1.1 Demonstrate how a child class can access a protected members of its parent class within the same package.

Explain with example what happens when the child is in a different package.

Sol: Accessing protecting members in same pakage:

```
package samepkg;
class x {
    protected int num = 10;
}
public class c extends x {
    void dis() {
        System.out.println("Number:" + num);
                                // accessible
    }
}
```

Explanation: In same package protected members are directly accessible like default.

Different package:

```
package otherpkg;
import samepkg.x;
public class cc extends x {
```

```
void dis( ){
    System.out.println(num);
                    // accessible because
                    child extends parents
    }
}
```

Explanation: Accessible only through inheritance (not by parent object).

1.2 Compare Abstract classes and ~~inheritance~~ interface in the term of multiple inheritance. When would you prefer to use an abstract class and when an interface?

Sol:

Abstract class	Interface
(i) Methods: Abstract + concrete	(i) Abstract
(ii) Single inheritance only	(ii) multiple inheritance
(iii) When classes are share state and ~~variable~~ behavior, they are usable.	(iii) When needs share ~~variable~~ behavior only

When to use?

Abstract class: If classes are closely related and share fields & methods.

Interface: To define common capability that many unrelated classes can implemented.

2.1: How does encapsulation ensure data security and integrity? show with a bankAccount class using private variables and validated method such as setAccountNumber(string), setInitialBalance(double), that reject null, negative or empty values.

Sol: Encapsulation (Data Security and Integrity)

```
class BankAccount {
    private string accountNumber;
    private double balance;
```

```
Public void SetAccountNumber (String acc) {
    if (acc != null && !acc.isEmpty()) {
        this.accountNumber = acc;
    }
    else
        System.out.println("Invalid account");
}

public void setInitialBalance (double bal) {
    if (bal >= 0) this.balance = bal;
    else System.out.println("Balance cannot
                             be negative");
}

public String getAccountNumber() {
    return accountNumber;
}
public double getbalance () {
    return balance;
}
}
```

How Encapsulation Ensures Security and Integrity ?

① Data is hidden

② Accessible through only valid methods.

3.1 (i) Find the Kth smallest element in an ArrayList?

(ii) Create a TreeMap to store the mappings of words to their frequencies in a given texts.

(iii) check if two LinkedList are equal.

Sol: (i)

```
import java.util.*;
public class KthSmallest {
    public static void main(String[] args){
        ArrayList list1
        ArrayList<Integer> list = new ArrayList<>();
        Collections.addAll(list, 7, 2, 9, 11, 19);
        int k = 3;
        Collections.sort(list);
    int kthsmall = list.get(K-1);
    System.out.println(k+"th smallest: "+
                                kth smallest);
}
}
```

(ii)
```
import java.util.*;
public class WordFrequency {
    public static void main(String[] args){
        String text = " apple banana apple banana";
        String[] words = text.split(" ");
        TreeMap<String, Integer> freqMap
                = new TreeMap<>();
    for(String word : words) {
        freqMap.put(word, freqMap.getOrDefault(
                word, 0)+1);
    }
    System.out.println(freqMap);
}
}
```

(iii)
```
import java.util.*;
public class EqualLinkedList {
    public static void main(String[] args)
    {
        LinkedList<Integer> list1 = new LinkedList
                <>();
        LinkedList<Integer> list2 = new LinkedList<>
```

```
List1.add(1); List1.add(2); List1.add(3);
List2.add(1); List2.add(2); List2.add(4);
boolean isEqual = List1.equals(List2);
System.out.println("List equal?" + isEqual);
}
}
```

3.2 project: using multithreading

```
import java.util.*;
// Parking Request
class RegisterParking {
    private final String carNumber;

    public RegisterParking(String carN) {
        this.carNumber = carN;
    }
    public String getCarNumber() {
        return carNumber;
    }

// Shared parking queue
class ParkingPool {
    private final Queue<RegisterParking> queue = new
                                    LinkedList<>();
```

```
public synchronized void addRequest(RegisterParking
                                    request){
    queue.add(request);
    sout("car"+request.getNumber()+" requested");
    notifyAll();
  }
public synchronized RegisterParking getRequest()
throws InterruptedException {
    while(queue.isEmpty()){
        wait();
    }
    return queue.poll();
}
// Parking Agent thread

class parkingAgent extends ~~Thr~~ Thread {
    private final ParkingPool pool;
    private final int agentID;

  public ~~A~~ parkingAgent(ParkingPool pool, int id)
  {
      this.pool = pool;
      this.agentID = id;
  }
```

@ Override

```
public void run() {
    while (true) {
        try {
            RegisterParking Request = pool.getRequest();
            System.out.println("Agent " + agentID + " parked car");
        } catch (InterruptedException e) {
            break;
        }
    }
}
```

// Main simulation

```
public class Main {
    public static void main(String[] args) throws InterruptedException {
        ParkingPool pool = new ParkingPool();
        ParkingAgent agent1 = new
```

4.1

JDBC is an API that allows Java Application to interact with relational DB.

It uses -

(I) Driver Manager

(II) Connection establishment

(III) prepared statement / statement

(IV) Resultset

Codes:

```
import java.sql.*;

public class JdbcExample {

    public static void main(String[] args) {

        String url = "jdbc:mysql://localhost:3306/testdb";

        String user = "root";
        String pass = "3135";

        Connection con = null;

        PreparedStatement stmt = null;
        ResultSet rs = null;
```

```
try{
Class.forName("com.mysql.cj.jdbc.Driver");
con = DriverManager.getConnection(url, user, pass);
stmt = con.prepareStatement("select id, name from Table1");
rs = stmt.executeQuery();
while(rs.next()){
System.out.println(rs.getInt("id")+rs.getString("name"));
}
}catch(Exception e){
e.printStackTrace();
}finally{
```

4.2 How does java handle XML data using DOM and SAX parsers? compare both approaches with respect to memory usage, processing speed and use cases. provide a scenario where SAX would be preferred over DOM.

SAX = Simple API for XML
DOM = Document Object Model

Sol: How they work?

- DOM: Builds a full in memory tree of the entire XML documents allowing random access and modification of nodes

- SAX: Event-driven, reads the XML sequentially and triggers callback without storing the whole document

Comparison:

Aspect	DOM	SAX
Parsing model	Tree in memory	Event-driven streaming
memory	High	Low
Speed on large files	slower	faster
Access pattern	Random access, easy navigation	forward only.

Modification	Supports Read/write and node edits	Read only during parse
Best for	small/medium docs	large/streamed doc.

5.1 How does Servlets and JSP work together in a web application following the MVC architecture? Provide a brief use case showing the servlet as a controller, JSP as a view and a Java class as the model.?

Sol: Servlets + JSP in MVC: How they work together

(i) In classic Java web MVC, the servlet is the controller, JSP is the view, and plain Java classes hold the Model/business/data logic.

(ii) the controller receives http request, invokes model code to get or

update data, store result in request scope and forwards to a JSP for rendering.

(III) The JSP reads attributes set by the controller and renders HTML; it

(IV) The model is independent Java code that encapsulates data and business rules.

Request flow (simple)

(I) client sends HTTP request to a URL mapped to a servlet (controller)

(II) Servlet handle input, calls model/ service classes to perform the work and obtain data

(III) Servlet stores request result in request attributes and forward to a JSP.

(v) JSP reads attributes and generates the HTML response sent back to the client.

Brief Use case : (controller = Servlet, View = JSP, Model = Java class)

- Model :

(i) Product : id, name, price

(ii) ProductService

- controller : (servlet)

(i) ProductServlet Mapped

(ii) Reads request parameter id, calls productservice.getById(id), puts the product into request then forword to JSP.

View (JSP):

- product.jsp reads request attribute product and renders its fields in HTML.
- JSP focuses on presentation only.

5.2: Explain the life cycle of a Java Servlet. What are the roles of the init(), service() and destroy methods? Discuss how servlets handle concurrent requests and how thread safety issues may arise.

Sol: Java Servlet Life Cycle and Concurrency

Life Cycle overview:

- The servlet container loads the class, creates one instance

calls init(); then handles many request via service() and then finally calls destroy() before unloading.

- For HttpServlet, service() dispatches to doGet/doPost/etc, based on HTTP.

Roles of Lifecycle Methods:

- init(): called once after instantiation to perform one-time set
- service(): called for every request
- destroy(): called once when the servlet is taken out of service to release resources

Concurrency model:

- Containers are multi threaded
- for each incoming request, the

controller creates req/res objects and invokes servlet service().

Thread safety and

- Instance fields are shared across all request threads
- Safe practices: Avoid mutable shared state, prefer local variables, use thread safe structure (concurrent HashMap)

5.3 Describe how the Mvc pattern separates concerns in a Java web application.

Explain the structure advantages of its structures in terms of maintainability and scalability. using a student registration system as an Example:

Sol: -MVC in a Java Web app: clear separation of concerns

(i) Model (ii) View (iii) controller

Advantages of maintainability and scalability:

(i) Clear roles reduce coupling

(ii) Easy to test

(iii) Reusable and consistent

(iv) Team productivity:

(v) Scalability and performance

(vi) Easier Evolution

Example with student registration

- Model
  - (i) student{id, name, email}
  - (ii) RegistrationService
  - (iii) student Repository (DAO)
  - (iv) controller (servlet) (vi) View (JSP)