Question: 1

Write a JDBC program to establish a connection to an Oracle database named college. Handle exceptions properly while connecting and closing the database connection.

Answer:

```
import java.sql.*;
public class JDBCExample {
  public static void main (String[] args) throws
  SQLException, classNotFoundException {
    String url = "jdbc:oracle:thin:@localhost:1521:
                                    college";
    String username = "YourUsername";
    String password = "YourPassword";
    Class.forName ("oracle.jdbc.driver.OracleDriver");
    Connection connection = DriverManager.getConnection(
                        url, username, password);
```

```
System.out.println("Connection Successful!");
connection.close();
System.out.println("Connection closed.");
}
}
```

Question -2.

Demonstrate how to insert multiple records into a Courses table using JDBC batch processing. Explain how batch execution improves performance.

Answer :

The process to insert multiple records into a 'Courses' table using JDBC batch processing is given below:

```
import java.sql.*;
public class BatchInsert {
  public static void main (String[] args) throws
  SQLException {
    Class.forName("oracle.jdbc.driver.OracleDriver");
try (Connection connection = DriverManager.getConnection
        ("jdbc:oracle:thin:@localhost:1521:college",
          "u", "p");
```

```
PreparedStatement statement = connection.prepareStat-
ement("INSERT INTO Courses (course_id, course_name,
course_code) VALUES (?, ?, ?)");
{
 for (int i=1 ; i<=24; i++)
 {
  statement.setString (1,"ID" +i);
  statement.setString (2,"Name" +i);
  statement.setString (3,"Code" +i);
   statement.addBatch ();
 }
 statement.executeBatch();
 }
}
}
```

P.T.O.

How Batch Processing Improves Performance:

1. Reduced Round Trips: Without batch processing, each INSERT statement would require a separate round trip to the database server. Batch processing sends multiple SQL statements to the server in one go, significantly reducing the network overhead and latency.

2. Server-Side Optimization: The database server can often optimize the execution of a batch of statements, leading to further performance gains.

3. Lower Database Overhead: The database processes multiple operations together, reducing parsing and execution overhead.

4. Efficient Resource Usage: Reusing the same connection and prepared statement for multiple operations saves resources.

5. Optimized Transaction Handling: Batch execution can be wrapped in a single transaction, reducing commit overhead.

Overall, batch processing reduces execution time and system load, especially for large sets of data.

Question: 3

Write a JDBC program to retrive a specific Student's details based on their ID from the Student table using a PrepareStatement. Display the result in the console.

Answer:

```
import java.sql.*;
public class GetStudent {
 public static void main(String[] args) throws
 SQLException, ClassNotfound Exception {
 Class.forName("oracle.Jdbc.driver.OracleDriver");
 try (Connection c = DriverManager.getConnection(
 "Jdbc:oracle:thin:@localhost:1521:college", "u","p");

PreparedStatement S = c.prepareStatement(" SELECT
name, major FROM Student WHERE id=?"))
{
   S.SetInt(1,123);
```

```
try (ResultSet r = s.executeQuery())
{
  if(r.next())
  System.out.println(r.getString(1)+" "+r.getString(2));
  else
  System.out.println("Not found");
    }
   }
  }
}
```

Example Output:

If a student with ID 123 is found, the output might look like:

John Doe Computer Science

If the student is not found, the output will be:

Not found.

Question: 4

Implement a JDBC transaction where a student's registration is inserted into the "Registration" table only if a course slot is available in the "Courses" table. Demonstrate how to use commit and rollback.

Answer:

```
import java.sql.*;
public class RegisterStudent{
  public static void main(String[] args) throws
SQL Execution, classNotfoundException {
class.forName("oracle.Jdbc.driver.OracleDriver");
try(Connection c = DriverManager.getConnection(
"Jdbc:oracle:thin:@localhost:1521:college","u","P"))
{
    c.setAutoCommit(false);
try(PreparedStatement PS = c.prepareStatement(
"SELECT available slots FROM Courses WHERE
```

```
course_id = ?");

PreparedStatement is = c.prepareStatement(
"INSERT INTO Registration (Student_id, course_id)
values(?,?)"))
{
    int cid = 101;
    ps.setInt(1, cid);
  try(ResultSet r = ps.executeQuery())
   {
     if(r.next() && r.getInt(1)>0)
     {
        is.setInt(1,123);
        is.setInt(2, cid);
        is.executeUpdate();
         c.commit();
         System.out.println("Registered");
     }
     else{
      System.out.println("No Slot");
      c.rollback();
}}}}
```

How commit Works:

1. Transaction Started:

c.setAutoCommit(false); begins the transaction.

2. Operation Performed:

The code checks for available slots and then inserts the registration (if spts are available)

3. C.Commit() called (on success):

If all operations are successful (no errors), the c.commit(); statement is executed. This permanently saves the changes made during the transaction to the database. The changes become visible to other database users.

4. Transaction Ends:

After c.commit(), the transaction is completed successfully.

How Rollback Works:

1. Transaction Started: c.setAutoCommit(false); starts a transaction. All database operations after this point are part of the transaction.

2. Error Occurs: The simulated error (or any actual SQL Exception) is caught in the catch block.

3. Rollback Called: c.rollback(); is called. This tells the database to undo all changes made within the current transaction. The database state is reverted to what it was before the transaction began.

4. Transaction Ends: The catch block handles the exception, and the transaction is effectively terminated (due to the rollback)

5. Compare Statement, PreparedStatement, and CallableStatement in JDBC. When should you use each and what are the advantages and disadvantages.

Answer:

Comparison of Statement, PreparedStatement and CallableStatement in JDBC:

1. Statement:

Statement is used to execute simple SQL queries without parameters.

Usage: Primarily used for executing static SQL queries where parameters are not involved.

Advantages:

- Simplicity: Easy to use for simple queries
- Performance: Useful for executing a single, simple SQL query

Disadvantages:

- Vulnerable to SQL Injection: Since parameters can't be bound, user input is directly included in the SQL, making it vulnerable to SQL injection attacks.
- Efficiency: Every time the query is executed, the SQL Statement is recompiled, which is less efficient than Prepared Statement.
- When to Use: Use when you need to execute a single, simple SQL query with no parameters.

2. PreparedStatement:

PreparedStatement is used to execute precompiled SQL queries with parameters providing an efficient way to run dynamic queries.

• Usage:

1. Used when you need to execute SQL queries with input parameters.

2. It allows for multiple executions of the the same query with different values, offering performance improvements due to precompilation.

• Advantages:

1. Precomplied SQL: The SQL query is precompile by the database, improving performance if executed multiple times.

2. Prevents SQL Injection: The Parameters are bound to placeholders (?), which protects against SQL injection attacks.

3. Reusability: Can be reused multiple times with different parametersvalues.

• Disadvantages:

1) Not flexible for complex Queries: It cannot be used for executing stored procedures or complex

dynamic queries.

- When to Use: Use when executing SQL queries with parameters (e.g., SELECT, INSERT, UPDATE, DELETE) when query permormance and security are important.

3. CallableStatement:

CallableStatement is used to execute SQL stored procedures and functions.

Usage:

1) It is used to call stored procedures or functions in the database.

2) It can handle both input and output parameters for stored procedures.

Advantages:

1) Execute Stored Procedures: Allows executing stored procedures, which can encapsulate complex logic in database.

2) Supports IN, OUT, and INOUT Parameters: Useful for procedures that take input parameters and return output parameters.

3) Performance: Stored procedures are often more efficient than executing multiple individual SQL queries, especially for complex business logic.

Disadvantages:

(1) Database Dependency: The use of stored procedures is database-specific, making your application less portable.

(2) Complexity: Working with stored procedures can add complexity to your application.

When to Use: Use when you need to execute a stored procedure in the database, especially when the logic is complex or needs to be reused across multiple applications.

Comparison:

Feature	Statement	Prepared Statement	Callable Statement
SQL Type	Static SQL	Dynamic SQL with parameters	Stored procedures or functions
Execution	Single execution, recompiled each time	Precompiled, executed multiple times	Execute stored procedures
Security	Vulnerable to SQL injection	Prevents SQL injection	Prevents SQL injection
Performance	Lower, recompiled each time	Higher, precompiled	High, Stored procedures are optimized.
Use Case	Simple queries without parameters	Dynamic queries with parameters	Calling stored procedures
Advantages	Simple to use	Efficient, reusable secure	Supports complex logic in DB
Disadvantages	Vulnerable to SQL injection	Not flexible for complex queries	Database specific more complex

When to use Each:

1. Use Statement:

→ For simple, one-time queries without parameters.

→ Example : SELECT * FROM Employees;

2. Use PreparedStatement:

→ When executing SQL queries with parameters (inserts, updates, selects with WHERE conditions).

→ Example : SELECT * FROM Students WHERE Student_id = ?;

→ Preferred for security (prevents SQL injection) and performance (precompiles the SQL).

3. Use CallableStatement:

→ When you need to execute stored procedures or functions.

→ Example : CALL procedure_name(?,?);

→ Useful for executing complex database logic encapsulated in stored procedures.

Question-6

Explain how to use stored procedures with JDBC. Write a JDBC program to call a stored procedure that retrives all students enrolled in a specific course.

Answer:

A stored procedure is a set of SQL queries that can be executed on the database server. In JDBC, you can call stored procedures using the CallableStatement interface. This is useful when you want to encapsulate complex SQL logic within the database for reusability and better performance.

Here's a simple step-by-step guide on how to use stored procedures in JDBC:

1. Create the Stored Procedure:

First, we must have the stored procedure in the database. This is done using SQL and the procedure will typically perform a task like querying, updateing or inserting data.

2. Create a callableStatement Object:

Use connection.prepareCall() to create a callableStatement

This is used to call the stored procedure.

3. Set Input Parameters (if any):

If the stored procedure requires input parameters, you set them using methods like ~~setIN~~ setINT(), setString() etc.

4. Execute the Stored Procedure:

You execute the stored procedure using executeQuery() (for SELECT) or executeUpdate() (for INSERT / UPDATE).

5. Process the Results:

If the procedure returns data, you can process it using a ResultSet.

JDBC Program :

```
import java.sql.* ;
public class GetStudentsByCourse {
  public static void main ( String[] args) throws SQLExce-
 eption, ClassNotFoundException {
      Class.forName ("Oracle.Jdbc.driver.OracleDriver");
    try ( Connection c = DriverManager. getConnection(
      "jdbc:oracle:thin:@localhost:1521:college","u","p");
CallableStatement s = c.prepareCall (
 "{call GET_STUDENTS_BY_COURSE (?,?)}"))
{    int cid = 101 ;
       s.registerOutParameter (2, Types.REF_CURSOR);
        s.execute ();
        try (ResultSet r = (ResultSet) s.getObject(2))
      {   while (r.next())
                 System.out.println(r.getString (1)+
                              " "+r.getString(2));
            }
         }
  }   }
```

Question-7

What is RowSet in JDBC? Explain the different types of RowSets available and their advantages over ResultSet.

Answer:

RowSet: In JDBC, a RowSet is an interface that holds data retrieved from a database in a set-like manner. It's similar to a ResultSet, but with added functionalities. Unlike a ResultSet, which is directly connected to the database and can only move forward, a Rowset can disconnect from the database and can move both forward and backward. It's like having a copy of the data that we can work with offline or manipulate more easily.

Types of RowSets:

There are several types of RowSets, each with different capabilities:

1. JdbcRowSet: The most basic RowSet. It's similar to a ResultSet with some extra features: It remains connected to the database.

2. CachedRowSet: This RowSet fetches all the data from the database into memory and then disconnects. This allows for offline operations, scrolling and data updates. It's serializable, making it easy to transfer data between application tiers.

3. WebRowSet: Extends CachedRowSet and is designed for XML-based data exchange over the network. It's also serializable and suitable for web services.

4. FilteredRowSet: This RowSet allows you to apply filters to the data without querying the database again.

5. JoinRowSet: This RowSet allows you to combine data from multiple tables (similar to a SQL JOIN) without writing the JOIN in the original SQL query

Advantages of RowSets over ResultSets:

1. Disconnectivity: CachedRowSet and WebRowSet can disconnect from the database after fetching data, freeing up database resources and enabling offline work. JdbcRowSet remains connected.

2. Scrollability: All RowSets are scrollable, allowing you to move the cursor forward and backward, unlike ResultSets.

3. Updatability: Most RowSets are updatable, letting you modify data and save changes back to the database. ResultSet updatability depends on the driver and database.

4. Serializability : CachedRowSet and WebRowSet are serializable, making them easy to transfer between application components or store in files. ResultSets are generally not serializable.

5. Flexibility: RowSets are versatile and can be used for data caching, transfer, and client-side manipulation.

6. Filtering and Joining: FilteredRowSet and JoinRowSe provide built-in functionalities for data manipulatio without database queries.

Question-8

Write a Hibernate program to update an existing student's email address in the database. Explain the use of Session.update() method.

Answer:

```
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

Public class UpdateStudentEmail {
    public static void main (String[] args) {
  Session session = new Configuration().configure(
                "hibernate.cfg.xml").addAnnota-
tedClass(Student.class).buildSessionfactory().
  getCurrentSession();

  try {
      session.beginTransaction();
    Student student = session.get(Student.class, 1);
```

```
if (student != null)
{
    student.setEmail("newemail@example.com");
    session.update(student);
}
session.getTransaction().commit();
System.out.println("Student's email updated
                    successfully!");
}
finally {
        session.close();
    }
  }
}
```

Use of Session.update() Method :

→ The session.update() method in Hibernate is used to update an existing record in the database.

→ It takes an object as an argument and performs an update operation on the database, using the primary key (id in this case) to identify the record.

→ The object you pass to update() must be in a persistent state, i.e. it should have been loaded from the database or its ID must be set correctly.

Summary:

- Hibernate allows easy updating of database records using Session.update().
- The update() method is used to update an existing entity based on its primary key.

Question-9

What is lazy loading in Hibernate? How does it affect performance, and how can you disable it? Provide an example.

Answer:

Lazy loading in Hibernate is a technique where associated entities are not fetched from the database until they are actually needed. Instead of loading all the related data upfront, Hibernate delays the loading of related entities until you explicity access them.

How Does Lazy Loading Affect Performance?

- Improved Performance: Lazy loading improves performance by reducing unnecessary database queries. It fetches related data only when required, rather than fetching it all at once.
- Problem with LazyInitializationException: If you try to access a lazily loaded association outside

of a session (after the session is closed, you'll get a LazyInitializationException.

How to Disable Lazy Loading:

Lazy loading can be disabled in two ways:

1. Using Fetch Type (Eager loading):

You can change the fetch type from LAZY to EAGER in your entity mappings, which will fetch the related data immediately.

2. Explicit Fetching:

You can use HQL (Hibernate Query Language) or criteria queries to fetch the related entities explicity.

Entity class with Lazy Loading:

```
import javax.persistence.*;
import java.util.List;
@Entity
public class Student {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String name;
  @OneToMany(fetch = FetchType.LAZY)
    private List<Course> courses;
    public List<course> getCourse() {
      return course;
    }
  public void setCourses(List<Course> courses){
    this.courses = courses;
  }
}
```

fetching Data With Lazy Loading :

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class LazyLoadingExample {
    public static void main(String[] args){
        SessionFactory factory = new Configuration()
            .configure("hibernate.cfg.xml")
            .addAnnotatedClass(Student.class)
            .addAnnotatedClass(Course.class)
            .buildSessionFactory();
        Session session = factory.getCurrentSession();
        try {
            session.beginTransaction();
            Student student = session.get(Student.class,1)
        System.out.println("Student: "+student.getName());
```

```
System.out.println("Courses: " + student.getCourses());
        session.getTransaction().commit();
    }
    finally {
        factory.close();
    }
  }
}
```

Disabling Lazy Loading (Eager Loading)

```
@OneToMany(fetch = FetchType.EAGER)
private List<Course> course;
```

Hibernate

Hibernate is a framework for Java that simplifies interacting with databases. It is an Object-Relational Mapping (ORM) tool that helps you map Java objects (classes) to database tables.

Key points about Hibernate:

1. Simplifies Database Operations:

It makes it easier to work with databases by allowing you to use Java objects instead of writting SQL queries manually.

2. Mapping Java Objects to Database Tables:

It automatically maps Java classes to database tables and Java object properties to table columns.

3. No Need for SQL: You can save, update, delete, and retrive data from the database using Java code without writing raw SQL

4. Automatic Query Generation:

Hibernate automatically generates SQL queries based on the Java objects and operations you perform.

5. Cross Database Compatibility:

Hibernate works with different types of databases (like MySQL, Oracle, SQL Server) without changing your code, making it flexible.

If you have a Student Java class, Hibernate will handle saving this object to the student table database:

```
@Entity
public class Student {

  @Id
  @GeneratedValue(Strategy = GenerationType.
                                IDENTITY)
  private int id;
  private String name;
}
```

Hibernate will map this Student class to a database table student and handle operations like inserting, updating and retrieving student records.

Why Hibernate is Useful:

→ Simplifies database interaction by eleminating the need for complex SQL

→ Handles CRUD operations (CREATE, READ, UPDATE, DELETE) automatically

→ Improves productivity by making Java code cleaner and easier to maintain.

In Short, Hibernate is a powerful tool for Java developers to work with databases in a more efficient and automated way.

Question-14

What is the purpose of the "struts-config.xml" file? Explain its key components, including Action mapping form beans and global forwards.

Answer:

The "struts-config.xml" file is a core configuration file used in Apache Struts framework.

It defines how the web application handles incoming requests, links them to appropriate actions (bussiness logic), and manages the flow between views and actions. It is the main configuration that controls the behavior of the application.

Key purposes:

1) Action mappings: Maps a URL or request to a specific Action class which contains the business logic for that request.

2. Form Beans:

Defines the form beans that are used to collect and transfer user input (like from a form submission) to the backend logic.

3. forward Mappings:

Defines the forward paths (URLs or JSP pages) to send a user after an action has been processed. These can be global or specific to an action.

4. Global Settings:

configures error handling, global forwards and other settings for the framework.

Struts-config.xml :-

→ ~~help~~ Struts map URLs to actions

→ manage data transfer (via form bean)

→ control the flow of the application.

Key Components of Struts-config.xml :

1. Action Mappings :

Purpose: Defines the relationship between a URL (request) and the corresponding Action class (which contains business logic)

Example :

```
<action path = "/login"
        type = "com.example.LoginAction"
        name = "loginform"
        scope = "request"
        validate = "true"
        forward = "/login.jsp">
</action>
```

Explanation : The configuration maps the /login URL to the LoginAction class. When a user visits this URL, the LoginAction class is executed. It also defines which form bean (loginform) is used and where to forward the result.

2. Form Beans:

Purpose: form beans are used to hold data submitted by the user and are associated with an action.

Example:

```
<form_beans>
  <form-bean name="loginform"
        type="com.example.loginform"/>
</form_beans>
```

Explanation: This defines a form bean loginform which is associated with the loginForm Java class to handle form data for login.

3. Global Forwards:

Purpose: Defines common or reusable forwards that can be used across different actions in the application. These are usually views or pages to which users can are forwarded.

after certain action.

Example:

```
<global-forwards>
  <forward name="success" path="/success.jsp"/>
  <forward name="error" path="/error.jsp"/>
</global-forwards>
```

Explantation: Defines global forwords for success and error pages that can be referenced by different actions in the application.

Summary:

- Action Mappings: Maps URLs to actions (business logic)
- Form Beans: Store user input data
- Global Forwards: Reusable forwards for common pages (like success/error pages)

Question-16

Explain how interceptors work in the Struts framework. Provide an example of an interceptor that logs request details before processing an Action.

Answer:

How interceptors works in the Struts framework:

In Struts 2, interceptors are used to perform tasks before or after an action is executed. They allow developers to add common functionality (like logging, validation or security checks) across multiple actions without modifying each action class.

Interceptors work in the following way:

1. Before an Action Execution: Interceptors can perform tasks before the actual action is

executed (like logging request details, checking user permissions)

2. After Action Execution: They can perform tasks after the action is executed (like modifying the response or handing any post-processing)

Example: Interceptor that Logs Request Details:

1. Create the Interceptor class:

```
import com.opensymphony.xwork2.ActionInvocation;
import com.opensymphony.xwork2.interceptor.MethodFilterI
                                                terceptor;
public class LogRequestInterceptor extends
                                    MethodFilterIntercepto
@ Override
protected String doIntercept (ActionInvocation invocation)
throws Exception {
System.out.println("Request Details: " + invocation.
getInvocationContext().getParameters());
```

~~invocation.invoke();~~

return invocation.invoke();

}

}

2. Configure the Interceptor in struts.xml:

```
<struts>
    <package name="default"
             extends="struts-default">
        <interceptors>
            <interceptor name="logRequest"
                         class="com.example.LogRequestInterceptor"/>
        </interceptors>

        <action name="login"
                class="com.example.LogInAction">
```

```
<interceptor_ref name = "logRequest"/>
    <result> /login.jsp </result>

  </action>
  </package>
</struts>
```

Summary:

→ Interceptors are used for cross-cutting concerns like logging, validation and security.

→ The interceptor runs before or after the action execution.

→ You define and configure interceptors in struts.xml.

Question-16

What is the difference between Struts 1 and Struts 2 ? Discuss improvements in Struts 2 and why it is preferred over Struts 1.

Difference between Struts1 and Struts 2 :

1. Architecture :

- Struts 1 : follows the traditional Model-View-Controller (MVC) architecture with a centralized ActionServlet for request processing.
- Struts 2 : follows MVC too but is based on ActionContext and uses filter-based architecture instead of a single servlet.

2. Action Handling :

- Struts 1: Requires developers to extend the Action class and override its execute() method.
- Struts 2: Makes use of POJOs (Plain Old Java objects) for actions, no need to extend a specific class. Also, action methods can directly return views.

3. Configuration :

- Struts 1: Configuration is done in struts-config.xml.
- Struts 2: Uses struts.xml for configuration, which is more flexible and cleaner.

4. Tag Libraries :

- Struts 1: Uses HTML tags (e.g. <html:form>) that are limited and not very flexible.
- Struts 2: Uses JSP tags like <s:form> and <s:textfield>, which are more powerful and easier to use.

5. Model Binding :

- Struts 1 : Manual form bean management and population is required.
- Struts 2 : Automatic model binding : automatically binds form fields to action properties.

6. Interceptors :

- Struts 1 : No built-in interceptors ; tasks like logging or validation had to be manually added.
- Struts 2 : Uses interceptors for reusable cross-cutting concerns (e.g. logging, validation, security checks), which makes the application cleaner and more modular.

7. AJAX Support :

- Struts 1 : AJAX support is limited and requires third-party libraries
- Struts 2 : Built-in AJAX support with easy integration of AJAX actions.

Improvements in Struts 2:

1. Simpler and more flexible architecture (filter-based, not servlet-based).
2. POJO-based Actions, no need for custom action classes.
3. Powerful tag libraries that are easier to use
4. Built-in Interceptors to handle repetitive tasks.
5. Automatic form data binding to Java objects (less manual coding).

Why Struts 2 is Preferred Over Struts 1:

- Easier to use: POJO actions and cleaner configuration
- Better performance: filter-based architecture, less overhead.
- More features: Built-in interceptors, AJAX support, automatic form binding.
- Greater flexibility: Easier to integrate with other frameworks.

In short, Struts 2 is more flexible, modern and easier to work with compared to Struts 1

Question:18

Compare forward() and sendRedirect() in Servlets. When should you use each, and how do they impact performance?

Answer:

forward():

→ RequestDispatcher.forward() forwards the request to another resource(like another servlet or JSP) within the same server.

How it works: The client doesn't know the request has been forwarded; the URL in the browser remains the same.

When to use: When we need to forward the request to another servlet or JSP in the same application.

Impact on performance: faster, as it doesn't involve a new request or response. It happens within the same server.

Example:

```
RequestDispatcher dispatcher = request.
getRequestDispatcher("anotherServlet");
dispatcher.forward(request, response);
```

sendRedirect():

→ HttpServletResponse.sendRedirect() sends a response to the client telling them to make a new request to another URL.

How it works:

The browser's URL changes to the new URL, and a new HTTP request is made.

When to use:

When you want to redirect the client to a different server or a different URL.

Impact on performance: Slower, because it involves sending a new HTTP request to the server.

Example:

```
response.sendRedirect("anotherPage.jsp");
```

Key Differences:

- forward(): No URL change, same request, within the same server.
- sendRedirect(): URL changes, new request, can be to another server.

When to use each:

- Use forward() when you want to stay within the same server without the client knowing.
- Use sendRedirect() when you need to redirect the client to a new page or another server.

Question-19

How does ServletContext differ from HttpSession? Provide examples of scenarios where each should be used.

Answer:

ServletContext: An interface that provides a way for servlets to interact with the web application's environment and share global data across all user and sessions.

Scope: Application-wide (shared across all users).

Purpose: Used to store global data that should be accessible to all users and across all sessions.

Lifetime: Exists as long as the application is running (until the web server is stopped).

Example Scenario:

- Store application-level settings such as database connections, or track global statistics like total page views.

Example code:

```
getServletContext().setAttribute("appVersion", "1.0");

String appVersion = (String) getServletContext().
                    getAttribute("appversion)
```

When to use: When you want to store data that should be shared by all users (e.g., application-wide settings, constants)

HttpSession:

HttpSession is an interface that allows you to store user-specific data across multiple requests during a session.

Scope: User-specific (unique for each user session)

Purpose: Used to store data that is specific to user's session, such as login information or user preferences.

Lifetime: Exists as long as the user's session is active, typically until the user logs out or the session times out.

Example Scenario:

- Store user-specific data such as the user's login status or shopping cart.

Example code:

```
session.getAttribute("username", "JohnDoe");
String username = (String) session.getAttribute
                          ("username");
```

When to use:

When you need to store data that is specific to an individual user, such as user login credentials, perferences, or session-specific data. (e.g. shopping cart items).

Key Differences:

ServletContext	HttpSession
Application-wide, shared by all users.	User-specific, unique for each session.
Lives as long as the application is running	Lives as long as the user's session is active
Store global or application-wide data.	Store user-specific data.
Example: Database connection, application version, global counters.	Example: User login info, shopping cart, user preferences.

Question-20

Explain the concept of filters in Java Servlets. Write an example of a Servlet filter that logs request parameter before passing the request to the target servlet.

Concept of Filters in Java Servlets:

A filter in Java Servlet is an object that perform filtering tasks on either the request or response or both before passing the request to a servlet or after the servlet has processed the request. Filter are used for tasks like logging, authentication, data compression and modifying request or response objects.

Key points:

- Request filtering: Manipulating or inspecting the incoming request before reaching the servlet.
- Response filtering: Manipulating or inspecting the response before it is sent to the client.

Lifecycle: Filters are configured in the web.xml file and invoked based on the pattern defined.

A filter that logs the request parameters before forwarding the request to the target servlet is given below:

1. Create the filter class:

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io*;

public class LogRequestFilter implements Filter{

 public void int (FilterConfig filterConfig)
 throws ServletException {

}
```

```
public void doFilter(ServletRequest request, ServletRes
ponse response, FilterChain chain) throws IOExceptio
ServletException {
    HttpServletRequest httpRequest = (HttpServletRequest)
                                        request;
    System.out.println("Request URL:" + httpRequest.
                                    getRequestURL());
    System.out.println("Request Parameters: ");
    httpRequest.getParameterMap().forEach((key, value) ->
        System.out.println(key + ": " + String.join(",", val
        ));
        chain.doFilter(request, response);
    }
    public void destroy(){
    }
}
```

Configure the filter in web.xml :

```
<filter>
    <filter-name>LogRequestFilter</filter-name>
    <filter-class>LogRequestFilter</filter-class>
</filter>

<filter-mapping>
    <filter-name>LogRequestFilter</filter-name>
    <url-pattern>/yourServletPath</url-
                                pattern>
</filter-mapping>
```

Explanation:

→ The filter LogRequestFilter logs the request parameters before passing the request to the servlet using chain.doFilter().

→ The web.xml file maps the filter to a servlet URL pattern, so it intercepts all requests to that URL.

Question

What

JSP

U

Question-21

What are scriptlets, expressions and declarations in JSP? Compare them and discuss when each should be used.

Answer:

In JSP (JavaServer Pages), there are three main types of code elements that are used to embed Java code in an HTML Page: Scriptlets, expressions and declarations.

Scriptlets: A scriptlet in JSP is a block of Java code that is embedded directly within the HTML code. It is written between <% and %> tags. When the page is requested, the Java code in the scriptlet is executed, and any output generated is inserted into the response.

Syntax:

```
<%
    // Java code goes here
%>
```

Example:

```
<%
    int x=10;
    out.println("The value of x is:"+x);
%>
```

When to use Scriptlets:

1) Scriptlets are useful when you need to execute Java logic or business rules on the server side before generating the dynamic content for the page.

2) They should be used sparingly, as they mix Java code with HTML, making the page harder to maintain and debug.

3) Typically, they are used for operations that generate dynamic content or perform server-side processing (e.g., setting up session date or calculations).

Expressions :

An expression in JSP is a shorthand for embedding Java code that outputs a result directly to the response. Expressions are evaluated and converted into a string that is then inserted into the generated HTML.

Syntax : <%= expression %>

Example :

```
<%= "Hello," + userName %>
```

When to use Expressions :

1. Use expressions when you need to display the result of a Java expression directly within the HTML
2. An expression is evaluated and immediately inserted into the response stream.
3. They are typically used for simple output, such as displaying dynamic values (e.g., user names, dates, etc.).

Declarations :

Declarations in JSP are used to declare variables and methods that can be used later in the page. They are written between <%! and %> tags. The code inside the declaration block is placed within a class definition and can be accessed by other parts of the JSP page.

Syntax:

```
<%!
    // Declare variables or methods
%>
```

Example:

```
<%!
    private int count=0;
    public int increment() {
        return ++count;
    }
%>
```

When to use declarations:

(1) Use declarations when you need to define variable or methods that will be used across multiple parts of the JSP page or across multiple requests

(2) Declarations are typically used for persistent data (e.g., counters or helper methods) that can be used across multiple requests within the same session.

Feature	Scriptlets	Expressions	Declarations
Purpose	Embed Java code that can generate dynamic content or perform operations	Output the result of a Java expression directly to the response	Declare variables or methods that can be used later in the JSP
Syntax	<% code %>	<%= expression %>	<%! code %>
Output	No direct output (typically modifies output with out.println())	Direct output to the response stream	No direct output; declares variables/ methods.
Use case	Execute Java logic (e.g., calculations, session handling)	Display values directly within HTML	Define helper methods or class-level variables
Performance	More resource-intensive due to multiple context switches	More efficient for simple output as it directly returns the result.	Should be used cautiously to avoid too many global variables

Question-22

Explain how JSP custom tags work. Write a simple JSP custom tag to display a welcome message dynamically.

Answer:

JSP custom tags allow you to define reusable components in a JSP page. These tags are written in Java and provide a way to encapsulate logic, making JSP code cleaner and more maintainable. Custom tags are defined in a tag handler class, and they are associated with a tag in the JSP page using the <%@taglib %> directive.

How Custom Tags Work:

1. Tag Handler class: A Java class that implements specific methods (like doStartTag() and doEndTag()) to define the behavior of the custom tag.

2. Tag Library Descriptor (TLD): A configuration file that maps the custom tag to the Java class.

3. JSP Page: The custom tag is used within the JSP page, like a regular HTML Tag.

Steps to Create a Custom Tag to Display a Welcome Message:

1. Create a Tag Handler class:

This class will define the behavior of the custom tag.

```
import javax.servlet.jsp.tagext.SimpleTagSupport;
import java.io.IOException;
public class WelcomeTagHandler extends
                SimpleTagSupport {
    private String name;
```

```
public void setName (String name){
    this.name= name;
}
@Override
  public void doTag() throws IOException{
  getJspContext().getOut().write("Welcome,"+name+"!");
  }
}
```

2. Create the Tag Library Descriptor (TLD) File:

This file maps the custom tag to the Java class.

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema
        -instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/
     j2ee http://java.sun.com/xml/ns/j2ee/web-
    jsptaglibrary_2_0.xsd">
```

```
<tlib-version> 1.0 </tlib-version>
<short-name> WelcomeTag </short-name>
<uri> /WEB-INF/tlds/ welcome-taglib </uri>

<tag>
    <name> welcome </name>
    <tag-class> WelcomeTagHandler </tag-class>
    <body-content> empty </body-content>
    <attribute>
        <name> name </name>
        <required> true </required>
    </attribute>
</tag>
</taglib>
```

3. Use the Custom Tag in the JSP Page:

Now, you can use the custom tag in your JSP page by referencing the tag library.

```
<%@ taglib uri = "/WEB-INF/tlds/welcome-taglib"
        prefix = "mytags" %>

<html>
<body>
    <h1> <mytags: welcome name = "John" /> </h1>

</body>
</html>
```

Question: 23

Compare and contrast FileReader, BufferedReader and Scanner in Java. Discuss their use cases, performance implications and any advantages or disadvantages you have observed while working with them.

Answer:

In Java, FileReader, BuffereReader and Scanner are all commonly used classes for reading data from files or other input sources, but they have different purposes, performance implications, and use cases.

Here's a comparison and contrast of these classes:

1. FileReader

- Purpose: FileReader is used to read the content of a file as a stream of characters. It is a basic reader class that provides a simple way to read individual characters from a file.

How it works: It reads the file character by character (or in small chunks) using a file input stream.

Use cases:

- Simple file reading when you need to process a file character by character.
- When you don't need additional features like buffering or parsing.

Performance:

- Slower compared to BufferedReader and Scanner because it reads one character at a time from the file.

Advantages:

- Simple and easy to use for small, simple tasks..
- Reads files as a sequence of characters, which is useful for text files.

Disadvantages:

- Inefficient for larger files because it doesn't use buffering, leading to multiple disk reads.
- No advanced features like line parsing or tokenization.

2. BufferedReader

- Purpose: BufferedReader is used to read text from an input stream more efficiently. It buffers characters, thus reducing the number of read operations required to read a file. It is typically used in combination with other reader (like fileReader).

- How it works : It reads chunks of data into a buffer, then processes it, allowing for faster reading operations compared to FileReader

Use cases :

- Reading large text files efficiently.
- Reading lines of text, as BufferedReader provides the readLine() method.

Performance :

- Much faster than FileReader for large files because it uses an internal buffer to minimize disk I/O.
- Efficient for line-by-line reading.

Advantages:

- Performance improvement over FileReader due to internal buffering.
- Provides the readLine() method, which makes it easy to read lines of text.

Disadvantages:

- It can only read text and cannot handle binary data.
- No built-in support for tokenization or parsing data.

3. Scanner

- Purpose: Scanner is a versatile class that can be used to read different types of data (like strings, integers etc.) from various sources, including files, strings, or user input.

How it works: It can read tokens (e.g., words, numbers) from a stream and also has methods to parse primitive types (like nextInt(), nextDouble()).

Use cases:

- Reading input with tokenization
- Parsing specific data types from a file
- Reading data from multiple sources.

Performance:

- Slower than BufferedReader because it performs additional operations for parsing tokens.
- Not as fast as BufferedReader for simple file reading due to internal parsing.

Advantages:

- Flexible: Can read different types of data (e.g., strings, numbers, etc) and can easily tokenize input.
- Convenient for interactive input.
- Supports advanced parsing features

Disadvantages:

- Slower than BufferedReader for simple file reading due to tokenization overhead.
- Not as efficient for large files as BufferedReader.
- More complex than FileReader or BufferedReader for simple tasks.

Question-25
Compare
wheth
of
m

FileReader	BufferedReader	Scanner
Read characters from a file	Read lines of text efficiently	Tokenize and parse different data types from a stream.
It reads character by-character	It reads buffered character stream.	It reads token-based reading
Speed is slow	Speed is fast	Slower than BufferedReader due to parsing overhead.
Not efficient for large files.	Efficient for large files.	Not efficient for large files.
It doesn't read line-by-line	It reads line-by line	It reads line-by-line
It has no parsing capabilities	It has no parsing capabilities	It has parsing capabilities
It's simple	It's simple to moderate	It's moderate

Question-25

Compare and analyze different approaches for checking whether a number is prime in Java. Discuss the efficiency of iterative methods (loop-based approach) versus recursive methods and built-in mathematical functions. Analyze their time complexity, memory usage, and real-world Applicability

Answer:

1. Iterative method (loop-based approach):

The iterative approach involves looping through numbers and checking if the given number n is divisible by any number between 2 and n-1. If no divisor is found, then the number is prime.

Code:

```
public boolean isPrimeIterative(int n) {
    if (n<=1) return false;
    for (int i=2; i<= Math.sqrt(n); i++) {
        if (n%i==0) {
            return false;
        }
    }
    return true;
}
```

Time Complexity:

Best Case : $O(1)$ [for $n=2$ or $n=3$]

Worst Case: $O(\sqrt{n})$

Memory usage:

Constant space $O(1)$, since we are only using a fixed amount of extra memory for the loop counter and a few variable

Real-world Applicability:

- The method is efficient enough for checking the primality of number up to moderate sizes (10^6 to 10^7)
- Good choice when you don't need extreme performance but want simplicity and clarity.

2. Recursive Method:

```
public boolean isPrimeRecursive (int n, int i) {
    if (n<=1) return false;
    if( n==Math.sqrt(n)+1) return true;
    if (n% i ==0) return false;
    return isPrimeRecursive (n, i+1);
  }
public boolean isPrime(int n) {
    return isPrimeRecursive( n, 2);
  }
```

Time Complexity:

Worst case: $O(\sqrt{n})$

Memory Usage:

Stack space $O(\sqrt{n})$, because each recursive call adds a new frame to the call stack, and we need about $\sqrt{n}$ calls before reaching the base case.

Real-World Applicability:

i. Less efficient

ii. Not recommended for large n

iii. Suitable for educational purposes or small-scale use.

3. Built-in Mathematical Function (BigInteger):

```
import java.math.BigInteger;
public boolean isPrimeBigInteger(long n){
    BigInteger bigN = BigInteger.valueOf(n);
    return bigN.isProbablePrime(1);
}
```

Time Complexity:

worst case: O(log n)

Memory Usage:

O(log n) space, as BigInteger needs to store the number in special data structure for large integer.

Real-World Applicability:

1) Ideal for very large number
2) very efficient for numbers with hundreads or thousands of digits.
3) Probabilistic
4) Best for cases when speed is more important than absolute correctness and the numbers are extremely large.

Question-27

What are the key principles of creating an immutable class in Java? If all fields are private and there are no setters, does that guarantee immutability? How does immutability impact performance and thread safety? Provide an example of an immutable class and explain its benefits.

Answer:

An Immutable class is a class whose objects cannot be modified after they are created. Once an instance of the class is ~~cant~~ constructed its state cannot change.

Key Principles of Creating an Immutable class in Java:

1. Declare the class as final: This prevents subclassing which could alter its behavior.
2. Make all fields private and final: This ensures fields can only be assigned once and cannot be modified.
3. Initialize fields in the constructor: Set values only in the constructor so they can't be changed later.
4. No setter methods: This prevents modification of fields after object creation.
5. Deep copy mutable objects: If the class holds references to mutable objects, return a copy of the object, not the original, to prevent external modifications.

Having all fields private and no setters not guaranteed immutability. If the class holds reference to mutable objects, those objects can still be changed externally. You must ensure that any mutable fields are deeply copied when accessing them.

Impacts ~~of~~ on Performance and thread safety:

- Performance: Immutable objects can be more efficient in some scenarios (e.g., caching, reuse), but creating deep copies may introduce overhead.
- Thread Safety: Immutable objects are inherently thread-safe because their state cannot change once they are created.

Example:

```
Public final class Person {
    private final String name;
    private final int age;
    // Constructor
    public Person (String name, int age) {
        this.name = name;
        this.age = age;
    } //Getters
    public String getName() {
        return name;
    }
    public int getAge() {
        return age;
    }
}
```

Benefits:

1) Thread Saftey
2) Simplicity: Fewer bugs because the state of the object cannot change after creation
3) Caching: Can be shared and reused without concern for unintended changes.

```
import java.math.BigInteger;
public class fibonacciBigInteger {
 public static BigInteger fibonacci (int n) {
   BigInteger a = BigInteger.ZERO;
   BigInteger b = BigInteger.ONE;
  for (int i=2; i<= n; i++) {
    BigInteger c = a.add(b);
         a=b;
         b=c;
   }
   return b;
  }
public static void main (String[] args)
 {
    int n=100;
    BigInteger result= fibonacci(n);
   System.out.println("fibonacci("+n+")=" +result);
  }
}
```

Answer to the Question no- 29

1. ArrayList:

Type: Dynamic Array

Performance:

1. Random access: O(1) - ArrayList provides fast access to elements by index.

2. Insertion / Deletions: O(n) is worst case

- Thread Safety: Not thread-safe.
- Use Case:

Best for scenarios where quick random access to elements by index is required, and the data is frequently accessed without many insertions or deletions in the middle of the list.

2. LinkedList:

Type: Doubly Linked List

Performance:

1) Random Access: O(n) - Slower compared to ArrayList because elements are stored in nodes

2) Insertion/Deletions: O(1) - Insertion and Deletions are fast when done at the beginning or end of the list

- Thread-Safety: Not thead-safe

Use Case: Best for scenarios where frequent insertions and deletions are required.

3. ConcurrentLinkedQueue:

- Type: Lock-free Queue (a thread-safe implementation of a queue)
- Performance:

1) Random access: Not applicable - it's a queque

meaning elements are processed in FIFO.

Random access to individual element is not possible.

2) Insertion/Deletions : O(1), it provides good performance even under high concurrency.

Thread safety : Thread-safe

Use case : Ideal for seenarios that require high concurrency where multiple threads need to add / remove

1. A system that needs fast random access to large Dataset → ArrayList

2. A concurrent task queue where multiple threads add and remove elements → ConcurrentLinkedQueue

3. A navigation system that stores locations in a sequential path but allows quick insertions and deletions at various point → LinkedList

4. A chat application where messages are frequently added at the end and old messages are removed periodically. → LinkedList

Answer to the Question no-31

If thread-safety is not critical : ArrayList for better performance and flexibility.

If thread-safety is important : Use vector or Arraylist with manual synchronization.

But prefer LinkedList for frequent insertions and deletions at both ends.

1. Synchronization and Thread-Safety:

- Vector : Thread-safe, but has synchronized methods, which can slow performance in multi-threaded environments.
- ArrayList & LinkedList :

Not thread-safe by default. If thread safety is needed, you can use synchronization methods.

2. Speed and Performance:

- ArrayList: Faster for random access (O(1)) and better performance in seenarios with fewer insertions/ removals
- LinkedList: Slower for random access (O(n)) but performs better for frequent insertion and deletions (O(1)) for add/remove at ends.
- Vector: Slower than both due to synchorization overhead.

3. Capacity:

- Vector: Doubles its size when capacity is exceeded.
- ArrayList: Increases by 50% when capacity is exceeded.
- LinkedList: No predefined capacity.

4. Enumeration and Iteration:

- Vector: Supports both
- ArrayList: Supports only Iterator.

5. Legacy:

- Vector: A legacy class from the early days of Java.
- ArrayList & LinkedList: Preferred over Vector as they are more modern and widely used.

Answer to the Question no-32

① Multiple Inheritance:

→ Interface allows multiple inheritance. A class can implement multiple interfaces

→ A class can only extend one abstract class due to Java's single inheritance model.

② Code Reusability:

→ Interface only defines method signatures It's best for defining a contract without any shared implementation.

→ Abstract class can have both abstract methods and concrete methods. It's useful for sharing common code.

③ Scalability :

→ Interface preferred when you expect classes to have different implementation but ~~have same~~ share common behavior.

→ Abstract class better for defining a common base class.

④ Impact on code:

→ Interface promotes flexibility in code

→ Abstract class encourages code reuse.

Interface Example:

```
Interface Flyable {
    void fly();
}
class Bird implements Flyable {
    public void fly() {
        System.out.println("Bird is flying");
    }
}
class Plane implements Flyable
{
    public void fly()
    {
        System.out.println("Plane is flying");
    }
}
```

Abstract class:

```
abstract class Vehicle {
    abstract void start();
    void stop() {
    System.out.println("Vehicle is stopping");
    }
}
class Car extends Vehicle {
    void start() {
    System.out.println("Car is starting");
    }
}
```

Answer to the Question no- 33

ExecutorService:

The ExecutorService is a higher-replacement for manually managing threads. It provides an easier and more efficient way to manage a pool of threads.

Advantages of ExecutorService:

1. Thread pool Management
2. Simpler API
3. Task Scheduling
4. Graceful Shutdown
5. Error Handling.

Disadvantages:

1. Overhead
2. Limited flexibility

~~Manual Thread :~~

Manual Thread Management:

In manual thread management, you create individual threads by extending thread or implementing runnable.

Advantages:

1. Flexibility
2. Simple for small Tasks

DisAdvantages:

1. Compleaity
2. Resource Management
3. Scaling Issues.

Answer to the Question no - 34:

① Try-catch-finally :-

Purpose: Used to handle exceptions directly within a method.

How it works:

- try: Block where exceptions might occur
- catch: Block that handles the exception if it occurs.
- finally: Optional block that always runs, regardless of whether an exception occurred.

Advantages:

① Code Readability
② System Stability

Disadvantages:

① Code clutter
② Limited Flexibility

When we want to handle exceptions immediately and recover from them within the same method.

② throws:

purpose: Used to declare that a method might throw an exception, which is handled by the caller.

How it works:

throws is used in the method signature to indicate that the method might handling it to the calling method.

Advantages:

① Produce clean code

② No cluttering

③ Flexibility.

Used when critical exceptions, special/ higher level handling

Question-38

How does the Java EE architecture ensure scalability and maintainability in enterprise applications, and what design patterns are commonly used to achieve these goals?

Answer:

Java EE (Enterprise Edition) provides a robust, scalable and maintainable architecture for developing enterprise applications. Here's how it achieves these goals:

- Scalability:

(1) Multi-tier architecture: Separates client, business logic, and database layers, supporting load balancing and horizontal scaling.

(2) Stateless components (e.g. stateless session beans) reduce memory overhead.

(3) Connection pooling: Efficiently manages database connections.

- Maintainability:

(1) Modularization: Components like EJBs, JSPs, and servlets promote easy updates and testing.

(2) Declarative configuration: Simplifies code with annotations and configuration files.

(3) Standardization: Reduces vendor lock-in using standards like JPA and JTA.

Design Patterns:

(1) MVC (Model-View-Controller):
Separates concerns for easier management.

(2) DAO (Data Access Object):
Abstracts database interactions.

(3) Singleton: Manages shared resources in a thread-safe manner.

(4) Factory: Creates objects dynamically for flexibility.

Java EE ensures scalability and maintainability with a modular, standardized architecture, using design patterns like MVC, DAO and Singleton for flexible efficient development.

Question - 39

How does Java ME's modular architecture (CLDC and CDC) enable application development for resource-constrained devices, and what are the trade-offs compared to full-fledged Java SE applications?

Answer:

Java ME (Micro Edition) is designed for resource-constrained devices like mobile phones, IoT devices and embedded systems. It uses a modular architecture with two main configurations:

1. CLDC (Connected Limited Device Configuration) It's a low-end devices with limited memory, processing power, and network connectivity.

Key Features of CDLC :

(1) Minimal set of APIs : Includes only essential Java features like basic I/O, networking and GUI.

(2) Compact JVM : The Java virtual Machine (JVM) is optimized for smaller devices, consuming less memory and power.

(3) Memory limitations : Typically, CLDC devices have limited heap memory and floating-point support.

2. CDC (Connected Device Configuration):

CDC targets at more powerful devices.

Key features:

1) Larger set of APIs: CDC provides a more extensive API set, similar to Java SE, but still optimized for constrained devices.

2) Full JVM: CDC support richer applications, with more memory and advanced features.

Benefits:

1) Optimized for Small devices:

Java ME is designed to work on devices with limited resources like memory and processing power.

2) Modular design: Only necessary features are included, making the apps lightweight

Trade-offs Compared to Java SE:

1) Limited functionality: Java ME has fewer APIs and features.

2) Performance: Slower than Java SE due to less powerful JVM and memory.

3) Development Complexity: Must be optimized for small resources.

Java ME's CLDC and CDC allow apps to run on small, low-power devices but with limited features and performance compared to full Java SE applications.

40.

Java ME : (Java Platform, Micro Edition) is a version of the Java platform specially designed for developing applications on embedded and mobile devices with limited resources, such as smartphones, feature phones, tablets and IoT devices.

It provides a simplified and optimized set of Java libraries and tools that are lightweight, offering a platform for developer to create small, resource-efficient application.

Key features:

1. Optimized for low resources
2. Modular structure
3. Cross-platform compatibility.
4. Support for networking and connectivity.

Advantages:

1. Cross-Platform Compatibility:
Java Me can run on a variety of devices with minimal changes to the code.

2. Lightweight: It's optimized for devices with limited resources (memory, CPU, battery), making it suitable for older or low-end hardware.

3. Mature Ecosystem: Java ME has been around for a long time, offering a robust set of libraries and tools for mobile development.

4. Security: Java ME has built-in security features (e.g. sandboxing) that are ideal for embedded systems.

Limitations :

1. Limited functionality :

Compared to modern mobile platforms (Android, iOS), Java ME has fewer features and API's, which limits app capabilities.

2. Performance : Java ME's JVM is less powerful, leading to slower performance than native mobile platforms.

3. Decline in Popularity : Modern mobile platforms (Android and iOS) have become the standard, reducing the demand for Java ME.

4. Smaller Ecosystem : Fewer developers and resources compared to Android/iOS development.

Java ME is useful for legacy devices and resource-constrained systems, but its limited features and declinging usage make it less viable . . . .