**Unit-1 java**

**Java :**

Java is a versatile, object-oriented, and platform-independent programming language. It was developed by James Gosling and his team at Sun Microsystems in the early 1990s and has since become one of the most widely used programming languages in the world.

**some key characteristics of Java:**

**Platform Independence (Write Once, Run Anywhere):**

Java is designed to be platform-independent, meaning that Java code can run on any device or platform with a Java Virtual Machine (JVM). This is achieved through the compilation of Java source code into an intermediate bytecode, which is then executed by the JVM.

**Object-Oriented Programming (OOP):**

Java follows the principles of object-oriented programming, emphasizing the use of classes and objects. This approach promotes modularity, reusability, and flexibility in code design.

**Robust and Secure:**

Java is designed with features like automatic memory management and strong type-checking, enhancing the robustness of applications. Security features, such as applet sandboxing, contribute to a secure computing environment.

**Rich Standard Library:**

Java comes with a comprehensive standard library that provides a wide range of pre-built classes and packages. This rich library simplifies common programming tasks and facilitates the development process.

**Community and Ecosystem:**

Java has a large and active developer community, contributing to an extensive ecosystem of libraries, frameworks, and tools. This community support enhances collaboration, problem-solving, and the availability of resources for developers.

**Scalability:**

Java's scalability makes it suitable for a variety of applications, from small mobile apps to large-scale enterprise systems, making it a versatile choice for developers.

**Multi-threading Support:**

Java provides built-in support for multithreading, allowing developers to create concurrent and parallel applications. This is essential for optimizing performance in applications that need to handle multiple tasks simultaneously.

**Community Support:**

Java benefits from a vibrant global community that fosters knowledge sharing, collaboration, and resource availability for troubleshooting and learning.

**Wide Range of Applications:**

Java is used in various domains, including web development, mobile app development (especially for Android), enterprise applications, scientific computing, and more. Its versatility makes it suitable for a broad range of projects.

**Memory Management:**

Java incorporates automatic memory management through a process known as garbage collection. This feature helps developers by automatically deallocating memory occupied by objects that are no longer in use, reducing the risk of memory leaks.

**JDK :**

JDK stands for Java Development Kit. It is a software development kit used by Java developers to build, compile, and run Java applications. The JDK includes tools and utilities necessary for Java development, such as the Java compiler (javac), the Java Virtual Machine (JVM), and various libraries and frameworks.

**=>JDK** Includes the Java compiler (javac), debugger, and other development tools.

**Key Elements:**

**Java Environment:**

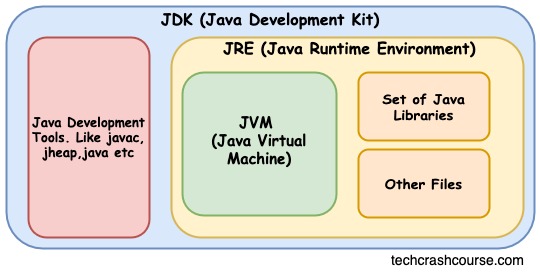
The Java environment refers to the combination of hardware and software components that enable the development and execution of Java applications.

**JRE (Java Runtime Environment):**

JRE is a software package that provides the necessary runtime components for executing Java applications. It includes the JVM, libraries, and other components required for running Java programs.

**JVM (Java Virtual Machine):**

The JVM is a virtual machine that executes Java bytecode. It provides a runtime environment for Java applications, interpreting or compiling Java bytecode into machine code suitable for a specific platform.



**Java Development Tools:** IDEs (Integrated Development Environments) like Eclipse or IntelliJ IDEA ,VS code .

**Java Class Library:** A set of pre-compiled classes and methods used by Java applications.

**Java Source code File Structure:**

**File Extension:** Java source files have a ".java" extension.

**Package Declaration (Optional):** Specifies the package to which the class belongs.

**Import Statements (Optional):** Include references to classes from other packages.

**Class Declaration:** Contains the class definition, main method, and other members.

**Methods and Fields:** Define the behavior and attributes of the class.

**Example of first java code:**

public class XYZ {

public static void main(String[] args) {

System.out.println("Hello, World!");

}

}

**Example-2 of java code :**

package xyz; **// Package declaration**

import java.util.\*; **// import statement**

public class MyClass { **// Class declaration**

**// Fields**

private int myField;

**// Constructor**

public MyClass(int value) {

this.myField = value;

}

**// Method**

public void myMethod() {

System.out.println("this is a method");

}

**// Main method & Driver code**

public static void main(String[] args) {

MyClass myObject = new MyClass(42);

myObject.myMethod();

}

}

**Compilation in java :**

**Compilation Process:**

**Step 1:** Write Java source code in a text editor or an IDE.

**Step 2:** Save the source code with a "class\_name.java" extension.

**Step 3:** Open a command prompt or terminal.

**Step 4:** Navigate to the directory containing the source file.

**Step 5:** Use the javac command to compile the source file into bytecode

javac MyClass.java

**Step 6:** Compilation generates a ".class" file containing bytecode.

**Step 7:** Execute the program using the java command.

java class\_name

**Method :**

In Java, a method is a block of code that performs a specific task or action. Methods are defined within classes and are used to encapsulate functionality, promote code reusability, and organize code into manageable units.

**Example:**

public class MyClass {

// Method definition

public static void printMessage(String message) {

System.out.println(message);

}

public static int f1(int a, int b) {

return a+b;

}

**public  static int gcd(int a, int b) {**

        int gcd = 1;

        int min = (a < b) ? a : b;

        int i = 2;

        while (i < min) {

            if (a % i == 0 && b % i == 0) {

                gcd = i;

            }

            i++;

        }

        return gcd;

    }

public static void main(String[] args) {

// Method invocation or calling

printMessage("Hello, world!"); // Output: Hello, world!

gcd(15,24);

}

}

**Constructors :**

In Java, constructors are special methods within a class that are responsible for initializing the object's state when an instance of the class is created. if you do not explicitly provide a constructor, Java generates a default constructor

**1.Default Constructor:**

A default constructor is automatically provided by Java if you do not define any constructor in your class.It takes no parameters and performs basic initialization.

class MyClass {

// Default Constructor

MyClass() {

System.out.println("hello");

}

}

public class Main

{

public static void main(String[] args) {

MyClass ob=new MyClass(); }}

**2.Parameterized Constructor:**

A parameterized constructor is one that takes parameters to initialize the object's state.

**class car {**

**String model;**

**int year;**

**car(String model, int year) {**

**this.model = model;**

**this.year = year;**

**}**

**}**

**public class Main**

**{**

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

**car ob=new car("ajay",22);**

**System.out.println(ob.model);//ajay**

**System.out.println(ob.year);//22 }**

**}**

this constructor allows you to create an instance of the **Car** class and set its **model** and **year** attributes during object creation.

**Constructor Overloading :**

Java supports constructor overloading, allowing you to define multiple constructors with different parameter lists but same name .

**Example :**

class car {

String model;

int year;

car(String model, int year) {

this.model = model;

this.year = year;

}

car(String model) {

this.model = model;

}

car(){

System.out.println("no parameter");

}

}

public class Main{

public static void main(String[] args) {

car ob=new car("ajay",22);

System.out.println(ob.model);//ajay

System.out.println(ob.year);//22

car ob1=new car("vijay");

System.out.println(ob1.model);//vijay

car ob2=new car();

}}

**Constructor Chaining:**

Constructors can call other constructors in the same class using this().

**Example:**

class car {

String model;

int year;

car(String model, int year) {

this();

this.model = model;

this.year = year;

}

car(){

System.out.println("no parameter");

}

}

public class Main

{

public static void main(String[] args) {

car ob=new car("ajay",22);

System.out.println(ob.model);//ajay

System.out.println(ob.year);//22

}}

**Comment:**

In Java, comments are used to add explanatory notes or remarks within the source code. Comments are ignored by the Java compiler and have no impact on the execution of the program. They are purely for the benefit of programmers to improve code readability and understanding. There are three types of comments in Java

**1.Single-Line Comments:**

Single-line comments start with // and extend to the end of the line.

They are used for short comments or annotations.

**Example:**

int x = 5; // Variable declaration

**2.Multi-Line Comments:**

They are used for longer comments or for commenting out multiple lines of code.

Example:

/\*

\* This is a multi-line comment.

\* It can span multiple lines.

\*/

int y = 10;

**3.Documentation Comments (Javadoc):**

Documentation comments start with /\*\* and end with \*/.

They are used for generating documentation automatically. Javadoc comments are commonly placed above classes, methods, and fields to provide information for documentation tools. Javadoc comments include tags like @param and @return to document parameters and return values. These comments can be processed by tools like Javadoc to generate API documentation

**Example:**

public class MyClass {

/\*\*

\* It provides information about the method.

\* @param num1 The first operand

\* @param num2 The second operand

\* @return The sum of num1 and num2

\*/

public int add(int num1, int num2) {

return num1 + num2;

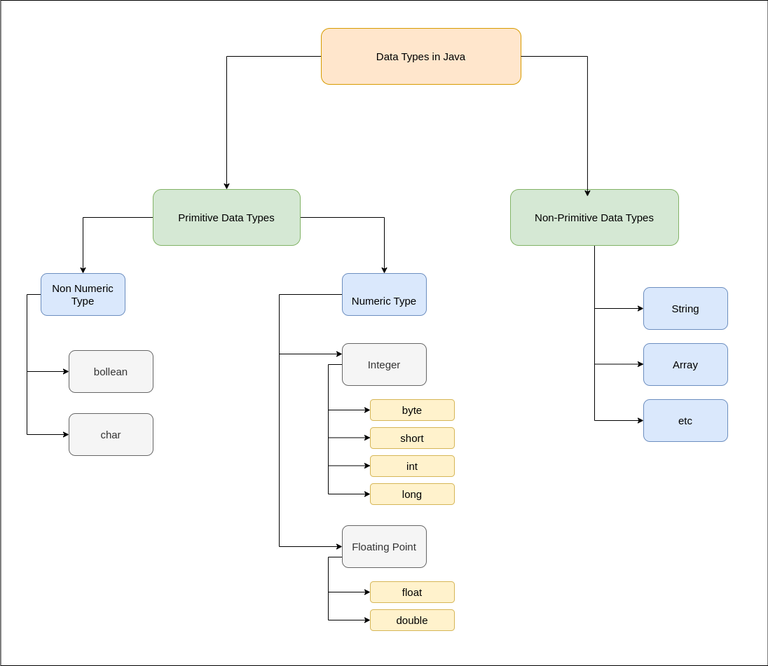
}

}

**data type :**

A data type is a classification that specifies which type of value a variable can hold in a programming language. It defines the characteristics of the data, such as the range of values it can take, the operations that can be performed on it, and the way it is stored in memory. Data types are fundamental to programming languages because they help in organizing and manipulating data efficiently.

Java is a statically-typed language, which means that you must declare the type of a variable before using it.



**1.Primitive Data Types:**

These are basic data types that represent single values.There are eight primitive data types in Java

**Integral Types:**

byte: 8-bit signed integer.

short: 16-bit signed integer.

int: 32-bit signed integer.

long: 64-bit signed integer.

int age = 25;

**Floating-Point Types:**

float: 32-bit floating-point.

double: 64-bit floating-point.

double salary = 50000.75;

**Character Type:**

char: 16-bit Unicode character.

char grade = 'A';

**Boolean Type:** boolean Represents true or false values.

boolean is Student = true;

**2.Non-Primitive Data Type or Reference Data Types :**

Reference data types are used to refer to objects.The Reference Data Types will contain a memory address of variable values because the reference types will not store the variable value directly in memory. They are strings, objects, arrays, interfaces etc.

**Example:**

String name = "John";

int[] numbers = {1, 2, 3, 4, 5};

The String class is an example of a reference data type. Even though String behaves like a primitive type in many scenarios, it is technically a class.

**Derived Data Types:**

These types are derived from primitive or reference types.

Examples include arrays, classes, and interfaces.

Example:

int[] numbers = {1, 2, 3, 4, 5}; // Array (Reference type)

class Person {

String name;

int age;

}

Person personObj = new PersonObj();

son(); // Class (Reference type)

**Arrays:**

An array is a collection of elements of the same data type.

Arrays can be of primitive types or reference types.

**Example:**

int[] numbers = {1, 2, 3, 4, 5};

String[] names = {"ajay", "vijay", "suraj"};

These data types provide the building blocks for writing Java programs. The choice of data type depends on the kind of values you need to represent and manipulate in your application.

**Variable:**

In Java, a variable is a named storage location that holds a value of a specific data type.

**Declaration:**

Before using a variable, it must be declared. Declaration involves specifying the data type and the name of the variable.

**Example:**

int age; // Declaration of an integer variable named 'age'

**Initialization:**

Initialization is the process of assigning an initial value to a variable at the time of declaration.

**Example:**

int age = 25; // Declaration and initialization of an integer variable named 'age’.

**Type of variable:**

**1.Local variables :**

Local variables in Java are variables that are declared within a method, constructor, or block.

key characteristics and details of local variables:

**Scope:**

Local variables are accessible only within the method, constructor, or block in which they are declared.They exist only within the scope of the block in which they are defined and cannot be accessed from outside that block.

**Lifetime:**

Local variables are created when the block they are declared in is entered and destroyed when the block is exited.

They have a shorter lifetime compared to instance variables or static variables, existing only as long as the block they are declared in is executing.

**Initialization:**

Local variables must be explicitly initialized before they are used. Unlike instance variables, they do not have default values.

**Accessing Local Variables:**

Local variables can only be accessed within the block they are declared in.

They are typically used for temporary storage of data within a method or block.

Example:

public class Main {

public static void main(String[] args) {

int localVar = 10;

System.out.println(localVar);//10

int result = localVar \* 2;

System.out.println(result);//20

}

}

**2.Instance variables:**

Instance variables also known as non-static variables, in Java are variables that are declared within a class but outside any method, constructor, or block.

=>Instance variables are associated with instances (objects) of the class. This means that each object created from the class will have its own copy of instance variables.

=>When an object of the class is instantiated (created), memory is allocated for each instance variable associated with that object.

=>Each object has its own set of instance variables, which can have different values.

**Initialization:**

=>Instance variables are initialized when an object is created, either with default values or with specific values provided by constructors or initialization blocks.

=>If not explicitly initialized, instance variables are initialized to default values (0 for numeric types, false for boolean, and null for reference types).

**Accessing Instance Variables:**

=>Instance variables can be accessed using the dot (.) operator with an instance of the class.

=>They cannot be accessed from static methods directly unless a reference to an object of the class is used.

**Example:**

public class Main {

int instanceVar; // Instance variable

public static void main(String[] args) {

Main obj1 = new Main();

Main obj2 = new Main();

obj1.instanceVar = 10;

obj2.instanceVar = 20;

System.out.println(obj1.instanceVar);//10

System.out.println(obj2.instanceVar);//20s

}}

**3. Static variables:**

It also known as class variables, in Java are variables that are declared with the static keyword within a class but outside any method, constructor, or block.

**Associated with the Class:**

Static variables are associated with the class itself rather than with instances of the class. This means there is only one copy of a static variable per class, regardless of how many objects (instances) are created from it.

**Shared Across All Instances:**

Since static variables belong to the class, they are shared among all instances (objects) of the class. Any change made to a static variable by one instance will be reflected in all other instances of the class.

**Initialization:**

Static variables are initialized only once, at the start of the execution of the program, before the initialization of any instance variables.

If not explicitly initialized, static variables are initialized to default values (0 for numeric types, false for boolean, and null for reference types).

**Accessing Static Variables:**

Static variables can be accessed using the class name followed by the dot (.) operator, without the need for an instance of the class.

They can also be accessed directly within other static methods of the class.

Example:

public class Main {

static int staticVar; // Static variable

public static void main(String[] args) {

Main.staticVar = 10;

System.out.println(Main.staticVar);//10

System.out.println(staticVar);//10

}

}

**operators:**

In Java, operators are special symbols that represent computations or operations on variables and values. Java supports a variety of operators, which can be categorized into different types.

**1.Arithmetic Operators:**

It Perform basic mathematical operations. Addition +, Subtraction -, Multiplication \*, Division /, Modulus % (remainder after division).

**Example:**

int a = 10;

int b = 3;

int sum = a + b; // 13

int difference = a - b; // 7

int product = a \* b; // 30

int quotient = a / b; // 3

int remainder = a % b; // 1

**2.Comparison Operators:**

It Compare two values and return a boolean result.

Equal to ==, Not equal to !=, Greater than >, Less than <, Greater than or equal to >=, Less than or equal to <=.

**Example:**

int x = 5;

int y = 8;

boolean n1= (x == y); // false

boolean n2 = (x != y); // true

boolean n3 = (x > y); // false

**3.Logical Operators:**

It Perform logical operations on boolean values.

Logical AND &&, Logical OR ||, Logical NOT ! .

**Example:**

boolean condition1 = true;

boolean condition2 = false;

boolean andResult = (condition1 && condition2); // false

boolean orResult = (condition1 || condition2); // true

boolean notResult = !condition1; // false

**4. Assignment Operators:**

It Assign values to variables.

Assignment =, Addition assignment +=, Subtraction assignment -=, Multiplication assignment \*=, Division assignment /=, Modulus assignment %=.

**Example:**

int a=5;

a+=5;

a-=5;

a\*=5;

a/=5;

a%=2;

System.out.println(a);//1

**5. Increment and Decrement Operators:**

**Increment Operator (++):** It adds 1 to the value of a variable. x++ increments the value of x after using its current value, while ++x increments x before using its value.

**Decrement Operator (--):** It subtracts 1 from the value of a variable. x-- decrements the value of x after using its current value, while --x decrements x before using its value.

**Example:**

public class Main

{

public static void main(String[] args) {

int a=5;

int res1=++a + ++a +a-- + a--; //6+7+7+6=26

int res2=a++ + a++ + --a + --a;//5+6+6+5=22

System.out.println(res1);//26

System.out.println(res2);//22

int a1=5;

System.out.println(a1++);//5

System.out.println(++a1);//7

System.out.println(a1--);//7

System.out.println(--a1);//5 }}

**6.bitwise operators:**

In Java, bitwise operators are used to perform operations on individual bits of integer operands. Java provides several bitwise operators:

1. **bitwise AND (&)**: This operator performs a bitwise AND operation. It returns a 1 in each bit position where both operands have a 1.
2. **bitwise OR (|)**: This operator performs a bitwise OR operation. It returns a 1 in each bit position where either operand has a 1.
3. **bitwise XOR (^)**: This operator performs a bitwise XOR (exclusive OR) operation. It returns a 1 in each bit position where the corresponding bits of the two operands are different.
4. **bitwise NOT (~)**: This operator performs a bitwise complement operation. It flips each bit of the operand, i.e., changes 1 to 0 and 0 to 1.
5. **bitwise Left Shift (<<)**: This operator shifts the bits of the left-hand operand to the left by a number of positions specified by the right-hand operand. It fills the vacant bits with zeroes.
6. **bitwise Right Shift (>>)**: This operator shifts the bits of the left-hand operand to the right by a number of positions specified by the right-hand operand. It fills the vacant bits differently depending on whether it's a logical or arithmetic shift.

**Example:**

public class Main

{

public static void main(String[] args) {

int a=5;

int b=3;

System.out.println(a|b);//7

System.out.println(a&b);//1

System.out.println(a^b );//6

System.out.println(~a);//-6

System.out.println(5<<2);//20

System.out.println(20>>2);//5

System.out.println(~-4)//3

}

}

**6.Conditional Operator (Ternary Operator):**

It is a shorthand way to write an if-else statement.

**Syntax:** condition ? expression\_if\_true : expression\_if\_false

**Example:**

int a = 10;

int b = 5;

int max = (a > b) ? a : b;

System.out.println(max);//10

**Conditional Statements & control flow :**

control flow refers to the order in which statements are executed in a program. Java provides several constructs for controlling the flow of execution, including conditional statements, looping constructs, and branching statements.

**Conditional Statements:**

**Example :**

if (condition1) {

// code to be executed if condition1 is true

} else if (condition2) {

// code to be executed if condition2 is true

} else {

// code to be executed if none of the conditions are true

}

**Example-2:**

**Q. Write a java program that uses nested if statements to check if x is greater than 5 and if y is greater than 2. If both conditions are met, print 'x is greater than 5, and y is also greater than 2.' If x is not greater than 5, print 'x is not greater than 5.?**

**public class Main {**

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

**int x = 6;**

**int y = 3;**

**if (x > 5) {**

**if (y > 2) {**

**System.out.println("x is greater than 5, and y is also greater than 2.");**

**} else {**

**System.out.println("x is greater than 5, but y is not greater than 2.");**

**}**

**} else {**

**System.out.println("x is not greater than 5.");**

**}**

**}**

**}**

**//output**

**x is greater than 5, and y is also greater than 2.**

**Switch Statement:**

The switch statement in Java is a control flow statement used to select one of many code blocks to execute, based on the value of a given expression. It provides a convenient way to write multi-way branches, making code more readable and concise compared to using multiple if-else statements.

**Example-1:**

public class Main {

public static void main(String[] args) {

int day = 3;

String dayString;

switch (day) {

case 1:

dayString = "Monday";

break;

case 2:

dayString = "Tuesday";

break;

case 3:

dayString = "Wednesday";

break;

case 4:

dayString = "Thursday";

break;

default:

dayString = "Invalid day";

break;

}

System.out.println("Today is " + dayString);//

}

}

**Example-2:**

int x=3;

switch(x){

default:System.out.println("default");

case 0: System.out.println("0");break;

case 1: System.out.println("1");

case 2: System.out.println("2");

// output

default

0

But for x= 1 it will be 1 ,2

**Looping Constructs:**

a loop is a control structure that allows you to execute a block of code repeatedly based on a certain condition.

**for Loop:** The for loop is used when it is known in advance how many times you want to execute a block of code.

for (initialization; condition; update) {//

}

**while Loop:**The while loop is used when you want to execute a block of code as long as a condition is true.

Value initialization

while (condition) {

// code to be executed repeatedly as long as the condition is true

}

**do-while Loop:**The do-while loop is similar to the while loop, but the condition is checked after the block of code is executed, so the block of code will always execute at least once.

Value initialization

do {

// code to be executed at least once, then repeatedly as long as the condition is true

} while (condition);

**Example:**

**public class PrintOddNumbers {**

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

**// Using a for loop with if statement**

**for (int i = 1; i <= 10; i++) {**

**if (i % 2 != 0) {**

**System.out.println(i);}**

**}**

**// Using a while loop with if statement**

**int j = 1;**

**while (j <= 10) {**

**if (j % 2 != 0) {**

**System.out.println(j);**

**}**

**j++;**

**}**

**// Using a do-while loop with if statement**

**int k = 1;**

**do {**

**if (k % 2 != 0) {**

**System.out.println(k);**

**}**

**k++;**

**} while (k <= 10); }**

**}**

**Branching Statements:**

**break Statement:**

it is Used to exit a loop or switch statement. The break statement is used to exit the current loop prematurely.

**Example:**

public class Main

{

public static void main(String[] args) {

for (int i = 0; i <=5; i++) {

if (i == 3) {

break; // exits the loop when i equals 3

}

System.out.print(i+" ");// 0,1,2

}

}

**continue Statement:**

it Skips the rest of the loop's code and moves to the next iteration.

**Example:**

public class Main

{

public static void main(String[] args) {

for (int i = 0; i <=5; i++) {

if (i == 3) {

continue; // exits the loop when i equals 3

}

System.out.print(i+" ");// 0,1,2,4,5

}}

**Method or function :**

In Java, methods are blocks of code that perform a specific task and are defined within classes. They are used to organize code into reusable components and to encapsulate functionality.

Methods can return values using the **return** statement. The return type in the method signature specifies the type of value that the method returns. If a method does not return any value, its return type should be **void**

**return Statement:** Exits a method and can return a value.

**Example:**

**public class Main{**

**static int sum(int a, int b) {**

**return a + b;**

**}**

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

**System.out.println(sum(2,3));//5**

**}**

**}**

**Arrays in Java:**

**Definition:**

An array in Java is a data structure that allows you to store multiple values of the same type under a single variable name. It is a container object that holds a fixed number of values of a specific type.

**Declaration:** To declare an array, you specify the type of its elements followed by square brackets [].

Example: int[] myArray;

**Initialization:** Arrays are initialized using the new keyword followed by the data type and the size of the array.

**method -1 :**

my\_Array = new int[5];

**method-2 :**

int[] myArray = {1, 2, 3, 4, 5};

**Accessing Elements from the array:**

Elements in an array are accessed by their index, starting from 0.

**Example:**

int[] myArray = {1, 2, 3};

System.out.println(myArray[0]);//1

System.out.println(myArray[1]);//2

System.out.println(myArray[2]);//3

System.out.println(myArray[3]);//java.lang.ArrayIndexOutOfBoundsException

**Length of array :**

The length of an array can be obtained using the length property.

**Example-1:** int length = my\_Array.length ;

**Example-2:**

int[] myArray = {1, 2, 3, 4, 5};

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

System.out.println(myArray[i]);

}

**2D Arrays in Java:**

**Definition:**

A 2D array, or a two-dimensional array, is a data structure that stores elements in a grid format. It can be visualized as an array of arrays. Each element in the 2D array is identified by two indices: the row index and the column index.

**Declaration:** To declare a 2D array, you specify the type of its elements followed by two sets of square brackets [][].

Example: int[][] my2DArray;

**Initialization:** 2D arrays are initialized using the new keyword followed by the data type, and the size of each dimension.

**Method-1:** my2DArray = new int[3][4];

**Method-2:** int[][] my2DArray = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}};

**Accessing Elements:**

Elements in a 2D array are accessed by specifying both row and column indices.

int[][] my2DArray = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}};

System.out.println(my2DArray[0][0]);//1

System.out.println(my2DArray[0][1]);//2

System.out.println(my2DArray[1][0]);//5

System.out.println(my2DArray[2][2]);//11

System.out.println(my2DArray[2][3]);//12

System.out.println(my2DArray[0][3]);//4

System.out.println(my2DArray[3][0]);//java.lang.ArrayIndexOutOfBoundsException

System.out.println(my2DArray[0][4]);//java.lang.ArrayIndexOutOfBoundsException

**Length of Rows and Columns:**

You can find the length of rows and columns in a 2D array.

**Example:**

int[][] my2DArray = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}};

int numRows = my2DArray.length;

int numCols = my2DArray[0].length;

System.out.println(numRows);//3

System.out.println(numCols);//4

**Strings in Java:**

In Java, a string is an object representing a sequence of characters. It is a class in the Java Standard Library (java.lang.String) and provides various methods for manipulating and working with sequences of characters. Strings in Java are immutable, meaning once a string object is created, it cannot be changed.

**Declaration:** Strings are declared using the String keyword.

**Example:** String str;

**Initialization:** Strings can be initialized using double-quotes.

String myString = "Hello";

**Concatenation of string :**

Strings can be concatenated using the + operator or the concat() method.

String r1 = "hello"+ "world";

String string1 = "Hello";

String string2 = "World";

String r2 = string1.concat(string2);

System.out.println(r1);//helloworld

System.out.println(r2);//HelloWorld

System.out.println("Java" + 1000 + 2000 + 3000);// Java100020003000

System.out.println(1000 + 2000 + 3000 + "Java"); // 6000Java

System.out.println(7.7 + 3.3 + "Java" + 3.3 + 7.7);// 11.0Java3.37.7

System.out.println("ONE" + 2 + 3 + 4 + "FIVE"); // ONE234FIVE

**Length of string:**

The length of a string can be obtained using the length() method.

Example: int length = myString.length();

**Accessing Characters:**

Characters in a string are accessed by their index using the charAt() method.

**Example:** char firstChar = myString.charAt(0);

**Comparison of string:**

Strings can be compared using the equals() method or other comparison methods. **Example-1:**

String str1 = "hello";

String str2 = "Hello";

String res = "hello";

System.out.println(str1==res);//true

System.out.println(str1.equals(str2));//false

System.out.println(str1.equalsIgnoreCase(str2));//true

**Example-2:**

String str1 = "abc";

String str2 = "bbc";

String res = "abc";

System.out.println(str1.compareTo(res));//0

System.out.println(str1.compareTo(str2));//-1

System.out.println(str2.compareTo(str1)); //1

**Substring:**

Substrings can be extracted using the substring() method.

Example:

String sub = myString.substring(startIndex, endIndex);

String str = "Hello, World!";

System.out.println(str.length());//13

System.out.println(str.substring(0,13));//Hello, World!

System.out.println(str.substring(3,12));//lo, World

**StringBuffer and StringBuilder**: These classes are mutable sequences of characters. They are used when you need to modify the content of a string frequently.

**StringBuffer:**

StringBuffer buffer = new StringBuffer();

buffer.append("Hello");

buffer.append(" ");

buffer.append("World");

String result = buffer.toString(); // Converts StringBuffer to String

System.out.println(result); // Output: Hello World

**StringBuilder:**

StringBuilder builder = new StringBuilder();

builder.append("Hello");

builder.append(" ");

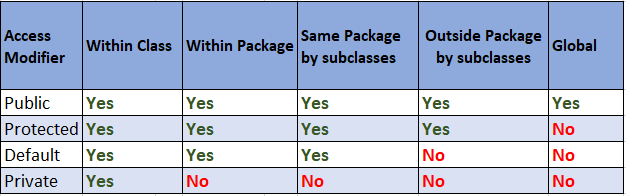
builder.append("World");

String result = builder.toString();

System.out.println(result); // Output: Hello World

**access modifiers :**

In Java, access specifiers (also known as access modifiers) are keywords used to define the visibility or accessibility of classes, methods, and fields within a program. There are four main access specifiers in Java:



**Example access modifiers :**

**Public:**

The public is most permissive access level.Classes, methods, and fields declared as public can be accessed by any other class.

public class MyClass {

**public** int myPublicField;

public void myPublicMethod() {

System.out.println(“this is public”);

}

}

**Private:**

The **Private** is most restrictive access level. Members (fields or methods) declared as private are only accessible within the class to which they belong.

public class MyClass {

**private** int myPrivateField;

private void myPrivateMethod() {

System.out.println(“this is private”);

}

}

**Protected:**

Members declared as protected are accessible within the same package and by subclasses, even if they are in a different package.

package mypackage;

public class MyBaseClass {

**protected** int myProtectedField;

**protected** void myProtectedMethod() {

System.out.println(“this is protected”);

}

}

package mypackage;

public class MyDerivedClass extends MyBaseClass {

System.out.println(“we can acess variable and method from MyBaseClass”);

}

**Default (Package-Private):**

If no access specifier is specified (package-private), it is visible to classes within the same package but not to classes outside that package.

package mypackage;

class MyPackagePrivateClass {

int myPackagePrivateField;

void myPackagePrivateMethod() {

// code here

}

}

**Static keyword :**

**Static Fields (Static Variables):** When a field is declared with the static keyword, it becomes a class variable. There will be only one copy of this variable that is shared among all instances of the class. It belongs to the class rather than to any particular instance.

**2.Static Methods:** Similar to static fields, static methods belong to the class rather than an instance. They can be called using the class name without creating an instance of the class.

**Example:**

class xyz1 {

    static int b;

   static String str2;

   static void f1(){

    System.out.println("valu of b & str2 is "+b+" "+str2);

   }

}

 class statics {

    public static void main(String[] args) {

        System.out.println(xyz1.b);

        System.out.println(xyz1.str2);

        xyz1.f1();}

}

//output

0

Null

valu of b & str2 is 0 null

**Final Members or final keyword :**

**Fields (Final Variables):** When a field is declared with the final keyword, its value cannot be changed after it has been initialized. This is often used to create constants.

public class MyClass {

final int finalVariable = 20;

// ...}

**Final Methods:** When a method is declared as final, it means that the method cannot be overridden by subclasses.

public class MyBaseClass {

final void myFinalMethod() {

// ...

}

// ...

}

**Final Classes:** When a class is declared as final, it means that the class cannot be extended or subclassed.

final public class FinalClass {

// ...

}

You can not extend a final class or override final methods.Combining static and final is also possible, and you may see public static final used together to create constants.

public class Constants {

public static final int MAX\_VALUE = 100;

}

**Example-1:**

final class xyz1 {

    final int b=3;//must initialize or fix value

   final String str2="abc";

  final  void f1(){

    System.out.println("valu of b & str2 is "+b+" "+str2);

   }

}

 class statics {

    public static void main(String[] args) {

        xyz1 o1=new xyz1();

        System.out.println(o1.b);

        System.out.println(o1.str2);

        o1.f1();

    }

}

Example-2:

final  class xyz1 {// class can not be static

    final static int b=3;//must initialize or fix value

   final static String str2="abc";

  final static void f1(){

    System.out.println("valu of b & str2 is "+b+" "+str2);

   }

}

 class statics {

    public static void main(String[] args) {

        System.out.println(xyz1.b);

        System.out.println(xyz1.str2);

        xyz1.f1();

    }

}

//output will be same for both ex-1 & ex-2

3

abc

valu of b & str2 is 3 abc

**class :**  
In Java, a class is a blueprint or a template for creating objects. Objects are instances of classes, and they encapsulate data (attributes) and behavior (methods). The class defines the structure and behavior of the objects that will be created based on it.

**Object :**

An object is an instance of a class. It's a concrete entity created based on the blueprint provided by the class. Objects have both attribute and behavior .

**Example -1**

class xyz{

void f1(int a,int b){

System.out.println(a+b);

}

}

public class Main

{

public static void main(String[] args) {

xyz ob=new xyz();

ob.f1(2,3);//5

}

}

**Example-2**

class Person {

// Properties or attributes or **Fields**

String name;

int age;

// Constructor

Person(String name, int age) {

this.name = name;

this.age = age;

}

// Method

void displayInfo() {

System.out.println("Name: " + name);

System.out.println("Age: " + age);s

}

}

class Main{

public static void main(String[] args) {

Person person = new Person("ajay", 30);

person.displayInfo();

}

}

**//output**

Name: ajay

Age: 30

**Object-Oriented Programming :**

Object-Oriented Programming (OOP) is a programming paradigm that revolves around the concept of objects, which can contain data in the form of fields (attributes or properties) and code in the form of methods or functions. Java is a strongly object-oriented programming language, and it follows the principles of OOPs.

**1.Inheritance :**

In Java, inheritance is a key feature of Object-Oriented Programming (OOP). It allows a class (subclass or derived class) to inherit the properties and behaviors of another class (**superclass or base class**). This promotes code reuse and establishes an "is-a" relationship between classes.

Example-1:

class Parent {

    int a, b;

Parent() {

        a = 10;

        b = 20;

        System.out.println("Parent Constant");

    }

  void f1(){

    System.out.println("this is f1 method");

  }

}

class Childd extends Parent {

    int x, y;

  Childd() {

        x = 100;

        y = 200;

    }

void f2() {

        System.out.println(a);// 10

        System.out.println(b);// 20

        System.out.println(x);// 100

        System.out.println(y); // 200

    }}

public class sk {

    public static void main(String[] args) {

        Childd ch = new Childd();

        ch.f2();

        ch.f1();}

}

Example-2:

class Parent {

    int a,b;

     Parent(){

        System.out.println("parent non paramterized constructer");

        a=40;

        b=50;

    }

Parent(int a, int b) {

        this.a = a;

        this.b = b;

        System.out.println("Parent paramterized constructer");

    }

  void f1(){

    System.err.println("this is f1 method");

  }

}

class Childd extends Parent {

    int x, y;

  Childd(int x,int y) {

        super(x,y);// for setting value of a & b, otherise it

        //will call by defualt non-parameterized constructer

        this.x = x;

        this.y = y;

    }

void f2() {

        System.out.println(a);//10

        System.out.println(b);//20

        System.out.println(x);//10

        System.out.println(y); //20

    }}

public class sk {

    public static void main(String[] args) {

        Childd ch = new Childd(10,20);

        ch.f2();

        ch.f1();}

}

Example -3

class Person {

String name;

int age;

Person(String name, int age) {

this.name = name;

this.age = age;

}

void displayInfo() {

System.out.println("Name: " + name);

System.out.println("Age: " + age);

}

}

// Derived class (subclass) inheriting from Person

class Student extends Person {

int studentId;

String course;

Student(String name, int age, int studentId, String course) {

super(name, age);

this.studentId = studentId;

this.course = course;

}

// Overriding the displayInfo method in the subclass

@Override

void displayInfo() {

super.displayInfo(); // Call the displayInfo method from the parent class

System.out.println("Student ID: " + studentId);

System.out.println("Course: " + course);

}

void study() {

System.out.println("Student is studying");

}

}

public class Main {

public static void main(String[] args) {

// Creating an instance of the base class

Person person = new Person("ajay", 25);

// Creating an instance of the derived class

Student student = new Student("vijay", 20, 12345, "Computer Science");

System.out.println("Person Information:");

person.displayInfo();

System.out.println();

// Accessing properties and methods of the derived class

System.out.println("Student Information:");

student.displayInfo();

student.study();

}

}

**// output**

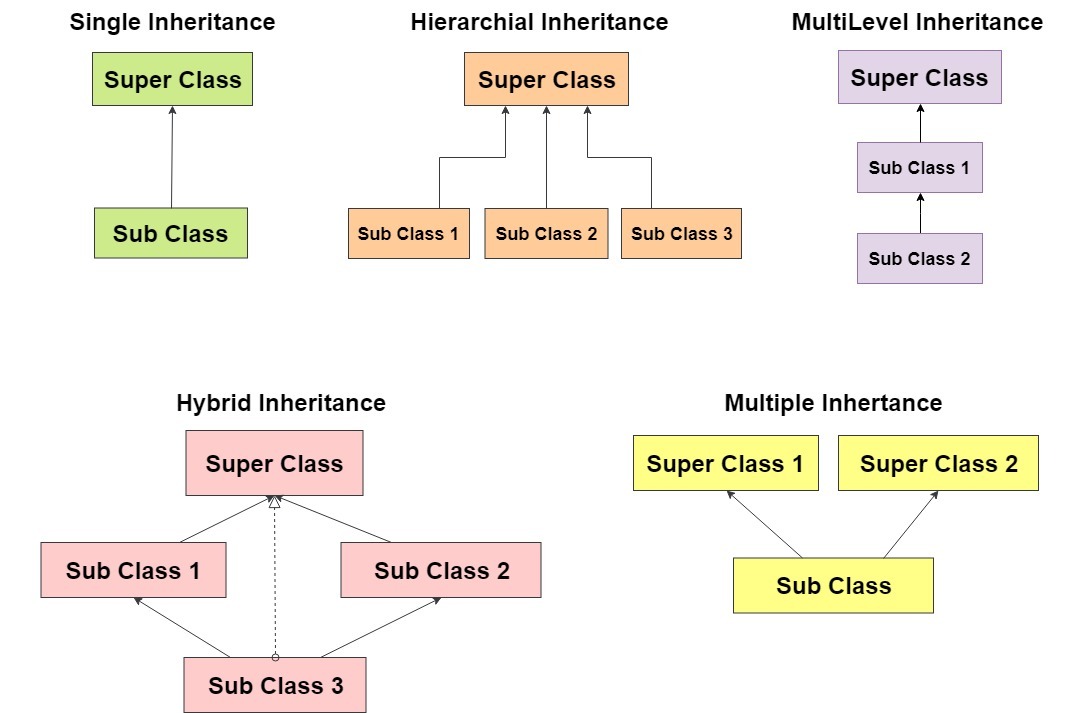
Age: 20

Student ID: 12345

Course: Computer Science

Student is studying

**Type of inheritance:**



**1.Single Inheritance:**

In single inheritance, a class extends only one superclass. The subclass inherits the properties and behaviors of the superclass.

class Animal {

void eat() {

System.out.println("Animal is eating...");

}

}

class Dog extends Animal {

void bark() {

System.out.println("Dog is barking...");

}

}

public class Main {

public static void main(String[] args) {

Dog dog = new Dog();

dog.eat(); // Inherited from Animal

dog.bark(); // Defined in Dog

}

}

//output

Animal is eating...

Dog is barking...

**2. Multilevel Inheritance:**

Multilevel inheritance occurs when a class extends another class, and that class is further extended by another class.

class Animal {

void eat() {

System.out.println("Animal is eating...");

}

}

class Dog extends Animal {

void bark() {

System.out.println("Dog is barking...");

}

}

class BabyDog extends Dog {

void weep() {

System.out.println("BabyDog is weeping...");

}

}

public class Main {

public static void main(String[] args) {

BabyDog babyDog = new BabyDog();

babyDog.eat(); // Inherited from Animal

babyDog.bark(); // Inherited from Dog

babyDog.weep(); // Defined in BabyDog

}

}

//output

Animal is eating...

Dog is barking...

BabyDog is weeping...

**3.Hierarchical Inheritance:**

Hierarchical inheritance occurs when multiple classes extend the same superclass.

class Shape {

void draw() {

System.out.println("Drawing a shape...");

}}

class Circle extends Shape {

void draw() {

System.out.println("Drawing a circle...");

}}

class Rectangle extends Shape {

void draw() {

System.out.println("Drawing a rectangle...");

}}

public class Main {

public static void main(String[] args) {

Circle circle = new Circle();

circle.draw(); // Defined in Circle

Rectangle rectangle = new Rectangle();

rectangle.draw(); // Defined in Rectangle

}}

//output

Drawing a circle...

Drawing a rectangle...

**Q. why Java does not support multiple inheritance for classes ?**

1. **Diamond Problem**: Multiple inheritance can lead to the diamond problem, where ambiguity arises if a subclass inherits from two superclasses that have a common ancestor. Resolving such ambiguity can be complex and error-prone.
2. **Complexity**: Supporting multiple inheritance adds complexity to the language, making it harder to understand and maintain code. Java aims to be simple and straightforward, and allowing multiple inheritance can introduce complications, especially for developers who are new to the language.
3. **Method Name Conflict**: If two parent classes have methods with the same name and signature, there's ambiguity about which method the subclass should inherit. This issue doesn't occur with interfaces because they only declare methods without providing implementations.

=> but through interface it supports .

interface A {

    void methodA();

}

interface B {

    void methodB();

}

class MyClass implements A, B {

    public void methodA() {

        System.out.println("Method A implementation");

    }

    public void methodB() {

        System.out.println("Method B implementation");

    }

}

public class Main {

    public static void main(String[] args) {

        MyClass obj = new MyClass();

        obj.methodA();

        obj.methodB();

    }

}

**Hybrid inheritance :**

Hybrid inheritance in the context of object-oriented programming typically refers to a combination of multiple types of inheritance within a programming language. It involves the use of more than one type of inheritance mechanism, such as single, multiple, hierarchical, or multilevel inheritance, in defining relationships between classes.

**Encapsulation :**

Encapsulation is one of the four fundamental Object-Oriented Programming (OOP) concepts, and it refers to the bundling of data and methods that operate on that data into a single unit known as a class. The main idea behind encapsulation is to hide the internal state of an object and to restrict direct access to some of its components, exposing only what is necessary for the outside world to interact with.

In Java, encapsulation is achieved through the use of access modifiers and the creation of getter and setter methods. Access modifiers control the visibility of fields and methods in a class.

**public:** Accessible from any class.

**protected:** Accessible within the same package and subclasses.

**default (no modifier):** Accessible only within the same package.

**private:** Accessible only within the same class.

**Example:**

class Student {

    // Private fields

    private String name;

    private int age;

  // Constructor

    public Student(String name, int age) {

        this.name = name;

        this.age = age;

    }

 // Getter methods

    public String getName() {

        return name;

    }

public int getAge() {

        return age;

    }

// Setter methods

public void setName(String name) {

        this.name = name;

    }

public void setAge(int age) {

        if (age >= 0) {

            this.age = age;

        } else {

            System.out.println("Age cannot be negative.");

        }

    }

}

public class student1 {

    public static void main(String[] args) {

        Student student1 = new Student("ajay", 20);

        System.out.println("Initial Name: " + student1.getName());

        System.out.println("Initial Age: " + student1.getAge());

        student1.setName("vijay");

        student1.setAge(22);

        System.out.println("Updated Name: " + student1.getName());

        System.out.println("Updated Age: " + student1.getAge());

    }

}

**#output**

Initial Name: ajay

Initial Age: 20

Updated Name: vijay

Updated Age: 22

Polymorphism :

Polymorphism is one of the core concepts in Object-Oriented Programming (OOP) and is characterized by the ability of a single function, method, or operator to work on different types of data or objects.

Java supports two types of polymorphism: compile-time (static) polymorphism and runtime (dynamic) polymorphism.

**1.Compile-time (Static) Polymorphism:**

=> Also known as method overloading.

=> The decision about which method to call is made by the compiler based on the method signature.

=>Method overloading occurs when a class has multiple methods with the same name but different parameter lists (number, type, or both). The compiler differentiates between these methods based on the number and types of parameters.

1. **Same Method Name**: Method overloading occurs when two or more methods within the same class have the same name.
2. **Different Parameter Lists**: The methods that are overloaded must have different parameter lists. This can involve differences in the number, order, or types of parameters.
3. **Return Type**: The return type of the methods can be the same or different. Method overloading is not determined by the return type; only the method signature (name and parameters) matters.
4. **Access Modifiers**: Overloaded methods can have different access modifiers (public, protected, private, or default). They can also have different access levels.
5. **Static and Instance Methods**: Overloaded methods can include both static and instance methods. Overloading can occur within the same class or across parent and child classes.

**Example:**

class MathOperations {

// Method overloading

public int add(int a, int b) {

return a + b;

}

public double add(double a, double b) {

return a + b;

}

public String add(String a, String b) {

return a + b;

}

}

public class Main {

public static void main(String[] args) {

MathOperations math = new MathOperations();

System.out.println(math.add(1, 2));

System.out.println(math.add(2.5, 3.5));

System.out.println(math.add("Hello", " World"));

}

}

**#output**

3

6.0

Hello World

**2.Runtime (Dynamic) Polymorphism:**

=>Also known as method overriding.

=>The decision about which method to call is made at runtime based on the actual type of the object.

=> Method overriding occurs when a subclass provides a specific implementation for a method that is already defined in its superclass. The overridden method in the subclass has the same signature (name, return type, and parameters) as the one in the superclass. It allows a subclass to provide a specialized implementation of a method defined in its superclass.

1. **Inheritance**: Method overriding is only applicable in the context of inheritance. It occurs when a subclass inherits a method from its superclass and provides a new implementation for that method.
2. **Signature**: The overridden method in the subclass must have the same signature (name, parameters, and return type) as the method in the superclass. The signature of the method is what determines whether it is an override.
3. **Access Modifiers**: The access modifier for the overriding method in the subclass cannot be more restrictive than the access modifier of the overridden method in the superclass. However, it can be less restrictive or the same.
4. **Return Type**: The return type of the overriding method can be a subtype of the return type of the overridden method, or it can be the same type (covariant return types). It cannot be a supertype.
5. **Annotations**: Annotations inherited from the overridden method can be overridden in the subclass.
6. **Runtime Polymorphism**: Method overriding is a key feature that enables runtime polymorphism, where the actual method invoked is determined at runtime based on the type of object.

**Example -1 of method overriding:**

class Shape {

void calculateArea() {

System.out.println("Area calculation not implemented for generic shape.");

}

}

class Rectangle extends Shape {

int length;

int width;

Rectangle(int length, int width) {

this.length = length;

this.width = width;

}

// Override the calculateArea method for rectangles

@Override

void calculateArea() {

int area = length \* width;

System.out.println("Area of the rectangle: " + area);

}

}

public class Main {

public static void main(String[] args) {

Rectangle rectangle = new Rectangle(5, 8);

// Call the overridden calculateArea method

rectangle.calculateArea();

}

}

**#output**

Area of the rectangle: 40

**Example-2:**

// Shape superclass

class Shape {

public void draw() {

System.out.println("Drawing a shape");

}

}

// Circle subclass

class Circle extends Shape {

// Method overriding

@Override

public void draw() {

System.out.println("Drawing a circle");

}

}

// Square subclass

class Square extends Shape {

// Method overriding

@Override

public void draw() {

System.out.println("Drawing a square");

}

}

// Triangle subclass

class Triangle extends Shape {

// Method overriding

@Override

public void draw() {

System.out.println("Drawing a triangle");

}

}

public class Main {

public static void main(String[] args) {

// Creating instances of different shapes

Shape shape1 = new Circle();

Shape shape2 = new Square();

Shape shape3 = new Triangle();

// Calling draw method on different shapes

shape1.draw(); // Calls draw() from Circle class

shape2.draw(); // Calls draw() from Square class

shape3.draw(); // Calls draw() from Triangle class

}

}

**#output**

Drawing a circle

Drawing a square

Drawing a triangle

**Abstraction :**

Abstraction is one of the core principles of Object-Oriented Programming (OOP) and refers to the concept of hiding complex implementation details and showing only the relevant features of an object. In Java, abstraction is achieved through abstract classes and interfaces.

**1.Abstract Classes:**

=>An abstract class is a class that cannot be instantiated and may contain abstract methods (methods without a body).

=>Abstract methods are meant to be overridden by concrete (non-abstract) subclasses.

=>Abstract classes can also have concrete methods with an implementation.

=>Abstract classes can have constructors.

**Example:**

abstract class Shape {

    // Abstract method for calculating the area

     abstract double calculateArea();

   // Concrete method

     void displayInfo() {

        System.out.println("This is a shape.");

    }

}

class Circle extends Shape {

     double radius;

    Circle(double radius) {

        this.radius = radius;

    }

    @Override

    public double calculateArea() {

        return Math.PI \* Math.pow(radius, 2);

    }

}

class Rectangle extends Shape {

     double length;

     double width;

    public Rectangle(double length, double width) {

        this.length = length;

        this.width = width;

    }

    @Override

     double calculateArea() {

        return length \* width;

    }

}

public class abstraction {

    public static void main(String[] args) {

        // Create instances of concrete subclasses

        Shape circle = new Circle(5.0);

        Shape rectangle = new Rectangle(4.0, 6.0);

       // Call methods on shapes

        System.out.println("Circle Area: " + circle.calculateArea());

        System.out.println("Rectangle Area: " + rectangle.calculateArea());

    }

}

**#output**

Circle Area: 78.53981633974483

Rectangle Area: 24.0

**Interfaces:**

=>An interface is a collection of abstract methods. It cannot have concrete methods with an implementation.

=>Classes implement interfaces by providing concrete implementations for all the methods declared in the interface.

=>Interfaces support multiple inheritance, allowing a class to implement multiple interfaces.

**Example:**

interface Shape {

    // Abstract method

   abstract void calculateArea();

}

class Circle implements Shape {

     double radius; double area;

     Circle(double radius) {

        this.radius = radius;

    }

    @Override

    public void calculateArea() {

        this.area= Math.PI \* radius \* radius;

        System.out.println(this.area);

    }

}

class Rectangle implements Shape {

     double length; double area;

     double width;

     Rectangle(double length, double width) {

        this.length = length;

        this.width = width;

    }

    @Override

    public void calculateArea() {

        this.area= length \* width;

      System.out.println(area);

    }

}

public class interfaces1 {

    public static void main(String[] args) {

        Shape circle = new Circle(5.0);

        Shape rectangle = new Rectangle(4.0, 6.0);

        circle.calculateArea();

        rectangle.calculateArea();

    }

}

**#output**

78.53981633974483

24.0

**Packages :**

In Java, packages are used to organize classes into namespaces, making it easier to manage and maintain large codebases.

**Example:**

package com.xyz.myproject;

This statement declares that all the classes defined in this file belong to the com.xyz.myproject package.

**Directory Structure:**

Java uses a directory structure to map packages to directories. For com.xyz.myproject package, you would create a directory structure like this

myproject/

│

└── com/

└── xyz/

└── myproject/

└── MyClass.java

**Setting CLASSPATH:**

The CLASSPATH is an environment variable that tells the Java Virtual Machine (JVM) where to find user-defined classes and packages.

**1.Setting CLASSPATH Temporarily:**

You can set the CLASSPATH temporarily using the command line:

**java -cp /path/to/your/classes com.xyz.myproject.Main**

This command tells the JVM to look for classes in the specified directory (/path/to/your/classes) and run the Main class in the com.xyz.myproject package.

**2.Setting CLASSPATH Permanently:**

You can set the CLASSPATH permanently by modifying your system environment variables. This method varies depending on your operating system. For example, in Unix-like systems, you would add a line like this to your .bashrc or .bash\_profile file.

**export CLASSPATH=/path/to/your/classes**

In Windows, you would set it through the System Properties or by using the setx command.

**Using Packages and import statement:**

Once you have defined your packages and set the CLASSPATH, you can import classes from other packages into your Java code using the import statement.

**Example:**

**import com.xyz.OtherClass;**

This statement allows you to use classes from the com.xyz.otherpackage package in your current file.

In Java, when you import packages or classes, you have two main options: regular import and static import.

**Example:**

**package com.xyz.myapp;**

public class Greeting {

public static void greet() {

System.out.println("Hello, World!");

}

}

**import com.xyz.myapp.Greeting;**

public class Main {

public static void main(String[] args) {

Greeting.greet();

}

}

**1.Regular Import:**

When you import a package or class using a regular import statement, you typically use the full package name or the specific class name. The naming convention for regular imports is to use.

**For example:**

import packageName.className;

import java.util.ArrayList;

import java.util.List;

**example-2:**

import java.util.ArrayList;

public class Main {

public static void main(String[] args) {

ArrayList<String> list1 = new ArrayList<>();

ArrayList<Integer> list2=new ArrayList<>();

list1.add("Java");

list1.add("Python");

list1.add("C++");

list2.add(1);

list2.add(2);

list2.add(3);

for (String i : list1) {

System.out.println(i);

}

for (int i=0;i<list2.size() ; i++){

System.out.println(list2.get(i));

}

}

}

**2.Static Import:**

Static import allows you to access static members (fields and methods) of a class directly without qualifying the class name. The naming convention for static imports is similar to regular imports, but it's used specifically for static members.

**For example:**

import static packageName.className.staticMember;

import static java.lang.Math.PI;

import static java.lang.Math.sqrt;

**example-2:**

import static java.lang.Math.\*;

public class Main {

public static void main(String[] args) {

double radius = 2.0;

double circumference = 2 \* PI \* radius;

double area = PI \* pow(radius, 2);

System.out.println(circumference);

System.out.println(area);

System.out.println(sqrt(625));

or System.out.println(Math.sqrt(25));

int r=(int)sqrt(25);

}

}

**Naming Convention Guidelines:**

**Clarity:** Use clear and concise names for packages, classes, and static members. Avoid ambiguous names to prevent confusion.

**Avoid Wildcard Imports:** While it's possible to use wildcard imports (import packageName.\*;), it's generally recommended to avoid them as they can make the code less readable and may lead to naming conflicts.

**Organize Imports:** Organize import statements logically. Group related imports together and separate them from other import statements. IDEs often have automated tools for organizing imports.

**Consistency:** Maintain consistency in naming conventions throughout your codebase. This improves readability and makes it easier for developers to understand the code.

**Follow Java Naming Conventions:** Adhere to the Java naming conventions for packages, classes, methods, and variables. This includes using camelCase for variable and method names, PascalCase for class names, and all-lowercase for package names.