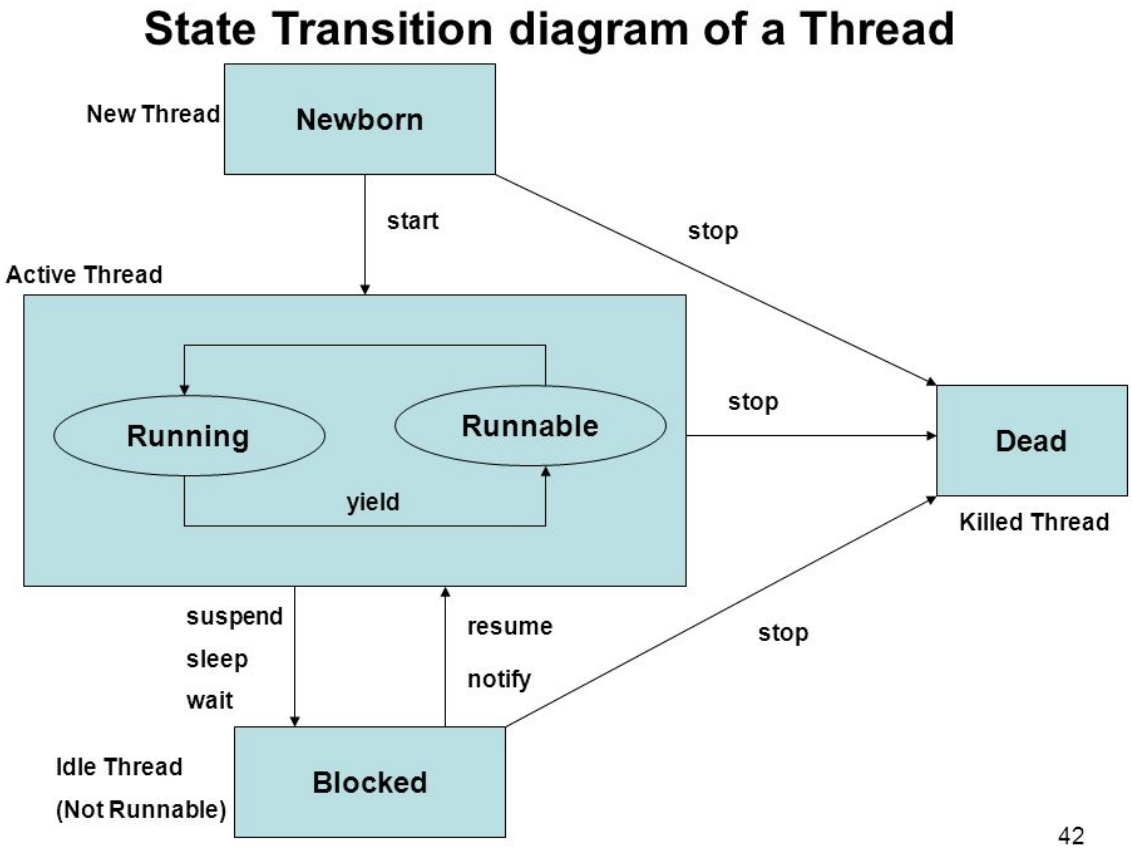
UNIT-V

Multithreading in Java

Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.



1. ***NEW*** – a newly created thread that has not yet started the execution
2. ***RUNNABLE*** – either running or ready for execution but it's waiting for resource allocation

3. ***RUNNING:*** the processor has given its time to the thread for its execution. A thread runs until it relinquishes(yield) the control on its own or pre-empted by higher priority thread
4. ***BLOCKED*** – a thread is prevented from entering into runnable state ans subsequently the running state (happens using suspend, wait and sleep)
5. ***DEAD*** – has completed its execution or end its life

Threads can be created by using two mechanisms :

1. Extending the Thread class
2. Implementing the Runnable Interface

Thread creation by extending the Thread class

We create a class that extends the **java.lang.Thread** class.

This class overrides the run() method available in the Thread class. A thread begins its life inside run() method. We create an object of our new class and call start() method to start the execution of a thread. Start() invokes the run() method on the Thread object.

STOPPING AND BLOCKING THREAD:

Stopping a thread:

Whenever we want to stop a thread from running further, we may do so by calling its *stop()* method. This causes a thread to stop immediately and move it to its *dead* state. It forces the thread to stop abruptly before its completion i.e. it causes premature death. To stop a thread we use the following syntax:

```
t1.stop();
```

Blocking a Thread:

A thread can also be temporarily suspended or blocked from entering into the runnable and subsequently running state by using either of the following thread methods:

sleep(t) // blocked for ‘t’ milliseconds

suspend() // blocked until further orders - resume() method is invoked

wait() // blocked until certain condition occurs- notify () is invoked

These methods cause the thread to go into the *blocked* (or *not-runnable*) state. The thread will return to the runnable state when the specified time is elapsed in the case of sleep( ), the resume() method is invoked in the case of suspend( ), and the notify( ) method is called in the case of wait().

Create a Thread by Extending a Thread Class

The first way to create a thread is to create a new class that extends **Thread** class using the following two simple steps.

This approach provides more flexibility in handling multiple threads created using available methods in Thread class.

1. create a new class that extends **Thread** class

   class ABC extends Thread

   {

   ….

   }

2. implement **run( )** method available in Thread class that is responsible for executing the sequence of code

   public void run( )
   {
   ….}

3. Thread object is created, you can start it by calling **start()** method, which executes a call to run( ) method.

```
ABC a=new ABC();
 a.start( );
```

Example

Here is the preceding program rewritten to extend the Thread −

```
class ThreadA extends Thread
{


  public void run()
{
    System.out.println("Running  ThreadA");

      for(int i = 0; i<= 3; i++)
    {
       System.out.println("Thread  A : i=  "  + i);

      }
   System.out.println(“Exit from A”);
}


}
class ThreadB extends Thread
{


  public void run()
{
    System.out.println("Running  ThreadB");

      for(int k = 0; k <=3; k++)
    {
       System.out.println("Thread B : k=  "  + k);

      }
   System.out.println(“Exit from B”);
}
```

```
}


public class TestThread {

  public static void main(String args[])
{
    ThreadA T1 = new ThreadA();
    T1.start();
   ThreadB T2 = new ThreadB();
    T2.start();

  }
}
```

This will produce the following result −

Output

Running Thread

Thread A: i=1

Thread B: k=1
Thread B: k=2

Thread A: i=2

Thread A: i=3
Exit from A

Thread B: k=3
Exit from B

2. Implementing Runnable interface- steps

1. create a new class that implements Runnable interface

   class ABC implements Runnable

   {

   ….

}

2. implement **run( )** method available in runnable interface that is responsible for executing the sequence of code

   public void run( )
   {
   ….}

3. create a Thread object
4. call the thread's , **start()** method, which executes a call to run( ) method.

   ABC a=new ABC();
   Thread x=new Thread(a);
   x.start( );

Java Thread Example by implementing Runnable interface

FileName: Multi3.java

```
1. class Multi3 implements Runnable{
2. public void run(){
3. System.out.println("thread is running...");
4. }
5.
6. public static void main(String args[]){
7. Multi3 m1=new Multi3();
8. Thread t1 =new Thread(m1);   // Using the constructor Thread(Runnable r)
9. t1.start();
10. }
11. }
```

Output:

```
thread is running...
```

Abstract Classes and Methods

- Data **abstraction** is the process of hiding certain details and showing only essential information to the user.
- Abstraction can be achieved with either **abstract classes** or **interfaces**
- The abstract keyword is a non-access modifier, used for classes and methods:

- **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).

- **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

Points to Remember

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

Example of abstract class

```
1. abstract class A
2. {
3. .....
4. }
```

Abstract Method in Java

A method which is declared as abstract and does not have implementation is known as an abstract method.

Example of abstract method

```
1. abstract void printStatus();  //no method body and abstract
```

An abstract class can have both abstract and regular methods:

```
abstract class Animal {
  public abstract void animalSound();
  public void sleep() {
    System.out.println("Zzz");
  }
}
```

From the example above, it is not possible to create an object of the Animal class:

```
Animal myObj = new Animal(); // will generate an error
```

To access the abstract class, it must be inherited from another class.

Example

```
// Abstract class
abstract class Animal {
  // Abstract method (does not have a body)
  public abstract void animalSound();
  // Regular method
  public void sleep() {
    System.out.println("it is sleeping");
  }
```

```
}


// Subclass (inherit from Animal)
class lion extends Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("Lion the King");
  }
}


class living {
  public static void main(String[] args) {
    lion m = new lion(); // Create a Pig object
    m.animalSound();
    m.sleep();
  }
}
```

Why And When To Use Abstract Classes and Methods?

To achieve security - hide certain details and only show the important details of an object.

Polymorphism in Java

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

Types of polymorphism

In Java polymorphism is mainly divided into two types:

- Compile-time Polymorphism
- Runtime Polymorphism

Type 1: Compile-time polymorphism

It is also known as static polymorphism. This type of polymorphism is achieved by function overloading or operator overloading.

Note: *But Java doesn't support the Operator Overloading.*

Method Overloading: When there are multiple functions with the same name but different parameters then these functions are said to be **overloaded**.

Functions can be overloaded by change in the number of arguments or/and a change in the type of arguments.

Example 1

- Java

```
class Helper {

     int Multiply(int a, int b)
    {

        return a * b;
    }


    double Multiply(double a, double b)
    {

                return a * b;
    }
}
```

```
class GFG {


    public static void main(String[] args)
    {


        System.out.println(Helper.Multiply(2, 4));
    System.out.println(Helper.Multiply(5.5, 6.3));

    }
}
```

Output:

8

34.65

Type 2: Runtime polymorphism

It is also known as Dynamic Method Dispatch. It is a process in which a function call to the overridden method is resolved at Runtime. This type of polymorphism is achieved by Method Overriding.

- The pure polymorphism

The pure polymorphism is **a technique used to define the same method with the same arguments but different implementations**. In a java programming language, pure polymorphism carried out with a method overriding concept.

Method overriding, on the other hand, occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

Example

- Java

```
// Java Program for Method Overriding

 class Parent {

    void Print()

    {

 System.out.println("parent class");

    }
}

class subclass1 extends Parent
```

```
{
  void Print()
   {
   System.out.println("subclass1");
    }
}
class subclass2 extends Parent
{
      void Print()
    {
        System.out.println("subclass2");
    }
}


// Main class
class GFG {
         public static void main(String[] args)
    {
         // Creating object of class 1
        Parent a;
         // Now we will be calling print methods
        a = new subclass1();
        a.Print();
        a = new subclass2();
        a.Print();
    }
}
```

Output:

subclass1

subclass2

Output explanation:

Here in this program, When an object of child class is created, then the method inside the child class is called. This is because The method in the parent class is overridden by the child class. Since The method is overridden, This method has more priority than the parent method inside the child class. So, the body inside the child class is executed.

Java instanceof

The **java instanceof operator** is used to test whether the object is an instance of the specified type (class or subclass or interface).

The instanceof in java is also known as type *comparison operator* because it compares the instance with type. It returns either true or false. If we apply the instanceof operator with any variable that has null value, it returns false.

Simple example of java instanceof

Let's see the simple example of instance operator where it tests the current class.

```
1. class Simple1
2. {
3.  public static void main(String args[]){
4.  Simple1 s=new Simple1();
5.  System.out.println(s instanceof Simple1);//true
6.  }
7. }
```

```
Output:true
```

Downcasting

- When Subclass type refers to the object of Parent class, it is known as downcasting.

  Syntax:

```
1. subclassname  subclassobject=(subclassname) superclassobject();
```

Downcasting example:

Eg: flute d=(flute)a

Downcasting can also be performed without the use of instanceof operator as displayed in the following example:

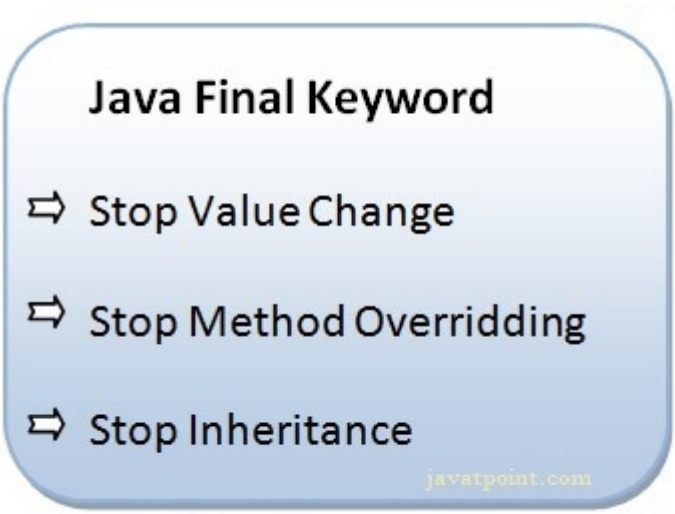
```
1.  class instrument
2.  {
3.  void guitar()
4.  {
5.          System.out.println("from instrument-guitar method");
6.   }
7.  }
8.  class flute extends instrument
9.  {
10.   void method()
11. {
12.       System.out.println("ok downcasting performed");
13.  }
14.   public static void main (String [] args) {
15.    instrument a=new flute();   //upcasting
16.   a. guitar();
17. flute f= (flute) a;   // downcasting
18. f.method();
19.  }
20. }
```

```
Output:ok downcasting performed
```

Final Keyword In Java

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. class



1) Java final variable

If you make any variable as final, you cannot change the value of final variable (It will be constant).

Example of final variable

There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

```
1.  class Bike9
2.  {
3.   final int speedlimit=90 ;  //final variable
4.   void run( )
5.  {
6.    speedlimit=400;
7.   }
8.  }
    class Main
    {
9.   public static void main(String args[])
10. {
11.  Bike9 obj=new  Bike9();
12.  obj.run();
13.  }
14. } //end of class
```

```
Output:Compile Time Error
```

2) Java final method

If you make any method as final, you cannot override it.

Example of final method

```
1. class Bike
2. {
3.   final void run( )
4. {
5. System.out.println("running");}
6. }
7. class Honda extends Bike
8. {
9.    void run( )
10. {
11. System.out.println("running safely with 100kmph");
12. }
13. }
14. class Vehicle
15. {
16.   public static void main(String args[])
17. {
18.   Honda honda= new Honda();
19.   honda.run();
20.   }
21. }
```

```
Output:Compile Time Error
```

3) Java final class

If you make any class as final, you cannot extend it.

Example of final class

```
1. final class Bike
2. {
```

```
3. void print()
4. {
5. System.out.println(“final class”);
6. }
7. }
8. class Honda1 extends Bike
9. {
10.  void run( )
11. {
12. System.out.println("running safely with 100kmph");
13. }
14. }
15. class vehicle
16. {
17.  public static void main(String args[])
18. {
19.  Honda1 honda= new Honda1();
20.  honda.run();
21.  }
22. }
Output:Compile Time Error
```

Q) Is final method inherited?

Ans) Yes, final method is inherited but you cannot override it. For Example:

```
1. class Bike
2. {
3.  final void run( )
4. {
5. System.out.println("running...");}
6. }
7. class Honda2 extends Bike
8. {
9.   public static void main(String args[])
10. {
11.   new Honda2().run();
12.  }
```

```
13. }
Output:running...
```

Java I/O (Input and Output) is used *to process the input* and *produce the output*.

Java uses the concept of a stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.

We can perform **file handling in Java** by Java I/O API.

Stream

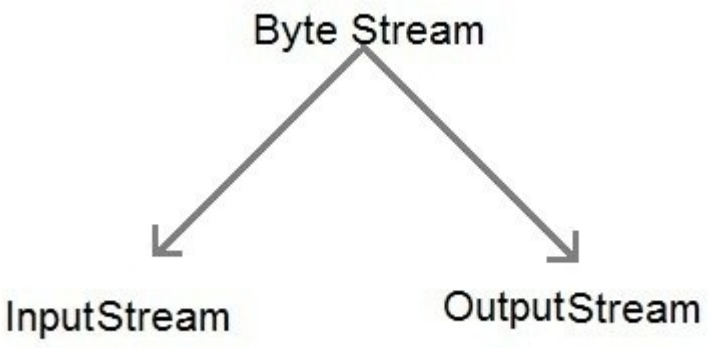
A stream is a sequence of data. In Java, a stream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow.

Java encapsulates Stream under **java.io** package. Java defines two types of streams. They are,

1. **Byte Stream :** It provides a convenient means for handling input and output of byte.
2. **Character Stream :** It provides a convenient means for handling input and output of characters.

Java Byte Stream Classes

Byte stream is defined by using two abstract class at the top of hierarchy, they are InputStream and OutputStream.



Stream class	Description
BufferedInputStream	Used for Buffered Input Stream.
BufferedOutputStream	Used for Buffered Output Stream.
DataInputStream	Contains method for reading java standard datatype
DataOutputStream	An output stream that contain method for writing java standard data type
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that write to a file.
InputStream	Abstract class that describe stream input.
OutputStream	Abstract class that describe stream output.

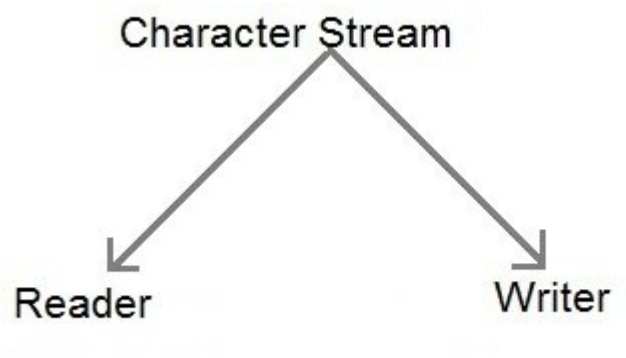
PrintStream	Output Stream that contain print() and println() method

These classes define several key methods. Two most important are

1. read() : reads byte of data.
2. write() : Writes byte of data.

Java Character Stream Classes

Character stream is also defined by using two abstract class at the top of hierarchy, they are Reader and Writer.



These two abstract classes have several concrete classes that handle unicode character.

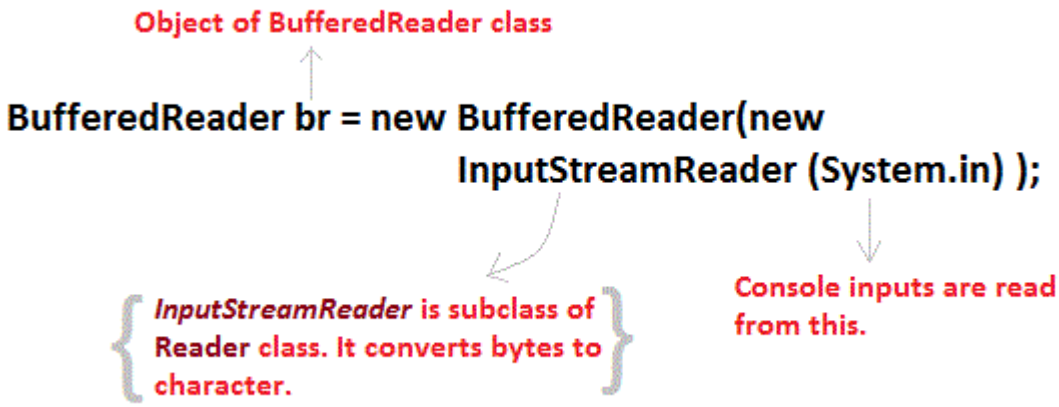
Some important Charcter stream classes

Stream class	Description

BufferedReader	Handles buffered input stream.
BufferedWriter	Handles buffered output stream.
FileReader	Input stream that reads from file.
FileWriter	Output stream that writes to file.
InputStreamReader	Input stream that translate byte to character
OutputStreamReader	Output stream that translate character to byte.
PrintWriter	Output Stream that contain `print()` and `println()` method.
Reader	Abstract class that define character stream input
Writer	Abstract class that define character stream output

Reading Console Input

We use the object of BufferedReader class to take inputs from the keyboard.



Reading Strings in Java

To read string we have to use readLine() function with BufferedReader class's object.

Program to take String input from Keyboard in Java

```
import java.io.*;

class MyInput

{

  public static void main(String[] args)

  {

    String text;

  BufferedReader br = new BufferedReader(new
```

```
                                InputStreamReader(System.in));
    text = br.readLine();    //Reading String
    System.out.println(text);
  }
}
```

Program to take integer/float input from Keyboard in Java

```
import java.io.*;
class MyInput
{
  public static void main(String[] args)
  {
    String text1,text2;
    BufferedReader br = new BufferedReader(new
                                InputStreamReader(System.in));
System.out.println("Enter any integer value");
    text1 = br.readLine();    //Reading String


int i= Integer.parseInt(text1);
System.out.println("integer is"+ i);
     System.out.println("Enter any float value");
```

```
    text2=br.readLine();

    float x=Float.valueOf(text2);

   System.out.println(“Float is”+ x);

 }
```

File OutputStream vs InputStream

The explanation of OutputStream and InputStream classes are given below:

OutputStream

Java application uses an output stream to write data to a destination; it may be a file, an array, peripheral device or socket.

InputStream

Java application uses an input stream to read data from a source; it may be a file, an array, peripheral device or socket.

Java FileOutputStream example :write string

```
1. import java.io.*;
2. public class FileOutputExample {
3.     public static void main(String args[]) {
4.           Try {
5.            FileOutputStream fout=new FileOutputStream("D:\\testout.txt");
6.            String s="Welcome to javastreamclass.";
7.            byte b[]=s.getBytes();//converting string into byte array
8.            fout.write(b);
9.            fout.close();
10.           System.out.println("success...");
11.          }
12. catch(Exception e)
13. {
14. System.out.println(e);
15. }
```

```
16.     }
17. }
```

Output:

```
Success...
```

The content of a text file **testout.txt** is set with the data **Welcome to javaTpoint.**

testout.txt

```
Welcome to javastreamclass.
```

Java FileInputStream Class

Java FileInputStream class obtains input bytes from a file. It is used for reading byte-oriented data (streams of raw bytes) such as image data, audio, video etc.

```
W
```

Java FileInputStream example : read all characters

```
1.  package com.javatpoint;
2.
3.  import java.io.FileInputStream;
4.  public class DataStreamExample {
5.      public static void main(String args[]){
6.          try{
7.            FileInputStream fin=new FileInputStream("D:\\testout.txt");
8.            int i=0;
9.            while((i=fin.read())!=-1)
10.       {
11.            System.out.print((char)i);
12.           }
13.           fin.close();
14.         }
15. catch(Exception e)
16. {
17. System.out.println(e);
18. }
19.         }
20.       }
```

Output:

```
Welcome to javastreamclass
```

Serialization in Java

Serialization in Java is a mechanism of *writing the state of an object into a byte-stream*. It is mainly used in RMI technology.

java.io.Serializable interface

The **Serializable** interface must be implemented by the class whose object needs to be persisted.

Student.java

```
1. import java.io.*;;
2. public class Student implements Serializable{
3.  int id;
4.  String name;
5.  public Student(int id, String name)
6.  {
7.   this.id = id;
8.   this.name = name;
9.  }
10. }
```

In the above example, ***Student*** class implements Serializable interface. Now its objects can be converted into stream. The main class implementation of is showed in the next code.

Example of Java Serialization

In this example, we are going to serialize the object of ***Student*** class from above code. The writeObject() method of ObjectOutputStream class provides the functionality to serialize the object. We are saving the state of the object in the file named f.txt.

Persist.java

```
1. import java.io.*;
2. class Persist
3. {
4.  public static void main(String args[])
5. {
6.   Try
7. {
8.   //Creating the object
9.   Student s1 =new Student(211,"harry");
10.  //Creating stream and writing the object
11.  FileOutputStream fout=new FileOutputStream("f.txt");
12.  ObjectOutputStream out=new ObjectOutputStream(fout);
13.  out.writeObject(s1);
14.  out.flush();
15.  //closing the stream
16.  out.close();
17.  System.out.println("success");
18.  }
19. catch(Exception e)
20. {
21. System.out.println(e);
22. }
23.  }
24. }
```

Generic Classes and Methods in Java

Generic Class

A **generic class** is a class that can work with different data types. It uses a **type parameter** (<T>) to specify the data type at the time of object creation.

Syntax

```
class ClassName<T> {
    T data;
```

```
    void setData(T data) {
        this.data = data;
    }

    T getData() {
        return data;
    }
}
```

Example of Generic Class

```
class Box<T> {
    T value;

    void setValue(T value) {
        this.value = value;
    }

    T getValue() {
        return value;
    }
}

public class Main {
    public static void main(String[] args) {
        Box<Integer> b1 = new Box<>();
        b1.setValue(100);

        Box<String> b2 = new Box<>();
        b2.setValue("Java");

        System.out.println(b1.getValue());
        System.out.println(b2.getValue());
    }
}
```

Output:

```
100
Java
```

Generic Method

A **generic method** is a method that can operate on different types of data independently of the class.

Syntax

```
<T> returnType methodName(T parameter) {
    // method body
}
```

Example of Generic Method

```
class Demo {
    public <T> void display(T value) {
        System.out.println(value);
    }
}

public class Main {
    public static void main(String[] args) {
        Demo d = new Demo();

        d.display(10);
        d.display("Java");
        d.display(5.5);
    }
}
```

Output:

```
10
Java
5.5
```

Advantages of Generics

1. **Type Safety** – Detects type errors at compile time.
2. **Code Reusability** – Same code can be used for multiple data types.
3. **No Type Casting Required** – Reduces explicit type conversions.
4. **Improved Readability** – Makes programs easier to understand and maintain.

Visibility in Java

Visibility refers to the accessibility of classes, variables, methods, and constructors from other parts of a program. Java provides **access modifiers** to control visibility.

Types of Visibility (Access Modifiers)

Modifier	Same Class	Same Package	Subclass	Other Packages
`Private`	✔️	✘	✘	✘
Default (no modifier)	✔️	✔️	✘	✘
`Protected`	✔️	✔️	✔️	✘*
`Public`	✔️	✔️	✔️	✔️

Example

```
class Student {
    private int id = 101;

    public void display() {
        System.out.println(id);
    }
}
```

Here, `id` is visible only within the `Student` class because it is declared as `private`.

Dependency in Java

Dependency is a relationship in which one class uses the services or objects of another class temporarily. Changes in the supplier class may affect the dependent class.

Characteristics

- Represents a **"uses-a"** relationship.
- One class depends on another to perform a task.
- Usually implemented through method parameters, local variables, or method calls.

Example

```
class Printer {
    void print() {
        System.out.println("Printing document");
    }
}

class Computer {
    void startPrint(Printer p) {
        p.print();
    }
}
```

In this example, `Computer` depends on `Printer` because it uses a `Printer` object in its method.

Difference Between Visibility and Dependency

Visibility	Dependency
Controls access to class members.	Represents a relationship between classes.
Implemented using access modifiers (`public`, `private`, etc.).	Implemented by one class using another class.
Focuses on security and accessibility.	Focuses on interaction and cooperation between classes.
Determines who can access data and methods.	Determines how classes rely on each other.

Definition

- **Visibility:** The accessibility level of class members controlled by access modifiers.
- **Dependency:** A relationship where one class temporarily uses another class to perform a specific operation.

Database Connectivity in Java (JDBC)

JDBC (Java Database Connectivity) is a standard Java API that enables Java applications to connect to relational databases such as MySQL, Oracle, PostgreSQL, SQL Server, and SQLite. It provides classes and interfaces to establish a connection, execute SQL statements, and process the results.

Steps to Connect a Database in Java

1. Import the JDBC package.
2. Load and register the JDBC driver.
3. Establish a connection to the database.
4. Create a `Statement` or `PreparedStatement` object.
5. Execute SQL queries.
6. Process the results (if any).
7. Close the database resources.

Prerequisites

Before running the program, ensure that:

- Java JDK is installed.
- MySQL Server is running.
- A database named `studentdb` exists.
- The **MySQL Connector/J (JDBC Driver)** JAR(java archive) is added to the project's classpath.
- Update the `username` and `password` in the program according to your MySQL configuration.

Steps to Connect and Execute Queries Using JDBC

1. Import JDBC Packages

The JDBC classes are available in the `java.sql` package. This package must be imported into the Java program.

```
import java.sql.*;
```

2. Load and Register the JDBC Driver

The JDBC driver acts as a bridge between the Java application and the database.

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

3. Establish Database Connection

The `DriverManager` class is used to establish a connection with the database.

```
Connection con = DriverManager.getConnection(url, username, password);
```

4. Create a Statement Object

A `Statement` object is used to execute SQL queries.

```
Statement stmt = con.createStatement();
```

5. Execute SQL Queries

SQL queries can be executed using methods such as:

- `executeUpdate()` → Used for INSERT, UPDATE, DELETE operations.
- `executeQuery()` → Used for SELECT operations.

6. Close the Connection

After completing database operations, the connection should be closed.

```
con.close();
```

Java Program to Connect and Execute Queries

Example: Create a Table and Insert Records Using JDBC

```
import java.sql.*;

public class JDBCExample {

    public static void main(String args[]) {

String url = "jdbc:mysql://localhost:3306/studentdb";
        String username = "root";
        String password = "root";

        try {
            // Step 1: Load JDBC Driver

Class.forName("com.mysql.cj.jdbc.Driver");

            // Step 2: Establish Connection
Connection con = DriverManager.getConnection(url,
username, password);

System.out.println("Database ConnectedSuccessfully");
```

```
            // Step 3: Create Statement Object
            Statement stmt = con.createStatement();

            // Step 4: Create Table Query
String c =
            "CREATE TABLE Student(" +
            "RollNo INT, " +
            "Name VARCHAR(30), " +
            "Department VARCHAR(20))";

            stmt.executeUpdate(c );

            System.out.println("Table Created
Successfully");


            // Step 5: Insert Records
String insertQuery = "INSERT INTO Student
                      VALUES(101,'John','CSE')";

            stmt.executeUpdate(insertQuery);

            System.out.println("Record Inserted
Successfully");


            // Step 6: Retrieve Data
            String selectQuery = "SELECT * FROM
Student";

       ResultSet rs= stmt.executeQuery(selectQuery);


            System.out.println("\nStudent Details:");

            while(rs.next()) {
```

```
                System.out.println(
                rs.getInt(1) + " " +
                rs.getString(2) + " " +
                rs.getString(3));
            }


            // Step 7: Close Connection
            con.close();

            System.out.println("Connection Closed");

        }

        catch(Exception e) {

            System.out.println(e);
        }
    }
}
```

Output

```
Database Connected Successfully

Table Created Successfully

Record Inserted Successfully

Student Details:

101 John CSE

Connection Closed
```

Explanation of Important JDBC Classes

Class / Interface	Purpose
`DriverManager`	Manages database drivers and creates database connections
`Connection`	Represents the connection between Java application and database
`Statement`	Executes SQL statements
`ResultSet`	Stores the result returned from SELECT queries
`SQLException`	Handles database-related errors

JDBC Query Execution Methods

1. executeQuery()

- Used for retrieving data from the database.
- Mainly used with the **SELECT** statement.
- Returns a `ResultSet` object.

Example:

```
ResultSet rs = stmt.executeQuery("SELECT * FROM Student");
```

2. executeUpdate()

- Used for modifying database records.
- Used with **INSERT, UPDATE, DELETE, and CREATE TABLE** statements.
- Returns the number of rows affected.

Example:

```
stmt.executeUpdate("INSERT INTO Student VALUES(102,'Alex','IT')");
```