JDBC Batch insetct :—

code:

```
import java.sql.*;
public class BatchInsert {
    public static void main(String [] args) throws
Exception {
try (Connection c=DriverManager.getConnection
    ("url", "username", "password");
preparestatement s=c.prepareStatement("Insetct
into courses values (?,?)");
con.setAutoCommit(false);
int [][] data ={{101,"math"},{102,"physics"},
                {103,"chemistry"}};
for (int[] d: data) {
    s.setInt(1,d[0]); s.setString(2,d[1]+"");
    s.addBatch();}
ps.executeBatch(); conn.commit(); con.close();
system.out.println("Batch Insert Done");
}}
```

③

```
import java.sql.*;
public class Fetchstudent {
    public static void main(String[] args) throws
    Exception {
        try Connection con = DriverManager.getConnection
        ("url", "username", "password");
    preparedStatement ps = con.preparedStatement
        ("select * FROM student WHERE id = ?");
    ps.setInt(1,101);
    ResultSet() rs = PS.executeQuery();
    if(rs.next()) System.out.println(rs.getInt(1)+
        " " + rs.getString(2));
    con.close();
    }
}
```

⇒ uses preparedstudent for security
⇒ fetches and prints student details.

④

```
import et java.sql.*;
public class JDBCTransaction {
  public static void main (String [] args)
  throws Exception {
  Connection conn = DriverManager.getConnection
    ("url", "username", "pass");
  conn.setAutoCommit (false);
try { preparedStatement check = conn.prepared
  Statement ("Select stold FROM courses WHERE
  check.setInt (1, 201);
Resultset rs = check.executeQuery ();
if (rs.next() && rs.getInt (1) > 0) }
conn.preparedStatement ("Insert INTO Registration
Values (101, 201)").executeUpdate ();
conn.commit();
system.out.println ("Reg successful");
  }
  else conn.rollback();
  }
```

```
catch(SQLException e) {
    conn.rollback();
    }
    conn.close();
}
}
```

(5)

Type	use case	Advantages	Disadvantages
Statement	simple queries	Easy to use, for static SQL	slow, prone to SQL injection
PreparedStatement	Queries with parameters	precompiled, prevents SQL injection	slightly more complex
CallableStatement	stored procedures, function	optimized for procedures, faster	DB-dependent

use statement for simple queries, Prepared-statement for dynamic queries, and callable-statement for stored procedures.

(e)

```
import java.sql.*;
public class StoredProcedureExample {
public static void main (String[] args)
throws Exception {
connection conn = DriverManger.getConnection
("uri", "username", "password");
callableStatement st = conn.prepareStatement
("call GetStudentsByCourse(?)}");
st.setInt(1,101);
ResultSet rs = st.executeQuery();
while (rs.next()) system.out.println(rs.getString(1));
conn.close();
}
}
```

Explanation: use callableStatement to executes stored procedures. Set parameters and retrieve results via executeQuery.

(7)

Rowset is an interface JDBC that provides a more flexible, disconnected alternative to ResultSet.
types include:
1. JdbcRowSet : connected, scrollable
2. CachedRowSet : Disconnected, scrollable, can be updated offline.
3. FilteredRowSet : Adds filtering capabilities.
4. JoinRowSet : combines multiple Rowset.
Advantages: Portable, supports offline manipulation, and automatic connection management.

(8)

Rowset is flexible, scrollable and updateable alternative to ResultSet. Types -
1. JdbcRowSet - Connected, Scrollable
2. CachedRowSet - Disconnected, editable offline.
3. FilteredRowSet - filters data dynamically
4. JoinRowSet - combines multiple Rowsets.
Advantage: works offline, auto-connects, portable.

(9)

Lazy Loading in Hibernate delays the loading of associated data until it's explicitly accessed, improving performance by fetching only necessary data. However, it may cause lazyInitializationException if accessed outside a session.
to disable lazy loding, use fetch = fetchType.Eage

```
@Entity
class Employee {
    @oneToMany(mappedBy = "employee", fetch
        = fetchType.EAGER)
    private List<TASK> tasks;
}
```

Eager Loading fetches related entities immediately, increasing memory usage but reducing additional queries.

(10)

Criteria API is used for dynamic, type-safe

queries, ideal when query structure varies. HQL is SQL-like, better for static, complex queries.
use Criteria API: When filtering dynamically.

code:

```
CriteriaBuilder cb = session.getCriteriaBuilder();
CriteriaQuery<Employee> cq = cb.createQuery(Employee
    class);
Root<Employee> root = cq.from(Employee.class);
cq.select(root).where(cb.equal(root.get("dept"),
IT));
List<Employee> employees = session.createQuery(cq)
        getResultList();
```

(11)

Hibernate caching boots performance by reducing database queries. First level cache is session-scoped, storing entities ~~who~~ within a transaction. Second-Level cache is session-factory scoped, shared across sessions.

code

```
session.get(Employee.class, 1);
session.get(Employee.class, 1);
```

caching minimizing DB hits, improving speed.

(12)

Hibernate Programme to Delete a student Record.

code:

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
student student = session.get(student.class, student
if (student != null) session.delete(student);
tx.commit();
session.close();
```

session.delete(): marks the entity for deletion.
and commit() removes it from the database.

(13)

Struts Action Class for login Authentication —

code:

```
public class LoginAction extends Action {
public ActionForward execute (ActionMapping mapping,
ActionForm form, HttpServletRequest req, HttpResponse res)
{ LoginForm loginForm = (LoginForm) form;
```

caching minimizing DB hits, improving speed.

(12)

Hibernate Programme to Delete a student Record.

code:

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
student student = session.get(student.class, student
if (student != null) session.delete(student);
tx.commit();
session.close();
```

session.delete(): marks the entity for deletion.
and commit() removes it from the database.

(13)

Struts Action Class for login Authentication —

code:

```
public class LoginAction extends Action {
public ActionForward execute (ActionMapping mapping,
ActionForm form, HttpServletRequest req, HttpResponse res)
{ LoginForm loginForm = (LoginForm) form;
```

```
if (isValidUser (loginForm)) return mapping.findForward
    ("success");
return mapping.findForward("failure");
}
private boolean isValidUser(LoginForm form){
return "admin".equals(form.getUsername()) &&
"pass".equals(form.getPassword());
}
}
```

it forwards requests to success.jsp or failure.jsp based on validation.

(14)

struts ~~-xml~~ - config.xml configures struut applications, defining navigation and reqesest handling.

Key concepts:

⇒ Action mapping
⇒ Form beans
⇒ Global forwards: Reusable navigation links.

Example —

<action path="/Login" type="LoginAction" name="LoginForm" Forward name="success" path="/Home.jsp"/>

it directs login requests to LoginAction and forwards response accordingly.

(15)

Interceptors in struts preprocess and postprocess requests dynamically. They execute before and after Action execution —

Example: Logging interceptors —

source code:

```
public class interceptor extends AbstractIn{
public string intercept(ActionInvocation invocation)
throws Exception{
system.out.println("Req received: "+invocation.
                    getAction());
return invocation.invoke();
}
}
```

it logs request details before executing the action.

(16)

Struts2 improves upon Struts1 with POJO based actions, interceptors and better integration with Spring and AJAX. it removes the need for ActionForm, supports dependency injection and simplifies validation. Unlike Strute1, which uses servlets, struts2 follow MVC with filters, making it more flexible and scalable.

(17)

code:

```
public class RegisterServlet extends HttpServlet
{ protected void doPost(HttpServletRequest req,
HttpServletResponse res) throws ServletException
{ try(connection c = DriverManager.get
  connection(DB_URL, USER, PASS);
prepared statement ps = c.preparedStatement("Inser
```

```
INTO users (name) values (?)");
    ps.setString (1, name);
    ps.executeUpdate();
  }
 }
}
```

(8)

Comparision of forward() and sendRedirect() in servlets:

features	forward()	sendRedirect()
machanism	internal server-side forward	client-side redirect
URL change	no	yes
performance	faster	slower
use case	within same app, sharing data	Different app, new request.

usage:-

- use forward() for internal navigation to improve performance.
- use sendRedirect() when redirecting to external resources or ensuring a fresh request.

(19)

Difference between ServletContext and Httpsession

feature	ServletContext	Httpsession
scope	Application-wide	per-user session
Lifetime	untill server stops	untill session expires
Data sharing	shared across all users	specific to one user

usage:

- use servletContext for app-wide configs
- use Httpsession for users-specific data.

(20)

filters in Java servlets:

filters intercept requests and responses.. allowing preprocessing. They run before the target servlet processes a request.

Example: Login filter

source code:

```
import java.io.IOException;
import javax.servlet.*;
public class LogFilter implements Filter {
public void doFilter(ServletRequest req, servlet
Response res, FilterChain chain) throws IOExcep
, servletException {
System.out.println("Request Params: "+req.
getParameterMap());
chain.doFilter(req, res);
}
}
```

(21)

feature	Scriptlet (<% ... %>)	Expression (<%= ... %>)	Declaration (<%! ... %>)
purpose	Embed Java code	output values in JSP	Declare methods
scope	Executes inside	inserts value in response	Defined as class level
Usage	Logic handling	Displaying dynamic data	Defining reusable methods

use: Avoid scriplets; use JSTL/EL for better MVC separation.

(22)

custom tags allow reusable components in JSP, improving code separation. Defined any Tag Library Description and Java tag Handler.

Java tag Handler —

source code:

```
import javax.servelet.jsp.target.*;
import javax.servelet.jsp.*;
import java.io.*;
public class welcomeTag extends simple
TagSupport{
public void doTag() throws JspException
,IOException{
getJSPContext().getOut().print("welcom
to JSP custom tags!");
}
}
```

(22)

feature	FileReader()	BufferReader()	Scanner
purpose	Reads characters	Reads line efficiently	parse inputs
performance	Slow(reads char by char	Fast (buffering inputs)	Slower than BufferReader

feature	FileReader	BufferReader	Scanner
use case	simple file Reading	Large files, better performance	Tokenizing, parsing different data types

Best practice: use BufferReaders for efficiency scanner for parsing and FileReader for basic reading.

(24)

source code:

```
import java.io.*;
import java.util.*;
public class productManager {
    static final String FILENAME = "products.csv";
public static void main (String[] args) throws
    IOException {
    Scanner sc = new Scanner (System.in);
    System.out.println ("1. Add, 2. search");
    int choice = sc.nextInt();
```

```
sc.nextLine();
if(choice==1){
 system.out.println("ID, name, Price: ");
try (BufferedWriter bw=new BufferWriter(new
FileWriter (FILE_NAME, true))){
   bw.write(sc.nextLine()+"\n");
    }
else if (choice==2){
 System.out.println("EnterID: ");
  }
 }
}
```

(25)

Comparision of prime number checking Approaches in JAVA

approach	time complexity	memory usage	Efficiency & use case
Loop-based	$O(\sqrt{n})$	Low	most efficient fo Large numbers
Recursive	$O(\sqrt{n})$	Higher	Less efficient, used for Learni recursion

Approach	time complexity	memory usage	Efficiency & use case
Build-in-function	varies	Low	useful for quick checks but limited

Best practice : use-loop-based for optimal performance.

(26)

```
import java.sql.*;
public class EmployeeDB {
static Connection c() throws SQLException {
return DriverManager.getConnection("url", "username", pass
}
static void exac(String q, object ... p) throws
SQLException {
try (connection c = con(); preparedStatement ps=c.
preparedStatement(q)) {
for (int i=0; i<p.length ; i++) ps.setObject(i+1,p[i]);
ps.executeUpdate();
}
public static void main (String[] a) throws
Exception {
exac("insert into employee values (?,?,?,?)",
```

```
try (connection c = con(); ResultSet rs = c.createStatem
    ().executeQuery("select * FROM employees")
} while (rs.next()). System.out.println(rs.getIn
)
exec("Update employees SET salary=?
where EmpId=?", 6000, 1);
exec("DELETE FROM employees set salary=?
where empId=?", 1);
    }
}
```

(27)

Key principles of Immutable class:

1. Declear class final to prevent subclassin
2. Make all fields private final
3. provide no setters, only getters
4. use deep copies for mutable objects

source code:

```
final class person{
  private final string name;
  person (string name) { this.name= name;}
  public string getName () { return name;}
}
```

(23)

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

analysis: BigInteger handles large fibonacci numbers efficiently, avoiding integer overflo but is slower than primitive types due to higher computation overhead.

(29)

Scenario	Best choice	Justification
Fast random access	ArrayList	O(1) index-based access
Concurrent task Queue	Concurrent LinkQueue	Thread-safe, non-blocking
Sequential path navigation	Linked list	Fast insertions/deletion (O(1)) at nodes)
Chat message storage	Linked list	Efficient for frequ insertions/removals

conclusion: ArrayList excels in access speed, LinkedList in modifications and concurrent-LinkedQueue in thread safety.

(30)

Here's the Java implementation based on the UML Diagram.

```
class TV {
private int channel=1, volume=10;
private boolean power = false;
void powerOn () { power = true; }
void powerOff () { power = false; }
void setChannel(int ch) { if (power) channel=ch; }
void setVolume (int vol) { if (power) volume=vol; }
void showStatus () { system.out.println ("power:
    " + power + ", channel: " + channel + ", volume
    : " + volume); }
}
public class TVSet {
public static void main (String[] args) {
    TV tv = new TV();
    tv.powerOn();
    tv.setChannel (5);
    tv.setVolume(15);
    tv.showStatus();
}
}
```

Criteria	Best choice	Justification
Synchronization & thread-safety	vector	synchronized, thread-safe
speed & performance	ArrayList	Fast random access, better cache locality
capacity	ArrayList	Dynamically grows, Less memory overhead
Enumeration & iterator	vector	supports Enumeration & iteration
Legacy	vector	part of old java versions

conclusion: use vector for thread-safety, ArrayList for speed and Linked list for frequent insertions/deletions.

(32)

source code:

```
interface Flyable { void fly(); }
class Bird implements Flyable { public void fly() {
system.out.println("Bird flies"); } }
class Airplane implements Flyable { public void fly()
{ system.out.println("plans flies"); } }
```

Impact:

⇒ Reusability: Ensures consistant behaviour across multiple classes.

⇒ Scalability: Easily extendable without modifying existing code.

⇒ multiple inheritance: A class can implement multiple interfaces, avoiding java's single inheritance limitation.

(33)

Filter & cookies in Servlet:

⇒ filter: used for request/response filtering

⇒ cookies: store small client-side data for

```
public double getArea(double radius){
    return Math.PI * radius * radius;
  }
}
```

2. client request

```
import circleApp.circle;
import org.omg.CORBA.ORB;
public class client {
public static void main (String[] args) {
ORB orb=ORB.init(args, null);
circle obj = circleHelper.narrow(orb.string_to_
  object("IOR_STRING"));
System.out.println("AREA: "+obj.getArea(5.0));
  }
}
```


Java EE Architecture for scalability & Maintainability:

⇒ scalability: supports load balancing, clustering and distributing computing

⇒ maintainability: user modular components common design patterns;

⇒ MVC : separates concerns for better maintenan.
⇒ singleton : manages shared resources efficien.
⇒ DAO : Encapsulates database logic
⇒ Factory : creates objects dynamically.
enhancing flexibility.

(39)

Java's ME's Modular Architecture :

- CLDC (connected Limited Device configuration)
  for low-memory devices.
- CDC (connected Device configuration)
  For ~~many~~ more capable devices.

Trade-offs vs. Java SE :

✓ Lightweight, optimized for constraints
✗ Limited libraries, weaker performance
reduced API support.

(40)