UNIT I: Java Fundamentals & Basic Networking

Part 1: Introduction to Java

- **What is Java?**
  - Java is a popular, high-level, object-oriented programming language.
  - It's known for "Write Once, Run Anywhere" (WORA).
- **How does WORA work? (The JVM)**
  1. You write Java code (`.java` files).
  2. You compile it using the Java compiler (`javac`) into **bytecode** (`.class` files). Bytecode is not machine code for a specific computer; it's instructions for a "virtual" computer.
  3. This bytecode can run on any computer that has a **Java Virtual Machine (JVM)** installed.
  4. The JVM reads the bytecode and translates it into actual machine code that the specific computer can understand and execute.
- **Key Features:**
  - **Object-Oriented:** Organizes code around "objects" which have data (fields) and behavior (methods). This promotes reusable and organized code (using concepts like Inheritance, Encapsulation, Polymorphism).
  - **Platform Independent:** Thanks to the JVM, Java code isn't tied to one operating system (like Windows, Mac, Linux).
  - **Simple (relatively):** Designed to be easier to learn than languages like C++.
  - **Secure:** Has features to prevent unauthorized access and malicious code.
  - **Robust:** Handles errors well (Exception Handling) and manages memory automatically (Garbage Collection).
  - **Multithreaded:** Can perform multiple tasks concurrently (Multithreading).

Part 2: Inheritance (Code Reusability)

- **What is Inheritance?**
  - It's a core Object-Oriented Programming (OOP) concept.
  - It allows a new class (called **subclass** or **child class**) to inherit properties (fields) and behaviors (methods) from an existing class (called **superclass** or **parent class**).
  - Think of it like a child inheriting traits from a parent.
- **Why Use Inheritance?**
  - **Code Reusability:** Avoid writing the same code multiple times. Define common features in a superclass, and let subclasses reuse them.
  - **Organization:** Creates an "is-a" relationship (e.g., a `Dog` *is an* `Animal`). Helps structure your code logically.
  - **Polymorphism:** Allows treating objects of different subclasses in a uniform way through the superclass type (more on this later if needed).
- **How to Use It (Keywords):**
  - `extends`: Used by the subclass to specify which superclass it inherits from.

```
class Animal { // Superclass
    void eat() {
        System.out.println("This animal eats food.");
    }
}

class Dog extends Animal { // Subclass inherits from Animal
    void bark() {
        System.out.println("Woof!");
    }
}

// Usage:
Dog myDog = new Dog();
myDog.eat();  // Inherited method from Animal
myDog.bark(); // Method defined in Dog
```

  - `super`: Used within a subclass to refer to its immediate superclass.

        - `super()`: Calls the superclass's constructor (often done implicitly, but needed explicitly if the superclass doesn't have a no-arg constructor).
        - `super.methodName()`: Calls a method from the superclass.
        - `super.fieldName`: Accesses a field from the superclass (less common, prefer methods).
- **Types of Inheritance (in Java):**
    - **Single Inheritance:** One class extends one other class (like the `Dog` example). This is what Java directly supports for classes.
    - **Multilevel Inheritance:** Class B extends Class A, Class C extends Class B (Chain: A -> B -> C). Allowed in Java.
    - **Hierarchical Inheritance:** Multiple classes extend the *same* superclass (e.g., `Dog extends Animal`, `Cat extends Animal`). Allowed in Java.
    - **Multiple Inheritance (Not for Classes):** A class extending *more than one* class directly. **Java does NOT support this for classes** to avoid complexity (the "Diamond Problem").
    - **Interfaces to the Rescue:** Java allows a class to *implement* multiple **Interfaces**, achieving a form of multiple inheritance for *type* and *contract*, but not implementation directly from multiple parent classes.

Part 3: Exception Handling (Dealing with Errors)

- **What is an Exception?**
    - An unexpected event or error that occurs during the execution of a program, disrupting the normal flow.
    - Examples: Trying to divide by zero, trying to access a file that doesn't exist, network connection lost.
- **Why Handle Exceptions?**
    - To prevent the program from crashing abruptly.
    - To gracefully manage errors, inform the user, or try alternative actions.
    - To separate error-handling code from the main program logic, making code cleaner.
- **Key Keywords:**
    - `try`: Encloses the block of code where an exception *might* occur.
    - `catch`: Follows a `try` block. Catches and handles a specific type of exception if it occurs within the `try` block. You can have multiple `catch` blocks for different exception types.
    - `finally`: Follows the `try` (and any `catch`) blocks. This block **always** executes, whether an exception occurred or not. Used for cleanup code (e.g., closing files or network connections).
    - `throw`: Used to manually *throw* an exception object, signaling that an error condition has been met.
    - `throws`: Used in a method signature to declare that the method *might* throw certain types of checked exceptions, shifting the responsibility of handling them to the calling method.
- **Example:**

```
FileReader reader = null;
try {
    reader = new FileReader("myFile.txt");
    // ... code to read from the file ...
    int result = 10 / 0; // This will cause an ArithmeticException
} catch (FileNotFoundException fnfEx) {
    System.err.println("Error: File not found!");
    // Handle the missing file case
} catch (ArithmeticException mathEx) {
    System.err.println("Error: Cannot divide by zero!");
    // Handle the math error
} catch (IOException ioEx) {
    System.err.println("Error reading file: " + ioEx.getMessage());
    // Handle other IO errors
} finally {
    System.out.println("Executing finally block.");
    if (reader != null) {
        try {
            reader.close(); // Always try to close the file
        } catch (IOException e) {
            System.err.println("Error closing file.");
        }
    }
}
System.out.println("Program continues after try-catch-finally.");
```

- **Types of Exceptions:**
    - **Checked Exceptions:** Exceptions that the Java compiler *forces* you to handle (using `try-catch` or `throws`). Usually related to external factors (like file I/O, network). Examples: `IOException`, `FileNotFoundException`, `SQLException`.
    - **Unchecked Exceptions (Runtime Exceptions):** Exceptions that the compiler *doesn't* force you to handle, though you often should. Usually indicate programming errors. Examples: `NullPointerException`, `ArrayIndexOutOfBoundsException`, `ArithmeticException`, `ClassCastException`. Subclass of `RuntimeException`.
    - **Errors:** Serious problems, usually outside the control of the application (like `OutOfMemoryError`, `StackOverflowError`). You typically don't try to catch these.

Part 4: Multithreading (Doing Multiple Things at Once)

- **What is a Thread?**
    - A single sequential flow of control within a program.
    - A program can have multiple threads running seemingly simultaneously, each performing a different task.
- **Process vs. Thread:**
    - **Process:** An independent program running in its own memory space (e.g., running Chrome, running Word). Processes are heavyweight.
    - **Thread:** A lightweight unit within a process. Multiple threads within the same process share the same memory space, making communication easier but also requiring careful management (synchronization).
- **Why Use Multithreading?**
    - **Responsiveness:** Keep user interfaces responsive while background tasks run (e.g., downloading a file without freezing the UI).
    - **Performance:** Utilize multiple CPU cores effectively by running tasks in parallel.
    - **Efficiency:** Threads share resources, potentially reducing overhead compared to multiple processes.
- **How to Create Threads:**
    1. **Extend the `Thread` class:**
        - Create a class that `extends Thread`.
        - Override the `run()` method with the code the thread should execute.
        - Create an instance of your class and call its `start()` method (which internally calls `run()`).

```
class MyThread extends Thread {
    public void run() {
        System.out.println("MyThread running...");
    }
}
// Usage:
MyThread t1 = new MyThread();
t1.start(); // Don't call run() directly!
```

2. **Implement the `Runnable` interface:** (Generally Preferred)
   - Create a class that `implements Runnable`.
   - Implement the `run()` method.
   - Create an instance of your class.
   - Create a `Thread` object, passing your `Runnable` instance to its constructor.
   - Call the `start()` method on the `Thread` object.

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("MyRunnable running...");
    }
}
// Usage:
MyRunnable myTask = new MyRunnable();
Thread t2 = new Thread(myTask);
t2.start();
```

- **Thread Life Cycle:**
  - **New:** Thread object created, but `start()` not yet called.
  - **Runnable:** `start()` called. Thread is ready to run, waiting for CPU time.
  - **Running:** Thread is actively executing its `run()` method code.
  - **Blocked/Waiting/Timed Waiting:** Thread is temporarily inactive (e.g., waiting for I/O, waiting for a lock (`synchronized`), `sleep()`, `wait()`).
  - **Terminated:** `run()` method completed execution, or an uncaught exception occurred. The thread is dead.
- **Synchronization (Preventing Chaos):**
  - When multiple threads access shared data, they can interfere with each other (e.g., one thread reads while another writes), leading to incorrect results (**Race Conditions**).
  - **Synchronization** ensures that only one thread can access a critical section of code or a shared resource at a time.
  - **`synchronized` keyword:**
    - **Synchronized Method:** `public synchronized void myMethod() { ... }` Only one thread can execute *any* synchronized method *on the same object instance* at a time.
    - **Synchronized Block:** `synchronized(someObjectLock) { ... critical code ... }` Only one thread can execute the code block that is synchronized *on the same `someObjectLock`* at a time. More granular control.
  - **Deadlock:** A situation where two or more threads are blocked forever, each waiting for a resource held by the other. Avoid complex nested locking.

Part 5: Applet Programming (Historical Context)

- **What *Were* Applets?**
  - Small Java programs designed to be embedded within web pages (HTML) and run inside the web browser using a Java plugin.
  - Popular in the early days of the web for adding dynamic content and interactivity.
- **Why Are They Obsolete?**
  - **Security Issues:** Required a browser plugin, which became a major security risk.
  - **Performance:** Could be slow to load.

    - **Alternatives:** Modern web technologies (HTML5, CSS3, JavaScript frameworks like React/Angular/Vue) provide better, safer, and more integrated ways to create interactive web content without plugins.
    - **Removed:** Browsers have removed support for Java plugins. Java itself has deprecated and marked the Applet API for removal.
- **Key Concepts (for historical understanding):**
    - Ran in a "sandbox" (restricted environment) for security.
    - Life Cycle Methods: `init()` (initialize once), `start()` (when page visible), `stop()` (when page hidden), `destroy()` (when applet unloaded), `paint()` (to draw content).
    - Extended `java.applet.Applet` or `javax.swing.JApplet`.
    - Embedded in HTML using `<applet>` or `<object>` tags.
- **Conclusion:** You might encounter applet code in legacy systems, but you should **not** use them for new development.

Part 6: Networking - Connecting to a Server (URL Connections)

- **Concept:** Your Java program acting as a **client** to fetch information from a web server (or other HTTP server).
- **Key Classes:**
    - `java.net.URL`: Represents a Uniform Resource Locator (like `http://www.example.com/index.html`). Used to identify the resource you want to connect to.
    - `java.net.URLConnection`: An abstract class representing a connection to the resource specified by a URL. You get an instance by calling `url.openConnection()`.
    - `java.net.HttpURLConnection`: A subclass of `URLConnection` specifically for handling HTTP and HTTPS connections. Provides methods to set request methods (GET, POST), handle headers, cookies, response codes, etc. You usually cast the result of `openConnection()` to this if you know it's an HTTP URL.
- **Basic Steps (e.g., Reading content from a URL):**
    1. **Create URL object:** `URL myUrl = new URL("http://www.example.com/");` (Handle `MalformedURLException`).
    2. **Open Connection:** `URLConnection connection = myUrl.openConnection();` (Handle `IOException`). Cast to `HttpURLConnection` if needed: `HttpURLConnection httpConn = (HttpURLConnection) connection;`
    3. **Configure Connection (Optional):** Set request method (`httpConn.setRequestMethod("GET");`), set headers (`httpConn.setRequestProperty("User-Agent", "MyJavaApp");`), enable input/output (`conn.setDoOutput(true);` for POST).
    4. **Connect (Often implicit):** Connecting usually happens when you try to get input/output streams or response info. You can call `conn.connect();` explicitly if needed.
    5. **Get Response:** Check response code (`httpConn.getResponseCode()`), get headers (`conn.getHeaderField(...)`).
    6. **Read Data:** Get an `InputStream` to read the server's response body: `InputStream inputStream = conn.getInputStream();` (Handle `IOException`). Read the stream (e.g., using a `BufferedReader`).
    7. **Close Streams:** Always close the `InputStream` in a `finally` block.
- **Example (Simple GET):**

```
import java.net.*;
import java.io.*;

public class UrlReader {
    public static void main(String[] args) {
        try {
            URL url = new URL("https://www.google.com");
            URLConnection conn = url.openConnection();
            BufferedReader reader = new BufferedReader(
                                        new InputStreamReader(conn.getInputStream()));
            String line;
            System.out.println("--- Content Start ---");
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
            System.out.println("--- Content End ---");
            reader.close();
        } catch (MalformedURLException e) {
            System.err.println("Invalid URL: " + e.getMessage());
        } catch (IOException e) {
            System.err.println("Error connecting or reading: " + e.getMessage());
        }
    }
}
```

Part 7: Networking - Implementing Servers & Socket Programming

- **Concept:** Your Java program acting as a **server**, waiting for clients to connect and then communicating with them. This uses lower-level **Sockets**.
- **What is a Socket?**
  - An endpoint for communication between two machines over a network.
  - Think of it like one end of a phone line. You need a socket on the client and a socket on the server to talk.
  - Sockets use underlying protocols, most commonly TCP (reliable, stream-based) or UDP (unreliable, packet-based). We'll focus on TCP.
- **Server-Side (TCP):**
  - **`java.net.ServerSocket`:** Used by the server to **listen** for incoming client connection requests on a specific port number (e.g., port 8080).
  - **Steps:**
    1. **Create ServerSocket:** `ServerSocket serverSocket = new ServerSocket(portNumber);` (Handle `IOException`).
    2. **Wait for Connection:** Call `serverSocket.accept();`. This method **blocks** (pauses the thread) until a client tries to connect. When a client connects, `accept()` returns a `java.net.Socket` object representing the connection *to that specific client*.
    3. **Communicate:** Get `InputStream` and `OutputStream` from the client `Socket` object returned by `accept()` to send and receive data with that client.
    4. **Handle Client:** Often, servers start a new Thread to handle each client connection so the main thread can go back to `accept()`ing new clients.
    5. **Close Sockets:** Close the client `Socket` and eventually the `ServerSocket` when done.
- **Client-Side (TCP):**
  - **`java.net.Socket`:** Used by the client to **connect** to a server at a specific IP address (or hostname) and port number.
  - **Steps:**
    1. **Create Socket:** `Socket clientSocket = new Socket(serverAddress, serverPort);` (Handle `IOException`). This attempts to establish the connection.
    2. **Communicate:** Get `InputStream` and `OutputStream` from the `clientSocket` to send data to and receive data from the server.
    3. **Close Socket:** Close the `clientSocket` when finished.
- **Communication (Streams):**

    - Once sockets are connected, use `InputStream` (e.g., `socket.getInputStream()`) to read bytes sent *from* the other end.
    - Use `OutputStream` (e.g., `socket.getOutputStream()`) to write bytes *to* the other end.
    - Often wrapped with higher-level readers/writers like `BufferedReader`, `PrintWriter`, `DataInputStream`, `DataOutputStream` for easier handling of text or primitive data types.
- **Example (Very Simple Echo Server):**

```java
// EchoServer.java (Server Side)
import java.net.*;
import java.io.*;

public class EchoServer {
    public static void main(String[] args) throws IOException {
        int port = 6789;
        System.out.println("Server listening on port " + port);
        ServerSocket serverSocket = new ServerSocket(port);
        try {
            // Wait for a client connection
            Socket clientSocket = serverSocket.accept();
            System.out.println("Client connected: " + clientSocket.getInetAddress());

            // Setup streams
            PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
            BufferedReader in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));

            String inputLine;
            while ((inputLine = in.readLine()) != null) {
                System.out.println("Received from client: " + inputLine);
                out.println("Server Echo: " + inputLine); // Echo back to client
                if (inputLine.equalsIgnoreCase("bye")) {
                    break;
                }
            }
            System.out.println("Client disconnected.");
            in.close();
            out.close();
            clientSocket.close();
        } finally {
            serverSocket.close();
            System.out.println("Server stopped.");
        }
    }
}
```

```
// EchoClient.java (Client Side)
import java.net.*;
import java.io.*;

public class EchoClient {
    public static void main(String[] args) throws IOException {
        String serverHostname = "localhost"; // Or server IP
        int port = 6789;
        System.out.println("Connecting to server " + serverHostname + " on port " + port);

        try (
            Socket echoSocket = new Socket(serverHostname, port);
            PrintWriter out = new PrintWriter(echoSocket.getOutputStream(), true);
            BufferedReader in = new BufferedReader(new InputStreamReader(echoSocket.getInputStream()));
            BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in))
        ) {
            System.out.println("Connected. Type messages to send (type 'bye' to quit):");
            String userInput;
            while ((userInput = stdIn.readLine()) != null) {
                out.println(userInput); // Send to server
                System.out.println("Server response: " + in.readLine()); // Read echo
                if (userInput.equalsIgnoreCase("bye")) {
                    break;
                }
            }
        } catch (UnknownHostException e) {
            System.err.println("Don't know about host " + serverHostname);
        } catch (IOException e) {
            System.err.println("Couldn't get I/O for the connection to " + serverHostname);
        }
        System.out.println("Client finished.");
    }
}
```

UNIT II: JavaBeans & Servlets

Part 1: What is a JavaBean?

- **Concept:** A JavaBean is a special kind of Java class designed to be a **reusable software component**.
- **Think of it as:** A well-behaved building block that other parts of your application (like JSPs or frameworks) can easily use and understand, often for holding data.
- **Not the same as EJB!** (Enterprise JavaBeans - covered next - are much more complex server-side components). A basic JavaBean is just a simple Java class following rules.

Part 2: Creating a JavaBean (The Rules/Conventions)

- To make a class a JavaBean, it *should* follow these rules (these help tools and frameworks interact with it automatically):

    1. **Public No-Argument Constructor:** It must have a constructor that takes no arguments (e.g., `public Employee() {}`). This allows tools to create instances easily.
    2. **Implements `java.io.Serializable`:** Allows the bean's state (its data) to be saved or sent over a network. Important for session management and other features.
    3. **Private Properties (Fields):** Data members should be declared `private` to encapsulate the data.
    4. **Public Getters and Setters:** For each private property (`propertyName`), provide public methods to access and modify it:
        - **Getter:** `public ReturnType getPropertyName()` (e.g., `public String getName()`)

        - **Setter:** `public void setPropertyName(DataType value)` (e.g., `public void setName(String name)`)
        - For boolean properties, the getter can be `isPropertyName()` (e.g., `public boolean isActive()`).

- **Example (Employee Bean):** (Based on provided notes)

```
package mypack;
import java.io.Serializable; // Rule 2

public class Employee implements Serializable { // Rule 2
    // Rule 3: Private properties
    private int id;
    private String name;

    // Rule 1: Public no-arg constructor
    public Employee() {
    }

    // Rule 4: Public Getters and Setters
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

Part 3: JavaBean Properties

- **What is a Property?**
    - A named feature or piece of data associated with the bean.
    - Properties are *accessed* through their getter and setter methods. The presence of these methods *defines* the property for external tools.
    - Example: The `Employee` bean above has properties named `id` and `name` because it has `getId`/`setId` and `getName`/`setName` methods.
- **Types of Properties:**
    - **Read/Write:** Has both a getter and a setter (like `id` and `name` above).
    - **Read-Only:** Has only a getter (e.g., if `Employee` had `public String getEmployeeCode() { return "EMP-" + id; }` but no `setEmployeeCode`, then `employeeCode` would be a read-only property).
    - **Write-Only:** Has only a setter (less common).

Part 4: Accessing and Using JavaBeans

- You access and use JavaBeans just like any other Java object.
- Frameworks like JSP (using `<jsp:useBean>`, `<jsp:getProperty>`, `<jsp:setProperty>`) rely on the JavaBean conventions to interact with bean instances.
- **Example (Simple Java Usage):** (Based on provided notes)

```
package mypack;

public class Test {
    public static void main(String args[]) {
        Employee e = new Employee(); // Create instance using no-arg constructor
        e.setName("Arjun");       // Use setter
        System.out.println(e.getName()); // Use getter
    }
}
```

Part 5: Advantages and Disadvantages of JavaBeans

- **Advantages:** (From notes)
  - **Reusability:** Components can be used in multiple applications.
  - **Encapsulation:** Hides internal data, exposes controlled access via methods.
  - **Tool Friendly:** Conventions allow visual development tools and frameworks to easily inspect and manipulate beans.
  - **Maintainability:** Encapsulates data and related logic.
- **Disadvantages:** (From notes)
  - **Mutability:** Standard JavaBeans are mutable (their state can be changed via setters). This can sometimes be a drawback compared to immutable objects in concurrent environments.
  - **Boilerplate Code:** Writing getters and setters for many properties can feel repetitive (though modern IDEs generate them easily).

Part 6: Types of Beans (Enterprise JavaBeans - EJB)

- **IMPORTANT Distinction:** The following types (**Session Beans**, **Entity Beans**, **Message-Driven Beans**) are **Enterprise JavaBeans (EJBs)**.
- EJBs are much more complex, **server-side components** used in large, distributed enterprise applications. They are *not* the simple data-holding JavaBeans we just discussed.
- **What are EJBs?** (From notes)
  - A specification (set of rules/APIs) for building robust, scalable, secure, distributed applications.
  - They run inside an **EJB Container** (part of an Application Server like JBoss, Glassfish, WebSphere).
  - The Container provides services like transaction management, security, lifecycle management, and pooling automatically.
- **When to Use EJBs?** (From notes)
  - When you need remote access (distributed application).
  - When the application needs to be highly scalable (handle many users).
  - When you need to encapsulate complex business logic on the server.

Part 7: EJB - Session Beans (Handling Business Logic/Tasks)

- **Purpose:** Perform tasks or business logic for a client. They represent interactions or conversations. They are typically **not persistent** (don't directly store data long-term in a database themselves, though they might *use* persistent data).
- **Types of Session Beans:**
  1. **Stateless Session Bean:**
     - **No Conversational State:** Does *not* remember anything specific about a particular client between method calls.
     - **Pooled:** The container keeps a pool of identical instances. Any instance can serve any client request.
     - **Scalable:** Good for handling many clients because fewer instances are needed.
     - **Use Cases:** Generic tasks, operations that don't depend on previous interactions, web services.
  2. **Stateful Session Bean:**
     - **Maintains Conversational State:** Remembers data specific to one client across multiple method calls within a single "session" or interaction.
     - **Dedicated Instance:** Each client typically interacts with its own dedicated instance.
     - **Not Shared:** State is not shared between clients.
     - **Use Cases:** Shopping carts, multi-step processes, wizards where data needs to be kept between steps for one user.
  3. **Singleton Session Bean:**
     - **One Instance Per Application:** Only a single instance of this bean exists for the entire application.
     - **Shared State:** Can hold state that needs to be shared across the whole application.
     - **Concurrency:** Needs careful handling if multiple clients access it concurrently.

        - **Use Cases:** Application-wide cache, application startup/shutdown tasks.

Part 8: EJB - Entity Beans (Representing Persistent Data - Older Approach)

- **Purpose:** Represented persistent data in a database (like a row in a table). They encapsulated data *and* the business logic related to that data.
- **Key Features:**
    - **Persistent:** Survived server crashes because their state was stored in a database.
    - **Primary Key:** Each entity bean instance had a unique identity (primary key).
    - **Shared:** Could be accessed by multiple clients.
    - **Recoverable:** Could be restored after failure.
- **Persistence Management:**
    - **Bean-Managed Persistence (BMP):** The *developer* had to write the JDBC code inside the bean to save/load data to/from the database. More flexible, but complex and tied to a specific DB.
    - **Container-Managed Persistence (CMP):** The *EJB Container* automatically handled saving/loading data based on deployment descriptor mappings. Simpler for the developer, but less flexible.
- **IMPORTANT NOTE: Entity Beans are largely obsolete.** They have been replaced by the **Java Persistence API (JPA)** standard. JPA uses simple annotated POJOs (@Entity classes, similar to basic JavaBeans but with persistence annotations) managed by an `EntityManager` (often provided by frameworks like Hibernate or EclipseLink) which is a much cleaner and more powerful approach.

Part 9: EJB - Message-Driven Beans (MDBs) (Handling Asynchronous Messages)

- **Purpose:** Act as asynchronous message consumers, typically listening to a **Java Message Service (JMS)** queue or topic.
- **How they work:**
    - They don't get called directly by clients.
    - The EJB Container listens to the JMS destination (queue/topic).
    - When a message arrives, the container takes an MDB instance from a pool and invokes its `onMessage()` method, passing the message.
- **Key Features:**
    - **Asynchronous:** Decouples message senders from receivers.
    - **Stateless (like Stateless Session Beans):** Don't maintain conversational state. Container manages a pool.
    - **Event-Driven:** React to incoming messages.
- **Benefits over raw JMS Listeners:** (From notes)
    - Container handles consumer creation, registration, transaction management, pooling, etc.

Part 10: Servlet Overview and Architecture

- **What is a Servlet?**
    - A Java program that runs **on a web server** (inside a Servlet Container).
    - Used to handle client requests (usually HTTP requests from web browsers) and generate dynamic responses (usually HTML web pages, but can be JSON, XML, etc.).
    - The server-side powerhouse behind many Java web applications.
- **Why Use Servlets?**
    - Generate dynamic content based on user input or data.
    - Process form submissions.
    - Interact with databases.
    - Manage user sessions.
    - Platform independent (Java!).
    - More efficient than older technologies like CGI (uses threads instead of processes per request).
- **Servlet Architecture (How a Request Flows):** (Based on notes)
    1. **Client (Browser):** Sends an HTTP Request (e.g., asking for `/myapp/login`).
    2. **Web Server:** Receives the request and passes it to the Servlet Container (e.g., Tomcat, Jetty).
    3. **Servlet Container:**
        - Finds which Servlet should handle the request based on the URL pattern (configured in `web.xml` or using annotations).
        - If the Servlet isn't loaded/running, it loads the class, creates an instance, and calls `init()`.
        - Creates `HttpServletRequest` and `HttpServletResponse` objects.
        - Creates a thread for this request and calls the Servlet's `service()` method, passing the request and response objects.
    4. **Servlet:**

        - The `service()` method (usually in `HttpServlet`) determines if it's a GET, POST, etc., request and calls the appropriate `doGet()` or `doPost()` method.
        - Your `doGet()`/`doPost()` code reads data from the `request` object (e.g., form parameters).
        - Performs business logic (talks to database, calls other classes).
        - Writes the response content (e.g., HTML) using the `response` object's `PrintWriter` or `OutputStream`.
5. **Servlet Container:** Sends the generated response back through the Web Server.
6. **Web Server:** Sends the HTTP Response back to the Client.
7. **Client (Browser):** Renders the received HTML.

Part 11: Servlet Interface and Life Cycle

- **`javax.servlet.Servlet` Interface:**
    - The core interface all servlets must implement (directly or indirectly).
    - Defines the essential methods for a servlet's life cycle and request handling.
- **Life Cycle Methods (Managed by the Container):**
    1. **`init(ServletConfig config):`**
        - Called **once** when the servlet instance is first created.
        - Used for one-time initialization tasks (e.g., loading database drivers, reading configuration).
        - The `ServletConfig` object provides initialization parameters and access to the `ServletContext`.
    2. **`service(ServletRequest req, ServletResponse res):`**
        - Called by the container **for every request** made to the servlet.
        - This is where the main request processing logic happens.
        - It receives request data via `ServletRequest` and sends the response via `ServletResponse`.
        - In `HttpServlet`, this method examines the HTTP method (GET, POST) and calls the corresponding `doGet()`, `doPost()`, etc.
    3. **`destroy():`**
        - Called **once** just before the servlet instance is taken out of service (e.g., when the web application shuts down).
        - Used for cleanup tasks (e.g., closing database connections, releasing resources).
- **Other `Servlet` Methods:**
    - `getServletConfig()`: Returns the `ServletConfig` object passed to `init()`.
    - `getServletInfo()`: Returns a string with information about the servlet (author, version).
- **Servlet States:** New -> Initialized (Ready) -> Destroyed. The servlet spends most of its life in the Ready state, handling requests via the `service` method.

Part 12: Servlet Implementation Classes

- You usually don't implement `Servlet` directly. You extend helper classes:
    1. **`javax.servlet.GenericServlet`:**
        - An abstract class that implements `Servlet` and `ServletConfig`.
        - Provides basic implementations for `init()`, `destroy()`, `getServletConfig()`, etc.
        - Makes writing servlets easier by requiring you only to implement the abstract `service()` method.
        - Protocol-independent (can handle protocols other than HTTP, though rarely used for that).
    2. **`javax.servlet.http.HttpServlet`:** (Most Common)
        - An abstract class that **extends `GenericServlet`**.
        - Specifically designed for handling **HTTP** requests.
        - Implements the `service()` method to parse the HTTP request and dispatch it to specific `doXXX()` methods based on the HTTP method (GET, POST, PUT, DELETE, etc.).
        - You typically **override** methods like `doGet()`, `doPost()`, etc., to handle specific request types, instead of overriding `service()`.

Part 13: Handling HTTP GET and POST Requests

- **HTTP Methods:**
    - **GET:** Used to request data from the server. Parameters are usually appended to the URL (query string). Should be idempotent (making the same request multiple times has the same effect). Used for fetching pages, searching. Browser default for links and typing URLs.
    - **POST:** Used to send data to the server to be processed (e.g., submitting a form, creating a resource). Parameters are sent in the request body, not the URL. Not necessarily idempotent (submitting the same form twice might create two entries).

- **Handling in `HttpServlet`:**
  - **Override `doGet(HttpServletRequest req, HttpServletResponse resp)`:** Implement this method to handle requests made using the HTTP GET method.
  - **Override `doPost(HttpServletRequest req, HttpServletResponse resp)`:** Implement this method to handle requests made using the HTTP POST method.
- **Common Tasks within `doGet`/`doPost`:**
  1. **Read Request Parameters:**
     - `String value = req.getParameter("parameterName");` (Gets a single value).
     - `String[] values = req.getParameterValues("parameterName");` (Gets multiple values, e.g., from checkboxes).
  2. **Read Request Headers:** `String userAgent = req.getHeader("User-Agent");`
  3. **Set Response Content Type:** `resp.setContentType("text/html; charset=UTF-8");` (Tell the browser what kind of data you're sending).
  4. **Get Writer/OutputStream:**
     - `PrintWriter out = resp.getWriter();` (For sending character/text data like HTML).
     - `ServletOutputStream outStream = resp.getOutputStream();` (For sending binary data like images).
  5. **Write Response:** Use the writer/output stream to send the content back to the client (`out.println("<html>...")`).
  6. **Close Writer/Stream:** `out.close();`
- **Example Structure:** (Combines GET/POST handling)

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

public class MyFormServlet extends HttpServlet {

    // Handles GET requests
    public void doGet(HttpServletRequest req, HttpServletResponse resp)
                            throws ServletException, IOException {
        resp.setContentType("text/html");
        PrintWriter out = resp.getWriter();
        out.println("<html><body>");
        out.println("<h2>GET Request Received</h2>");
        String name = req.getParameter("name");
        if (name != null) {
            out.println("Hello, " + name + "!");
        } else {
            out.println("Please provide a name parameter in the URL.");
        }
        // Display the form (could be in a separate JSP)
        out.println("<form method='POST' action='myForm'>"); // Submit via POST
        out.println("Name: <input type='text' name='name'><br>");
        out.println("<input type='submit' value='Submit via POST'>");
        out.println("</form>");
        out.println("</body></html>");
        out.close();
    }

    // Handles POST requests
    public void doPost(HttpServletRequest req, HttpServletResponse resp)
                            throws ServletException, IOException {
        resp.setContentType("text/html");
        PrintWriter out = resp.getWriter();
        out.println("<html><body>");
        out.println("<h2>POST Request Received</h2>");
        String name = req.getParameter("name"); // Get param from request body
        if (name != null && !name.trim().isEmpty()) {
            out.println("Thank you for submitting your name: " + name);
        } else {
            out.println("You submitted an empty name.");
        }
        out.println("<br><a href='myForm'>Go back</a>"); // Link back (makes a GET request)
        out.println("</body></html>");
        out.close();
    }
}
```

Part 14: Session Tracking (Remembering Users)

- **Problem:** HTTP is **stateless**. Each request from a client is independent; the server doesn't automatically know if multiple requests come from the same user.
- **Need:** We often need to maintain a "session" or "conversation" with a user across multiple requests (e.g., for login status, shopping carts).
- **Session Tracking:** The process of associating multiple requests with the same user.
- **Techniques:**
    1. **Cookies:** (Most Common for session IDs)

2. **Hidden Form Fields:** Include hidden `<input type="hidden">` fields in forms to pass data (like a session ID) back to the server. Works only for sequences of form submissions. Clunky.
3. **URL Rewriting:** Append a session ID directly to URLs (`/myapp/page?sessionid=12345`). Looks ugly, can break bookmarks, security concerns. Used as a fallback if cookies are disabled.
4. **HttpSession API:** (Built on top of Cookies or URL Rewriting) Java Servlet API provides a convenient way to manage sessions using the `HttpSession` object.

Part 15: Cookies (Small Pieces of Client-Side Data)

- **What is a Cookie?**
  - A small piece of text data that the server sends to the client's browser.
  - The browser stores the cookie.
  - The browser automatically sends the same cookie back to the *same server* with every subsequent request.
- **How They Work (Session Tracking Use Case):**
  1. User makes first request.
  2. Server generates a unique session ID.
  3. Server creates a Cookie containing this session ID (`JSESSIONID` is the common name).
  4. Server adds this Cookie to the HTTP response.
  5. Browser receives the response and stores the Cookie.
  6. User makes subsequent requests to the same server.
  7. Browser automatically includes the session ID Cookie in the HTTP request header.
  8. Server reads the Cookie from the request, gets the session ID, and can then retrieve the user's specific session data (stored on the server).
- **Key Java Classes/Methods:**
  - `javax.servlet.http.Cookie`: Class representing a cookie.
    - Constructor: `Cookie(String name, String value)`
    - `setMaxAge(int seconds)`: Sets how long the cookie should live (0 deletes, negative means non-persistent/session cookie, positive value is persistence time).
    - `getName()`, `getValue()`, `setValue(String value)`.
  - `HttpServletResponse.addCookie(Cookie ck)`: Adds a cookie to the response to be sent to the client.
  - `HttpServletRequest.getCookies()`: Returns an array (`Cookie[]`) of all cookies sent by the client with the request. You need to loop through this array to find the cookie you're interested in.
- **Types of Cookies:**
  - **Non-Persistent (Session Cookie):** Default. Deleted when the browser closes. Used for tracking sessions. `setMaxAge` is negative or not set.
  - **Persistent Cookie:** Stored on the user's disk until an expiration date. Used for remembering logins ("Remember Me"), preferences, etc. `setMaxAge` is positive.
- **Advantages:** Simple mechanism, supported by all browsers.
- **Disadvantages:** Users can disable cookies, limited size (~4KB), security concerns (can be stolen - use `HttpOnly` and `Secure` flags), only stores text.
- **Example:** See `FirstServlet`/`SecondServlet` in provided notes - demonstrates creating a cookie in one servlet and reading it in another.

Part 16: HttpSession API (The Preferred Way for Sessions)

- While Cookies (or URL rewriting) handle *sending* the session ID back and forth, the `HttpSession` API provides the server-side mechanism to *manage* the actual session data associated with that ID.
- **Getting the Session:**
  - `HttpSession session = request.getSession();` // Gets current session, creates one if none exists.
  - `HttpSession session = request.getSession(false);` // Gets current session, returns `null` if none exists (doesn't create one).
- **Storing Data:** `session.setAttribute("attributeName", attributeValueObject);` (Stores any Java object).
- **Retrieving Data:** `Object value = session.getAttribute("attributeName");` (Returns the object, needs casting).
- **Removing Data:** `session.removeAttribute("attributeName");`
- **Invalidating Session:** `session.invalidate();` (Logs the user out, discards all session data).
- **Other Useful Methods:**
  - `session.getId()`: Gets the unique session ID string.

    - `session.isNew()`: Checks if the session was just created.
    - `session.setMaxInactiveInterval(int seconds)`: Sets the session timeout period.
- **How it works:** The `request.getSession()` method typically looks for the `JSESSIONID` cookie (or rewritten URL), finds the corresponding session object on the server, and returns it. If none is found (and argument is not `false`), it creates a new session object, generates a new ID, sets the `JSESSIONID` cookie in the response, and returns the new session object.
