Unit 2

Objects and classes:

Java is an object-oriented programming language. The core concept of the object-oriented approach is to break complex problems into smaller objects.

An object is any entity that has a state and behavior. For example, a bicycle is an object. It has

- States: idle, first gear, etc
- Behaviors: braking, accelerating, etc.

Before we learn about objects, let's first know about classes in Java.

Java Class: A class is a blueprint for the object. Before we create an object, we first need to define the class.

We can think of the class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows, etc. Based on these descriptions we build the house. House is the object.

Since many houses can be made from the same description, we can create many objects from a class.

We can create a class in Java using the class keyword. For example,

```
class ClassName {
  // fields
  // methods
}
```

Here, fields (variables) and methods represent the state and behavior of the object respectively.

- fields are used to store data

- methods are used to perform some operations

For our bicycle object, we can create the class as

```
class Bicycle {
  // state or field
  private int gear = 5;
  // behavior or method
  public void braking() {
    System.out.println("Working of Braking");
  }
}
```

In the above example, we have created a class named Bicycle. It contains a field named gear and a method named braking().

Here, Bicycle is a prototype. Now, we can create any number of bicycles using the prototype. And, all the bicycles will share the fields and methods of the prototype.

What is an Object in Java?

Objects model an entity that exists in the real world. Modeling entities require you to identify the state and set of actions that can be performed in that object. This way of thinking is key to object-oriented programming. It is important to disambiguate an Object with an instantiated object in Java.

- An Object is the root class of all instantiated objects in Java.
- Instantiated objects are names that refer to an instance of the class.

Declaring Objects in Java

```
Sphere sphere = new Sphere(10);
```

```
// 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);

    }

}
```

Difference between Java Class and Objects: The differences between class and object in Java are as follows:

Class	**Object**
Class is the blueprint of an object. It is used to create objects.	An object is an instance of the class.
No memory is allocated when a class is declared.	Memory is allocated as soon as an object is created.
A class is a group of similar objects.	An object is a real-world entity such as a book, car, etc.
Class is a logical entity.	An object is a physical entity.
A class can only be declared once.	Objects can be created many times as per requirement.
An example of class can be a car.	Objects of the class car can be BMW, Mercedes, Ferrari, etc.

What are Branching Statements in Java?

Branching statements allow the flow of execution to jump to a different part of the program. The common branching statements used within other control structures include: break, continue, and return.

Types of Branching Statement in Java:

In Java, there are three Branching Statements. They are as follows:

1. Break Statement
2. Continue Statement
3. Return Statement

Break statement in Java: Using break, we can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop.

```
class BreakStatementDemo

{

   public static void main(String args[])

   {

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

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

      {

         // terminate loop when i is 5.

         if (i == 5)

            break;

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

      }

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

   }

}
```

Continue Statement in Java:

The continue statement is used to skip a part of the loop and continue with the next iteration of the loop. It can be used in combination with for and while statements.

```
class ContinueStatementDemo
{
   public static void main(String args[])
   {
      for (int i = 0; i < 10; i++)
      {
         // If the number is even
         // skip and continue
         if (i%2 == 0)
            continue;
         // If number is odd, print it
         System.out.print(i + " ");
      }
   }
}
```

Return Statement in Java: This is the most commonly used branching statement of all. The return statement exits the control flow from the current method and returns to where the method was invoked. The return statement can return a value or may not return a value. To return a value, just put the value after the return keyword. But remember data type of the returned value must match the type of the method's declared return value. And so when a method is declared void, use return without a return value.

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

```
    {
       boolean t = true;
       System.out.println("Before the return.");
       if (t)
          return;
       // Compiler will bypass every statement
       // after return
       System.out.println("This won't execute.");
    }
}
```

Iteration in Java: Java provides a set of looping statements that executes a block of code repeatedly while some condition evaluates to true. Looping control statements in Java are used to traverse a collection of elements, like arrays.

while Loop: The while loop statement is the simplest kind of loop statement. It is used to iterate over a single statement or a block of statements until the specified boolean condition is false. The while loop statement is also called the entry-control looping statement because the condition is checked prior to the execution of the statement and as soon as the boolean condition becomes false, the loop automatically stops. You can use a while loop statement if the number of iterations is not fixed. Normally the while loop statement contains an update section where the variables, which are involved in while loop condition, are updated.

```
public class WhileLoopDemo {
  public static void main(String args[]) {
    int num = 10;
    while (num > 0) {
      System.out.println(num);
      // Update Section
      num--;
    }
  }
}
```

do-while Loop: The Java do-while loop statement works the same as the while loop statement with the only difference being that its boolean condition is evaluated post first execution of the body of the loop. Thus it is also called exit controlled looping statement. You can use a do-while loop if the number of iterations is not fixed and the body of the loop has to be executed at least once.

```
public class Main {
  public static void main(String args[]) {
    int num = 10;
    do {
      System.out.println(num);
      num--;
    } while (num > 0);
  }
}
```

for Loop: Unlike the while loop control statements in Java, a for loop statement consists of the initialization of a variable, a condition, and an increment/decrement value, all in one line. It executes the body of the loop until the condition is false. The for loop statement is shorter and provides an easy way to debug structure in Java. You can use the for loop statement if the number of iterations is known.

In a for loop statement, execution begins with the initialization of the looping variable, then it executes the condition, and then it increments or decrements the looping variable. If the condition results in true then the loop body is executed otherwise the for loop statement is terminated.

```
public class ForLoopDemo {
  public static void main(String args[]) {
    for (int num = 10; num > 0; num--) {
System.out.println( "The value of the number is: " + num  );
  }
}}
```

for-each Loop: The for-each loop statement provides an approach to traverse through elements of an array or a collection in Java. It executes the body of the loop for each element of the given array or collection. It is also known as the Enhanced for loop statement because it is easier to use than the for loop statement as you don't have to handle the increment operation. The major difference between the for and for-each loop is that for loop is a general-purpose loop that we can use for any use case, the for-each loop can only be used with collections or arrays.

In for-each loop statement, you cannot skip any element of the given array or collection. Also, you cannot traverse the elements in reverse order using the for-each loop control statement in Java.

```
public class ForEachLoopDemo {
  public static void main(String args[]) {
    int[] array = {+ 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };
    System.out.println("Elements of the array are: ");
    for (int elem : array) System.out.println(elem);
  }
}
```

Decision-Making Statements: Decision-making statements in Java are similar to making a decision in real life, where we have a situation and based on certain conditions we decide what to do next.

For example, if it's raining outside then we need to carry an umbrella. In programming also, we have some situations when we need a specific code to be executed only when a condition is satisfied.

The decision control statements help us in this task by evaluating a boolean expression and accordingly controlling the flow of the program.

There are four types of decision-making statements in Java:

1. if Statement: These are the simplest and yet most widely used control statements in Java. The if statement is used to decide whether a particular block of code will be executed or not based on a certain condition. If the condition is true, then the code is executed otherwise not.

```
String role = "admin";

if(role == "admin") {

   System.out.println("Admin screen is loaded");}
```

2. **if-else Statement:** The if statement is used to execute a block of code based on a condition. But if the condition is false and we want to do some other task when the condition is false, how should we do it? That's where else statement is used. In this, if the condition is true then the code inside the if block is executed otherwise the else block is executed.

```
String role = "user";

if(role == "admin") {

   System.out.println("Admin screen is loaded");

}

else {

   System.out.println("User screen is loaded");}
```

3. **Nested if-else Statement:** Java allows us to nest control statements within control statements.Nested control statements mean an if-else statement inside other if or else blocks. It is similar to an if-else statement but they are defined inside another if-else statement. Let us see the syntax of a specific type of nested control statements where an if-else statement is nested inside another if block.

```
int age = 20;

String gender = "male";

if (age > 18) {

   // person is an adult

   if (gender == "male") {

   // person is a male

   System.out.println(

      "You can shop in the men's section on the 3rd Floor"

   );

   } else {

   // person is a female

   System.out.println("You can shop in the women's section on 2nd Floor");

   }

} else {

   // person is not an adult

   System.out.println("You can shop in the kid's section on 1st Floor");

}
```

4. switch Statement: Switch statements are almost similar to the if-else-if ladder control statements in Java. It is a multi-branch statement. It is a bit easier than the if-else-if ladder and also more user-friendly and readable. The switch statements have an expression and based on the output of the expression, one or more blocks of codes are executed. These blocks are called cases. We may also provide a default block of code that can be executed when none of the cases are matched similar to the else block.

```
String browser = "chrome";

switch (browser)

{

   case "safari":

       System.out.println("The browser is Safari");

       break;

   case "edge":

       System.out.println("The browser is Edge");

       break;

   case "chrome":

       System.out.println("The browser is Chrome");

       break;

   default:

       System.out.println("The browser is not supported");

}
```

Constructor: Constructor in java is used to create the instance of the class. Constructors are almost similar to methods except for two things - its name is the same as the class name and it has no return type. Sometimes constructors are also referred to as special methods to initialize an object.

If a class doesn't have a constructor, the Java compiler automatically creates a default constructor during run-time. The default constructor initializes instance variables with default values. For example, the int variable will be initialized to 0. A constructor cannot be abstract or static or final.A constructor can be overloaded but can not be overridden.

```
class Main {
 private String name;
 // constructor
 Main() {
  System.out.println("Constructor Called:");
  name = "Programiz";
 }
 public static void main(String[] args) {
  // constructor is invoked while
  // creating an object of the Main class
  Main obj = new Main();
  System.out.println("The name is " + obj.name);
 }
}
```

Features of Constructors in Java:

- The constructor name has the same as the Class Name.

  Explanation: Compiler uses this character to differentiate constructors from the other member functions of the class.

A constructor must not have any return type. Not even void.

  Explanation: It automatically calls and is commonly used for initializing values.

The constructor automatically gets calls when an object creates for the class.

Unlike, Constructors in C++, Java doesn't allow the constructor definition outside the class.

Constructors usually make use to initialize all the data members (variables) of the class.

In addition, more than one constructor can also declare inside a class with different parameters. It's called "Constructor Overloading."

Difference between java constructor and java method:

Java Constructor	Java Method
Name of a constructor is same name as that of the class.	Name of the method can be different from the class name.
Constructor has no return type.	Method must have a return type.
Constructors are invoked implicitly by system.	Methods are invoked explicitly by programmer.
Constructors are executed only when the object is created.	Methods are executed when we explicitly call it.

Java Data Types:

Every individual bit of data that is processed every day is categorized into types. The type of data is known as datatype. Java uses various kinds of data types.

However the data types are mainly of two categories:

a. Primitive Data Types- These data types are already hard coded into the compiler to be recognized when the program is executed. Examples are- int,float etc.

b. Non-Primitive Data Types- These data types are special types of data which are user defined, i,e, the program contains their definition. Some examples are- classes, interfaces etc.

1. Primitive Data Types in Java

Java primitive data types are the ones which are predefined by the programming language which in this case is Java. Without primitive data types it would be impossible to frame programs. Primitive data types are also the building blocks of Non-primitive data types.

Primitive Data Types
01
Int
02
Float
03
Char
04
Boolean
05
Byte
06
Short
07
Long
08
Double

There are 8 types of Java primitive data types namely:
a. Int
b. Float
c. Char
d. Boolean
e. Byte
f. Short
g. long
h. Double.

a. Integer Datatype in Java: int is used for storing integer values. Its size is 4 bytes and has a default value of 0. The maximum values of integer is 2^31 and the minimum value is -2^31. It can be used to store integer values unless there is a need for storing numbers larger or smaller than the limits

Example- int a=56;

b. Float Datatype in Java: float is used for storing decimal values. Its default value is 0.0f and has a size of 4 bytes. It has an infinite value range. However its always advised to use float in place of double if there is a memory constraint. Currency should also never be stored in float datatype. However it has a single precision bit.

Example- float a=98.7f;

c. Character Datatype in Java: char as the name suggests is useful for storing single value characters. Its default value is '\u0000' with the max value being '\uffff' and has a size of 2 bytes.

Example- char a='D';

It must be confusing for you to see this new kind of data '/u000'. This is the unicode format which java uses inplace of ASCII.

d. Boolean Datatype in Java: boolean is a special datatype which can have only two values 'true' and 'false'. It has a default value of 'false' and a size of 1 byte. It comes in use for storing flag values.

Example- boolean flag=true;

e. Byte Datatype in Java: It's an 8 bit signed two's complement . The range of values are -128 to 127. It is space efficient because it is smaller than integer datatype. It can be a replacement for int datatype usage but it doesn't have the size range as the integer datatype.

Example- byte a = 10;

f. Short Datatype in Java: This datatype is also similar to the integer datatype. However it's 2 times smaller than the integer datatype. Its minimum range is -32,768 and maximum range is 32,767. It has a size of

Example- short a= 54;

g. Long Datatype in Java: This datatype primarily stores huge sized numeric data. It is a 64 bit integer and ranges from -2^63 to +(2^63)-1. It has a size of 8 bytes and is useful when you need to store data which is longer than int datatype.

Example- long a= 1273762;

h. Double Datatype in Java: This is similar to the float datatype. However it has one advantage over float datatype i.e, it has two bit precision over the float datatype which has one bit precision. However it still shouldnt be used for precision sensitive data such as currency. It has a range of -2^31 to (2^31)-1.

Example- double DataFlair=99.987d;

Java program to illustrate the different types of datatypes:

```
Class variable{

public static void main(String[] args) throws IOException {

int a = 10;

short s = 2;

byte b = 6;

long l = 1253621332231;

float f = 65.20298f;

double d = 876.765d;

System.out.println("The integer variable is " + a);

System.out.println("The short variable is " + s);

System.out.println("The byte variable is " + b);

System.out.println("The long variable is " + l);

System.out.println("The float variable is " + f);

System.out.println("The double variable is " + d);
```

}}

2. Non-primitive Data Types in Java :These are the datatypes which have instances like objects. Hence they are called reference variables. They are primarily classes, arrays, strings or interfaces.

Non-Primitive Data Types
01
Classes
02
Interfaces
03
Strings
04
Arrays

a. Classes in Java: These are the special user defined data type. It has member variables and class methods. They are blueprinted by objects.

```
class DataFlair {

int a,

b,

c;

void teach() {

System.out.println("Hi I am teaching at DataFlair");

}

void learn() {

System.out.println("Hi I am learning at DataFlair");.

}

}
```

b. Strings in Java: You may be knowing string as a collection of characters but in Java String is a completely different class altogether. It's located in java.lang.String. However, strings end with a ‘\0’ character. Java has a lot of methods for manipulating strings such as substring, length and many more.

Example:

```
String s=”DataFlair is a fun place to learn”;
String sub=s.substring(0,9);
System.out.println(sub);
```

c. Arrays in Java : Arrays are special memory locations that can store a collection of homogeneous data. They are indexed. Arrays always start indexing from 0. Dynamic allocation of arrays is there in Java. Arrays in Java can be passed as method parameters, local variables and static fields.

Example- int arr[] = new int[100];

This creates a storage space for 100 integers.

Java Identifiers: Identifiers are used to name classes, methods and variables. It can be a sequence of uppercase and lowercase characters. It can also contain '_' (underscore) and '$' (dollar) sign. It should not start with a digit(0-9) and not contain any special characters.

Declaring Variables: Variables are also known as identifiers in Java. It is a named memory location which contain a value. We are allowed to declare more than one variable of the same type in a statement.

```
int var=100;
int g;
char c,d; // declaring more than one variable in a statement
```

Explanation:

- In the first line,we are declaring a variable of type *int* and name *var* and initialising it with a value.
- The second line is just the declaration of an integer.
- The third line is declaring two characters in one statement.

Rules for Naming a Variable-

- A variable may contain characters, digits and underscores.
- The underscore can be used in between the characters.
- Variable names should be meaningful which depicts the program logic.

- Variable name should not start with a digit or a special character.
- A variable name should not be a keyword.

Java Keywords: Keywords are also known as reserved words in Java. They carry some predefined meaning and are used throughout the program. They cannot be used as a variable name. Example: abstract,package,class

Java Modifiers: Access Modifiers or Access Specifiers are used to define the scope of classes, methods or constructors. There are 4 types of access modifiers in Java.

1. Default- When no access modifier is specified, then it is said to have the default access modifier. It cannot be accessed from outside the package and is only accessible within that package.
2. Public- The public access modifier is defined using the public keyword. This has the widest scope among access modifiers and this can be accessed anywhere within the program like within the class, outside the class, within the package and outside the package.
3. Private- The private access modifier is defined using the private keyword. Methods or variables defined as private can be accessed within that class and not outside it.
4. Protected- The protected access modifier is defined using the protected keyword. Methods or classes defined as protected can be accessed within that package and outside package by subclasses.

Structure of Java Program

Apart from the above sections, there are some sections that play a crucial role in Java.

Comments: Comments are non-executable parts of a program. A comment is not executed by a compiler. It is used to improve the readability of the code.

Writing comments is a good practice as it improves the documentation of the code. There are 3 types of comments- single line, multi line and documentation comments.

- Single line comment- It is a comment that starts with // and is only used for a single line.

// Single line comment

- Multi line comment- It is a comment that starts with /* and ends with */ and is used when more than one line has to be enclosed as comments.
- Documentation comment- It is a comment that starts with /** and ends with */

Methods : A method is a collection of statements that perform a specific task. The method names should start with a lowercase letter. It provides the reusability of code as we write a method once and use it many times by invoking the method. The most important method is the main() method in Java. In the method declaration, the following are the important components.

- Modifier- Defines access type i.e., the scope of the method.
- The return data type-It is the data type returned by the method or void if does not return any value.
- Method Name- The method names should start with a lowercase letter (convention) and should not be a keyword.
- Parameter list- It is the list of the input parameters preceded with their data type within enclosed parenthesis. If there are no parameters, you must put empty parentheses ().
- Exception list- The exceptions that a method might throw are specified.
- Method body- It is enclosed between braces {}. The code which needs to be executed is written inside them.
- Method signature-It consists of the method name and a parameter list (number, type and order of the parameters). The return data type and exceptions are not a part of it.

EXAMPLE:

```
public void add(int a, int b){
   int c = a + b;
   System.out.println(c);
}
```

Program File Name: So, you must be wondering about the process of naming your file in Java. In Java, we have to save the program file name with the same name as that of the name of public class in that file. The above is a good practice as it tells JVM that which class to load and where to look for the entry point (main method).

The extension should be java. Java programs are run using the following two commands:

```
javac fileName.java // To compile the Java program into byte-code
java fileName // To run the program
```

Arrays: An array is a homogenous non-primitive data type used to save multiple elements (having the same data type) in a particular variable.

Arrays in Java can hold primitive data types (Integer, Character, Float, etc.) and non-primitive data types(Object). The values of primitive data types are stored in a memory location, whereas in the case of objects, it's stored in heap memory.

Assume you have to write a code that takes the marks of five students. Instead of creating five different variables, you can just create an array of lengths of five. As a result, it is so easy to store and access data from only one place, and also, it is memory efficient. An array is a data structure or container that stores the same typed elements in a sequential format.

Instantiating an Array in Java

When an array is declared, only a reference of an array is created. To create or give memory to the array, you create an array like this: The general form of *new* as it applies to one-dimensional arrays appears as follows:

```
var-name = new type [size];
```

Here, type specifies the type of data being allocated, size determines the number of elements in the array, and var-name is the name of the array variable that is linked to the array. To use new to allocate an array, you must specify the type and number of elements to allocate.

Example:

```
int intArray[];    //declaring array

intArray = new int[20];  // allocating memory to array
```

OR

```
int[] intArray = new int[20]; // combining both statements in one
```

Array Literal: In a situation where the size of the array and variables of the array are already known, array literals can be used.

```
 int[] intArray = new int[]{ 1,2,3,4,5,6,7,8,9,10 };

 // Declaring array literal
```

- The length of this array determines the length of the created array.
- There is no need to write the new int[] part in the latest versions of Java.

Accessing Java Array Elements using for Loop: Each element in the array is accessed via its index. The index begins with 0 and ends at (total array size)-1. All the elements of array can be accessed using Java for Loop.

```
 // accessing the elements of the specified array

for (int i = 0; i < arr.length; i++)

  System.out.println("Element at index " + i +

                              " : "+ arr[i]);
```

Implementation: Java

```
  // Java program to illustrate creating an array

  // of integers,  puts some values in the array,

  // and prints each value to standard output.

  class G {

      public static void main(String[] args)

      {

          // declares an Array of integers.

          int[] arr;

          // allocating memory for 5 integers.
```

```
        arr = new int[5];
        // initialize the first elements of the array
        arr[0] = 10;
        // initialize the second elements of the array
        arr[1] = 20;
        // so on...
        arr[2] = 30;
        arr[3] = 40;
        arr[4] = 50;
        // accessing the elements of the specified array
        for (int i = 0; i < arr.length; i++)
            System.out.println("Element at index " + i
                               + " : " + arr[i]);
    }
}
```

Multidimensional Arrays: Multidimensional arrays are arrays of arrays with each element of the array holding the reference of other arrays. These are also known as Jagged Arrays. A multidimensional array is created by appending one set of square brackets ([]) per dimension.

Syntax :

datatype [][] arrayrefvariable;

or

datatype arrayrefvariable[][];

Syntax: Java

```
import java.io.*;
 class GFG {
    public static void main (String[] args) {
      // Syntax
       int [][] arr= new int[3][3];
      // 3 row and 3 column
    }
 }
```

```
public class multiDimensional {
    public static void main(String args[])
    {
        // declaring and initializing 2D array
        int arr[][]
            = { { 2, 7, 9 }, { 3, 6, 1 }, { 7, 4, 2 } };
        // printing 2D array
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++)
                System.out.print(arr[i][j] + " ");
            System.out.println();
        }
    }
}
```

Inheritance:

Inheritance in Java is one of the key features of Object-Oriented Programming. It is a concept by which objects of one class can acquire the behavior and properties of an existing parent class. In simple terms, we can create a new class from an existing class. The newly created class is called subclass (child class or derived class) that can have the same methods and attributes as the existing class (superclass or parent class or base class).

Extends keyword: We use the " extends" keyword to demonstrate Inheritance in Java by creating a derived class from the parent class. The extended keyword is used to indicate that a specific class is being inherited by another class.For example: class A extends B which means class A is being derived from the parent class B.

Some important terms used to define inheritance in Java

- **Class:** A class is a blueprint or a template for creating objects.
- **Subclass:** It is also known as the child class or the derived class that inherits the properties of the parent class.
- **Parent class:** It is also known as the superclass or the base class from which other subclasses inherit the properties.
- **Reusability:** It is a concept by which we can reuse the methods and the fields of an existing class in our newly derived class without actually writing the code again and again.

Example (inheritance in Java): In this Java example code, we have two classes: A (parent class) and B (derived class, inherited from class A). To demonstrate the concept of Inheritance in Java, we have firstly created an object obj of derived class B. We can now access the fields and methods of the parent class with the help of the object of the derived class.

Code:

```
class A {
  String name;
  public void getName(String str) {
    System.out.println("I am " + str);
  }
}
// B is the derived class and A is the parent class
class B extends A {
  public void display() {
    System.out.println("I am derived from class A: " + name);
  }
}
class Main {
  public static void main(String[] args) {
    // Object of derived class B
    B obj = new B();
    obj.name = "Inheritance";
    obj.display();

    obj.getName("learning");

  }

}
```

Some important facts about inheritance in Java

- A subclass can have only one superclass but a superclass can have any number of subclasses.
- A subclass can inherit all the members (methods and fields) of the parent class except the private members.
- The constructors are not considered members of a class and are not inherited by subclasses.
- While constructors are not inherited by subclasses, subclasses can call their parent class's constructors using the super keyword, and a default constructor is automatically created by the Java compiler only if no explicit constructors are defined in the class.

<u>Method Overriding In Java Inheritance:</u> Have you ever wondered what will happen if a specific method is present both in the subclass and the superclass? In such cases, methods of the subclass override the methods of the parent class and hence, this is known as method overriding in Java. For method overriding, methods must have the same name & the same number of parameters in the parent and the child class. This concept is important to achieve run-time polymorphism.

Let's take an example to understand this.

```
// Parent class

class One {

  public void name(String str) {

    System.out.println("I am a member of class One: " + str);

  }

}
```

```
// Subclass derived from class One
class Two extends One {
  @Override
  public void name(String str) { //Overriding the method - name()
    System.out.println("I am a member of class Two: " + str);
  }
}
class Main {
  public static void main(String[] args) {
    // Object of the subclass
    Two obj = new Two();
    obj.name("Inheritance");
  }
}
```

Types Of Inheritance In Java: There are five types of Java Inheritances and now, we will study them with their implementation code in Java :

- Single Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Multiple Inheritance
- Hybrid Inheritance

1. Single Inheritance : In single inheritance, a single subclass is extended from a single superclass and inherits its properties.

```
// Parent class
class One {
  public void print_single() {
    System.out.print("Single ");
  }
}
// Subclass
class Two extends One {
  public void print_inheritance() {
    System.out.println("Inheritance");
  }
}
class Main {
  public static void main(String[] args) {
    Two obj = new Two();
    // Accessing method of superclass
    obj.print_single();
    obj.print_inheritance();
  }
}
```

2. Multilevel Inheritance: In multilevel inheritance, a child class is derived from a parent class and that child class further acts as a parent class and is used to derive another child class. This is a single increasing chain of inheritance.

Code:

```
// Parent class

class One {



  public void print_multi() {

    System.out.print("Multi");

  }

}

// Derived class from class One

class Two extends One {

  public void print_level() {

    System.out.println("level");

  }

}

// Derived class from class Two and

// it can further access members of class One

// which is inherited by class Two

class Three extends Two {

  public void print_inheritance() {
    System.out.println("Inheritance");

  }

}

class Main {

  public static void main(String[] args) {

    Three obj = new Three();



    obj.print_multi();

    obj.print_level();

    obj.print_inheritance();

  }

}
```

3. Hierarchical Inheritance : In Hierarchical Inheritance, one superclass is used to derive more than one subclass or we can say that multiple subclasses extend from a single superclass.

For example:

```
// Parent Class

class A {

  public void print_A() {

    System.out.println("Class A");

  }

}
```

```
// Subclass B derived from class A
class B extends A {
  public void print_B() {
    System.out.println("Class B");
  }
}

// Subclass C derived from class A
class C extends A {
  public void print_C() {
    System.out.println("Class C");
  }
}
class Main {
  public static void main(String[] args) {
    B obj = new B();
    obj.print_A();
    obj.print_B();
    C obj2 = new C();
    obj2.print_A();
    obj2.print_C();
  }

}
```

4. Multiple Inheritance (Not Supported by Java): In Multiple Inheritance, one subclass is derived from more than one superclass. Java doesn't support multiple inheritances because of high complexity and logic issues but as an alternative, we can implement multiple inheritances in Java using interfaces.

For example:

```
// Implementing Multiple Inheritance using interfaces

interface One {

  // Only function definition here

  public void print_Hierarchical();

}


interface Two {

  public void print_Inheritance();

}


interface Three extends One, Two {

  public void print_Implementation();

}


// class implementing interface

class Child implements Three {
  // Body of function definition
  public void print_Hierarchical() {
    System.out.print("Hierarchical ");
  }
  public void print_Inheritance() {
    System.out.print("Inheritance ");
  }
  public void print_Implementation() {
    System.out.print("Implementation");
  }
}
class Main {
  public static void main(String[] args) {
    // Object of Child class to access methods implemented by interfaces
    Child obj = new Child();
    obj.print_Hierarchical();
    obj.print_Inheritance();
    obj.print_Implementation();
  }
}
```

}

5. Hybrid Inheritance: Hybrid Inheritance in Java is a combination of two or more types of inheritance. In simple terms, more than one type of inheritance is being observed in the case of Hybrid inheritance. As we know that Java doesn't support multiple inheritance with classes, similarly we can't implement hybrid inheritance that consists of multiple inheritance in Java.

In the case of demonstrating multiple inheritance in Java, we must have to use **interfaces** for implementation purposes as we have seen in the example of multiple inheritances.

For example:

```
class C {

  public void display() {

    System.out.println("C");

  }

}

class A extends C {

  public void display() {

    System.out.println("A");

  }

}

class B extends C {

  public void display() {

    System.out.println("B");

  }

}
```

```
class D extends A {
  public void display() {
    System.out.println("D");
  }
}
class Main {
  public static void main(String args[]) {
    D obj = new D();
    obj.display();
  }
}
```

Encapsulation: Data Encapsulation can be defined as wrapping the code or methods(properties) and the related fields or variables together as a single unit. In object-oriented programming, we call this single unit - a class, interface, etc. We can visualize it like a medical capsule (as the name suggests, too), wherein the enclosed medicine can be compared to fields and methods of a class.
The variables or fields can be accessed by methods of their class and can be hidden from the other classes using private access modifiers. One thing to note here is that data hiding and encapsulation may sound the same but different.
Example of Data Encapsulation: Below is a program to calculate the perimeter of a rectangle

```
class Perimeter {
  int length;
  int breadth;
  Perimeter(int length, int breadth) {
    this.length = length;
    this.breadth = breadth;

  }

  public void getPerimeter() {

    int perimeter = 2 * (length +  breadth);

    System.out.println("Perimeter of Rectangle : " + perimeter);

  }

}

class Main {

  public static void main(String[] args) {

    Perimeter rectangle = new Perimeter(3, 6);

    rectangle.getPerimeter();

  }
}
```

Data Hiding

Access modifiers are used to achieve data hiding. Access modifiers are the keywords that specify the accessibility or the scope of methods, classes, fields, and other members.

Difference between encapsulation and data hiding is that Encapsulation is a way of bundling data whereas Data Hiding prevents external unauthorized access to the sensitive data.

The four types of access modifiers in Java are:

- Public: A public modifier is accessible to everyone. It can be accessed from within the class, outside the class, within the package, and outside the package.

```
class A {

 // public variables

 public int a;
 public int b;

 // public methods

 public void method_1() {

    // Method logic

 }}
```

- **Private**: A private modifier can not be accessed outside the class. It provides restricted access. Example:

```
class A {

  // private variables

  private String a;

  private int b;

}
```

- **Protected**: A protected modifier can be accessed from the classes of the same package and outside the package only through inheritance.

```
class A {

   // protected variables

   protected String a;

   protected int b;

}
```

A Private Access Modifier is used for the purpose of Data Hiding. Example of Data Hiding:

```
class Employee {

  // private variables
  private String name;

  private int age;

  private int salary;

}

public class Main {


  public static void main(String[] args) {

    //object of class Employee

    Employee newObj = new Employee();

    newObj.name = "James";

    System.out.println(newObj.name);

  }
}
```

Getter and Setter Methods: As we can't directly read and set the values of private variables/fields, Java provides us with getter and setter methods to do so.

How to implement Encapsulation in Java?

We need to perform two steps to achieve the purpose of Encapsulation in Java.

- Use the private access modifier to declare all variables/fields of class as private.
- Define public getter and setter methods to read and modify/set the values of the abovesaid fields.

```
class Book {

  //private fields

  private String author;

  private String title;

  private int id;
  //public getter and setter  methods for each field
  public String getAuthor() {
    return author;
  }
  public void setAuthor(String a) {
    this.author = a;
  }
  public String getTitle() {
    return title;
  }
  public void setTitle(String t) {
    this.title = t;
  }
  public int getId() {
    return id;
  }
  public void setId(int i) {
    this.id = i;
  }
}
public class Main {
 public static void main(String[] args) {

  Book newObj = new Book();

  newObj.setAuthor("Jane Austen");



  newObj.setTitle("Pride and Prejudice");

  newObj.setId(775);

    System.out.println("Book Title: " +newObj.getTitle() +"\n" + "Book
Author: " +newObj.getAuthor() + "\n" +"Book Id: " +newObj.getId() );

 }

}
```

Benefits of Encapsulation java

- Cleaner, more organized and less complex code.
- More flexible code as can modify a unit independently without changing any other unit.
- Makes the code more secure.
- The code can be maintained at any point without breaking the classes that use the code.

Polymorphism: Polymorphism is one of the main aspects of Object-Oriented Programming(OOP). The word polymorphism can be broken down into Poly and morphs, as “Poly” means many and “Morphs” means forms.
Let us understand the definition of polymorphism by an example; a lady can have different characteristics simultaneously. She can be a mother, a daughter, or a wife, so the same lady possesses different behavior in different situations.
Another example of polymorphism can be seen in carbon, as carbon can exist in many forms, i.e., diamond, graphite, coal, etc. We can say that both woman and carbon show different characteristics at the same time according to the situation. This is called polymorphism.
The definition of polymorphism can be explained as performing a single task in different ways. A single interface having multiple implementations is also called polymorphism.

How can polymorphism be achieved in Java?

Polymorphism in Java can be achieved in two ways i.e., method overloading and method overriding.
Polymorphism in Java is mainly divided into two types.

- Compile-time polymorphism
- Runtime polymorphism

Compile-time polymorphism can be achieved by method overloading, and Runtime polymorphism can be achieved by method overriding.

Runtime Polymorphism: Runtime polymorphism is also called Dynamic method dispatch. Instead of resolving the overridden method at compile-time, it is resolved at runtime.

Here, an overridden child class method is called through its parent's reference. Then the method is evoked according to the type of object. In runtime, JVM figures out the object type and the method belonging to that object.

Runtime polymorphism in Java occurs when we have two or more classes, and all are interrelated through inheritance. To achieve runtime polymorphism, we must build an "IS-A" relationship between classes and override a method.

Method overriding

If a child class has a method as its parent class, it is called method overriding. If the derived class has a specific implementation of the method that has been declared in its parent class is known as method overriding.

Rules for overriding a method in Java

1. There must be inheritance between classes.
2. The method between the classes must be the same(name of the class, number, and type of arguments must be the same).

```
// Parent class to illustrate run-time polymorphism

class Parent {

  // creating print method

  void print() {

    System.out.println("Hi I am parent");
  }

}

// Child class extends Parent class

class Child extends Parent {

  // overriding print method

  void print() {

    System.out.println("Hi I am children");

  }

}
```

Compile-time polymorphism: This type of polymorphism in Java is also called static polymorphism or static method dispatch. It can be achieved by method overloading. In this process, an overloaded method is resolved at compile time rather than resolving at runtime.
Method overloading: Consider a class where multiple methods have the same name. It will be difficult for the compiler to distinguish between every method.
To overcome this problem, we pass a different number of arguments to the method or different types of arguments to the method. In this way, we achieve method overloading.
In other words, a class can have multiple methods of the same name, and each method can be differentiated either by bypassing different types of parameters or bypassing a different number of parameters.

```
// Overload class to illustrate compile-time polymorphism

class Overload {

  // Creating a statement method

  void statement(String name) {

    System.out.println("Hi myself " + name);

  }
  // overloading statement method

  void statement(String fname, String lname) {

    System.out.println("Hi myself " + fname + " " + lname);

  }

}

public class Main {

  public static void main(String[] args) {

    // creating instance of parent

    Parent obj1;

    obj1 = new Parent();

    obj1.print();

    obj1 = new Child();

    obj1.print();

    // creating instance of overload

    Overload obj2 = new Overload();

    obj2.statement("Soham.");

    obj2.statement("Soham", "Medewar.");

  }
}
```

Abstraction: Abstraction in Java is a process of hiding the implementation details from the user and showing only the functionality to the user. It can be achieved by using abstract classes, methods, and interfaces. An abstract class is a class that cannot be instantiated on its own and is meant to be inherited by concrete classes.

An abstract method is a method declared without an implementation. Interfaces, on the other hand, are collections of abstract methods.

Abstraction is a key concept in OOP and in general as well. Think about real world objects, they are made by combining raw materials like electrons, protons, and atoms, which we don't see due to the abstraction that nature exposes to make the objects more understandable. Similarly, in computer science, abstraction is used to hide the complexities of hardware and machine code from the programmer.
This is achieved by using higher-level programming languages like Java, which are easier to use than low-level languages like assembly.
Abstraction in Java can be achieved using the following tools it provides :

- Abstract classes
- Interface

Interface:

An interface is a fully abstract class. It includes a group of abstract methods (methods without a body).
We use the interface keyword to create an interface in Java. For example,

```
interface Language {
  public void getType();

  public void getVersion();
}
```

Here,

- Language is an interface.
- It includes abstract methods: getType() and getVersion(

Implementing an Interface

Like abstract classes, we cannot create objects of interfaces.
To use an interface, other classes must implement it. We use the implements keyword to implement an interface.
Example 1: Java Interface

```
interface Polygon {
  void getArea(int length, int breadth);
}

// implement the Polygon interface
```

```
class Rectangle implements Polygon {

  // implementation of abstract method
  public void getArea(int length, int breadth) {
    System.out.println("The area of the rectangle is " + (length * breadth));
  }
}

class Main {
  public static void main(String[] args) {
    Rectangle r1 = new Rectangle();
    r1.getArea(5, 6);
  }
}
```

Output

```
The area of the rectangle is 30
```

In the above example, we have created an interface named Polygon. The interface contains an abstract method getArea().

Here, the Rectangle class implements Polygon. And, provides the implementation of the getArea() method.

Similar to classes, interfaces can extend other interfaces. The extends keyword is used for extending interfaces. For example,

```
interface Line {
  // members of Line interface
}

// extending interface
interface Polygon extends Line {
  // members of Polygon interface
  // members of Line interface
}
```

Here, the Polygon interface extends the Line interface. Now, if any class implements Polygon, it should provide implementations for all the abstract methods of both Line and Polygon.

Advantages of Interface in Java

Now that we know what interfaces are, let's learn about why interfaces are used in Java.

- Similar to abstract classes, interfaces help us to achieve abstraction in Java. Here, we know getArea() calculates the area of polygons but the way area is calculated is different for different polygons. Hence, the implementation of getArea() is independent of one another.
- Interfaces provide specifications that a class (which implements it) must follow.
  In our previous example, we have used getArea() as a specification inside the interface Polygon.
  This is like setting a rule that we should be able to get the area of every polygon.
  Now any class that implements the Polygon interface must provide an implementation for the getArea() method.
- Interfaces are also used to achieve multiple inheritance in Java.

Abstract class:

Generally, an abstract class in Java is a template that stores the data members and methods that we use in a program. Abstraction in Java keeps the user from viewing complex code implementations and provides the user with necessary information.
We cannot instantiate the abstract class in Java directly. Instead, we can subclass the abstract class. When we use an abstract class as a subclass, the abstract class method implementation becomes available to all of its parent classes.
The important rules that we need to follow while using an abstract class in Java are as follows:

- The keyword "abstract" is mandatory while declaring an abstract class in Java.
- Abstract classes cannot be instantiated directly.
- An abstract class must have at least one abstract method.
- An abstract class includes final methods.
- An abstract class may also include non-abstract methods.
- An abstract class can consist of constructors and static methods.

Program:

```
// 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();
    }
}
```

Difference between interface and abstract class:

Interface	Abstract Class
Keyword used: interface	Keyword used: abstract

Subclasses can implement an interface	Subclasses have to extend abstract class
Multiple interfaces can be implemented	One abstract class can be extended
Supports Multiple Inheritance	Cannot support Multiple Inheritance

Now that the differences between an interface and abstract class are clear, let us move forward. The next part will explore the crucial advantages and disadvantages that we must consider while using an abstract class in Java

Packages:

A Java package is a collection of similar types of sub-packages, interfaces, and classes. In Java, there are two types of packages: built-in packages and user-defined packages. The package keyword is used in Java to create Java packages.
Let's find out why we even need packages in the first place. Say you have a laptop and a bunch of data you want to store. Data includes your favorite movies, songs, and images.
So, do you store them all in a single folder or make a separate category for each one and store them in their corresponding folder?
It is obvious that anyone would like to create separate folders for images, videos, songs, movies, etc. And the reason is the ease of accessibility and manageability.

How does Packages in Java improve accessibility and manageability?

If you had clustered everything inside a single folder, you would probably spend a lot of time finding the correct file. But, separating your files and storing them in the correct folder saves you a lot of time.
This way, we're improving accessibility. Accessibility and manageability go kind of hand in hand. Managing a file is related to how easy it's to modify a specified file to incorporate future needs.
Though management includes other factors as well, like how well the code is written, having those files in a structured format is an essential requirement.

You can think of it like that – “What if we write this article in a not-so-structured format and combine everything inside a single paragraph?” You obviously will have a hard time reading it. Isn’t it?
So using the same analogy in application development, but here, packages in Java provide little more than that. In simple words, packages in Java are nothing but a folder that contains related files.
These files could be of type Java classes or interfaces or even sub-packages (don’t forget the storing data inside the laptop analogy).

Types of Packages in Java
Packages in Java can be categorised into 2 categories.

1. Built-in / predefined packages
2. User-defined packages.

Built-in packages
When we install Java on a personal computer or laptop, many packages are automatically installed. Each of these packages is unique and capable of handling various tasks. This eliminates the need to build everything from scratch. Here are some examples of built-in packages:

- java.lang
- java.io
- java.util
- java.applet
- java.awt
- java.net

Let’s see how you can use an inbuilt package in your Java file.

```
import java.lang.*;

public class Example {

  public static void main(String[] args) {
    double radius = 5.0;
    double area = Math.PI * Math.pow(radius, 2);
    System.out.println("Area: " + area);

    String message = "Hello, World!";
    int length = message.length();
    System.out.println("Length: " + length);

    int number = 42;
    String binary = Integer.toBinaryString(number);
    System.out.println("Binary: " + binary);
  }
}
```

User-defined packages

User-defined packages are those that developers create to incorporate different needs of applications. In simple terms, User-defined packages are those that the users define. Inside a package, you can have Java files like classes, interfaces, and a package as well (called a sub-package).

Sub-package:A package defined inside a package is called a sub-package. It's used to make the structure of the package more generic. It lets users arrange their Java files into their corresponding packages. For example, say, you have a package named cars. You've defined supporting Java files inside it.

Extended and tagging:

S.No.	Extends	Implements
1.	By using "extends" keyword a class can inherit another class, or an interface can inherit other interfaces	By using "implements" keyword a class can implement an interface
2.	It is not compulsory that subclass that extends a superclass override all the methods in a superclass.	It is compulsory that class implementing an interface has to implement all the methods of that interface.
3.	Only one superclass can be	A class can implement any number

S.No.	**Extends**	**Implements**
	extended by a class.	of an interface at a time
4.	Any number of interfaces can be extended by interface.	An interface can never implement any other interface

Implements: In Java, the implements keyword is used to implement an interface. An interface is a special type of class which implements a complete abstraction and only contains abstract methods. To access the interface methods, the interface must be "implemented" by another class with the implements keyword and the methods need to be implemented in the class which is inheriting the properties of the interface. Since an interface is not having the implementation of the methods, a class can implement any number of interfaces at a time.

Extends: In Java, the extends keyword is used to indicate that the class which is being defined is derived from the base class using inheritance. So basically, extends keyword is used to extend the functionality of the parent class to the subclass. In Java, multiple inheritances are not allowed due to ambiguity. Therefore, a class can extend only one class to avoid ambiguity.