ਜਗਤ ਗੁਰੂ ਨਾਨਕ ਦੇਵ
ਪੰਜਾਬ ਸਟੇਟ ਓਪਨ ਯੂਨੀਵਰਸਿਟੀ
ਪਟਿਆਲਾ

JAGAT GURU NANAK DEV

PUNJAB STATE OPEN UNIVERSITY, PATIALA

(Established by Act No. 19 of 2019 of the Legislature of State of Punjab)

The Motto of the University

(SEWA)

SKILL ENHANCEMENT EMPLOYABILITY WISDOM ACCESSIBILITY

Bachelor of Computer Applications (BCA)
Course Name: Object Oriented Programming
Course Code: BCA-4-03T

ADDRESS: C/28, THE LOWER MALL, PATIALA-147001
WEBSITE: www.psou.ac.in

JAGAT GURU NANAK DEV
PUNJAB STATE OPEN UNIVERSITY PATIALA
(Established by Act No.19 of 2019 of Legislature of the State of Punjab)

PROGRAMME COORDINATOR :
Dr. Monika Pathak
Assistant Professor, School of Sciences and Emerging Technologies
Jagat Guru Nanak Dev Punjab State Open University, Patiala

PROGRAMME CO-COORDINATOR :
Dr. Gaurav Dhiman
Assistant Professor, School of Sciences and Emerging Technologies
Jagat Guru Nanak Dev Punjab State Open University, Patiala

COURSE COORDINATOR :
Dr. Karan Sukhija
Assistant Professor, School of Sciences and Emerging Technologies
JGND PSOU, Patiala

JAGAT GURU NANAK DEV
PUNJAB STATE OPEN UNIVERSITY PATIALA
(Established by Act No.19 of 2019 of Legislature of the State of Punjab)

PREFACE

Jagat Guru Nanak Dev Punjab State Open University, Patiala was established in Decembas 2019 by Act 19 of the Legislature of State of Punjab. It is the first and only Open Universit of the State, entrusted with the responsibility of making higher education accessible to all especially to those sections of society who do not have the means, time or opportunity to pursue regular education.

In keeping with the nature of an Open University, this University provides a flexible education system to suit every need. The time given to complete a programme is double the duration of a regular mode programme. Well-designed study material has been prepared in consultation with experts in their respective fields.

The University offers programmes which have been designed to provide relevant, skill-based and employability-enhancing education. The study material provided in this booklet is self instructional, with self-assessment exercises, and recommendations for further readings. The syllabus has been divided in sections, and provided as units for simplification.

The Learner Support Centres/Study Centres are located in the Government and Government aided colleges of Punjab, to enable students to make use of reading facilities, and for curriculum-based counselling and practicals. We, at the University, welcome you to be a part of this institution of knowledge.

Prof. G. S. Batra,
Dean Academic Affairs

BCA-4-03T: Object Oriented Programming

Total Marks: 100
External Marks: 70
Internal Marks: 30
Credits: 4
Pass Percentage:40%

INSTRUCTIONS FOR THE PAPER SETTER/EXAMINER

1. The syllabus prescribed should be strictly adhered to.
2. The question paper will consist of three sections: A, B, and C. Sections A and B will have four questions from the respective sections of the syllabus and will carry 10 marks each. The candidates will attempt two questions from each section.
3. Section C will have fifteen short answer questions covering the entire syllabus. Each question will carry 3 marks. Candidates will attempt any ten questions from this section.
4. The examiner shall give a clear instruction to the candidates to attempt questions only at one place and only once. Second or subsequent attempts, unless the earlier ones have been crossed out, shall not be evaluated.
5. The duration of each paper will be three hours.

INSTRUCTIONS FOR THE CANDIDATES

Candidates are required to attempt any two questions each from the sections A and B of the question paper and any ten short questions from Section C. They have to attempt questions only at one place and only once. Second or subsequent attempts, unless the earlier ones have been crossed out, shall not be evaluated.

Course: Object Oriented Programming	
Course Code: BCA-4-03T	
Course Outcomes (COs) After the completion of this course, the students will be able to:	
CO1	Develop understanding of writing object-oriented programs that combine functions and data.
CO2	Gain a thorough understanding of the core principles of OOP, including encapsulation, inheritance, and polymorphism.
CO3	Learn how to apply OOP concepts to solve programming problems, design software systems, and develop reusable code.
CO4	Understand how to create classes and objects in a programming language that supports OOP
CO5	Learn how to use inheritance to create hierarchies of classes and reuse code efficiently.

Detailed Contents:

Module	Module Name	Module Contents
Section-A		
Module I	**Introduction to OOP**	**Introduction to OOP:** • Basic concepts (objects, classes, inheritance, polymorphism, encapsulation) • Advantages of OOP over procedural programming **Classes and Objects:** • Declaring classes • Creating objects • Access specifiers (public, private, protected) • Constructors and destructors • Static members
Module II	**Inheritance and Polymorphism**	**Inheritance:** • Base and derived classes • Types of inheritance (single, multiple, multilevel, hierarchical) • Access control in inheritance **Polymorphism:** • Function overloading • Operator overloading • Virtual functions and runtime polymorphism • Abstract classes and pure virtual functions
Section-B		
Module III	**Encapsulation and Interfaces and Abstract Classes**	**Encapsulation:** • Data hiding • Accessor and mutator methods • Benefits of encapsulation **Interfaces and Abstract Classes:** • Declaring interfaces • Implementing interfaces • Abstract classes and methods
Module IV	**Exception Handling and File Handling**	**Exception Handling:** • Handling exceptions using try-catch blocks • Throwing exceptions • Custom exceptions **File Handling:** • Reading from and writing to files • File streams (File Input Stream, File Output Stream, etc.)

Reference Books:

- Herbert Schildt, “The Complete Reference C++”, Tata McGraw-Hill.
- Deiteland Deitel, “C++ How to Program”, Pearson Education.
- Robert Lafore, “Object Oriented Programming in C++”, Galgotia Publications.
- Bjarne Strautrup, “The C++ Programming Language”, Addition-Wesley Publication Co.
- Stanley B. Lippman, Josee Lajoie, “C++ Primer”, Pearson Education.
- E. Balagurusamy, “Object Oriented Programming with C++”, Tata McGraw-Hill

Section A

Module I

Introduction to OOP:

OOP stands for Object-Oriented Programming. Object-Oriented Programming & System (OOPS) concepts in Java helps reduce code complexity and enables the reusability of code. Programmers feel like working with real-life entities or objects. Object-oriented programming is a programming paradigm that brings together data and methods in a single entity called object. This promotes greater understanding as well as flexibility and maintenance of code over a long period of time. The basic concepts about oops in java are given below:

Basic Concepts of oops:

Objects & Classes:

Objects are the basic unit of OOPS representing real-life entities. They are invoked with the help of methods. These methods are declared within a class. Usually, a new keyword is used to create an object of a class in Java. Class is a predefined or user-defined template from which objects are created. It represents properties/methods that are common to all objects of the same class. It has several features, such as access modifiers, class names, interfaces, and class bodies.

Abstraction

Abstraction means showing only the relevant details to the end-user and hiding the irrelevant features that serve as a distraction. For example, during an ATM operation, we only answer a series of questions to process the transaction without any knowledge about what happens in the background between the bank and the ATM.

Example Program of Abstraction in Java

```
abstract class Bike
{
        Bike()
        {
                System.out.println("The Street Bob. ");
        }
                abstract void drive();
        void weight()
        {
                System.out.println("Light on its feet with a hefty : 630 lbs.");
        }
}
class HarleyDavidson extends Bike
```

```
{
        void drive()
        {
                System.out.println("Old-school yet relevant.");
        }
 }
public class Abstraction
{
        public static void main (String args[])
        {
                Bike obj = new HarleyDavidson();
                obj.drive();
                obj. weight();
        }
 }
```

Output:

The Street Bob.
Old-school yet relevant.
Light on its feet with a hefty: 630 lbs.

3. Encapsulation

Encapsulation is a means of binding data variables and methods together in a class. Only objects of the class can then be allowed to access these entities. This is known as data hiding and helps in the insulation of data.

Example Program of Encapsulation in Java

```
class Encapsulate
{
   private String Name;
   private int Height;
   private int Weight;
   public int getHeight()
   {
     return Height;
   }
   public String getName()
   {
     return Name;
   }
   public int getWeight()
   {
      return Weight;
   }

   public void setWeight( int newWeight)
   {
     Weight = newWeight;
   }
  public void setName(String newName)
```

```
   {
     Name = newName;
   }
   public void setHeight( int newHeight)
   {
     Height = newHeight;
   }
}
public class TestEncapsulation
{
   public static void main (String[] args)
   {
      Encapsulate obj = new Encapsulate ();
      obj.setName("Abi");
      obj.setWeight(70);
      obj.setHeight(178);
      System.out.println("My name: " + obj.getName());
      System.out.println("My height: " + obj.getWeight());
      System.out.println("My weight " + obj.getHeight());
   }
}
```

Output:

```
My name: Abi
My height: 70
My weight: 178
```

4. Inheritance – Single, Multilevel, Hierarchical, and Multiple

Inheritance is the process by which one class inherits the functions and properties of another class. The main function of inheritance is the reusability of code. Each subclass only has to define its features. The rest of the features can be derived directly from the parent class.

Single Inheritance – Refers to a parent-child relationship where a child class extends the parent class features. Class Y extends Class X.

Multilevel Inheritance – Refers to a parent-child relationship where a child class extends another child's class. Class Y extends Class X. Class Z extends Class Y.

Hierarchical Inheritance – This refers to a parent-child relationship where several child classes extend one class. Class Y extends Class X, and Class Z extends Class X.

Multiple Inheritance – Refers to a parent-child relationship where one child class is extending from two or more parent classes. JAVA does not support this inheritance.

Example Program of Inheritance in Java

```
class Animal
{
        void habit()
        {
                System.out.println("I am nocturnal!! ");
        }
}
```

```
class Mammal extends Animal
{
	void nature()
	{
		System.out.println("I hang upside down!! ");
	}
}
class Bat extends Mammal
{
	void hobby()
	{
		System.out.println("I fly !! ");
	}
}
public class Inheritance
{
	public static void main(String args[])
	{
		Bat b = new Bat();
		b.habit();
		b.nature();
		b.hobby();
	}
}
```

Output:

I am nocturnal!!
I hang upside down!!
I fly !!

5. Polymorphism – Static and Dynamic

It is an object-oriented approach that allows the developer to assign and perform several actions using a single function. For example, "+" can be used for addition as well as string concatenation. Static Polymorphism is based on Method Overloading, and Dynamic Polymorphism is based on Method Overriding.

Example Program of Static Polymorphism with Method Overloading
Method Overloading

```
class CubeArea
{
	double area(int x)
	{
		return 6 * x * x;
	}
}
class SphereArea
{
	double area(int x)
{
```

```
                return 4 * 3.14 * x * x;
        }
}
class CylinderArea
{
        double area(int x, int y)
        {
                 return x * y;
}
}
public class Overloading
{
        public static void main(String []args)
        {
                CubeArea ca = new CubeArea();
                SphereArea sa = new SphereArea();
                CylinderArea cia = new CylinderArea();
                System.out.println("Surface area of cube = "+ ca.area(1));
                System.out.println("Surface area of sphere= "+ sa.area(2));
                System.out.println("Surface area of cylinder= "+ cia.area(3,4));
         }
}
```

Output:

```
        Surface area of cube = 6.0
        Surface area of sphere= 50.24
        Surface area of cylinder= 12.0
```

Example Program of Dynamic Polymorphism with Method Overriding

```
class Shape
{
   void draw()
   {
      System.out.println("Your favorite shape");
   }
   void  numberOfSides()
   {
       System.out.println("side = 0");
   }
}
class Square extends Shape
{
   void draw()
   {
      System.out.println("SQUARE ");
   }
   void numberOfSides()
   {
      System.out.println("side = 4 ");
   }
}
```

```
class Pentagon extends Shape
{
   void draw()
   {
      System.out.println("PENTAGON ");
}
   void numberOfSides()
   {
      System.out.println("side= 5");
   }
}
class Hexagon extends Shape
{
   void draw()
{
      System.out.println("HEXAGON ");
 }
void numberOfSides()
{
      System.out.println("side = 6 ");
}
}
public class Overriding
{
         public static void main(String []args)
         {
         Square s = new Square();
         s.draw();
         s.numberOfSides();
         Pentagon p = new Pentagon();
         p.draw();
         p.numberOfSides();
         Hexagon h = new Hexagon();
         h.draw();
         h.numberOfSides();
    }
}
```

Output:

```
         SQUARE;
         side = 4;
         PENTAGON;
         side= 5
         HEXAGON
         side = 6
```

Advantages of OOP over procedural programming

Procedural programming is about writing procedures or methods that perform operations on the data, while object-oriented programming is about creating objects that contain both data and methods. Object-oriented programming has several advantages over procedural

programming:

- OOP is faster and easier to execute
- OOP provides a clear structure for the programs
- OOP helps to keep the Java code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
- OOP makes it possible to create full reusable applications with less code and shorter development time.

Classes and Objects:

Java is an object-oriented programming language. Everything in Java is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an object. The car has attributes, such as weight and color, and methods, such as drive and brake. A Class is like an object constructor, or a "blueprint" for creating objects.

Java Classes

A class in Java is a set of objects which shares common characteristics/ behavior and common properties/ attributes. It is a user-defined blueprint or prototype from which objects are created. For example, Student is a class while a particular student named Ravi is an object.

Properties of Java Classes

1. Class is not a real-world entity. It is just a template or blueprint or prototype from which objects are created.
2. Class does not occupy memory.
3. Class is a group of variables of different data types and a group of methods.
4. A Class in Java can contain:
    - Data member
    - Method
    - Constructor
    - Nested Class
    - Interface

Class Declaration in Java:

access_modifier **class** <*class_name*>

```
{
   data member;
   method;
   constructor;
   nested class;
```

```
   interface;
}
```

Example:

```
// Java Program for class example
class Student
{
        // data member (also instance variable)
        int id;
        // data member (also instance variable)
        String name;
        public static void main(String args[])
        {
                // creating an object of
                // Student
                Student s1 = new Student();
                System.out.println(s1.id);
                System.out.println(s1.name);
        }
}
```

Components of Java Classes

In general, class declarations can include these components, in order:

1. **Modifiers**: A class can be public or has default access (Refer this for details).
2. **Class keyword:** class keyword is used to create a class.
3. **Class name:** The name should begin with an initial letter (capitalized by convention).
4. **Superclass (if any):** The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
5. **Interfaces (if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
6. **Body:** The class body is surrounded by braces, {}.

Java Objects

An object in Java is a basic unit of Object-Oriented Programming and represents real-life entities. Objects are the instances of a class that are created to use the attributes and methods of a class. A typical Java program creates many objects, which as you know, interact by invoking methods. An object consists of:

1. **State**: It is represented by attributes of an object. It also reflects the properties of an object.
2. **Behavior**: It is represented by the methods of an object. It also reflects the response of an object with other objects.
3. **Identity**: It gives a unique name to an object and enables one object to interact with other objects.

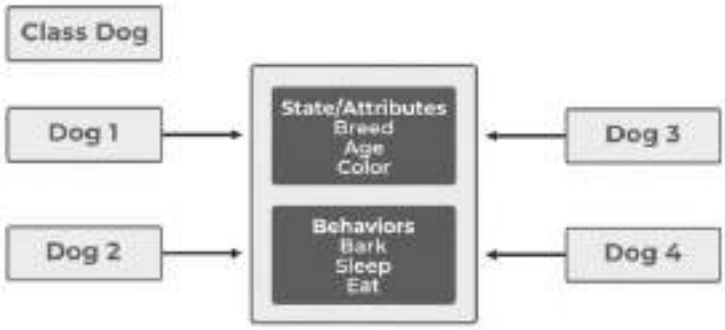
Example of an object: dog

Objects correspond to things found in the real world. For example, a graphics program may have objects such as “circle”, “square”, and “menu”. An online shopping system might have objects such as “shopping cart”, “customer”, and “product”.

Creating Objects

When an object of a class is created, the class is said to be **instantiated**. All the instances share the attributes and the behavior of the class. But the values of those attributes, i.e. the state are unique for each object. A single class may have any number of instances.

Example:



As we declare variables like (type name;). This notifies the compiler that we will use the name to refer to data whose type is type. With a primitive variable, this declaration also reserves the proper amount of memory for the variable. So for reference variables , the type must be strictly a concrete class name. In general, we **can’t** create objects of an abstract class or an interface.

Syntax: Dog tuffy;

If we declare a reference variable(tuffy) like this, its value will be undetermined(null) until an object is actually created and assigned to it. Simply declaring a reference variable does not create an object.

Initializing a Java object

The new operator instantiates a class by allocating memory for a new object and returning a reference to that memory. The new operator also invokes the class constructor.

Example:

```
// Class Declaration
```

```
public class Dog
{
   // Instance Variables
   String name;
   String breed;
   int age;
   String color;
   // Constructor Declaration of Class
   public Dog (String name, String breed, int age, String color)
   {
      this.name = name;
      this.breed = breed;
      this.age = age;
      this.color = color;
   }
   // method 1
   public String getName() { return name; }
   // method 2
   public String getBreed() { return breed; }
   // method 3
   public int getAge() { return age; }
   // method 4
   public String getColor() { return color; }
   @Override public String toString()
   {
      return ("Hi my name is " + this.getName()
            + ".\nMy breed,age and color are "
            + this.getBreed() + "," + this.getAge()
            + "," + this.getColor());
   }
   public static void main(String[] args)
   {
      Dog tuffy
         = new Dog("tuffy", "papillon", 5, "white");
      System.out.println(tuffy.toString());
   }
}
```

Output:

Hi my name is tuffy.
My breed,age and color are papillon,5,white

Ways to Create an Object of a Class

There are four ways to create objects in Java. Strictly speaking, there is only one way (by using a *new* keyword), and the rest internally use a *new* keyword.

i. Using new keyword

It is the most common and general way to create an object in Java.

Example:

```
// creating object of class Test
Test t = new Test();
```

ii. Using Class.forName (String className) method

There is a pre-defined class in java.lang package with name Class. The forName(String className) method returns the Class object associated with the class with the given string name. We have to give a fully qualified name for a class. On calling the new Instance() method on this Class object returns a new instance of the class with the given string name.

```
// creating object of public class Test
// consider class Test present in com.p1 package
Test obj = (Test)Class.forName("com.p1.Test").newInstance();
```

iii. Using clone() method

clone() method is present in the Object class. It creates and returns a copy of the object.

```
// creating object of class Test
Test t1 = new Test();
// creating clone of above object
Test t2 = (Test)t1.clone();
```

Access Specifier in Java

In Java, Access modifiers help to restrict the scope of a class, constructor, variable, method, or data member. It provides security, accessibility, etc to the user depending upon the access modifier used with the element. Let us learn about Java Access Modifiers, their types, and the uses of access modifiers in this article.

Types of Access Modifiers in Java

There are four types of access modifiers available in Java:

1. Default – No keyword required
2. Private
3. Protected
4. Public

1. Default Access Modifier

When no access modifier is specified for a class, method, or data member – It is said to be having the default access modifier by default. The data members, classes, or methods that are not declared using any access modifiers i.e. having default access modifiers are accessible only within the same package. In this example, we will create two packages and the classes in the packages will be having the default access modifiers and we will try to access a class from one package from a class of the second package.

```
// Java program to illustrate default modifier
package p1;
// Class Geek is having Default access modifier
```

```
class Geek
{
   void display()
   {
      System.out.println("Hello World!");
   }
}
```

2. Private Access Modifier

The private access modifier is specified using the keyword private. The methods or data members declared as private are accessible only within the class in which they are declared. Any other class of the same package will not be able to access these members. Top-level classes or interfaces cannot be declared as private because private means “only visible within the enclosing class”. protected means “only visible within the enclosing class and any subclasses”, Hence these modifiers in terms of application to classes, apply only to nested classes and not on top-level classes. In this example, we will create two classes A and B within the same package p1. We will declare a method in class A as private and try to access this method from class B and see the result.

```
// Java program to illustrate error while
// Using class from different package with
// Private Modifier
package p1;
// Class A
class A
{
        private void display()
        {
                System.out.println("GeeksforGeeks");
        }
}
// Class B
class B
{
        public static void main(String args[])
        {
                A obj = new A();
                // Trying to access private method of another class
                obj.display();
   }
}
```

Output:

```
error:  display() has private access in A
        obj.display();
```

3. Protected Access Modifier

The protected access modifier is specified using the keyword protected. The methods or data members declared as protected are accessible within the same package or subclasses in different packages. In this example, we will create two packages p1 and p2. Class A in p1 is made public, to access it in p2. The method display in class A is protected and class B is inherited from class A and this protected method is then accessed by creating an object of class B.

Example:

```
// Java Program to Illustrate
// Protected Modifier
package p1;
// Class A
public class A
{
   protected void display()
{
      System.out.println("GeeksforGeeks");
}
}
```

4 Public Access modifier:

The public access modifier is specified using the keyword public. The public access modifier has the widest scope among all other access modifiers. Classes, methods, or data members that are declared as public are accessible from everywhere in the program. There is no restriction on the scope of public data members.

Example:

```
// Java program to illustrate
// public modifier
package p1;
public class A
{
public void display()
   {
      System.out.println("GeeksforGeeks");
   }
}
```

Constructor and Destructor in Java

In Java, a constructor is a particular method that initializes an object when it is first formed. It guarantees that the item begins its trip with predetermined values and configurations. Consider it a blueprint for the object, outlining how it should be initialized. In Java, constructors have the same name as the class they belong to. They have no return type, not

even void, which distinguishes them from conventional methods. When you build an instance of a class using the 'new' keyword, the constructor is automatically invoked, setting the stage for the object to perform its duties.

Syntax:

```
public class Car
{
        // Constructor
        public Car()
        {
                // Initialization logic goes here
        }
}
```

Constructors exist in a variety of types; parameterized constructors enable you to supply data during object formation, whilst default constructors are used when there is no explicit constructor declared. Unlike several programming languages, Java does not have explicit destructors. Instead, it depends on the garbage collector to automatically reclaim memory used by things that are no longer in use. This technique, referred to as garbage collection, relieves the programmer of manual memory management duties.

Java doesn't offer a traditional destructor, developers can implement the finalize() method. However, it's essential to note that relying solely on finalize() for cleanup is not recommended due to its unpredictable nature.

```
public class Car
{
   // Finalize method for cleanup
   @Override
   protected void finalize() throws Throwable {
      // Cleanup logic goes here
      super.finalize();
   }
}
```

To summarise, constructors give life to Java objects by purposefully initialising them, whereas destructors, or their substitutes, provide a graceful farewell when an object's trip is over. The ability to generate and deconstruct objects is critical for developing strong and efficient Java programs.

Static members in Java

The static keyword in Java is mainly used for memory management. The static keyword in Java is used to share the same variable or method of a given class. The users can apply static keywords with variables, methods, blocks, and nested classes. The static keyword belongs to

the class than an instance of the class. The static keyword is used for a constant variable or a method that is the same for every instance of a class.

The static keyword is a non-access modifier in Java that is applicable for the following:

- Blocks
- Variables
- Methods
- Classes

Note: To create a static member (block, variable, method, nested class), precede its declaration with the keyword static.

Characteristics of static keyword:

Shared memory allocation: Static variables and methods are allocated memory space only once during the execution of the program. This memory space is shared among all instances of the class, which makes static members useful for maintaining global state or shared functionality.

Accessible without object instantiation: Static members can be accessed without the need to create an instance of the class. This makes them useful for providing utility functions and constants that can be used across the entire program.

Associated with class, not objects: Static members are associated with the class, not with individual objects. This means that changes to a static member are reflected in all instances of the class, and that you can access static members using the class name rather than an object reference. Cannot access non-static members: Static methods and variables cannot access non-static members of a class, as they are not associated with any particular instance of the class. Can be overloaded, but not overridden: Static methods can be overloaded, which means that you can define multiple methods with the same name but different parameters. However, they cannot be overridden, as they are associated with the class rather than with a particular instance of the class.

When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object. For example, in the below java program, we are accessing static method m1() without creating any object of the Test class.

```
// Java program to demonstrate that a static member
// can be accessed before instantiating a class
class Test
{
   // static method
   static void m1()
   {
      System.out.println("from m1");
```

```
    }
    public static void main(String[] args)
    {
        // calling m1 without creating
        // any object of class Test
         m1();
    }
}
```

Output:

from m1

Module II

Inheritance

Java, Inheritance is an important pillar of OOP(Object-Oriented Programming). It is the mechanism in Java by which one class is allowed to inherit the features (fields and methods) of another class. In Java, Inheritance means creating new classes based on existing ones. A class that inherits from another class can reuse the methods and fields of that class. In addition, you can add new fields and methods to your current class as well.

Why Do We Need Java Inheritance?

- **Code Reusability:** The code written in the Superclass is common to all subclasses. Child classes can directly use the parent class code.
- **Method Overriding:** Method Overriding is achievable only through Inheritance. It is one of the ways by which Java achieves Run Time Polymorphism.
- **Abstraction:** The concept of abstract where we do not have to provide all details is achieved through inheritance. Abstraction only shows the functionality to the user.

Base and Derive Class in Java Inheritance

- **Class:** Class is a set of objects which shares common characteristics/ behavior and common properties/ attributes. Class is not a real-world entity. It is just a template or blueprint or prototype from which objects are created.
- **Super Class/Parent Class:** The class whose features are inherited is known as a superclass (or a base class or a parent class).
- **Sub Class/Child Class:** The class that inherits the other class is known as a subclass (or a derived class, extended class, or child class). The

subclass can add its own fields and methods in addition to the superclass fields and methods.

- **Reusability:** Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

How to Use Inheritance in Java?

The **extends keyword** is used for inheritance in Java. Using the extends keyword indicates you are derived from an existing class. In other words, “extends” refers to increased functionality.

Syntax:

```
class DerivedClass extends BaseClass
{
  //methods and fields
}
```

Inheritance is one of the key features of OOP that allows us to create a new class from an existing class. The new class that is created is known as subclass (child or derived class) and the existing class from where the child class is derived is known as superclass (parent or base class). The extends keyword is used to perform inheritance in Java.

For example:

```
class Animal
{
        // methods and fields
}

        // use of extends keyword
        // to perform inheritance
```

```
class Dog extends Animal
{
        // methods and fields of Animal
        // methods and fields of Dog
}
```

In the above example, the Dog class is created by inheriting the methods and fields from the Animal class. Here, Dog is the subclass and Animal is the superclass.

Example 1: Java Inheritance

```
class Animal
{
        // field and method of the parent class
        String name;
        public void eat()
        {
                System.out.println("I can eat");
        }
}

// inherit from Animal
class Dog extends Animal
{
        // new method in subclass
        public void display()
        {
                System.out.println("My name is " + name);
        }
}

class Main
{
        public static void main(String[] args)
        {
                // create an object of the subclass
                Dog labrador = new Dog();
                // access field of superclass
                labrador.name = "Rohu";
                labrador.display();
```

```
        // call method of superclass
        // using object of subclass
        labrador.eat();
    }
}
```

Output:

```
My name is Rohu
I can eat
```

Types of Inheritance in Java

Have you ever wondered about the ways in which Java allows for code reusability and hierarchical organization through inheritance? Java supports several types of inheritance: single inheritance through class extension, multilevel inheritance to create a chain of class relationships, hierarchical inheritance for multiple classes to share a single superclass, and multiple inheritance through interfaces for a class to adopt methods from multiple sources. Hybrid inheritance combines these approaches, offering flexibility in complex software design. The following sections elaborate the types of reusability support by java.

- Single-level inheritance
- Multi-level Inheritance
- Hierarchical Inheritance
- Multiple Inheritance
- Hybrid Inheritance

NOTE: Java does not support Multiple Inheritance and Hybrid Inheritance directly through classes due to its language design, it facilitates aspects of these inheritance types using interfaces and other mechanisms.

Single Inheritance

In this type of java inheritance, the class inherits the properties of some other class. It allows derived classes to take properties and behavior from a single-parent class. In turn, this will make it possible to reuse current code and give it new functionalities. Here, Class A serves as the parent class, while Class B, the child class, inherits the traits and characteristics of the

parent class. The following code shows a comparable idea:

```
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();
	}
}
```

Multi-Level Inheritance

Multi-level type of java inheritance comes with a chain of inheritance. This indicates that we feature a parent class which a derived class inherits. The derived class then serves as the parent to the next class, and so forth. There is a dog class descended from the Animal class, sticking with the Animal class example from below. Another great option is the puppy class – a young dog descended from the Dog class. This way, you can possess a tiered inheritance.

```
class Animal
{
	void eat()
	{
		System.out.println(“eating…”);
	}
}
class Dog extends Animal
{
	void bark()
	{
		System.out.println(“barking…”);
	}
}
class Puppy extends Dog
{
	void weep()
	{
		System.out.println(“weeping…”);
	}
}
class TestInheritance2
{
	public static void main(String args[])
	{
		Puppy d=new Puppy();
		d.weep();
		d.bark();
		d.eat();
	}
}
```

Multiple Inheritances

The concept of inheritance, which enables classes to adopt features and attributes from other classes, is fundamental to object-oriented programming. Due to Java's support for single inheritance, a class can only descend from one superclass. However, Java offers a method for achieving multiple inheritances through interfaces, enabling a class to implement many interfaces. We will examine the idea of multiple inheritance in Java, how it is implemented using interfaces, and use examples to help us understand.

Understanding Multiple Inheritance A class's capacity to inherit traits from several classes is referred to as multiple inheritances. This notion may be quite helpful when a class needs features from many sources. Multiple inheritances, however, can result in issues like the diamond problem, which occurs when two superclasses share the same method or field and causes conflicts. Java uses interfaces to implement multiple inheritances in order to prevent these conflicts.

Java interfaces

A Java interface is a group of abstract methods that specify the behavior that implementing classes must follow. It serves as a class blueprint by outlining each class's methods. Interfaces offer a degree of abstraction for specifying behaviors but cannot be instantiated like classes. In Java, a class can successfully implement several interfaces to achieve multiple inheritance.

Syntax of implementing multiple interfaces:

```
class MyClass implements Interface1, Interface2, Interface3
{
   // class body
}
```

The classes "MyClass" and "Interface1", "Interface2", and "Interface3" can now inherit and implement methods from other interfaces. As a result, the class is able to display the behaviours specified in every interface it implements.

Example - 1

Let's look at an example situation to demonstrate multiple inheritance using Java interfaces. Imagine that you and I are creating a game with a variety of characters, such as warriors and magicians. We also carry a variety of weaponry, including swords and wands. Although we want to keep character kinds and weapon types apart, we also want our characters to be able to utilize weapons. The following is how multiple inheritances through interfaces may help us

do this:

```
interface Character
{
   void attack();
}
interface Weapon
{
   void use();
}
class Warrior implements Character, Weapon
{
        public void attack()
        {
                System.out.println("Warrior attacks with a sword.");
        }
        public void use()
        {
                System.out.println("Warrior uses a sword.");
        }
}
class Mage implements Character, Weapon
{
        public void attack()
        {
                System.out.println("Mage attacks with a wand.");
        }
        public void use()
        {
                System.out.println("Mage uses a wand.");
        }
}
public class MultipleInheritance
{
```

```
    public static void main(String[] args)
    {
        Warrior warrior = new Warrior();
        Mage mage = new Mage();
        warrior.attack(); // Output: Warrior attacks with a sword.
        warrior.use(); // Output: Warrior uses a sword.
        mage.attack(); // Output: Mage attacks with a wand.
        mage.use(); // Output: Mage uses a wand.
    }
}
```

Output:

```
Warrior attacks with a sword.
Warrior uses a sword.
Mage attacks with a wand.
Mage uses a wand.
```

Explanation: The interfaces "Character" and "Weapon" in the example above specify the behaviour that classes that implement them must have. As a result of the classes "Warrior" and "Mage" implementing both interfaces, the necessary behaviors may be inherited and shown. The main method shows how to instantiate these classes' objects and call their corresponding behaviors.

Hierarchical Inheritance

This type of java inheritance is where many subclasses inherit from one single class. Basically it is a combination of more than one type of java inheritance. When a class contains several child classes or subclasses, or, to put it another way, when multiple child classes share the same parent class, this type of inheritance is referred to as hierarchical.

```
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();
	}
}
```

Hybrid Inheritance

Hybrid type of java inheritance is a combination of more than two types of java inheritances single and multiple. A hybrid inheritance combines a single or more of the inheritance types we've covered so far. Any combination, though, leads to a form of multiple inheritances that Java does not support.

Hybrid Inheritance Example:

Class A and B extends class C → Hierarchical inheritance

Class D extends class A → Single inheritance

```
class C
{
  public void disp()
  {
        System.out.println("C");
  }
}

class A extends C
{
  public void disp()
  {
        System.out.println("A");
  }
}

class B extends C
{
  public void disp()
  {
        System.out.println("B");
  }

}

class D extends A
{
  public void disp()
  {
        System.out.println("D");
  }
  public static void main(String args[]){
```

```
        D obj = new D();
        obj.disp();
    }
}
```

Output: D

Access control in inheritance

Java provides a rich set of modifiers. They are used to control access mechanisms and also provide information about class functionalities to JVM. They are divided into two categories namely:

- **Access modifiers**
- **Non-access modifiers**

Java's access modifiers are public, private, and protected. Java also defines a default access level (called package-private).

- public: When a member of a class is modified by public, then that member can be accessed by any other code.
- private: When a member of a class is specified as private, then that member can only be accessed by other members of its class.
- default: It is also referred to as no modifier. Whenever we do not use any access modifier it is treated as default where this allows us to access within a class, within a subclass, and also non-sun class within a package but when the package differs now be it a subclass or non-class we are not able to access.
- protected: With the above default keyword we were facing an issue as we are getting closer to the real world with the above default modifier but there was a constriction as we are not able to access class sub-class from a different package. So protected access modifier allows not only to access class be it subclass or non-sub class but allows us to access subclass of the different package which brings us very close to a real-world and hence strong understanding of inheritance is required for understanding and implementing this keyword.

The following table elaborate the details about access modifiers provided.

	default	private	protected	public
same class	yes	yes	yes	yes
same package subclass	yes	no	yes	yes
same package non-subclass	yes	no	yes	yes
different package subclass	no	no	yes	yes
different package non-subclass	no	no	no	yes

Note: Now you can understand why main( ) has always been preceded by the public modifier. It is called by code that is outside the program—that is, by the Java run-time system. When no access modifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package. protected applies only when inheritance is involved.

Polymorphism

Polymorphism is derived from two Greek words, "poly" and "morph", which mean "many" and "forms", respectively. Hence, polymorphism meaning in Java refers to the ability of objects to take on many forms. In other words, it allows different objects to respond to the same message or method call in multiple ways.

Polymorphism in Java Example

As previously explained, polymorphism in Java helps an object take on many different forms. In this section, we will provide different examples of polymorphism to show how it works. The Animal class has a makeSound() method that outputs "Animal making a sound…" while the subclasses Dog, Cat, and Elephant, each provide their own implementation of the same function to produce individual noises.

```
class Animal
{
   void makeSound()
{
      System.out.println("Animal making a sound...");
}
}
class Dog extends Animal
{
```

```
    void makeSound() {
        System.out.println("Dog barking...");
    }
}
class Cat extends Animal
{
        void makeSound()
        {
                System.out.println("Cat meowing...");
        }
}
class Elephant extends Animal
{
        void makeSound()
        {
                System.out.println("Elephant trumpeting...");
        }
}
class TestPolymorphism2
{
    public static void main(String args[])
{
        Animal animal;
        animal = new Dog();
        animal.makeSound();
        animal = new Cat();
        animal.makeSound();
        animal = new Elephant();
        animal.makeSound();
    }
}
```

Output:

Dog barking…

Cat meowing…

Elephant trumpeting…

Function Overloading in Java

Function Overloading in Java occurs when there are functions having the same name but have different numbers of parameters passed to it, which can be different in data like int, double, float and used to return different values are computed inside the respective overloaded method. Function overloading is used to reduce complexity and increase the efficiency of the program by involving more functions that are segregated and can be used to distinguish among each other with respect to their individual functionality. Overloaded functions are related to compile-time or static polymorphism. There is also a concept of type conversion, which is basically used in overloaded functions used to calculate the conversion of type in variables.

Overloaded functions have the same name but different types of arguments or parameters assigned to them. They can be used to calculate mathematical or logical operations within the number of assigned variables in the method. The syntax of the overloaded function can be given below, where there are up to N number of variables assigned.

Syntax:

```
public class OverloadedMethod
{
	public int FunctionName(int x, int y) //Two parameters in the function
	{
		return (x + y); //Returns the sum of the two numbers
	}
// This function takes three integer parameters
	public int FunctionName(int x, int y, int z)
	{
		return (x + y + z);
	}
// This function takes two double parameters
	public double FunctionName(double x, double y)
	{
		return (x + y);
	}
//Many more such methods can be done with different number of parameters
// Code used to input the number and
	public static void main(String args[])
	{
		FunctionName s = new FunctionName();
		System.out.println(s.FunctionName(10, 20));
		System.out.println(s. FunctionName(10, 20, 30));
		System.out.println(s. FunctionName(10.5, 20.5));
	}
}
```

Explanation: Function overloading works by calling different functions having the same name, but the different number of arguments passed to it. There are many coding examples that can be shown in order to identify the benefits and disadvantages of function overloading properly.

Operator overloading in Java

Operator overloading aims to redefine an operator that has been defined and has certain functions to complete more detailed and specific operations and other functions. From an object-oriented perspective, it means an operator can be defined as a method of a class, so the function of the operator can be used to represent a certain behaviour of the object.

There are at least two benefits to being able to perform operator overloading for numeric operations of non-primitive types.

1. The code is simpler to write and less error-prone.
2. The code is easier to read without many parentheses.

How to Implement Operator Overloading in Java

The implementation of operator overloading in Java still uses Manifold. Manifold allows you to overload Java operators in various scenarios, such as arithmetic operators (including +,-, *, /, and %), comparison operators (>, >=, <, <=, ==, and !=), and index operators ([]). Please see Java's Missing Feature: Extension Methods for more information about the integration of Manifold.

Arithmetic Operator

Manifold is a function that maps each overload of an arithmetic operator to a specific name. For example, if you define a plus(B) method in class A, that class can be called using a + b instead of a.plus(b). The following chart describes the mappings:

Operator	Method Call
c = a + b	c = a.plus(b)
c = a – b	c = a.minus(b)
c = a * b	c = a.times(b)
c = a / b	c = a.div(b)
c = a % b	c = a.rem(b)

Those familiar with Kotlin should know that this is an imitation of Kotlin's operator overloading.

Let's define a numeric Num to facilitate illustration.

```
public class Num
{

	private final int v;
	public Num(int v)
	{
		this.v = v;
	}

	public Num plus(Num that)
	{
		return new Num(this.v + that.v);
	}

	public Num minus(Num that)
	{
		return new Num(this.v - that.v);
	}

	public Num times(Num that)
	{
		return new Num(this.v * that.v);
	}
}
```

For the following code:

```
Num a = new Num(1);
Num b = new Num(2);
Num c = a + b - a;
```

Virtual functions and runtime polymorphism

A member function that has the keyword virtual used in its declaration in the base class and is redefined (Overridden) in the derived class is referred to as a virtual function. The late binding instruction instructs the compiler to execute the called function during runtime by matching the object with the appropriately called function. Runtime Polymorphism refers to this method.

1. No matter what kind of reference (or pointer) is used to invoke a function, virtual functions make sure the right function is called for an object.
2. Their primary purpose is to implement runtime polymorphism.
3. In base classes, functions are declared using the virtual keyword.
4. Runtime resolution of function calls is carried out.

Polymorphism is a term used to describe the capacity to assume several shapes. If there is a hierarchy of classes connected to one another by inheritance, it happens. Polymorphism, which is defined as "showing diverse traits in different contexts," can be summarised as "showing different characteristics in a variety of situations" and "polymorphism."

What is the use of virtual functions?

To achieve Runtime Polymorphism, virtual functions are primarily used. Only a base class type pointer (or reference) can enable runtime polymorphism. A base class pointer can also point to both objects from the base class and those from derived classes.

Also, without even knowing the type of derived class object, we can use virtual functions to compile a list of base class pointers and call any of the derived classes' methods.

```
#include<iostream>
using namespace std;
class B
{
public:
        virtual void s()
        {
                cout<<" In Base \n";
        }
};
class D: public B
{
        public:
                void s()
                {
                        cout<<"In Derived \n";
                }
};
int main(void)
{
        D d; // An object of class D
        B *b= &d; // A pointer of type B* pointing to d
        b->s(); // prints "D::s() called"
        return 0;
}
```

Output: In Derived

What are the rules for virtual functions?

I. Virtual functions are not permitted to be static or friendly to other classes.

II. Pointers or references of base class type are required to access virtual functions.

III. Both the base class and any derived classes should use the same function prototype.

IV. There cannot be a virtual constructor in a class. However, it might have a virtual

destroyer.

v. The base class always defines them, and the derived class redefines them.

What is runtime polymorphism?

Runtime polymorphism is the process of binding an object at runtime with a capability. Overriding methods is one way to implement runtime polymorphism. At runtime, not at compilation time, the Java virtual machine decides which method to invoke. Additionally known as dynamic binding or late binding. The parent class's method is overridden in the child class, according to this concept. The term "method overriding" refers to the situation where a child class implements a method specifically that was supplied by one of its parent classes. You can see runtime polymorphism in the example that follows.

Example

```
class Test
{
	public void method()
	{
		System.out.println("Method 1");
	}
}
public class DEMO extends Test
{
	public void method()
	{
		System.out.println("Method 2");
	}
	public static void main(String args[])
	{
		Test test = new DEMO();
		test.method();
	}
}
```

Output: Method 2

What are the limitations of virtual functions?

Slower: The virtual mechanism causes the function call to take a little longer, making it harder for the compiler to optimize as it is unsure which function will be called at compilation time. Virtual functions can make it slightly more challenging to determine where a function is being called from in complicated systems, which makes them more challenging to debug.

Abstract Class in Java

In Java, abstract class is declared with the abstract keyword. It may have both abstract and non-abstract methods (methods with bodies). An abstract is a Java modifier applicable for classes and methods in Java but not for Variables. In this article, we will learn the use of abstract classes in Java. Furthermore, Java abstract class is a class that cannot be initiated by itself, it needs to be subclassed by another class to use its properties. An abstract class is declared using the "abstract" keyword in its class definition.

Illustration of Abstract class

```
abstract class Shape
{
   int color;
   // An abstract function
   abstract void draw();
}
```

In Java, the following some important observations about abstract classes are as follows:

1. An instance of an abstract class can not be created.
2. Constructors are allowed.
3. We can have an abstract class without any abstract method.
4. There can be a final method in abstract class but any abstract method in class(abstract class) can not be declared as final or in simpler terms final method can not be abstract itself as it will yield an error: "Illegal combination of modifiers: abstract and final".
5. We can define static methods in an abstract class.
6. We can use the abstract keyword for declaring top-level classes (Outer class) as well as inner classes as abstract.
7. If a class contains at least one abstract method then compulsory should declare a class as abstract.
8. If the Child class is unable to provide implementation to all abstract methods of the Parent class then we should declare that Child class as abstract so that the next level Child class should provide implementation to the remaining abstract method.

Example of Java Abstract Class

```
// Abstract class
abstract class Sunstar
{
    abstract void printInfo();
}

// Abstraction performed using extends
class Employee extends Sunstar {
    void printInfo()
    {
        String name = "avinash";
        int age = 21;
        float salary = 222.2F;

        System.out.println(name);
        System.out.println(age);
        System.out.println(salary);
    }
}

// Base class
class Base {
    public static void main(String args[])
    {
        Sunstar s = new Employee();
        s.printInfo();
    }
}
```

Output:

```
avinash
21
222.2
```

Pure Virtual Function

Pure virtual function is a virtual function for which we don't have implementations. An abstract method in Java can be considered as a pure virtual function. Let's take an example to understand this better.

Example of Pure Virtual Function:

```
abstract class Dog
{
```

```
        final void bark()
        {
                System.out.println("woof");
        }
                abstract void jump(); //this is a pure virtual function
}
class MyDog extends Dog
{
        void jump()
        {
                System.out.println("Jumps in the air");
        }
}
public class Runner
{
        public static void main(String args[])
        {
                Dog ob1 = new MyDog();
                ob1.jump();
        }
}
```

Output: Jumps in the air

This is how virtual function can be used with abstract class.

Run-Time Polymorphism

Run-time polymorphism is when a call to an overridden method is resolved at run-time instead of compile-time. The overridden method is called through the reference variable of the base class.

Output: Java Certification Course

```
class Edureka
{
	public void show()
	{
		System.out.println("welcome to edureka");
	}
}
class Course extends Edureka
{
	public void show()
	{
		System.out.println("Java Certification Program");
	}
public static void main(String args[])
{
	Edureka ob1 = new Course();
	ob1.show();
}
}
```

Points To Remember

- For a virtual function in Java, you do not need an explicit declaration. It is any function that we have in a base class and redefined in the derived class with the same name.
- The base class pointer can be used to refer to the object of the derived class.
- During the execution of the program, the base class pointer is used to call the derived class functions.

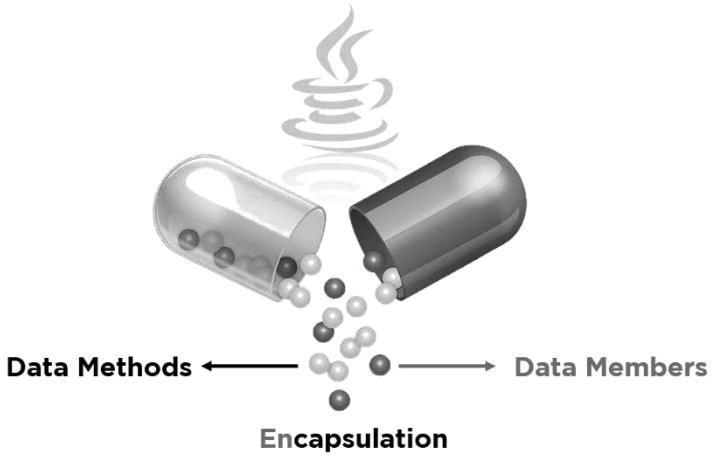
This brings us to the end of this article where we have learned about the Virtual Function In Java. I hope you are clear with all that has been shared with you in this tutorial.

Section B

Module III

Encapsulation

Encapsulation is a powerful mechanism for storing the data members and data methods of a class together. It is done in the form of a secure field accessible by only the members of the same class. Encapsulation in Java refers to integrating data (variables) and code (methods) into a single unit. In encapsulation, a class's variables are hidden from other classes and can only be accessed by the methods of the class in which they are found.



Source: simplilearn

Encapsulation in Java is an object-oriented procedure of combining the data members and data methods of the class inside the user-defined class. It is important to declare this class as private. It refers to the bundling of data and methods that operate on the data within a single unit, typically a class. This concept helps in hiding the internal state of an object and only exposing necessary functionalities through methods. By encapsulating data, Java ensures better data security and code maintainability. Understanding encapsulation is fundamental for building robust and organized Java applications, making a Java Course essential for mastering this concept. Next, we will understand the Syntax to be followed while implementing encapsulation in Java.

Syntax:

```
<Access_Modifier> class <Class_Name>
{
 private <Data_Members>;
 private <Data_Methods>;
}
```

For enhancing the understanding of the encapsulation process, let us go through the following sample program.

Example:

```
package dc;
public class c
{
	public static void main (String[] args)
	{
		Employee e = new Employee();
		e.setName("Robert");
		e.setAge(33);
		e.setEmpID(1253);
		System.out.println("Employee's name: " + e.getName());
		System.out.println("Employee's age: " + e.getAge());
		System.out.println("Employee's ID: " + e.getEmpID());
	}
}
package dc;

public class Employee
{
private String Name;
private int EmpID;
private int Age;
public int getAge()
```

```
{
        return Age;
}
public String getName()
{
        return Name;
}
public int getEmpID()
{
        return EmpID;
}
public void setAge(int newAge)
{
        Age = newAge;
}
public void setName(String newName)
{
        Name = newName;
}
public void setRoll(int newEmpID)
{
        EmpID = newEmpID;
}
public void setEmpID(int EmpID)
{
}
}
```

Output:

Employee's name: Robert

Employee's age: 33

Employee's ID: 1253

Data Hiding in Java

Data hiding is a procedure done to avoid access to the data members and data methods and their logical implementation. Data hiding can be done by using the access specifiers. We have four access specifiers, which are as follows.

Default

Default is the first line of data hiding. If any class in Java is not mentioned with an access specifier, then the compiler will set ‘default’ as the access specifier. The access specifications of default are extremely similar to that of the public access specifier.

Public

The public access specifier provides the access specifications to a class so that it can be accessed from anywhere within the program.

Example:

```
package Simplilearn;
class vehicle
{
public int tires;
public void display()
{
System.out.println("I have a vehicle.");
System.out.println("It has " + tires + " tires.");
}
}
```

```
public class Display
{
public static void main(String[] args)
{
vehicle veh = new vehicle();
veh.tires = 4;
veh.display();
}
}
```

//Output:

I have a vehicle.

It has four tires.

Private

The private access specifier provides access to the data members, and the data methods limit to the class itself.

Example:

```
package Simplilearn;
class Student
{
	private int rank;
	public int getRank()
	{
		return rank;
	}
	public void setRank(int rank)
	{
		this.rank = rank;
```

```
	}
}
public class school
{
	public static void main(String[] args)
	{
		Student s = new Student();
		s.setRank(1022);
		System.out.println("Student rank is " + s.getRank());
	}
}
```

//**Output**:

Student rank is 1022

Protected

The protected access specifier protects the class methods and members similar to the private access specifier. The main difference is that the access is limited to the entire package, unlike only a class with the private access specifier.

Example:

```
package Simplilearn;
class human
{
	protected String stream;
	protected void display()
	{
		System.out.println("Hello, I am a " + stream + " Student");
	}
}

public class Student extends human
{
```

```
    public static void main(String[] args)
    {
        Student s = new Student();
        s.stream = "Computer Science and Engineering Technology";
        s.display();
    }
}
```

//Output:
Hello, I am a Computer Science and Engineering Technology Student

Accessor and Mutator methods in Java

In object-oriented programming, encapsulation is a fundamental concept that refers to the practice of hiding the implementation details of an object and providing an interface to access its properties and behaviors. Accessor and mutator methods are two important concepts related to encapsulation in Java.

Accessor
Accessor methods, also known as getter methods, are methods that allow you to retrieve the value of an object's private instance variables. These methods provide read-only access to the object's state. By using accessor methods, you can ensure that the object's state is not modified accidentally or maliciously by external code.

Mutator
Mutator methods, also known as setter methods, are methods that allow you to modify the value of an object's private instance variables. These methods provide write-only access to the object's state. By using mutator methods, you can ensure that the object's state is modified only through a controlled interface.

Let's take a look at an example to understand the concept of accessor and mutator methods in

Java. Suppose we have a class called Person that has three private instance variables: name, age, and email. We want to provide access to these variables using accessor and mutator methods.

```
public class Person
{
	private String name;
	private int age;
	private String email;

	public String getName()
	{
		return name;
	}
	public void setName(String name)
	{
		this.name = name;
	}
	public int getAge()
	{
		return age;
	}
	public void setAge(int age)
	{
		this.age = age;
	}
	public String getEmail()
	{
		return email;
	}
	public void setEmail(String email)
	{
		this.email = email;
	}
```

}

In this example, we have defined three accessor methods: getName(), getAge(), and getEmail(), and three mutator methods: setName(), setAge(), and setEmail(). The accessor methods return the value of the corresponding instance variable, while the mutator methods set the value of the corresponding instance variable.

Naming Convention

The naming convention for accessor and mutator methods is important in Java. Accessor methods should be named starting with "get" followed by the name of the variable, with the first letter capitalized. Mutator methods should be named starting with "set" followed by the name of the variable, with the first letter capitalized. This naming convention makes it clear what each method does and makes the code more readable and maintainable.

Accessor and mutator methods are an essential part of encapsulation in Java. They allow you to control access to an object's state, ensuring that it is accessed and modified only through a controlled interface. By following the naming convention for accessor and mutator methods, we can make your code more readable and maintainable.

Example:

Person.java

```
import java.util.Scanner;
public class Person
{
        private String name;
        private int age;
        private String email;
            public String getName()
            {
                    return name;
            }
            public void setName(String name)
```

```
{
        this.name = name;
}
public int getAge()
{
        return age;
}
public void setAge(int age)
{
        this.age = age;
}
public String getEmail()
{
        return email;
}
public void setEmail(String email)
{
        this.email = email;
}
public static void main(String[] args)
{
        Scanner scanner = new Scanner(System.in);
        Person person = new Person();
        System.out.print("Enter name: ");
        String name = scanner.nextLine();
        person.setName(name);
        System.out.print("Enter age: ");
        int age = scanner.nextInt();
        person.setAge(age);
        scanner.nextLine(); // Consume the newline character left by nextInt()
        System.out.print("Enter email: ");
        String email = scanner.nextLine();
        person.setEmail(email);
        System.out.println("\nName: " + person.getName());
```

```
                System.out.println("Age: " + person.getAge());
                System.out.println("Email: " + person.getEmail());
    }
}
```

Output:

Name: Manoj

Age: 21

Email: manoj@gmail.com

In this example, the main method creates a new instance of the Person class and uses the mutator methods (setName, setAge, and setEmail) to set the values of the object's private instance variables. Then, the accessor methods (getName, getAge, and getEmail) are used to retrieve the values of the instance variables and display them on the console.

Advantage of Using Accessor and Mutator

Advantage of using accessor and mutator methods is that they allows us to add validation and other processing logic to the process of getting and setting an object's state. For example, you could add a validation check to the "setAge(int age)" method to ensure that the age value is within a certain range, or add formatting logic to the "getName()" method to capitalize the first letter of the person's name.

Benefits of Encapsulation

Implementing the process of encapsulation in Java has proven to be highly effective and beneficial while programming in real-time. The following are the significant benefits of encapsulation.

- A class can have complete control over its data members and data methods.
- The class will maintain its data members and methods as read-only.
- Data hiding prevents the user from the complex implementations in the code.
- The variables of the class can be read-only or write-only as per the programmer's requirement.
- Encapsulation in Java provides an option of code-reusability.
- Using encapsulation will help in making changes to an existing code quickly.

- Unit testing a code designed using encapsulation is elementary.
- Standard IDEs have the support of getters and setters; this makes coding even faster.

Interfaces and Abstract Classes

Interfaces and abstraction are essential features of object-oriented programming. They provide a way to define contracts and hide implementation details. In Java, they play a crucial role in achieving code flexibility and maintainability.

In Java, interfaces and abstraction are powerful concepts that enable developers to design and implement flexible and extensible software. In this comprehensive guide, we'll delve into the concepts of interfaces and abstraction, provide code examples, discuss key differences, explore new features in Java, and offer best practices to make your Java code more robust and maintainable.

Interface: An Introduction

An interface is a contract that specifies a set of methods without providing their implementations. It acts as a blueprint for classes that implement it. Interfaces enable multiple classes to share a common set of methods without forcing them into a specific inheritance hierarchy.

Declaring interfaces:

An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

Syntax:

```
interface <interface_name>
{

    // declare constant fields
    // declare methods that abstract
    // by default.
```

}

Defining an Interface

In Java, you define an interface using the interface keyword:

```
public interface Shape
{
   double area();
   void draw();
}
```

Implementing an Interface:

A class implements an interface by using the implements keyword. It must provide implementations for all the methods defined in the interface:

```
public class Circle implements Shape
{
        private double radius;
        public Circle(double radius)
        {
                this.radius = radius;
        }
        @Override
        public double area()
        {
                return Math.PI * radius * radius;
        }
        @Override
        public void draw()
        {
                System.out.println("Drawing a circle");
        }
}
```

Extending Interfaces

One interface can inherit another by the use of keyword extends. When a class implements an interface that inherits another interface, it must provide an implementation for all methods required by the interface inheritance chain.

Example 1:

```
interface A
{
    void method1();
    void method2();
}
// B now includes method1 and method2
interface B extends A
{
    void method3();
}
// the class must implement all method of A and B.
class gfg implements B
{
        public void method1()
        {
                System.out.println("Method 1");
        }
        public void method2()
        {
                System.out.println("Method 2");
        }
        public void method3()
```

```
        {
                System.out.println("Method 3");
        }
}
```

Example 2:

```
interface Student
{
   public void data();
}
class avi implements Student
{
   public void data ()
   {
      String name="avinash";
      int rollno=68;
      System.out.println(name);
      System.out.println(rollno);
   }
}
public class inter_face
{
   public static void main (String args [])
   {
      avi h= new avi();
      h.data();
   }
}
```

Output:

avinash

68

In a Simple way, the interface contains multiple abstract methods, so write the implementation in implementation classes. If the implementation is unable to provide an implementation of all abstract methods, then declare the implementation class with an abstract modifier, and complete the remaining method implementation in the next created child classes. It is possible to declare multiple child classes but at final we have completed the implementation of all abstract methods.\

In general, the development process is step by step:

- Level 1 – interfaces: It contains the service details.
- Level 2 – abstract classes: It contains partial implementation.
- Level 3 – implementation classes: It contains all implementations.
- Level 4 – Final Code / Main Method: It have access of all interfaces data.

Key Points to Remember (Interface)

- Interfaces define a contract with method signatures but no method bodies.
- A class can implement multiple interfaces.
- Interfaces are used to achieve multiple inheritances in Java.
- Interfaces can be used to define constants (variables with public static final).

Advantages of Interfaces in Java

The advantages of using interfaces in Java are as follows:

- Without bothering about the implementation part, we can achieve the security of

the implementation.

- In Java, multiple inheritances are not allowed, however, you can use an interface to make use of it as you can implement more than one interface.

Abstraction: An Introduction

Abstraction is the process of hiding complex implementation details and showing only the necessary features of an object. In Java, you achieve abstraction using abstract classes and methods.

Abstract Classes

An abstract class is a class that cannot be instantiated but can contain abstract methods (methods without implementation) and concrete methods (methods with implementation).

```
public abstract class Vehicle
{
	public abstract void start();
	public void stop()
	{
		System.out.println("Vehicle stopped");
	}
}
```

Abstract Methods

An abstract method is declared using the abstract keyword and must be implemented by any concrete subclass.

```
public class Car extends Vehicle
{
	@Override
	public void start()
	{
		System.out.println("Car started");
	}
```

}

Key Points to Remember:

- Abstract classes can’t be instantiated.
- Abstract classes can have both abstract and concrete methods.
- Subclasses of an abstract class must implement its abstract methods.
- Abstraction is used to hide implementation details.

Key Differences

1. Interface vs. Abstract Class

An interface can’t have instance variables, while an abstract class can. A class can implement multiple interfaces, but it can extend only one abstract class. Interfaces provide a strong contract, and multiple inheritance is achieved through them.

2. Method Definitions

Interfaces define methods without implementation, while abstract classes can contain both abstract and concrete methods.

3. Usage

Use interfaces when you want to define a contract for a group of classes. Use abstract classes when you want to provide a common base class with some default behavior.

Module IV

Exception Handling

An *exception* is an error event that can happen during the execution of a program and disrupts its normal flow. Java provides a robust and object-oriented way to handle exception scenarios known as Java Exception Handling.

Exceptions in Java can arise from different kinds of situations such as wrong data entered by the user, hardware failure, network connection failure, or a database server that is down. The code that specifies what to do in specific exception scenarios is called exception handling.

Throwing and Catching Exceptions

Java creates an *exception object* when an error occurs while executing a statement. The exception object contains a lot of debugging information such as method hierarchy, line number where the exception occurred, and type of exception.
If an exception occurs in a method, the process of creating the exception object and handing it over to the runtime environment is called *"throwing the exception"*. The normal flow of the program halts and the Java Runtime Environment (JRE) tries to find the handler for the exception. Exception Handler is the block of code that can process the exception object.

- The logic to find the exception handler begins with searching in the method where the error occurred.
- If no appropriate handler is found, then it will move to the caller method.
- And so on.

So. if the method's call stack is A->B->C and an exception is raised in method C, then the search for the appropriate handler will move from C->B->A.
If an appropriate exception handler is found, the exception object is passed to the handler to process it. The handler is said to be *"catching the exception"*. If there is no appropriate exception handler, found then the program terminates and prints information about the exception to the console. Java Exception handling framework is used to handle runtime errors only. The compile-time errors have to be fixed by the developer writing the code else the program won't execute.

Java Exception Handling Keywords

Java provides specific keywords for exception handling purposes.

1. **throw** – We know that if an error occurs, an exception object is getting created and then Java runtime starts processing to handle them. Sometimes we might want to generate exceptions explicitly in our code. For example, in a user authentication program, we should throw exceptions to clients if the password is null. The throw keyword is used to throw exceptions to the runtime to handle it.

2. **throws** – When we are throwing an exception in a method and not handling it, then we have to use the throws keyword in the method signature to let the caller program know the exceptions that might be thrown by the method. The caller method might handle these exceptions or propagate them to its caller method using

the throws keyword. We can provide multiple exceptions in the throws clause, and it can be used with the main() method also.

3. **try-catch** – We use the try-catch block for exception handling in our code. try is the start of the block and catch is at the end of the try block to handle the exceptions. We can have multiple catch blocks with a try block. The try-catch block can be nested too. The catch block requires a parameter that should be of type Exception.
4. **finally** – the finally block is optional and can be used only with a try-catch block. Since exception halts the process of execution, we might have some resources open that will not get closed, so we can use the finally block. The finally block always gets executed, whether an exception occurred or not.

Example:

ExceptionHandling.java

```
package com.journaldev.exceptions;
import java.io.FileNotFoundException;
import java.io.IOException;
public class ExceptionHandling
{

    public static void main(String[] args) throws FileNotFoundException, IOException
    {
        try
        {
            testException(-5);
            testException(-10);
        }
        catch(FileNotFoundException e)
        {
            e.printStackTrace();
        }
        catch(IOException e)
        {
            e.printStackTrace();
```

```
            }
            finally
            {
                System.out.println("Releasing resources");
            }
            testException(15);
        }

        public static void testException(int i) throws FileNotFoundException, IOException
        {
            if (i < 0)
            {
                FileNotFoundExceptionmyException=new FileNotFoundException
                ("Negative Integer " + i);
                throw myException;
            }
            else if (i > 10)
            {
                throw new IOException("Only supported for index 0 to 10");
            }
        }
}
```

Explanations:

- The testException() method is throwing exceptions using the throw keyword. The method signature uses the throws keyword to let the caller know the type of exceptions it might throw.
- In the main() method, I am handling exceptions using the try-catch block in the main() method. When I am not handling it, I am propagating it to runtime with the throws clause in the main() method.
- The testException(-10) never gets executed because of the exception and then the finally block is executed.

The printStackTrace() is one of the useful methods in the Exception class for debugging purposes.

This code will output the following:

Output:

```
java.io.FileNotFoundException: Negative Integer -5
        at
com.journaldev.exceptions.ExceptionHandling.testException(ExceptionHandling.java:24)
        at com.journaldev.exceptions.ExceptionHandling.main(ExceptionHandling.java:10)
Releasing resources
Exception in thread "main" java.io.IOException: Only supported for index 0 to 10
        at
com.journaldev.exceptions.ExceptionHandling.testException(ExceptionHandling.java:27)
        at com.journaldev.exceptions.ExceptionHandling.main(ExceptionHandling.java:19)
```

Some important points to note:

- We can't have catch or finally clause without a try statement.
- A try statement should have either catch block or finally block, it can have both blocks.
- We can't write any code between try-catch-finally blocks.
- We can have multiple catch blocks with a single try statement.
- try-catch blocks can be nested similar to if-else statements.
- We can have only one finally block with a try-catch statement.

Java Exception Hierarchy

As stated earlier, when an exception is raised an exception object is getting created. Java Exceptions are hierarchical and inheritance is used to categorize different types of exceptions. Throwable is the parent class of Java Exceptions Hierarchy and it has two child objects – Error and Exception. Exceptions are further divided into Checked Exceptions and

Runtime Exceptions.

1. **Errors**: Errors are exceptional scenarios that are out of the scope of application, and it's not possible to anticipate and recover from them. For example, hardware failure, Java virtual machine (JVM) crash, or out-of-memory error. That's why we have a separate hierarchy of Errors and we should not try to handle these situations. Some of the common Errors are OutOfMemoryError and StackOverflowError.

2. **Checked Exceptions**: Checked Exceptions are exceptional scenarios that we can anticipate in a program and try to recover from it. For example, FileNotFoundException. We should catch this exception and provide a useful message to the user and log it properly for debugging purposes. The Exception is the parent class of all Checked Exceptions. If we are throwing a Checked Exception, we must catch it in the same method, or we have to propagate it to the caller using the throws keyword.

3. **Runtime Exception**: Runtime Exceptions are caused by bad programming. For example, trying to retrieve an element from an array. We should check the length of the array first before trying to retrieve the element otherwise it might throw ArrayIndexOutOfBoundException at runtime. RuntimeException is the parent class of all Runtime Exceptions. If we are throwing any Runtime Exception in a method, it's not required to specify them in the method signature throws clause. Runtime exceptions can be avoided with better programming.

Custom Exception

In Java, we can create our own exceptions that are derived classes of the Exception class. Creating our own Exception is known as custom exception or user-defined exception. Basically, Java custom exceptions are used to customize the exception according to user need.

Consider the example in which InvalidAgeException class extends the Exception class. Using the custom exception, we can have your own exception and message. Here, we have passed a string to the constructor of superclass i.e. Exception class that can be obtained using getMessage() method on the object we have created.

Why use custom exceptions?

Java exceptions cover almost all the general type of exceptions that may occur in the programming. However, we sometimes need to create custom exceptions.

Following are few of the reasons to use custom exceptions:
To catch and provide specific treatment to a subset of existing Java exceptions. Business logic exceptions: These are the exceptions related to business logic and workflow. It is useful for the application users or the developers to understand the exact problem. In order to create custom exception, we need to extend Exception class that belongs to java.lang package.

Consider the following example, where we create a custom exception named WrongFileNameException:

```
public class WrongFileNameException extends Exception
{
        public WrongFileNameException(String errorMessage)
        {
                super(errorMessage);
        }
}
```

Example:
Let's see a simple example of Java custom exception. In the following code, constructor of InvalidAgeException takes a string as an argument. This string is passed to constructor of parent class Exception using the super() method. Also the constructor of Exception class can be called without using a parameter and calling super() method is not mandatory.

TestCustomException1.java

```
// class representing custom exception
class InvalidAgeException  extends Exception
{
   public InvalidAgeException (String str)
   {
```

```
        // calling the constructor of parent Exception
        super(str);
    }
}

// class that uses custom exception InvalidAgeException
public class TestCustomException1
{

        // method to check the age
        static void validate (int age) throws InvalidAgeException
        {
                if(age < 18)
                {
                        // throw an object of user defined exception
                        throw new InvalidAgeException("age is not valid to vote");
                }
                else
                {
                        System.out.println("welcome to vote");
                }
        }

    // main method
    public static void main(String args[])
    {
        try
        {
            // calling the method
            validate(13);
        }
        catch (InvalidAgeException ex)
        {
            System.out.println("Caught the exception");
```

```
            // printing the message from InvalidAgeException object
            System.out.println("Exception occured: " + ex);
        }
        System.out.println("rest of the code...");
    }
}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestCustomException1.java

C:\Users\Anurati\Desktop\abcDemo>java TestCustomException1
Caught the exception
Exception occured: InvalidAgeException: age is not valid to vote
rest of the code...
```

File Handling in Java

In Java, with the help of File Class, we can work with files. This File Class is inside the java.io package. The File class can be used by creating an object of the class and then specifying the name of the file.

Why File Handling is Required?

File Handling is an integral part of any programming language as file handling enables us to store the output of any particular program in a file and allows us to perform certain operations on it.

In simple words, file handling means reading and writing data to a file.

```
// Importing File Class
import java.io.File;
class GFG
{
    public static void main(String[] args)
    {

        // File name specified
        File obj = new File("myfile.txt");
```

```
        System.out.println("File Created!");
    }
}
```

Output:
File Created!

File Operations in Java

In Java, a File is an abstract data type. A named location used to store related information is known as a File. There are several File Operations like creating a new File, getting information about File, writing into a File, reading from a File and deleting a File. Before understanding the File operations, it is required that we should have knowledge of Stream and File methods. If you have knowledge about both of them, you can skip it.

Stream

A series of data is referred to as a stream. In Java, Stream is classified into two types, i.e., Byte Stream and Character Stream.

- **Byte Stream:** Byte Stream is mainly involved with byte data. A file handling process with a byte stream is a process in which an input is provided and executed with the byte data.

- **Character Stream:** Character Stream is mainly involved with character data. A file handling process with a character stream is a process in which an input is provided and executed with the character data.

We can perform the following operation on a file:

- Create a File
- Get File Information
- Write to a File
- Read from a File
- Delete a File

Create a File

Create a File operation is performed to create a new file. We use the createNewFile() method of file. The createNewFile() method returns true when it successfully creates a new file and returns false when the file already exists. Let's take an example of creating a file to understand how we can use the createNewFile() method to perform this operation.

Example:

CreateFile.java

```
// Importing File class
import java.io.File;
// Importing the IOException class for handling errors
import java.io.IOException;
 class CreateFile
{
                public static void main(String args[])
                {
                   try
                  {
                        // Creating an object of a file
                        File f0 = new File("D:FileOperationExample.txt");
                        if (f0.createNewFile())
                        {
                        System.out.println("File " + f0.getName() + " is created successfully.");
                        }
                        else
                        {
                        System.out.println("File is already exist in the directory.");
                        }
                  }
                catch (IOException exception)
                {
                     System.out.println("An unexpected error is occurred.");
```

```
                exception.printStackTrace();
            }
        }
}
```

Output:

File Operations in Java

File Operations in Java

Explanation: In the above code, we import the File and IOException class for performing file operation and handling errors, respectively. We create the f0 object of the File class and specify the location of the directory where we want to create a file. In the try block, we call the createNewFile() method through the f0 object to create a new file in the specified location. If the method returns false, it will jump to the else section. If there is any error, it gets handled in the catch block.

Write to a File

The next operation which we can perform on a file is "writing into a file". In order to write data into a file, we will use the FileWriter class and its write() method together. We need to close the stream using the close() method to retrieve the allocated resources. Let's take an example to understand how we can write data into a file.

Example:

WriteToFile.java

```
// Importing the FileWriter class
import java.io.FileWriter;
// Importing the IOException class for handling errors
import java.io.IOException;
class WriteToFile
{

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

```
            try
            {
                FileWriter fwrite = new FileWriter("D:FileOperationExample.txt");
                // writing the content into the FileOperationExample.txt file
                fwrite.write("A named location used to store related information is
    referred                to as a File.");
                // Closing the stream
                fwrite.close();
                System.out.println("Content is successfully wrote to the file.");
            }
            catch (IOException e)
            {
                System.out.println("Unexpected error occurred");
                e.printStackTrace();
            }
   }
}
```

Output:

File Operations in Java

File Operations in Java

Explanation: In the above code, we import the java.io.FileWriter and java.io.IOException classes. We create a class WriteToFile, and in its main method, we use the try-catch block. In the try section, we create an instance of the FileWriter class, i.e., fwrite. We call the write method of the FileWriter class and pass the content to that function which we want to write. After that, we call the close() method of the FileWriter class to close the file stream. After writing the content and closing the stream, we print a custom message. If we get any error in the try section, it jumps to the catch block. In the catch block, we handle the IOException and print a custom message.

Read from a File

The next operation which we can perform on a file is "read from a file". In order to write data

into a file, we will use the Scanner class. Here, we need to close the stream using the close() method. We will create an instance of the Scanner class and use the hasNextLine() method nextLine() method to get data from the file. Let's take an example to understand how we can read data from a file.

Example:

ReadFromFile.java

```
// Importing the File class
import java.io.File;
// Importing FileNotFoundException class for handling errors
import java.io.FileNotFoundException;
// Importing the Scanner class for reading text files
import java.util.Scanner;
class ReadFromFile
{
	public static void main(String[] args)
	{
		try
		{
			// Create f1 object of the file to read data
			File f1 = new File("D:FileOperationExample.txt");
			Scanner dataReader = new Scanner(f1);
			while (dataReader.hasNextLine()) {
			String fileData = dataReader.nextLine();
			System.out.println(fileData);
		}
	dataReader.close();
      }
	catch (FileNotFoundException exception)
	{
		System.out.println("Unexcpected error occurred!");
		exception.printStackTrace();
	}
  }
```

```
}
```

Output:

File Operations in Java

Explanation: In the above code, we import the "java.util.Scannner", "java.io.File" and "java.io.IOException" classes. We create a class ReadFromFile, and in its main method, we use the try-catch block. In the try section, we create an instance of both the Scanner and the File classes. We pass the File class object to the Scanner class object and then iterate the scanner class object using the "While" loop and print each line of the file. We also need to close the scanner class object, so we use the close() function. If we get any error in the try section, it jumps to the catch block. In the catch block, we handle the IOException and print a custom message.

Delete a File:

The next operation which we can perform on a file is "deleting a file". In order to delete a file, we will use the delete() method of the file. We don't need to close the stream using the close() method because for deleting a file, we neither use the FileWriter class nor the Scanner class. Let's take an example to understand how we can write data into a file.

Example:

DeleteFile.java

```
// Importing the File class
import java.io.File;
class DeleteFile
{
	public static void main(String[] args)
	{
		File f0 = new File("D:FileOperationExample.txt");
		if (f0.delete())
		{
			System.out.println(f0.getName()+ " file is deleted successfully.");
		}
		else
		{
```

```
                    System.out.println("Unexpected error found in deletion of the file.");
                }
        }
}
```

Output:

File Operations in Java

Explanation: In the above code, we import the File class and create a class DeleteFile. In the main() method of the class, we create f0 object of the file which we want to delete. In the if statement, we call the delete() method of the file using the f0 object. If the delete() method returns true, we print the success custom message. Otherwise, it jumps to the else section where we print the unsuccessful custom message.

File Streams in Java

In Java, a sequence of data is known as a stream. This concept is used to perform I/O operations on a file. You can create a file stream from the filename, a File object, or a FileDescriptor. object. Use file streams to read data from or write data to files on the file system.

There are two types of streams in java:

1. File Input Stream
2. File Output Stream

1. File Input Stream:

The Java InputStream class is the superclass of all input streams. The input stream is used to read data from numerous input devices like the keyboard, network, etc. InputStream is an abstract class, and because of this, it is not useful by itself. However, its subclasses are used to read data. There are several subclasses of the InputStream class, which are as follows:

- AudioInputStream
- ByteArrayInputStream
- FileInputStream
- FilterInputStream
- StringBufferInputStream
- ObjectInputStream

Creating an Input Stream:

Syntax:

```
// Creating an InputStream
InputStream obj = new FileInputStream();
```

Here, an input stream is created using FileInputStream.

Note: We can create an input stream from other subclasses as well as InputStream.

Methods of Input Stream:

1	read()	Reads one byte of data from the input stream.
2	read(byte[] array)()	Reads byte from the stream and stores that byte in the specified array.
3	mark()	It marks the position in the input stream until the data has been read.
4	available()	Returns the number of bytes available in the input stream.
5	markSupported()	It checks if the mark() method and the reset() method is supported in the stream.
6	reset()	Returns the control to the point where the mark was set inside the stream.
7	skips()	Skips and removes a particular number of bytes from the input stream.
8	close()	Closes the input stream.

2. Output Stream:

The output stream is used to write data to numerous output devices like the monitor, file, etc. OutputStream is an abstract superclass that represents an output stream. OutputStream is an abstract class and because of this, it is not useful by itself. However, its subclasses are used to write data. There are several subclasses of the OutputStream class which are as follows:

- ByteArrayOutputStream
- FileOutputStream
- StringBufferOutputStream
- ObjectOutputStream
- DataOutputStream
- PrintStream

Creating an Output Stream

Syntax:

// Creating an OutputStream
OutputStream obj = new FileOutputStream();
Here, an output stream is created using FileOutputStream.

Note: We can create an output stream from other subclasses as well as OutputStream.

Methods of Output Stream:

1. write() Writes the specified byte to the output stream.
2. write(byte[] array) Writes the bytes which are inside a specific array to the output stream.
3. close() Closes the output stream.
4. flush() Forces to write all the data present in an output stream to the destination.