MODULE 1

Features of Java

- **Object Oriented** − In Java, everything is an Object. Java can be easily extended since it is based on the Object model.
- **Platform Independent** − Java code can run on multiple platforms directly, which can be converted into byte code at the compile time. The byte code is a platform-independent code that can run on multiple platforms

.Simple − Java is very easy to learn, and its syntax is simple, clean and easy to understand. Java syntax is based on C++ .Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc.

- **Secure** − With Java's secure feature it enables to develop virus-free, systems.
- **Architecture-neutral** − Java compiler generates an architecture-neutral object file format, which makes the compiled code executable on many processors, with the presence of Java runtime system.

Portable − Java is portable because it facilitates you to carry the Java bytecode to any platform. It doesn't require any implementation.

.Robust − Java is robust because ,It uses strong memory management. There is a lack of pointers that avoids security problems. There are exception handling and the type checking mechanism in Java. All these points make Java robust.

- **Multithreaded** − With Java's multithreaded feature it is possible to write programs that can perform many tasks simultaneously. This design feature allows the developers to construct interactive applications that can run smoothly.
- **Interpreted** − The development process is more rapid and analytical since the linking is an incremental and light-weight process.

. **High Performance -**Java is faster than other programming languages because of Java bytecode.

. **Distributed** − Java is designed for the distributed environment of the internet.

- **Dynamic** − Java is a dynamic language. It supports the dynamic loading of classes. It means classes are loaded on demand.

Characteristics of OOP(Object Oriented Programming)

- Object-Oriented Development
- Classes and Objects:
- Encapsulation:
- Abstraction:
- Inheritance
- Polymorphism

ADVANTAGES OF JAVA

1. Simple

Java is a simple programming language since it is easy to learn and easy to understand. Its syntax is based on C++. Java has also removed the features like explicit pointers, operator overloading, etc., making it easy to read and write

2. Object-Oriented

Java uses an object-oriented paradigm, which makes it more practical. Everything in Java is an object which takes care of both data and behavior. Java uses object-oriented concepts like object, class, inheritance, encapsulation, polymorphism, and abstraction.

3. Secured

Java is a secured programming language because it doesn't use Explicit pointers

4. Robust

Java is a robust programming language since it uses strong memory management. We can also handle exceptions through the Java code. Also, we can use type checking to make our code more secure. It doesn't provide explicit pointers so that the programmer cannot access the memory directly from the code.

5. Platform independent

Java code can run on multiple platforms directly, which can be converted into byte code at the compile time. The byte code is a platform-independent code that can run on multiple platforms.

6. Multi-Threaded

Java uses a multi-threaded environment in which a bigger task can be converted into various threads and run separately.

TOOLS AVAILABLE OF JAVAPROGRAMMING

(JDK and IDE)

(I) JDK(Java Development Kit)

The Java Development Kit (JDK) is a software development environment which is used to develop java application. It physically exists. It contains JRE (Java Runtime Environment)and development tools.

The JDK contains a private Java Virtual Machine (JVM) and a few other resources such as an interpreter/loader (Java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc) etc to complete the development of a Java Application. Components of JDK are java, javac ,javadoc etc

(II) Java IDE (Integrated Development Environment)

It is a software application that enables users to **write** and **debug** Java programs more easily. Most IDEs have features such as syntax highlighting and code completion that helps users to code more easily. Usually, Java IDEs include a **code editor, a compiler, a debugger**, and **an interpreter** that the developer may access via a single graphical user interface.

The Java IDE or Integrated Development Environment provides considerable support for the application development process. Through using them, we can save time and effort and set up a standard development process for the team or company. **Eclipse, NetBeans,** many other IDE's are most popular in the Java IDE's that can be used according to our requirements.

(a)Eclipse

It is a Java-based open-source platform.This platform is also suitable for beginners to create user-friendly and more sophisticated applications. Eclipse provides powerful tools for different software development processes, such as charting, reporting, checking, etc. so that Java developers can build the

application as quickly as possible.Eclipse can be used on platforms such as MacOS, Linux, and Windows

(b) NetBeans

NetBeans is a Java-based IDE and basic application platform framework. NetBeans supports java PHP, C/C++,and HTML5 and some other languages. It is a free and open-source Java IDE .NetBeans is available for various operating systems, such as Linux, MacOS, Windows etc. That is NetBeans may be used on different operating systems such as MacOS, Windows, and Linux.

OBJECT AND CLASS

Class –A class is a blueprint from which individual objects are created.

-- It is a template or blueprint from which objects are created.

--A class in Java can contain: Fields, Methods, Constructors ,Blocks.

Syntax to declare a class:

```
class   classname {
     field;
    method;
}
```

Example of class:

```
class  Cat {
      int age;
   String color;
   void eating() {
    }
  void sleeping() {
    }
}
```

Object –

.. An object is an instance of a class. Objects have states and behaviors.

If we consider the real-world, we can find many objects around us, cars, dogs, humans, etc. All these objects have a state and a behavior.

If we consider a dog, then its state is - name, color, and the behavior is - barking, running.

Software objects also have a state and a behavior. A software object's state is stored in **fields** and behavior is shown via **methods.**

Creating an Object:::

A class provides the blueprints for objects. So basically, an object is created from a class. In Java, the **new** keyword is used to create new objects..

Following is an example of creating an object −

Example

```
public  class Student{
      public static void main(String [] args) {
      // Following statement would create an object obj
      Student obj = new Student();
   }
}
```

Here obj is called "Reference variable"

A class can contain any of the following variable types.

- **Local variables** − Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method.
- **Instance variables** − Instance variables are variables within a class but outside any method. Instance variables can be accessed from inside any method, constructor or blocks of that particular **class**.
- **Class/static variables** − Class variables are variables declared within a class, outside any method, with the static keyword.

A class can have any number of methods to access the value of various kinds of methods. In the above example, eating() and sleeping() are methods.

Java Methods

A **method** is a block of code which only runs when it is called.

You can pass data, known as parameters, into a method.

Methods are used to perform certain actions, and they are also known as **functions**.

To reuse code: methods are used.(define the code once, and use it many times.)

Create a Method

A method must be declared within a class. It is defined with the name of the method, followed by parentheses **()**. Java provides some pre-defined methods, such as **System.out.println(),** but you can also create your own methods to perform certain actions:

Example:::

Create a method inside class Demo:

```
class Demo {
        void myMethod() {
// code to be executed
 }
}
```

Call a Method

To call a method in Java, write the method's name followed by two parentheses **()** and a semicolon**;**

Parameters and Arguments

Information can be passed to methods as parameter. Parameters act as variables inside the method.Parameters are specified after the method name, inside the parentheses.

Java - Constructors

constructor has the same name as its class and is syntactically similar to a method. However, constructors have no explicit **return type**.

Typically, we will use a constructor to give initial values to the instance variables defined by the class,

All classes have constructors, whether you define one or not, because Java automatically provides a default constructor that initializes all member variables to zero.

Syntax

Following is the syntax of a constructor −

```
class ClassName {
   ClassName() {
   }
}
```

In the above syntax **ClassName()** is the constructor

```
Example:  class Student {
   Student() {
   }
}
```

In the above example **Student()** is the constructor

Two types of constructors –

- Default Constructors
- Parameterized Constructors

(a) Default Constructors

As the name specifies the default constructors or no argument constructors of Java does not accept any parameters . Using these constructors the instance variables of a method will be initialized with fixed values for all objects.

```
class MyClass {
   Int num;
   MyClass() {
      num = 100;
   }
}
```

calling default constructor is as follows

```
class Demo {
  public static void main(String args[]) {
     MyClass  ob1 = new MyClass();
           }
}
```

(b)Parameterized Constructors

This constructor accepts one or more parameters. Parameters are added to a constructor in the same way that they are added to a method, just declare them inside the parentheses after the constructor's name.

Example

```
class MyClass {
   int x;
      // Following is the constructor
   MyClass(int i ) {
      x = i;
   }
}
```

calling parameterized constructor is follows –

```
public class  Demo {
   public static void main(String args[]) {
      MyClass  obj = new MyClass( 10 );
            }
}
```

Constructor overloading in Java

In Java, we can overload constructors like methods. The constructor overloading can be defined as the concept of having more than one constructor with different parameters so that every constructor can perform a different task.

Java - Access Modifiers(visibility modifiers)

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

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

(1)Default Access Modifier - No Keyword

Default access modifier means we do not explicitly declare an access modifier for a class, field, method, etc.

A variable or method declared without any access control modifier is available to any other class in the same package.

(2)Private Access Modifier - Private

Methods, variables, and constructors that are declared private can only be accessed within the declared **class** itself.

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

(3)Public Access Modifier - Public

A class, method, constructor, interface, etc. declared public can be accessed from any other class. Therefore, fields, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe.

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

(4)Protected Access Modifier – Protected

Variables, methods, and constructors, which are declared protected in a superclass can be accessed only by the subclasses in other **package** .The protected access modifier cannot be applied to class and interfaces. **Methods, fields** can be declared protected, however methods and fields in a interface cannot be declared protected.

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Yes	No	No	No
Default	Yes	Yes	No	No
Protected	Yes	Yes	Yes	No
Public	Yes	Yes	Yes	Yes

Static variables and static method

The **static keyword** in Java is used for memory management mainly. We can apply static keyword with variables, methods.

The static can be:

1. Variable (also known as a class variable)
2. Method (also known as a class method)

(1) static variables::

If we declare any variable as **static**, it is known as a static variable.

Eg: static String college;

- .Static variables can be accessed using the **class name** followed by a dot and the name of the variable
- The static variable can be used to refer to the common property of all objects. for example, the company name of employees, college name of students, etc.

- The static variable gets memory only once in the class area at the time of class loading.

Advantages of static variable

It makes the program **memory efficient** (i.e., it saves memory).

Static variables are also known as **class variables**..

(b) static method::

If we apply **static** keyword with any method, it is known as static method.

Eg: **static void** change(){
college = "BBDIT"; }

- A static method can be invoked(called) without the need for creating an object of a class.
- Static methods can be accessed using the **class name** followed by a dot and the name of the variable or method.

Encapsulation in Java

Encapsulation in Java is a *process of wrapping code and data together into a single unit*,. for example, a capsule which is mixed of several medicines.

Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the **variables** of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as **data hiding**


Capsule

Encapsulation is one of the four fundamental OOP concepts. The other three are inheritance, polymorphism, and abstraction.

.To achieve encapsulation in Java −Declare the variables of a class as **private.**

That is We can create a fully encapsulated class in Java by making all the data members of the class **private.**

Example:

```
class  EncapTest {
  private String name;
  private String  idNum;
  private int age;
  public int getAge() {
     return age;
  }
```

Polymorphism in Java

Polymorphism in Java is a concept by which we can perform a *single action in different ways*. Polymorphism is derived from 2 Greek words: poly and morphs. The word **"poly"** means **many** and **"morphs"** means **forms**. So polymorphism means "**many forms"**.

There are two types of polymorphism in Java:

(1)compile-time polymorphism

(2)Runtime polymorphism.

We can perform polymorphism in java by **method overloading and method overriding.**

(1)compile-time polymorphism:::

Method Overloading in Java

If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.

If we have to perform only one operation, having same name of the methods increases the readability of the program.

There are two ways to **overload** the method in java

1. By changing number of arguments
2. By changing the data type

Example :

```
void  myMethod(int x)
void  myMethod(float x)
void  myMethod(double x, double y)
```

(2)Runtime Polymorphism in Java:

Runtime polymorphism or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

In this process, an overridden method is called through the **reference variable** of a superclass.

Method overriding:

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.

In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding. Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.
3. There must be an inheritance.

Example:

```
class Bike{

     void run(){
   System.out.println("running");}
                    }
class Splendor extends Bike{
  void run(){
System.out.println("running safely with 60km");
                  }
    public static void main(String args[]){
```

```
    Bike obj = new Splendor();
   obj.run();
  }
}
```

In this example, we are creating two classes Bike and Splendor. Splendor class extends Bike class and overrides its **run()** method. We are calling the run method by the **reference variable of Parent class**. Since it refers to the subclass object and subclass method overrides the Parent class method, the subclass method is invoked at runtime.

Exception

In Java, an exception is an event that disrupts the normal flow of the program.

Types of Java Exceptions:

1) ArithmeticException

If we divide any number by zero, there occurs an ArithmeticException.

Eg: **int** a=50/0;//ArithmeticException

2) NullPointerException

If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

Eg: String s=**null**;
System.out.println(s.length()); //NullPointer Exception

3)ArrayIndexOutOfBoundsException occurs

When an array exceeds to it's size, the ArrayIndexOutOfBounds Exception occurs.

Eg: **int** a[]=**new int**[5];

a[10]=50; //ArrayIndexOutOfBoundsException

Exception Handling

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, etc.

The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions.

statement 1;

statement 2;

statement 3;

statement 4;

statement 5;//**exception occurs**

statement 6;

statement 7;

statement 8;

statement 9;

statement 10;

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

Java try-catch block

Java try block

Java **try** block is used to enclose the code that might throw an exception. It must be used within the method.

If an exception occurs at the particular statement in the try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception.

Java try block must be followed by either catch or finally block.

Syntax of Java try-catch

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

Java catch block

Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (i.e., **Exception**) or the generated exception type (eg :ArithmeticException).

Java Multiple-catch block

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block. At a time only one exception occurs and at a time only one catch block is executed.

```
public class MultipleCatchBlock1 {
    public static void main(String[] args) {
          try{
        int a[]=new int[5];
        a[5]=30/0;
        }
        catch(ArithmeticException e)
         {
          System.out.println("Arithmetic Exception occurs");
         }
        catch(ArrayIndexOutOfBoundsException e)
         {
          System.out.println("ArrayIndexOutOfBounds Exception occurs");
         }
        catch(Exception e)
```

```
            {
             System.out.println("Parent Exception occurs");
            }
          System.out.println("rest of the code");
    }
}
```

Java finally block

Java finally block is a block used to execute important code such as closing the connection, etc.

Java finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.

The finally block follows the try-catch block.

```
Eg:        try {
               }
               catch(Exception  e){
                    System.out.println(e);
                    }
              finally {
                  System.out.println("finally block is always executed");
                    }
```

Java Data Types

Data types are divided into two groups:

(1)**Primitive data types** -
includes `byte`, `short`, `int`, `long`, `float`, `double`, `boolean` and `char`

(2)Non-Primitive Data Types

Non-primitive data types are called **reference types** because they refer to objects.

Examples of non-primitive types are Strings, Arrays, Classes, Interface, etc.

The main difference between **primitive** and **non-primitive** data types are:

- Primitive types are predefined (already defined) in Java. Non-primitive types are created by the programmer and is not defined by Java (except for `String`).
- Non-primitive types can be used to call methods to perform certain operations, while primitive types cannot.
- A primitive type has always a value, while non-primitive types can be `null`.
- A primitive type starts with a lowercase letter, while non-primitive types starts with an uppercase letter.
- The size of a primitive type depends on the data type, while non-primitive types have all the same size.

pass “object” as an argument in Java methods

When we pass a primitive type (int, float, etc) to a method, it is **pass by value.** we can pass **objects** as arguments in Java.

But when we pass an **object** to a method, it is called as **call-by-reference**. when we pass reference variable (object) to a method, the parameter that receives it will refer to the same object as that referred to by the argument.

Returning Objects

In java, a method can return any type of data, including **objects.**