Thread class provide constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of Thread class:

- ❑ Thread()
- ❑ Thread(String name)
- ❑ Thread(Runnable r)
- ❑ Thread(Runnable r, String name)

Commonly used methods of Thread class:

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread. JVM calls the run() method on the thread.
3. **public void sleep(long milliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long milliseconds):** waits for a thread to die for the specified milliseconds.

**2. Runnable interface :** The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run ().

1. **public void run() :** is used to perform action for a thread.

**Starting a thread :** Start () method of Thread class is used to start a newly created thread. It performs following tasks:

- ❑ A new thread starts (with new callstack).
- ❑ The thread moves from New state to the Runnable state.
- ❑ When the thread gets a chance to execute, its target run() method will run.

Java Thread Example by extending Thread class

```
class Multi extends Thread{
    public void run(){
        System.out.println("thread is running...");
    }
    public static void main(String args[]){
        Multi t1=new Multi();
        t1.start();
    }
}
```



Output: thread is running...

#### Q.4. How to set thread priorities?

**Ans.** Each thread has a priority. Priorities are represented by a number between 1 and 10. In most cases, thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

3 constants defined in Thread class :

- (i) public static int MIN\_PRIORITY
- (ii) public static int NORM\_PRIORITY
- (iii) public static int MAX\_PRIORITY

Default priority of a thread is 5 (NORM\_PRIORITY). The value of MIN\_PRIORITY is 1 and value of MAX\_PRIORITY is 10.

Example of priority of a Thread:

```
class TestMultiPriority1 extends Thread{
    public void run(){
        System.out.println("running thread name is:"+Thread.currentThread().getName());
        System.out.println("running thread priority is:"+Thread.currentThread().getPriority());

        public static void main(String args[]){
            TestMultiPriority1 m1=new TestMultiPriority1();
            TestMultiPriority1 m2=new TestMultiPriority1();
            m1.setPriority(Thread.MIN_PRIORITY);
            m2.setPriority(Thread.MAX_PRIORITY);
            m1.start();
            m2.start();
        }
    }
}
```

Output:

```
running thread name is: Thread-0
running thread priority is: 10
running thread name is: Thread-1
running thread priority is: 1
```

#### Q.5. Explain synchronizing threads.

**Ans.** When we start two or more threads within a program, there may be a situation when multiple threads try to access the same resource and finally they can produce unforeseen result due to concurrency issues. For example, if multiple threads try to write within a same file then they may corrupt the data because one of the threads can override data or while one thread is using the same file at the same time another thread might be closing the same file.



So there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time. This is implemented using a concept called monitors. Each object in Java is associated with a monitor, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor.

Java programming language provides a very handy way of creating threads and synchronizing their task by using synchronized blocks. You keep shared resources within this block. Following is the general form of the synchronized statement :

#### Syntax

```
synchronized(objectidentifier)
{
    // Access shared variables and other shared resources
}
```

Here, the **objectidentifier** is a reference to an object whose lock associates with the monitor that the synchronized statement represents. Now we are going to see two examples, where we will print a counter using two different threads. When threads are not synchronized, they print counter value which is not in sequence, but when we print counter by putting inside synchronized() block, then it prints counter very much in sequence for both the threads.

**Multithreading Example without Synchronization :** Here is a simple example which may or may not print counter value in sequence and every time we run it, it produces a different result based on CPU availability to a thread.

#### Example

```
class PrintDemo {
    public void printCount() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Counter --- " + i);
            }
        } catch (Exception e) {
            System.out.println("Thread interrupted.");
        }
    }
}

class ThreadDemo extends Thread {
    private Thread t;
    private String threadName;
    PrintDemo PD;
    ThreadDemo( String name, PrintDemo pd) {
        threadName = name;
        PD = pd;
    }
    public void run() {
        PD.printCount();
    }
}
```



```

        System.out.println("Thread " + threadName + " exiting.");
    }

    public void start () {
        System.out.println("Starting " + threadName);
        if (t == null) {
            t = new Thread (this, threadName);
            t.start ();
        }
    }
}

public class TestThread {
    public static void main(String args[]) {
        PrintDemo PD = new PrintDemo();
        ThreadDemo T1 = new ThreadDemo("Thread - 1", PD);
        ThreadDemo T2 = new ThreadDemo("Thread - 2", PD);
        T1.start();
        T2.start();
        // wait for threads to end
        try {
            T1.join();
            T2.join();
        } catch (Exception e) {
            System.out.println("Interrupted");
        }
    }
}

```

This produces a different result every time you run this program -

### Output

Starting Thread - 1

Starting Thread - 2

Counter --- 5

Counter --- 4

Counter --- 3

Counter --- 5

Counter --- 2

Counter --- 1



Counter --- 4  
Thread Thread - 1 exiting.  
Counter --- 3  
Counter --- 2  
Counter --- 1  
Thread Thread - 2 exiting.

**Multithreading Example with Synchronization :** Here is the same example which prints counter value in sequence and every time we run it, it produces the same result.

### Example

```
class PrintDemo {
    public void printCount() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Counter --- " + i);
            }
        } catch (Exception e) {
            System.out.println("Thread interrupted.");
        }
    }
}

class ThreadDemo extends Thread {
    private Thread t;
    private String threadName;
    PrintDemo PD;

    ThreadDemo( String name, PrintDemo pd) {
        threadName = name;
        PD = pd;
    }

    public void run() {
        synchronized(PD) {
            PD.printCount();
        }

        System.out.println("Thread " + threadName + " exiting.");
    }

    public void start () {
        System.out.println("Starting " + threadName);
        if (t == null) {
            t = new Thread (this, threadName);
        }
    }
}
```



```
        t.start ();
    }
}

public class TestThread {
    public static void main(String args[]) {
        PrintDemo PD = new PrintDemo();
        ThreadDemo T1 = new ThreadDemo( "Thread - 1 ", PD );
        ThreadDemo T2 = new ThreadDemo( "Thread - 2 ", PD );
        T1.start();
        T2.start();
        // wait for threads to end
        try {
            T1.join();
            T2.join();
        } catch ( Exception e) {
            System.out.println("Interrupted");
        }
    }
}
```

This produces the same result every time you run this program -

### Output

Starting Thread - 1

Starting Thread - 2

Counter --- 5

Counter --- 4

Counter --- 3

Counter --- 2

Counter --- 1

Thread Thread - 1 exiting.

Counter --- 5

Counter --- 4

Counter --- 3

Counter --- 2

Counter --- 1

Thread Thread - 2 exiting.

