Ans-1 : A protected member of a parent class is accessible by a child within the same package

Example:

```
package pack1;
class parent {
  protected int x=10;
}
class child extends Parents {
    void show(){
        System.out.println (x); // Accesible
    }
}
```

If the parent class is in the different package, it can be still access the protected member, but only through inheritence, not by object reference. (But in same package can access protected member via parent object)

Example:

```
package pack1;
public class Parent {
    protected int x=10;
}
```

```
package pack2;
import pack1.Parent;
class Child extends Parent {
    void show(){
        System.out.println (x); // accesible
} }
```

```
package pack2;
import pack1.Parent;
class Other {
    void (){
        parent p= new Parent();
        System.out.println(p.x); // compile-time error
    }
}
```

Ans 2: Multiple inheritence support:

Abstract class: Java doesn't support multiple inheritence with classes, so a class can extends only one abstract class.

Interface: Java support multiple inheritence with interfaces

A class can implement multiple interfaces

Use abstract class:

i) When you need to provide some common code

ii) When methods may have default behavior

iii) When all child classes share 'is-a' relationship

Use interface:

i) When you want to achieve multiple inheritance
ii) When you need to define only method signatures
iii) When unrelated classes need to share common capability

Example:

```
abstract class A{
    abstract void showA();
}
abstract class B{
    abstract void showB();
}
class C extends A,B{  // compilation Error
}

interface Flyable{
    void fly();
}
interface Swimmable{
    void swim();
}
class Duck implements Flyable, Swimmable{
    void fly(){System.out.println("Duck flying");
    void swim(){ "  " ("Duck swimming")
}
```

Ans-3 Encapsulation is a principle of OOP where data are kept private and can only be accessed or modified through public methods (getters / setters). This ensure

i) Data security - Prevents direct access to sensitive data

ii) Data integrity - validate inputs before changing state

iii) Controlled access - we can enforce rules

Example:

```
public class BankAccount {
    private String accountNumber;
    private double balance;
    public void setAccountNumber (String accNo) {
        if (accNo != null && !accNo.isEmpty()) {
            accountNumber = accNo;
        } else {
            System.out.println ("Invalid account Number");
        }
    }
    public void setInitialBalance (double bal) {
        if (bal >= 0) {
            balance = bal;
        } else {
            System.out.println ("Initial balance cannot be
                                negative");
        }
    }
```

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

Ans-6 Java provides two common ways to handle XML:

i) DOM (Document Object Model)

ii) SAX (Simple API for XML)

DOM Parser:

i) Loads entire XML file into memory as a tree

ii) Allows random access, easy navigation and modification

```
DocumentBuilderFactory factory = DocumentBuilderFactory.
                                    newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
Document doc = builder.parse("data.xml");
```

SAX Parser:

i) Reads XML sequentially

ii) Doesn't store whole document in memory

iii) Faster and memory-efficient for large XML files

```
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser parser = factory.newSAXParser();
parser.parse("data.xml", new DefaultHandler());
```

Feature	DOM	SAX
Memory Usage	High (loads whole file)	Low (reads line by line)
Processing Speed	Slower for large files	Faster for large files
Access type	Random read/write access	Sequential read-only
Use Cases	Small/medium XML with edits	Large XML, read-only parsing

If you're parsing a 500MB XML log file to extract specific tags (like <error>), SAX is preffered. It processes line by line without memory overload

* Use DOM for erase and editing

Ans 7 : In react, the virtual DOM (VDOM) is a lightweight copy of the real DOM kept in memory. It updates the actual DOM only when necessary, improving performance.

It improves performance by:

i) Updating the real DOM is slow

ii) React creates a new virtual DOM when state/props change

iii) It compares the new virtual DOM with the old one (diffing algo)

iv) Only the changed parts are updated in the real DOM - not the whole page

Feature	Virtual DOM	Traditional DOM
Updates	Efficient (only diffs)	Full re-render on change
Speed	Faster	Slower
Control	controlled by React	Manual

Diffing Algorithm Example:

```
function Greet(){
    return <h1>Hello</h1>;
}
```

If the state changes and React needs to render:

```
function Greet() {
    return <h1>Hello, World!</h1>
}
```

React:

i) Creates a new virtual DOM

ii) Compares with old virtual DOM

iii) Finds the difference ("Hello" → "Hello, World!")

iv) Updates only that in the Real DOM

Ans-8 : Event delegation is a technique where a single event listener is added to a parent element to hadle events on its child elements, even if they are added later. It optimize performance by -

i) Reducing the number of event listener

ii) Saves memory

iii) Works well with dynamically added elements.

Example:

```
<ul id = "list">
    <li> Item 1 </li>
    <li> Item 2 </li>
</ul>
<script>
document.getElementById("list").addEventListener
                ("click", function(e)) {
    if(e.target.tagName === "LI"){
        alert("Clicked: " + e.target.textContent);
    }
});
```

// Dynamically added item

```
const newItem = document.createElement("li");
newItem.textContent = "Item 3";
document.getElementById("list").appendChild(newItem);
</script>
```

Listener is added only on <ul>, not each

Works for new <li> like "Item 3" too

DOM traversal via e.target handles delegation

Ans-9 In Java, Regular Expression (regex) are used to validate, search or extract patterns from string - such as validating emails, passwords, phone numbers etc.

Java provides two core classes for regex:

- Java.util.regex.pattern : Compiles the regex pattern
- java.util.regex.Matcher: Applies the pattern to input string

Email validation Regex:

```
String regex = "^[A-Za-z0-9+_.-]+@[A-Za-z0-9.-]+$";
```

This pattern checks:

username: letter, digits, +, -, ., -

@ symbol

Domain name - letters, digits, dots, hyphen

```
import java.util.regex.*;
public class EmailValidation {
    public static void main(String[] args){
        String email = "test.user@gmail.com";
        String regex = "^[A-Za-z0-9+_.-]+@[A-Za-z0-9.-]+$";
        Pattern pattern = Pattern.compile(regex);
        Matcher matcher = pattern.matcher(email);
        if(matcher.matches()){
            System.out.println("Valid email");
        }else{
            System.out.println("Invalid email");
        }
    }
}
```

Pattern.compile() → compiles the regex

matcher() → applies the pattern to the input string

matches() → checks if the input fully matches the pattern

Ans-10 : Custom annotations are user-defined metadata that can be added to code elements (classes, methods etc) and processed at runtime using reflection

Steps to use Custom Annotation with Reflection:

i) Define the annotation using @interface

ii) Annotate elements with it

iii) Use reflection to read and process the annotation at runtime

Let's build a custom annotation @RunImmediately that marks method to be invoked automatically at runtime

Define the annotation:

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME) // Keeps annotation info during runtime
@Target(ElementType.METHOD) // Can be applied to methods only
public @Interface RunImmediately {
    int times() default 1;
}
```

times(): Optional attribute to control how many times to run the method

Use Annotation in a class

```
public class MyService {
    @RunImmediately(times=3)
    public void sayHello() {
        System.out.println("Hello");
    }
}
```

Use Reflection to process Annotation at RunTime

```
import java.lang.reflect.Method;
public class AnnotationProcessor {
    public static void main (String[] args) throws Exc {
        MyService service = new MyService();
        Method[] methods = MyService.class.getDeclared
                                            Methods();
        for (Method method : methods) {
            if (method.isAnnotationPresent (Run... .class)) {
                Run... annotation = method.getAnno(
                                        Runtime... class)
                for(int i=0; i<annotation.times(); i++) {
                    method.invoke(service);
                }
            }
        }
    }
```

Ans 12 JDBC is an API that enables Java applications to connect and interact with relational databases (like MySQL, PostgreSQL, Oracle etc). JDBC manages communicati

i) Driver Manager loads the appropriate database driver

ii) Establishes a connection to the database

iii) Sends SQL queries to the DB engine

iv) Retrives ResultSet for SELECT queries

v) Closes resources (ResultSET, Statement, Connection) after use

```
import java.sql.*;
public class JDBCExample {
    public static void main(String args[]){
        Connection conn = null;
        Statement stmt = null;
        ResultSet rs = null;
        try{
          Class.forName("com.mysql.cj.jdbc.Driver");
          // load driver
          conn = DriverManager.getConnection("jdbc:my
                sql://localhost:3306/testdb", "root", "pass");
          //connection to DB
          stmt = conn.createStatement(); //create statement
          rs = stmt.executeQuery("SELECT * FROM users");
          // Execute SELECT query
          while(rs.next()){
              System.out.println(rs.getString("username");
          }//process result
        } catch(Exception e){
            e.printStackTrace(); // Handle exception
        } finally{
            try{
              if(rs != null) rs.close();
              if(stmt != null) st.close();
              if(conn != null) conn.close();
```

```
        }catch (SQLException ex){
            ex.printStackTrace();
        }
    }
}
}
```

Ans 13 In a web application following the MVC pattern:

- Servlet acts as the controller: It handles client requests, processes input, interacts with the model, and decides which view to display.
- Java class acts as the Model: It contains the business logic and data (like fetching data from the database or processing it)
- JSP acts as the view: It presents data to the user in the form of HTML pages, displaying the results sent by the controller

They work together:

1) The client sends a request to the Servlet (controller)
2) The servlet calls the model class to process the data or fetch required information
3) The model returns data to the Servlet

4) The servlet sets this data as request attributes and forwards the request to a JSP page (view)

5) The JSP uses the data to display the output to the user

Model (Java class):

```
public class User {
    private String name;
    public User(String name) { this.name = name; }
    public String getName() { return name; }
}
```

Controller (Servlet):

```
protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
            IOException {
    User user = new User("Alice");
    request.setAttribute("user", user);
    RequestDispatcher rd = request.getRequestDispatcher
    rd.forward(request, response);
}
```

("welcome.jsp");

View (JSP - welcome.jsp):

```
<html>
<body>
    <h1>Welcome, ${user.name}!</h1>
<body>
<html>
```