**Activity-01**

**Write down how to configure environment variables for java.**

# Understanding Environment Variables

Environment variables are system-wide or user-specific settings that define the execution environment for applications. In the case of Java, setting the correct environment variables ensures that Java tools (e.g., java, java c) can be accessed globally.

# Key Environment Variables for Java

* **JAVA\_HOME**: This variable points to the root directory of the Java Development Kit (JDK). It allows other applications and build tools (e.g., Maven, Gradle) to locate the Java installation.
* **PATH**: This variable contains a list of directories where the operating system looks for executable files. Adding JAVA\_HOME/bin to PATH allows users to run Java commands from any terminal or command prompt.

# Steps for Configuration

## Determine Java Installation Path

* + Locate the directory where the JDK is installed.
  + Common locations include C:\Program Files\Java\jdk-<version> (Windows) and

/usr/lib/jvm/java-<version> (Linux).

## Define JAVA\_HOME

* + Assign the installation path to the JAVA\_HOME variable. o

This serves as a reference for Java-based applications.

## Modify PATH Variable

* + Append JAVA\_HOME/bin to the system’s PATH

variable.

* + This enables execution of Java commands from any location in the terminal.

## Apply and Verify

* + Refresh the environment variable settings.
  + Use java -version and java c -version to confirm successful configuration.

**Activity-02**

**Write a program to illustrate creating multiple objects by one type only.**

java

// Class to represent a 'Person' with name and age class Person {

String name; int age;

// Constructor to initialize name and age public Person(String name, int age) {

this.name = name; this.age = age;

}

// Method to display person details public void displayDetails() {

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

}

}

public class Main {

public static void main(String[] args) {

// Creating multiple objects of the same class 'Person' Person person1 = new Person("John", 25);

Person person2 = new Person("Alice", 30); Person person3 = new Person("Bob", 22);

// Displaying details of each person object System.out.println("Person 1 Details:"); person1.displayDetails();

System.out.println("\nPerson 2 Details:"); person2.displayDetails();

System.out.println("\nPerson 3 Details:"); person3.displayDetails();

}

}

## Output:

Person 1 Details: Name: John Age: 25

Person 2 Details: Name: Alice Age: 30

Person 3 Details: Name: Bob Age:22

**Activity-03**

**Code a program to use different types of constructors.**

# Types of Constructors Used:

1. **Default Constructor** – No parameters, initializes with default values.
2. **Parameterized Constructor** – Takes arguments to initialize values.
3. **Copy Constructor** – Creates a new object by copying another object's values.

# Java Program:

// Class with different types of constructors class Person { String name; int age;

// 1. Default Constructor Person() { this.name = "Unknown";

this.age = 0;

}

// 2. Parameterized Constructor Person(String name, int age) { this.name = name; this.age

= age;

}

// 3. Copy Constructor Person(Person p) { this.name = p.name; this.age = p.age;

}

// Method to display details void display() {

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

}

}

// Main class public class ConstructorExample { public static void main(String[] args) {

// Using Default Constructor Person p1 = new Person();

// Using Parameterized Constructor Person p2 = new Person("Alice", 25);

// Using Copy Constructor Person p3 = new Person(p2);

// Display details p1.display(); p2.display(); p3.display();

}

}

# Output:

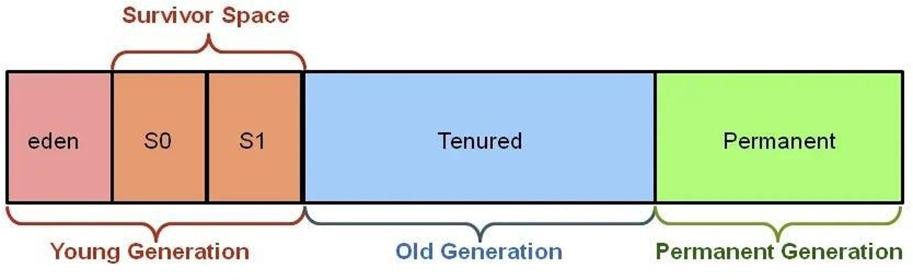
Name: Unknown, Age: 0 Name: Alice, Age: 25 Name: Alice, Age: 25

**Activity-04 Write a note on JVM generations.**

JVM (Java Virtual Machine) Generation:

The **Java Virtual Machine (JVM)** is the engine that provides a runtime environment to execute Java bytecode. It is a crucial part of the Java Runtime Environment (JRE) and helps Java achieve platform independence by converting compiled Java bytecode into machine code that can be executed by the underlying operating system.

JVM is organized into various components, and one key aspect of JVM is its memory management structure, which is divided into different generations. These generations are part of the **Java heap** and are designed to improve the performance of garbage collection (GC). The **heap** is where JVM stores objects, and objects are organized into different regions based on their lifecycle.



## There are typically four main generations in the JVM heap:

1. **Young Generation (Minor GC)**

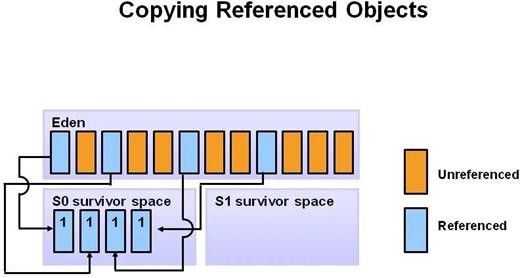
The **Young Generation** is where all new objects are created. This area is focused on objects that are short-lived, meaning they are typically discarded after a short period.

## The Young Generation is subdivided into three parts:

**Eden Space:** This is where all new objects are initially allocated. When the Eden space fills up, a **Minor GC** (Garbage Collection) is triggered to clean up unused objects and promote surviving objects to the next generation.

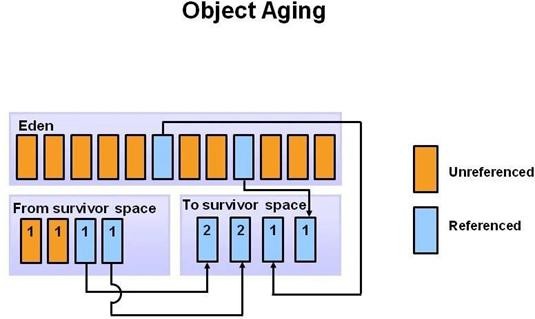
**Survivor Spaces (S0 and S1):** These are two smaller spaces where objects that survive the Minor GC are promoted from Eden. Objects that survive for a while in the survivor spaces are then promoted to the Old Generation.

**Minor GC:** A minor garbage collection occurs when the Eden space fills up. It collects garbage in the Young Generation and compacts the remaining objects into the survivor spaces. It's faster than a full garbage collection but still can be time-consuming when large objects are created frequently.

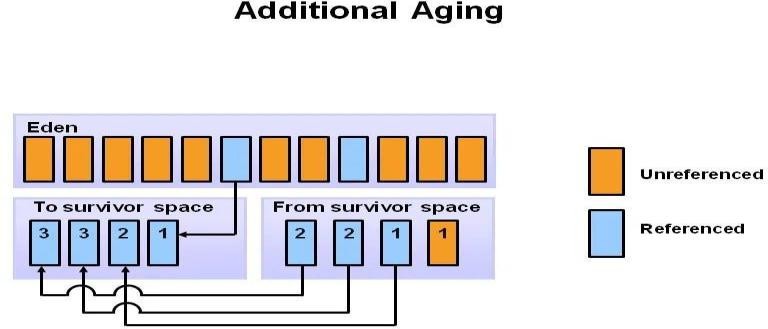


The **Young Generation** is where the most frequent garbage collections occur. This is because most objects in the young generation have short lifespans (i.e., they are quickly discarded). A **Minor Garbage Collection** (often referred to as a **Young GC**) typically happens here and is designed to quickly reclaim memory by collecting only the young objects.

This pattern continues until the eden space becomes full once more, triggering a new cycle. The events described in the previous paragraph are repeated, but with a slight difference. In this cycle, since **S0** has already been populated, all the marked objects that survive from both the eden space and S0 are moved to the second part of the survivor space known as **S1**. In the diagram below, these sections are labeled as the “**from survivor space**” and the “to survivor space” respectively.



One very important thing to note that any objects that make it to the survivor space gets tagged with an age counter. The algorithm will check this to see if it meets a threshold to go to the next stage: the old generation. More on that in a later section.



The provided diagram should help illustrate the switching of the “from” and “to” survivor spaces. The key takeaway here is that objects don’t necessarily move from **S0** to **S1** within the survivor space. Instead, they alternate their destination with each minor garbage collection event.

## Key Points of Young Generation:

* Objects are created and garbage-collected quickly.
* The Minor GC happens frequently but typically takes less time.
* It's optimized for the fast creation and deletion of short-lived objects.

## Old Generation (Major GC)

The **Old Generation** is where long-lived objects are stored. Objects that have survived several cycles of the Young Generation's garbage collection (minor GCs) are promoted to the Old Generation. These objects typically have a longer lifecycle and are less likely to be garbage collected quickly.

**Major GC** (also known as **Full GC**) occurs when the Old Generation is full. The garbage collector scans through the Old Generation to reclaim memory by removing unused objects. Major GCs are more expensive and time-consuming than minor GCs since they involve the entire heap.

## Key Points of Old Generation:

* + Contains long-lived objects.
  + Less frequent but more time-consuming garbage collection (major GC).
  + Can lead to performance degradation if the Old Generation becomes too large or full.

## Permanent Generation (or Metaspace in newer JVMs)

The **Permanent Generation** (PermGen) was a part of the heap memory used in older versions of Java (prior to Java 8) to store metadata related to class definitions, method definitions, and constant pool information. This area was managed separately from the heap and was not involved in garbage collection.

**Metaspace** (in Java 8 and beyond) replaced PermGen. Unlike PermGen, the **Metaspace**is not part of the heap, and its memory is managed by the operating system. It is dynamically sized, and its size can grow automatically as needed, unlike PermGen, which had a fixed size.

## Key Points of Permanent Generation (or Metaspace):

* + Used to store metadata related to loaded classes and methods.
  + In Java 8 and newer, the Metaspace replaced PermGen and is now managed separately outside the heap.
  + Can lead to memory issues like \*OutOfMemoryError\* if the space is exhausted.

## Code Cache (or Native Method Stack)

The **Code Cache** is used to store compiled native code (JIT-compiled code) during runtime. This includes code generated by the Just-In-Time (JIT) compiler, which compiles bytecode into native machine code to improve performance. The Code Cache allows the JVM to reuse compiled code for faster execution.

## Key Points of Code Cache:

* + Stores JIT-compiled code.
  + Speeds up program execution by caching compiled methods.
  + Can lead to an \*OutOfMemoryError\* if the cache fills up.

## Garbage Collection in JVM Generations:

* + **Minor GC:** Targets only the Young Generation, reclaiming memory for short-lived objects. It is fast but occurs more frequently.
  + **Major (Full) GC:** Targets the Old Generation and the entire heap. It is more time- consuming because it involves checking all live objects in the heap.
  + **Promotion:** Objects that survive in the Young Generation (after Minor GCs) get promoted to the Old Generation. Over time, objects that live longer are eventually moved to the Old Generation.

## Conclusion:

The JVM memory generations are a crucial part of Java's memory management system. They help optimize the performance of garbage collection and reduce overhead by categorizing objects based on their lifecycles. Understanding the different generations and how garbage collection works in each area can help developers optimize memory usage and performance in Java applications.

**Activity-05**

**Write a java program to check weather the given number is prime or not.**

java

import java.util.Scanner;

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

// Create a Scanner object to take input from the user Scanner scanner = new Scanner(System.in);

// Ask the user for the number to check System.out.print("Enter a number: "); int number = scanner.nextInt();

// Check if the number is prime if (isPrime(number)) {

System.out.println(number + " is a prime number.");

} else {

System.out.println(number + " is not a prime number.");

}

// Close the scanner object scanner.close();

}

// Method to check if a number is prime public static boolean isPrime(int num) {

// Handle edge case for numbers less than 2 if (num <= 1) {

return false;

}

// Check for factors other than 1 and the number itself for (int i = 2; i <= Math.sqrt(num); i++) {

if (num % i == 0) {

return false; // Number is divisible by i, hence not prime

}

}

return true; // Number is prime if no divisors found

}

}

## Output:

Enter a number: 29 29 is a prime number.

Enter a number: 15

15 is not a prime number

**Activity-06 Define encapsulation and list all its benefits.**

# Encapsulation

Encapsulation is a fundamental concept in object-oriented programming (OOP) that binds together the data and the methods that manipulate that data, keeping both safe from outside interference and misuse. In encapsulation, the data is hidden from the outside world, and only the necessary information is exposed through public methods.

# Benefits of Encapsulation

1. **Data Hiding:** Encapsulation helps to hide the internal details of an object from the outside world, making it harder for other objects to access or modify the data directly.
2. **Code Security:** By hiding the data and behavior of an object, encapsulation makes it more difficult for malicious code to access or modify the data.
3. **Improved Code Organization:** Encapsulation promotes code organization by grouping related data and methods together, making it easier to understand and maintain

.

1. **Reduced Coupling:** Encapsulation helps to reduce coupling between objects by hiding the internal implementation details, making it easier to change or replace one object without affecting others.
2. **Increased Flexibility:** Encapsulation makes it easier to change or modify the internal implementation of an object without affecting other parts of the program.
3. **Easier Testing:** Encapsulation makes it easier to test individual objects or modules by providing a clear interface and hiding the internal implementation details.
4. **Better Error Handling:** Encapsulation helps to improve error handling by providing a clear and controlled interface for accessing and modifying data.
5. **Improved Reusability:** Encapsulation promotes code reusability by providing a self- contained module that can be easily reused in other parts of the program.
6. **Simplified Debugging:** Encapsulation makes it easier to debug programs by providing a clear and controlled interface for accessing and modifying data.
7. **Enhanced Code Readability:** Encapsulation improves code readability by providing a clear and concise interface for accessing and modifying data.

By using encapsulation, developers can write more robust, maintainable, and scalable code that is easier to understand and modify.

**Activity-07**

# Write a java program to find reverse of the String.

**Java Program:**

java

import java.util.Scanner;

public class ReverseString {

public static void main(String[] args) {

// Create a Scanner object to take input from the user

Scanner scanner = new Scanner(System.in);

// Ask the user to enter a string System.out.print("Enter a string: "); String inputString =

scanner.nextLine();

// Reverse the string String reversedString =

reverse(inputString);

// Output the reversed string System.out.println("Reversed

string: " + reversedString);

// Close the scanner object scanner.close();

}

// Method to reverse the string

public static String reverse(String str)

{

StringBuilder reversed = new

StringBuilder(str);

return reversed.reverse().toString();

}

}

# Output:

Enter a string: Hello Reversed string: olleH

Enter a string: Java Reversed string: avaJ

**Activity-08**

**Explain Multi-level Inheritance with an example program.**

## Multi-Level Inheritance

Multi-level inheritance is a type of inheritance where a derived class inherits properties and behavior from a base class, and the base class itself inherits properties and behavior from another base class. This creates a hierarchical relationship between classes.

## Example Program:

// Base class: Animal class Animal {

void eat() { System.out.println("Eating...");

}

}

// Intermediate class: Mammal (inherits from Animal) class Mammal extends Animal {

void walk() { System.out.println("Walking...");

}

}

// Derived class: Dog (inherits from Mammal) class Dog extends Mammal {

void bark() { System.out.println("Barking...");

}

}

public class MultiLevelInheritance {

public static void main(String[] args) { Dog dog = new Dog();

dog.eat(); // Inherited from Animal dog.walk(); // Inherited from Mammal dog.bark(); // Specific to Dog

}

}

**Output:** Eating... Walking... Barking...

**Activity-09**

**Write a program of Static binding and Dynamic binding.**

# Static Binding Program:

// Animal class with a static method class Animal {

public static void sound() { System.out.println("Animal makes a sound");

}

}

// Dog class that extends Animal class Dog extends Animal {

public static void sound() { System.out.println("Dog barks");

}

}

public class StaticBinding {

public static void main(String[] args) {

// Create a Dog object Dog dog = new Dog();

// Call the sound() method using the Dog object dog.sound(); // Output: Dog barks

// Call the sound() method using the Animal class Animal.sound(); // Output: Animal makes a sound

// Create an Animal reference and assign it a Dog object Animal animal = new Dog();

// Call the sound() method using the Animal reference animal.sound(); // Output: Animal makes a sound (static binding)

}

}

## Output:

Dog barks

Animal makes a sound Animal makes a sound

# Dynamic Binding Program:

// Animal class with a method class Animal {

public void sound() { System.out.println("Animal makes a sound");

}

}

// Dog class that extends Animal class Dog extends Animal {

@Override

public void sound() { System.out.println("Dog barks");

}

}

// Cat class that extends Animal class Cat extends Animal {

@Override

public void sound() { System.out.println("Cat meows");

}

}

public class DynamicBinding {

public static void main(String[] args) {

// Create an Animal reference Animal animal;

// Assign a Dog object to the Animal reference animal = new Dog();

animal.sound(); // Output: Dog barks

// Assign a Cat object to the Animal reference animal = new Cat();

animal.sound(); // Output: Cat meows

}

}

**Output:** Dog barks Cat meows

**Activity-10**

**Differentiate between Class and Interface.**

|  |  |  |
| --- | --- | --- |
| **Instantiation** | A class can be instantiated, i.e., objects of a class can be created. | An interface cannot be instantiated directly, instead, it is implemented by a class or a struct. |
| **Inheritance** | Classes do not support multiple inheritance. | Interface supports multiple inheritance. |
| **Inheritance Mechanism** | A class can inherit another class using the keyword extends. | A class can implement an interface using the keyword “implements”. An interface can inherit another interface using “extends”. |
| **Constructors** | It can contain constructors. | It cannot contain constructors. |
| **Methods** | Methods in a class can be abstract, concrete, or both. | An interface contains abstract methods by default (before Java 8) or default/static methods (from Java 8 onward). |
| **Access Specifiers** | Variables and methods in a class can be declared using any access specifier(public, private, default, protected). | All variables and methods in an interface are declared public |
| **Variables** | Variables in a class can be static, final, or neither. | All variables are static and final. |
| **Purpose** | A class is a blueprint for creating objects and encapsulates data and behaviour. | An interface specifies a contract for classes to implement by focusing on capabilities rather than implementation. |

**Activity-11**

**Write a program to handle checked and unchecked exception.**

# Java Program:

import java.io.File; import java.io.FileReader;

import java.io.IOException;

public class ExceptionHandling {

public static void main(String[] args) {

// Checked Exception: IOException try {

File file = new File("non-existent-file.txt"); FileReader reader = new FileReader(file);

System.out.println("File found and read successfully.");

} catch (IOException e) {

System.out.println("Checked Exception: " + e.getMessage());

}

// Unchecked Exception: ArithmeticException try {

int num = 10 / 0; System.out.println("Division result: " + num);

} catch (ArithmeticException e) {

System.out.println("Unchecked Exception: " + e.getMessage());

}

// Unchecked Exception: NullPointerException try {

String str = null; System.out.println(str.length());

} catch (NullPointerException e) { System.out.println("Unchecked Exception: " + e.getMessage());

}

}

}

## Output:

Checked Exception: non-existent-file.txt (No such file or directory) Unchecked Exception: / by zero

Unchecked Exception: Cannot invoke "String.length()" because "str" is null

**Activity-12**

**Write a note on built-in java annotations.**

# Built-in Java Annotations

Java provides several built-in annotations that can be used to provide metadata about classes, methods, and variables. These annotations are predefined in the Java language and can be used to achieve specific goals.

## @Override

* + Used to inform the compiler that the method is intended to override a method in the superclass.
  + Helps catch errors at compile-time if the method does not exist in the superclass.

## @Deprecated

* + Indicates that a class, method, or variable is deprecated and should no longer be used.
  + The compiler generates a warning if the deprecated element is used.

## @SuppressWarnings

* + Suppresses specific compiler warnings.
  + Can be used to suppress warnings for deprecated methods, unchecked operations, etc.

## @SafeVarargs

* + Suppresses warnings for variable argument methods (varargs) that cannot be proven safe.
  + Introduced in Java 7.

## @FunctionalInterface

* + Indicates that an interface is intended to be a functional interface (i.e., it has only one abstract method).
  + Introduced in Java 8.

**Activity-13**

**Write a program to connect java application to database using JDBC.**

JDBC (Java Database Connectivity) is an API that allows Java applications to interact with relational databases. Below is a **Java program that connects to a MySQL database** using JDBC.

# Steps to Connect Java Application to Database Using JDBC

1. **Load the JDBC Driver** – The driver helps Java communicate with the database.
2. **Establish a Connection** – Use DriverManager.getConnection() to connect to the database.
3. **Create a Statement** – Use Statement or PreparedStatement to execute queries.
4. **Execute SQL Query** – Perform SELECT, INSERT, UPDATE, or DELETE operations.
5. **Process the Result** – Retrieve data from the database if applicable.
6. **Close the Connection** – Always close the database connection to free resources.

# JDBC Program to Connect Java to MySQL Database

import java.sql.Connection; import java.sql.DriverManager; import java.sql.ResultSet; import java.sql.SQLException; import java.sql.Statement;

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

// Database URL, Username, and Password

String url = "jdbc:mysql://localhost:3306/mydatabase"; // Change 'mydatabase' to your DB name

String user = "root"; // Change 'root' to your username

String password = "password"; // Change 'password' to your DB password

// JDBC Connection and Query Execution try {

// 1. Load the MySQL JDBC Driver Class.forName("com.mysql.cj.jdbc.Driver");

// 2. Establish Connection

Connection con = DriverManager.getConnection(url, user, password); System.out.println("Connected to the database successfully!");

// 3. Create a Statement

Statement stmt = con.createStatement();

name

// 4. Execute Query (Example: Retrieve Data)

String query = "SELECT \* FROM employees"; // Replace with your table ResultSet rs = stmt.executeQuery(query);

// 5. Process Results while (rs.next()) {

System.out.println("ID: " + rs.getInt("id") +

", Name: " + rs.getString("name") + ", Salary: " + rs.getDouble("salary"));

}

// 6. Close Resources rs.close(); stmt.close(); con.close();

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

} catch (ClassNotFoundException e) { System.out.println("MySQL JDBC Driver not found!"); e.printStackTrace();

} catch (SQLException e) { System.out.println("Database connection error!"); e.printStackTrace();

}

}

}

# Prerequisites

1. Install **MySQL** and create a database (e.g., mydatabase).
2. Add **MySQL JDBC Driver (mysql-connector-java.jar)** to your Java project.
3. Ensure your **database credentials** (username, password, and database name) are correct.

# Conclusion

This program demonstrates how to connect a Java application to a MySQL database using JDBC. You can modify it to perform **INSERT, UPDATE, DELETE** operations based on your requirements.