JAVA UNIT 4 NOTES

Advanced Java - Unit IV: Remote Method Invocation (RMI) & Persistence with Hibernate (HB)

This unit introduces two important concepts in Advanced Java: Remote Method Invocation (RMI) for building distributed applications and Hibernate (HB) as a framework for managing data persistence, particularly Object-Relational Mapping (ORM).

1. The Roles of Client and Server

- **What is it?**
  - The Client-Server model is a fundamental architectural pattern in distributed computing. It describes how programs running on different computers interact over a network. (Notes Image 2).
- **Why is it important?**
  - Most modern applications, from simple websites to complex enterprise systems, are built using this model. Understanding these roles is crucial for designing and implementing distributed systems like those using RMI or accessing remote databases via ORM frameworks.
- **Key Concepts:**
  - **Client:**
    - A program or computer that **initiates requests** for services or data. (Notes Image 2).
    - Typically interacts directly with the user.
    - Connects to a server to get what it needs.
    - Usually interacts with only one server for a specific task, but can interact with multiple servers overall. (Notes Image 2). (Your web browser is a prime example – it's a client requesting pages from a web server).
  - **Server:**
    - A program or computer that **listens for and responds to requests** from clients, providing services or data. (Notes Image 2).
    - Runs continuously, waiting for client connections.
    - Can serve **multiple clients concurrently** (at the same time). (Notes Image 2).
    - Often manages shared resources like databases or application logic.
  - **Interaction:** Client and server communicate over a computer network (like the internet) using common communication **protocols** (like HTTP, or custom protocols used by RMI). Sometimes they might reside on the same physical machine (e.g., for testing or specific architectures). (Notes Images 2, 3).

    - **Characteristics:**
        - Works on a request-response principle. (Notes Image 3).
        - Requires a common communication protocol. (Notes Image 3).
        - A server can handle a limited number of requests simultaneously (can use priority or queuing systems). (Notes Image 3).
        - Vulnerable to Denial of Service (DoS) attacks if overwhelmed with false requests. (Notes Image 3).
    - **Advantages:** Data centralization (easier management, security, authorization, authentication), efficient data access (server can be optimized), easy to update/replace clients or servers independently. (Notes Image 5).
    - **Disadvantages:** Server can get overloaded, single point of failure (if the server goes down, clients can't get services), potentially high cost of setup/maintenance. (Notes Image 6).
    - **Vs. Peer-to-Peer (P2P):** In P2P, nodes are generally equal and share resources directly, without a central server. Client-Server is often seen as a subcategory of distributed computing, contrasting with P2P. (Notes Image 4).

(No code example needed for this architectural concept)

- **References:** Notes Images 2-6 cover the roles, characteristics, advantages, disadvantages, and comparison to P2P computing.

2. Remote Method Invocation (RMI) - What is it?

- **What is it?**
    - Remote Method Invocation (RMI) is Java's native API (Application Programming Interface) for creating distributed applications in Java. (Notes Image 7).
    - It enables a Java object running in one Java Virtual Machine (JVM) to invoke methods on a Java object running in another JVM, potentially on a different physical machine. (Notes Image 7).
    - **Goal:** To make calls to remote objects appear and behave as much as possible like calls to local objects, hiding the complexities of network communication.
- **Why is it important?**
    - It allows you to distribute the logic of your application across multiple machines. For example, you can have a powerful server performing complex calculations or accessing data, and thinner client applications running on user machines that simply call methods on the server objects as needed.
    - Facilitates the creation of multi-tiered enterprise applications where different tiers (e.g., business logic tier, data access tier) might run on separate servers.
- **Key Concepts:**
    - **Distributed Computing Model:** RMI is Java's implementation for object-to-object communication in this model. (Notes Image 7).

- **Remote Object:** An object whose methods can be invoked from a different JVM. (Notes Image 8).
- **Remote Interface:** A Java `interface` that defines the methods of the Remote Object that can be called remotely. (Notes Image 15 shows an example `MySearch` interface).
  - Must extend `java.rmi.Remote` (a marker interface).
  - All methods declared in the interface must declare that they `throws java.rmi.RemoteException`. This is a checked exception indicating that network-related errors can occur during a remote call. (Notes Image 15, 23).
- **Remote Object Implementation:** The actual Java class that provides the implementation for the methods declared in the Remote Interface. (Notes Image 16 shows an example `SearchQuery` class).
  - It must implement the Remote Interface.
  - To make an instance of this class available remotely, it typically extends `java.rmi.server.UnicastRemoteObject` or is explicitly exported using `UnicastRemoteObject.exportObject()`. This handles listening on an anonymous port and creating the necessary network connections. (Notes Image 16).
  - The constructor of the implementation class must declare `throws RemoteException` because the superclass constructor (`UnicastRemoteObject`) does. (Notes Image 16).
- **Stub (Client-side Proxy):** (Notes Images 7, 8, 9, 11 diagram).
  - A proxy object that resides in the **Client's JVM**.
  - It implements the *same* Remote Interface as the actual Remote Object.
  - When the client code calls a method on the Stub object (because it looks like the real object based on the interface), the Stub doesn't execute the business logic itself. Instead, it:
    - Initiates a connection to the server JVM. (Notes Image 9).
    - Marshals (packages) the method name and parameters into a byte stream. (Notes Image 9).
    - Sends the request byte stream over the network to the server. (Notes Image 9).
    - Waits for the response from the server. (Notes Image 9).
    - Unmarshals (unpackages) the return value or exception received from the server. (Notes Image 9).
    - Returns the result or throws the exception back to the client code. (Notes Image 9).
  - The client code interacts only with the Stub object, which acts as a gateway to the remote object. (Notes Images 7, 9).
- **Skeleton (Server-side Helper):** (Notes Images 7, 8, 10, 11 diagram).
  - A server-side object that receives the request from the Stub. (Notes Images 8, 10).

        - Unmarshals (unpackages) the method name and parameters from the incoming byte stream. (Notes Image 10).
        - Invokes the actual method on the real Remote Object Implementation instance that resides in the server JVM. (Notes Image 10).
        - Marshals (packages) the result (return value or exception) from the method call. (Notes Image 10).
        - Sends the response byte stream back to the client's Stub. (Notes Image 10).
        - **Note:** In Java 2 SDK (and later), the need for explicit Skeleton classes generated by the `rmic` tool was largely eliminated. The RMI framework handles this on the server side automatically. (Notes Image 11 explicitly mentions this).
    - **RMI Registry (`java.rmi.registry.Registry`):** (Notes Image 17, 18, 19, 20 show usage via `Naming` class).
        - A simple naming service running on a specific port (default 1099). Think of it as a "phonebook" or "lookup service".
        - The **Server** application registers (binds) its Remote Object Implementation instance with the Registry under a specific name (a URL-like string, e.g., `rmi://localhost:1099/MyServiceName`). (Notes Images 18, 19).
        - The **Client** application looks up (accesses) the Remote Object in the Registry using that same name to obtain the Stub object that points to the remote object. (Notes Images 20).
- **References:** Notes Images 7-12 explain RMI, its components (stub, skeleton), and requirements for distributed applications.

3. Setup for Remote Method Invocation (The RMI Example Steps)

- **What is it?**
    - RMI requires several distinct steps to define the remote service, implement it, make it available on the server, and allow clients to access it. (Notes Image 14 lists 6 steps).
- **Why is it important?**
    - RMI is not a simple "write code and run" technology for distributed systems. It has a specific setup and deployment process that must be followed to make remote objects accessible and usable over the network.
- **Key Concepts & Steps:** (Notes Image 14 outlines the 6 steps).
    1. **Define the Remote Interface:** Create a Java interface that extends `java.rmi.Remote` and declares the methods callable by clients, all throwing `RemoteException`. (Notes Image 15 provides an example `MySearch` interface).
    2. **Implement the Remote Interface:** Create a Java class that implements the Remote Interface. This class contains the actual logic. Extend `java.rmi.server.UnicastRemoteObject` (or export the object) and provide a constructor that throws `RemoteException`. (Notes Image 16 provides an example `SearchQuery` implementation).

3. **Compile the Implementation Class and Create Stub/Skeleton:**
    - Compile the interface and implementation `.java` files using `javac`.
    - *(Older RMI):* Run the RMI compiler (`rmic`) on the implementation class (`rmic SearchQuery`). This tool generated the Stub (`_Stub.class`) and Skeleton (`_Skel.class`) files. (Notes Image 17 mentions using `rmic` to produce stub/skeleton).
    - *(Modern RMI):* With `UnicastRemoteObject` or explicit export, the RMI framework often generates and handles the stub/skeleton dynamically or automatically. The `rmic` step is often not explicitly needed by the developer.
4. **Start the RMI Registry:**
    - The RMI Registry must be running on the server machine (or the machine where the server application will bind the object).
    - You can start it as a separate process from the command line: `rmiregistry [port]` (default port is 1099). (Notes Image 17).
    - Alternatively, you can start the registry programmatically within your server application using `java.rmi.registry.LocateRegistry.createRegistry(port);`. (Notes Images 18, 19).
5. **Write and Start the Server Application:**
    - Create the main Java server application class.
    - Create an instance of your Remote Object Implementation class.
    - Register (bind) this object instance with the running RMI Registry using `java.rmi.Naming.rebind(String name, Remote obj)`. The `name` is a URL-like string specifying the host, port (optional), and a unique service name (e.g., `rmi://localhost:1099/MyServiceName`). (Notes Images 18, 19).
    - Keep the server application running so the remote object remains available.
6. **Write and Start the Client Application:**
    - Create the main Java client application class.
    - Look up the Remote Object in the RMI Registry using `java.rmi.Naming.lookup(String name)`. Use the same URL-like name the server used. (Notes Image 20).
    - Cast the returned `Object` to your Remote Interface type (this object is actually the Stub). (Notes Image 20).
    - Call methods on the obtained Stub object as if it were a local object. Handle `RemoteException` for all remote method calls. (Notes Image 20).

- **Simple Code Example (Based on Unit 1 Practical Program 21 structure and Unit 4 Slides/Notes 15-20):**

  1. Calculator.java (Remote Interface)

```java
import java.rmi.Remote;
import java.rmi.RemoteException; // Must declare RemoteException
```

```java
public interface Calculator extends Remote { // Must extend Remote
    int add(int a, int b) throws RemoteException; // All methods throw
RemoteException
    // Add other methods like subtract, multiply, divide as needed
}
```

2. CalculatorImpl.java (Remote Object Implementation)

```java
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject; // Helper class for exporting

// Implements the Remote Interface and extends UnicastRemoteObject
public class CalculatorImpl extends UnicastRemoteObject implements
Calculator {

    // Constructor must declare throwing RemoteException (because
UnicastRemoteObject's constructor does)
    public CalculatorImpl() throws RemoteException {
        super(); // Calls the constructor of UnicastRemoteObject
    }

    // Implement the remote method defined in the interface
    @Override
    public int add(int a, int b) throws RemoteException {
        System.out.println("Server received add request for " + a + " and
" + b); // Server side print
        return a + b; // Perform the actual calculation
    }

    // Implement other methods from the interface
    // @Override
    // public int subtract(int a, int b) throws RemoteException { ... }
}
```

5. CalculatorServer.java (Server Application)

```java
import java.rmi.Naming; // Used for binding/lookup
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry; // Used for creating registry

public class CalculatorServer {
    public static void main(String[] args) {
        try {
            // Step 4: Start the RMI Registry programmatically (or run
'rmiregistry' command separately)
            LocateRegistry.createRegistry(1099); // Default RMI registry
port

            // Step 5: Create an instance of the remote object
implementation
            CalculatorImpl service = new CalculatorImpl();
            System.out.println("Calculator service implementation
created.");

            // Step 5: Bind (register) the remote object instance with
the RMI Registry
            // Use a URL format: rmi://hostname:port/serviceName
            // "localhost" means the registry is on the same machine
            Naming.rebind("rmi://localhost:1099/CalculatorService",
service); // Binds the object

            System.out.println("Calculator service bound in Registry.");
            System.out.println("Server is ready.");

        } catch (RemoteException e) {
            System.err.println("RMI Server error: " + e.getMessage());
            e.printStackTrace();
        } catch (Exception e) { // Catch other exceptions like
MalformedURLException from Naming.rebind
             System.err.println("Server error: " + e.getMessage());
             e.printStackTrace();
        }
    }
}
```

6. CalculatorClient.java (Client Application)

```
import java.rmi.Naming; // Used for binding/lookup
import java.rmi.RemoteException;
// No need to import CalculatorImpl, only the Remote Interface is needed
// import CalculatorImpl; // Not needed

public class CalculatorClient {
    public static void main(String[] args) {
        try {
            // Step 6: Look up the remote object (get the Stub) from the
RMI Registry
            // Use the same URL format used by the server for binding
            // Replace "localhost" with the server's IP/hostname if on a
different machine
            Calculator stub = (Calculator)
Naming.lookup("rmi://localhost:1099/CalculatorService"); // Looks up the
object, returns the Stub

            System.out.println("Stub obtained from Registry.");

            // Step 6: Call the remote method on the Stub object
            int result = stub.add(10, 20); // Calling the method on the
Stub

            System.out.println("Result of remote add(10, 20): " +
result); // Prints 30

        } catch (RemoteException e) {
            System.err.println("RMI Client error during remote call: " +
e.getMessage());
            e.printStackTrace();
        } catch (Exception e) { // Catch exceptions from Naming.lookup
(e.g., NotBoundException, MalformedURLException, RemoteException)
            System.err.println("Client lookup error: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

- **Explanation of Code:**
  - We define the `Calculator` interface extending `Remote` with methods throwing `RemoteException`.
  - `CalculatorImpl` implements `Calculator`, extends `UnicastRemoteObject`, provides the actual `add` logic, and has the required constructor.
  - The `CalculatorServer` class's `main` method creates an RMI registry, instantiates `CalculatorImpl`, and binds it to a name ("CalculatorService") in the registry. This makes the object available remotely.
  - The `CalculatorClient` class's `main` method looks up "CalculatorService" in the registry to get the Stub object (which implements the `Calculator` interface). It then calls the `add` method on this Stub. RMI handles sending the request to the server JVM, executing the `add` method on the `CalculatorImpl` instance there, and returning the result back to the client.
  - Both client and server must handle `RemoteException` because of the interface definition.
- **How to Run:**

1. Compile all `.java` files: `javac Calculator.java CalculatorImpl.java CalculatorServer.java CalculatorClient.java`
2. *(If not creating registry in server code)* Start RMI Registry: Open a command prompt and run `rmiregistry`. (Keep this window open).
3. Start Server: Open a *new* command prompt and run `java CalculatorServer`. (Keep this window open. It should print "Server is ready.").
4. Start Client: Open a *new* command prompt and run `java CalculatorClient`. (It should print "Stub obtained..." and "Result of remote add(10, 20): 30". The server window might print "Server received add request...").

- **References:** Notes Images 13-20 detail the 6 steps of the RMI setup process and provide the code structure and snippets for the Remote Interface, Implementation, Server, and Client applications. Practical Program 21 in Unit 1's practical file also provides a working RMI example setup.

4. Parameter Passing in Remote Methods

- **What is it?**
  - When a method is called remotely in RMI, arguments are sent from the client to the server, and a return value (or exception) is sent back from the server to the client. Parameter passing defines how these objects and primitive values are transferred. (Notes Image 21).
- **Why is it important?**
  - Understanding parameter passing is crucial because it affects how changes to objects on one side are (or are not) reflected on the other side, and what types of objects can be transferred.
- **Key Concepts:** (Notes Images 21, 22, 23 explain this).
  - **Serializable Objects:** Any argument or return value in an RMI method call must be **serializable**. This includes Java primitive types (int, float, boolean, etc., which are automatically serializable) and objects that implement the `java.io.Serializable` interface. (Notes Image 21).
  - **Serialization:** The process of converting an object's state into a byte stream so it can be sent over the network or saved to a file.
  - **Deserialization:** The process of reconstructing an object from its byte stream representation.
  - **Pass By Value (for Non-Remote Objects):** (Notes Images 21, 22).
    - Applies to primitive types and objects that implement `java.io.Serializable` but are *not* Remote Objects themselves.
    - How it works: The object is serialized on the sending side, the byte stream is sent over the network, and a *new, distinct copy* of the object is deserialized on the receiving side. (Notes Image 22).
    - Effect: Changes made to the object (mutating its state) on the receiving side **do not affect the original object** on the sending side because the receiving side is working on a copy. Think of sending a photocopy – changing the photocopy doesn't change the original.

- **Pass By Reference (for Remote Objects):** (Notes Image 23).
    - Applies *only* to objects that are themselves Remote Objects (i.e., they implement a `Remote` interface and have been exported by an RMI server).
    - How it works: Instead of sending the object's state, RMI sends a **Stub** object (a reference to the remote object) over the network. The Stub is a serializable object that represents the remote object's endpoint. (Notes Image 23).
    - Effect: The receiving side gets a Stub that points back to the *original* Remote Object instance on the sending side. Any method calls made on the received Stub are executed on the single, original remote object instance. Think of sending a remote control – using the remote control operates the original device.
- **Exception Handling:** Remote methods must declare `throws RemoteException`. Clients calling these methods must handle this exception. This acknowledges that network issues can prevent the call from succeeding or the result from returning. (Notes Images 15, 20, 23). The notes mention designing interfaces with "failure semantics in mind," implying that the client might not know the exact state of the server after a failed remote call. (Notes Image 23).
- **Class Loading:** If the class of an argument or return value is not available locally on the receiving side, the JVM might try to load it dynamically from the sending side's codebase using mechanisms like the RMIClassLoader, depending on security settings and configuration. (Notes Image 21).

- **References:** Notes Images 21-23 detail parameter passing by value (Serializable objects) and by reference (Remote objects), and the implications for exception handling.

5. Introduction of Hibernate (HB) / Object-Relational Mapping (ORM)

- **What is it?**
    - **Persistence:** The ability of an object's state to survive the process that created it (e.g., saving data to a database or file so it's still there after the program finishes).
    - **Database Interaction:** Traditionally, saving/loading Java objects to/from relational databases involved writing significant amounts of manual code using JDBC (Java Database Connectivity) to:
        - Translate object fields into database table columns for saving (`INSERT`, `UPDATE`).
        - Translate database rows/columns back into Java object fields for loading (`SELECT`).
        - Handle relationships between objects (like one-to-many) and map them to foreign keys and join queries.
    - **Object-Relational Impedance Mismatch:** The fundamental differences between how Java objects are structured (inheritance, references) and how relational databases are structured (tables, rows, foreign keys, flat data) create a challenge in mapping between the two.
    - **ORM (Object-Relational Mapping):** A programming technique that maps objects in an object-oriented language (like Java) to data in a relational database.

  - **Hibernate (HB):** A popular, open-source, lightweight Java framework that provides an implementation of ORM. (Notes Image 24).
  - It implements the specifications defined by the **JPA (Java Persistence API)**. JPA is the *standard* Java API for persistence, while Hibernate is a popular *provider* (implementation) of that standard. (Notes Image 24).
- **Why is it important?**
  - **Simplifies Data Persistence:** Hibernate automates the mapping between Java objects and database tables, generating most of the required SQL and JDBC code. This significantly reduces the amount of manual "boilerplate" code developers have to write for data persistence. (Notes Image 24, 25 - "Less Code").
  - **Increased Productivity:** Developers can focus more on business logic and less on repetitive database access tasks.
  - **Improved Maintainability:** Code is cleaner, easier to understand, and less prone to errors associated with manual JDBC coding.
  - **Database Independence:** By working through the ORM layer, you can often switch between different database systems (e.g., from MySQL to PostgreSQL) with minimal code changes, mainly by adjusting configuration settings. Hibernate handles the database-specific SQL dialects. (Notes Image 25).
- **Key Concepts:**
  - **Entities (Persistent Objects):** Regular Java classes (often POJOs - Plain Old Java Objects) that are mapped to database tables. (Notes Image 27 diagram shows "Persistent Object"). Marked with annotations like `@Entity` (from JPA) or defined in Hibernate mapping files (`.hbm.xml`).
  - **Mapping:** The configuration that tells Hibernate how each Entity class maps to a database table, how each Entity field maps to a table column, and how relationships between Entities map to foreign keys/join tables. Done via annotations within the Entity classes or separate XML mapping files.
  - **JPA (Java Persistence API):** The standard Java specification for ORM. Hibernate is a popular *implementation* of JPA. The `javax.persistence` (or `jakarta.persistence` in Jakarta EE) package contains the JPA classes and interfaces (`@Entity`, `@Id`, `EntityManager`, `EntityManagerFactory`). Hibernate uses these. (Notes Image 24 mentions JPA and the `javax.persistence` package).
- **Advantages of Hibernate (Notes Image 25 lists these):**
  1. Open Source and Lightweight.
  2. Fast Performance (due to internal caching - First-Level and Second-Level cache).
  3. Database Independent Queries (HQL/JPQL generates database-specific SQL).
  4. Automatic Table Creation (can generate database schema from mappings).
  5. Simplifies Complex Joins (ORM handles relationship queries).

   6. Provides Query Statistics and Database Status.

- **References:** Notes Images 24-25 introduce Hibernate as an ORM tool and JPA provider, explain the purpose of ORM, and list the key advantages.

6. Hibernate Architecture

- **What is it?**
  - The Hibernate architecture describes the main components (objects and configuration files) that make up a Hibernate application and how they interact. (Notes Image 26).
- **Why is it important?**
  - Understanding the architecture is necessary to correctly set up, configure, and use Hibernate for data persistence in your application. Each component has a specific role and lifecycle.
- **Key Concepts & Components:** The Hibernate architecture is typically viewed in layers, with several key objects involved. (Notes Images 26, 27, 28, 29 illustrate this).
  1. **Configuration:** Represents the settings required for Hibernate to connect to the database and define its behavior. (Notes Image 27 shows "Configuration File").
     - Loaded from configuration files (e.g., `hibernate.cfg.xml`, `persistence.xml` for JPA) or directly configured in code.
     - Includes database connection details (driver, URL, username, password), database dialect, mappings to Entity classes, caching settings, etc.
  2. **SessionFactory (or `EntityManagerFactory` in JPA):** (Notes Images 26, 27, 28, 29).
     - Created **once per application** (typically during application startup) using the `Configuration`.
     - It is a **factory** for `Session` objects.
     - It is **thread-safe**. Multiple threads in your application can safely use the same `SessionFactory` instance.
     - It holds database connection properties, mappings, and manages Hibernate's Second-Level Cache (optional cache shared across sessions). (Notes Image 29). It is expensive to create, hence created only once.
  3. **Session (or `EntityManager` in JPA):** (Notes Images 26, 27, 28, 29).
     - Represents a **single-threaded unit of work** or conversation with the database.
     - Obtained from the `SessionFactory` whenever database interaction is needed (e.g., at the beginning of a request or business transaction).
     - It is **NOT thread-safe**. A `Session` should only be used by one thread at a time.
     - It is **short-lived**. You typically open a session, perform some database operations, and then close the session.
     - This is the **primary object you interact with** to perform CRUD (Create, Read, Update, Delete) operations and execute queries on your persistent objects (Entities). (Notes Image

29 lists methods like insert, update, delete, Query, Criteria).

    - Manages a **First-Level Cache** (mandatory cache specific to the current session). Objects loaded or saved within this session are cached here to avoid redundant database calls within the same session. (Notes Image 29).

4. **Transaction (or `EntityTransaction` in JPA):** (Notes Images 26, 28, 29).
    - Represents an atomic unit of work with the database. All operations within a transaction succeed or fail together.
    - Obtained from the `Session` (`session.beginTransaction()`).
    - You perform database operations, then either `commit()` the transaction (make changes permanent) or `rollback()` (undo changes) if an error occurs. (Notes Image 29). Optional if using auto-commit mode (not recommended for complex operations).
5. **Persistent Objects (Entities):** Your application's Java objects that are mapped to the database and managed by Hibernate/JPA. (Notes Images 26, 27, 28).
6. **Connections:** Hibernate uses a `ConnectionProvider` (often managed internally or via a connection pool) to obtain JDBC connections to the database. (Notes Image 29).

- **Hibernate Layers:** (Notes Image 26 lists four). This is a conceptual view:
    1. **Java Application Layer:** Your code, which uses Hibernate APIs (Session, Transaction) to interact with persistent objects.
    2. **Hibernate Framework Layer:** The core Hibernate engine, including SessionFactory, Session, caches, transaction management, and mapping capabilities.
    3. **Backend API Layer:** Hibernate uses underlying APIs like JDBC (for database communication), JTA (Java Transaction API for distributed transactions), JNDI (Java Naming Directory Interface, potentially for looking up DataSources or SessionFactories). (Notes Image 26).
    4. **Database Layer:** The actual relational database.
- **Basic Workflow (Simplified - Based on Notes Image 17 from provided text):**
    1. **Load Configuration:** (Done once at startup).
    2. **Create SessionFactory:** (Done once at startup).
    3. **Open Session:** Get a `Session` from `SessionFactory` (at the start of each unit of work).
    4. **Begin Transaction:** Get a `Transaction` from the `Session` and begin it.
    5. **Execute Logic:** Use the `Session` object to save, retrieve, update, or delete persistent objects.
    6. **Commit/Rollback Transaction:** Commit the transaction if successful, rollback if an error occurred.
    7. **Close Session:** Close the `Session` to release resources.
- **References:** Notes Images 24-29 explain Hibernate as ORM, its advantages, architecture components (Configuration, SessionFactory, Session, Transaction, Entities, Connections), layers, and basic workflow.

This concludes the detailed notes for Advanced Java Unit 4, covering the Client-Server roles, RMI (Introduction, Setup Steps, Parameter Passing), and Hibernate (Introduction/ORM, Advantages, Architecture, Workflow). The explanations are based on the provided materials, incorporating the conceptual breakdown.