ONESHOT ADV JAVA

JAVA UNIT 1 NOTES - ULTRA CONCISE

A. Java Versions & Key Features `[PYQ]`

- **Early (JDK 1.0-1.4):** Core lang, Applets, AWT, Swing, JavaBeans, JDBC, RMI, `java.io`, `java.net`. JDK 1.4: NIO.
- **Java 5 (JDK 1.5/5.0):** Generics (`List<String>`), Enums, Autoboxing/Unboxing, Enhanced `for` loop, Varargs, Annotations (`@Override`).
- **Java 6 (JDK 6):** Performance, DB integration, Web services (JAX-WS, JAXB).
- **Java 7 (JDK 7):** Diamond operator (`<>`), try-with-resources, NIO.2.
- **Java 8 (JDK 8):** Lambda Expressions, Stream API, Default Methods in Interfaces, New Date/Time API.
- **Java 9+:** Modularity (Project Jigsaw), `var` keyword, Text Blocks, Pattern Matching.

B. JRE vs. JDK vs. JVM `[PYQ]`

- **JVM (Java Virtual Machine):** Abstract spec & runtime instance; executes bytecode; Manages memory (GC); "Write Once, Run Anywhere."
- **JRE (Java Runtime Environment):** JVM + Libraries (Java API classes); Minimum to *run* Java programs.
- **JDK (Java Development Kit):** JRE + Development Tools (compiler `javac`, debugger `jdb`, `jar`, `javadoc`). Needed to *develop, compile, and run*.

C. How Java Compilation Works `[PYQ]`

1. **Compilation:** `Your_Code.java` (Source) --(`javac`)--> `Your_Code.class` (Bytecode - platform-independent).
2. **Execution:** `java Your_Code` --> Launcher starts JVM --> JVM loads `.class` --> Verifies bytecode --> Executes bytecode (often with JIT compilation to native code for speed).

D. Security Features of Java `[PYQ]`

1. **Bytecode Verifier:** Checks `.class` file integrity & adherence to Java rules before execution.
2. **Security Manager (Sandbox):** Restricts untrusted code (e.g., Applets) permissions (file access, network). `[PYQ]`
3. **Exception Handling:** Prevents crashes, allows graceful error recovery. `[PYQ]`
4. **Automatic Memory Management (Garbage Collection):** Prevents memory leaks & buffer overflows.
5. **Type Safety:** Strong typing + compiler/JVM checks prevent type errors; Generics enhance this.

Unit I: Deep Dive

1. Introduction to Java / Advanced Java (Core vs. Enterprise) `[PYQ]`

- **Core Java (J2SE):** Foundation; basic language, I/O, collections; For desktop/CLI apps.
- **Advanced Java (JEE):** Specs/APIs on Core Java; for large-scale, distributed, multi-tiered enterprise apps (Servlets, JSP, EJB). Highlights need for networking (Sockets, URLConnections).

2. Inheritance `[PYQ]`

- **Concept:** Subclass inherits properties/methods from superclass (`class Dog extends Animal {}`). `super` keyword.
- **Benefits:** Code Reusability, Maintainability, Polymorphism, Organization.
- **Types:** Single, Multilevel (`ElectricCar extends Car extends Vehicle`), Hierarchical. (Direct multiple class inheritance not supported, use interfaces).
- **Example:** `class Animal { void eat(){...} } class Dog extends Animal { void bark() {...} }`

3. Exception Handling `[PYQ]`

- **Concept:** Structured way to deal with runtime errors (exceptions). `try-catch-finally` blocks.
  - `try`: Code that might throw.
  - `catch (ExceptionType e)`: Handles specific exception.
  - `finally`: Always executes (resource cleanup).
  - `throw`: Explicitly throw an exception.
  - `throws`: Declare exceptions a method might throw.
- **Types:**
  - **Checked Exceptions:** Compiler forces handling (e.g., `IOException`, `SQLException`).
  - **Unchecked (Runtime) Exceptions:** Often programming errors (e.g., `NullPointerException`, `ArithmeticException`, `ArrayIndexOutOfBoundsException`).
  - **Errors:** Serious problems, usually not caught (e.g., `OutOfMemoryError`).
  - **Custom Exceptions:** `class MyEx extends Exception {}`.

4. Multithreading `[PYQ]`

- **Concept:** Concurrent execution of multiple parts (threads) of a program.
- **Benefits:** Responsiveness, Performance (parallelism), Handling multiple requests.
- **Creating Threads:**
  1. `class MyThread extends Thread { public void run() {...} }`
  2. `class MyRunnable implements Runnable { public void run() {...} }` then `new Thread(myRunnableInstance).start();` (Preferred).
- **Starting:** `thread.start();` (calls `run()`).

- **Lifecycle:** New, Runnable, Running, Blocked/Waiting, Timed Waiting, Terminated.
- **Synchronization:** Critical for shared resources.
  - `synchronized` keyword (methods/blocks).
  - `wait()`, `notify()`, `notifyAll()` (Object methods for inter-thread communication in `synchronized` context).
  - `java.util.concurrent.locks.Lock` interface (more flexible).

5. Applet Programming `[PYQ]`

- **Concept:** Small Java programs embedded in HTML, run in browser (with plugin). Now largely deprecated.
- **Key Class:** `java.applet.Applet`.
- **Lifecycle Methods:** `[PYQ]`
  - `init()`: Called once on load.
  - `start()`: Called after `init` and when applet page is revisited.
  - `paint(Graphics g)`: Called to redraw applet content. `[PYQ]`
  - `stop()`: Called when user leaves applet page.
  - `destroy()`: Called once when applet is unloaded.
- **HTML Tag:** `<applet code="MyApplet.class" width="W" height="H"></applet>`.
- **Applet Viewer:** Tool to run applets without a browser.
- **Security:** Runs in a sandbox. `[PYQ]`
- **Example:** `public class MyApplet extends Applet { public void paint(Graphics g){ g.drawString("Hi",10,10); } }`

6. Connecting to a Server / Making URL Connections `[PYQ]`

- **Concept:** Using `java.net.URL` and `java.net.URLConnection` (or `HttpURLConnection`) for HTTP/HTTPS communication.
- **`URL`:** Represents resource address. `URL url = new URL("http://example.com");`
- **`HttpURLConnection`:** For HTTP. Get via `(HttpURLConnection)url.openConnection();`.
- **Steps (Simplified):**
  1. Create `URL` object.
  2. `openConnection()` from `URL`. Cast to `HttpURLConnection`.
  3. Configure: `setRequestMethod("GET" / "POST")`, `setRequestProperty("Header","Value")`. For POST: `setDoOutput(true)`.
  4. Send Data (POST): Get `OutputStream`, write data.
  5. Get Response Code: `getResponseCode()`. (200 OK).
  6. Receive Data: Get `InputStream`, read data (often with `BufferedReader`).

7. Close: `disconnect()`.

- **Example (GET):**

```
URL url = new URL("...");
HttpURLConnection con = (HttpURLConnection) url.openConnection();
con.setRequestMethod("GET");
BufferedReader in = new BufferedReader(new
InputStreamReader(con.getInputStream()));
// read from 'in'
in.close(); con.disconnect();
```

7. Implementing Servers / Socket Programming `[PYQ]`

- **Concept:** Lower-level TCP/IP or UDP communication using `java.net.Socket` and `java.net.ServerSocket`.
- **`ServerSocket` (Server Side):** Listens for client connections on a port.
  - `ServerSocket ss = new ServerSocket(port);`
  - `Socket clientSocket = ss.accept();` (Blocks until client connects). `[PYQ]`
- **`Socket` (Client Side & Server's client handler):** Represents one endpoint of a connection.
  - Client: `Socket s = new Socket("host", port);`
- **Streams:** Get `InputStream` and `OutputStream` from `Socket` for communication.
  - `clientSocket.getInputStream();`, `clientSocket.getOutputStream();` (Often wrapped). `[PYQ]`
- **Communication:** Write to `OutputStream`, read from `InputStream`.
- **Closing:** Close streams, `Socket`, and `ServerSocket`.
- **Example (Server Snippet):**

```
ServerSocket server = new ServerSocket(5000);
Socket client = server.accept();
PrintWriter out = new PrintWriter(client.getOutputStream(), true);
out.println("Hello Client");
// ... close resources
```

- **Example (Client Snippet):**

```
Socket socket = new Socket("localhost", 5000);
BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
System.out.println(in.readLine());
// ... close resources
```

8. Java NIO (New Input/Output) `[PYQ]`

- **Concept:** More efficient I/O, especially for large files/network. Introduced JDK 1.4.
- **Key Features:**
  - **Channels:** Connection to I/O source/destination (file, socket). Data read/written via Buffers.
  - **Buffers:** Memory blocks for temporary data storage during I/O.
  - **Selectors:** For non-blocking I/O; monitor multiple Channels.
- **`java.nio.file` Package:** Modern file system interaction.
  - **`Path`:** Platform-independent file/directory path. `Paths.get("dir", "file.txt");`
  - **`Files`:** Static utility methods for file ops (read, write, copy, delete).
    - `Files.exists(path)`, `Files.readAllLines(path)`, `Files.write(path, bytes)`.
- **Old I/O (`java.io`):** Stream-oriented, blocking.
- **New I/O (`java.nio`):** Channel/Buffer-oriented, non-blocking possible.
- **Example (Read file with NIO):**

```
Path filePath = Paths.get("mydata.txt");
if (Files.exists(filePath)) {
    List<String> lines = Files.readAllLines(filePath);
    lines.forEach(System.out::println);
}
```

JAVA UNIT 2 NOTES - ULTRA CONCISE

A. JavaBeans `[PYQ]`

- **What:** Reusable Java software components following conventions. Encapsulate data/logic.
- **Conventions:** `[PYQ]`
  1. **Public No-Argument Constructor:** `public MyBean() {}`.
  2. **Serializable:** `implements java.io.Serializable`. `[PYQ]`
  3. **Properties with Getters/Setters:** Private fields, public accessors.
     - Getter: `public <Type> getPropertyName()` or `public boolean isPropertyName()`. `[PYQ]`
     - Setter: `public void setPropertyName(<Type> value)`. `[PYQ]`
- **Property:** Named feature accessed via getter/setter.
- **Advantages:** Reusability, Encapsulation, Interoperability (tools), Maintainability.
- **Disadvantages:** Mutable by default, boilerplate code for many properties.
- **Example (`Employee.java`):** `[PYQ]`

```
public class Employee implements Serializable {
    private int id; private String name;
```

```
    public Employee(){}
    public int getId(){return id;} public void setId(int id){this.id=id;}
    public String getName(){return name;} public void setName(String n)
{name=n;}
}
```

- **Accessing:** `Employee e = new Employee(); e.setName("Test"); System.out.println(e.getName());`

B. Enterprise Java Beans (EJBs)

- **What:** Server-side components in EJB container (Java EE App Server like JBoss, Glassfish).
- **Why:** Handle enterprise concerns (lifecycle, transactions, security, concurrency, remote access, scalability) allowing focus on business logic.
- **Types of EJBs:** `[PYQ]` (for Q5 implicitly)
  1. **Session Beans:** Business logic/tasks. Not persistent.
     - **Stateless:** No client-specific state between calls. Pooled. Efficient. `[PYQ]`
       - Syntax: `@Stateless public class MyStatelessBean implements MyService {...}`
     - **Stateful:** Maintains client-specific conversational state. One-to-one client-bean.
       - Syntax: `@Stateful public class MyStatefulBean implements MyCart {...}`
     - **Singleton:** Single instance for entire app. Shared resources.
       - Syntax: `@Singleton @Startup public class AppConfigBean {...}`
  2. **Entity Beans (Largely Replaced by JPA):** Represented persistent DB data. Mapped to DB rows.
     - **Persistence:** BMP (Bean-Managed), CMP (Container-Managed).
     - Note: Modern Java EE uses JPA (POJOs with `@Entity`) for persistence. `[PYQ]`
  3. **Message-Driven Beans (MDBs):** Asynchronous message consumers (JMS).
     - Listen to JMS queue/topic. `onMessage(Message msg)` method processes messages.
     - Syntax: `@MessageDriven(activationConfig = {...}) public class MyMDB implements MessageListener {...}`
- **Difference: Session vs. Entity Beans:**
  - Primary Key: Entity (yes), Session (no).
  - State: Entity (stateful DB record), Session (stateless/stateful conversation).
  - Span: Entity (persists beyond session), Session (limited to conversation/app lifecycle).
  - Persistence: Entity (manages persistent data), Session (doesn't store data persistently itself).

C. Servlets `[PYQ]`

- **What:** Java programs on web server extending server capabilities; handle dynamic web content via HTTP.
- **Why:** Dynamic content, Handle HTTP Requests/Responses (GET, POST), Improved performance (vs CGI), Platform independent, Robust, Scalable, Integration (JDBC, JSP).
- **Key Concepts:**
  - Run in Web Container (Tomcat, Jetty). Manages lifecycle, maps URLs, handles threads.
  - Architecture: Browser -> Web Server -> Web Container -> Servlet.
  - Request Flow: Client req -> Container finds servlet (web.xml/@WebServlet) -> `service()` called -> Servlet processes (HttpServletRequest/Response) -> Servlet writes response -> Container sends.
- **Types (Implementation path):**
  1. `implements Servlet`: Implement all 5 methods (`init`, `service`, `destroy`, etc.).
  2. `extends GenericServlet`: Abstract class, override `service()`. Protocol-independent.
  3. `extends HttpServlet`: Abstract, for HTTP. `service()` dispatches to `doGet()`, `doPost()`, etc. Override `doXxx()`. Most common. `[PYQ]`
- **Servlet Life Cycle:** `[PYQ]`
  - Loading -> Instantiation (once, no-arg constructor) -> Initialization (`init(ServletConfig)`) (once) -> Request Handling (`service()`/`doXxx()`) (per request) -> Destruction (`destroy()`) (once).
  - Arrow Text Diagram: `[PYQ]` `Load -> Instantiate -> init() -> Ready --(Request)--> service() --(Done)--> Ready --(Shutdown)--> destroy() -> Unloaded`
- **Interface `Servlet` Methods:** `init()`, `service()`, `destroy()`, `getServletConfig()`, `getServletInfo()`.
- **Create & Deploy:** Dir structure (WEB-INF/classes, WEB-INF/lib) -> Create class (extends HttpServlet) -> Compile (need servlet-api.jar) -> `web.xml` (or `@WebServlet`) for mapping -> Package WAR -> Deploy to server -> Access via URL.

D. Handling HTTP GET Requests `[PYQ]`

- **What:** GET requests data; params in URL query string (`?p1=v1&p2=v2`).
- **Servlet:** `extends HttpServlet`, override `doGet(HttpServletRequest req, HttpServletResponse res)`.
- **`HttpServletRequest req`:** Get params: `req.getParameter("name")`, `req.getParameterValues("name")`, `req.getParameterNames()`.
- **`HttpServletResponse res`:** Set content type: `res.setContentType("text/html")`. Get writer: `PrintWriter out = res.getWriter()`. Write HTML: `out.println(...)`. Close: `out.close()`.
- **Example:**

```
protected void doGet(HttpServletRequest req, HttpServletResponse res)
throws IOException {
    String name = req.getParameter("user");
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    out.println("<h1>Hello " + (name != null ? name : "Guest") + "
</h1>");
}
```

E. Handling HTTP POST Requests `[PYQ]`

- **What:** POST sends data to server (create/update resource, form submission). Params in request body.
- **Servlet:** `extends HttpServlet`, override `doPost(HttpServletRequest req, HttpServletResponse res)`.
- **`HttpServletRequest req`:** Get params from body: `req.getParameter("name")` (same as GET).
- **`HttpServletResponse res`:** Same as for GET.
- **Example:**

```
protected void doPost(HttpServletRequest req, HttpServletResponse res)
throws IOException {
    String email = req.getParameter("email");
    res.getWriter().println("<p>Email submitted: " + email + "</p>");
}
```

F. Session Tracking, Cookies `[PYQ]`

- **What:** HTTP is stateless. Session tracking maintains user state across multiple requests.
- **Techniques:** Cookies, Hidden Form Fields, URL Rewriting, `HttpSession`.
- **Cookies (`javax.servlet.http.Cookie`):** Small text files; server sends to browser, browser stores and sends back with subsequent requests. `[PYQ]`
  - `Cookie c = new Cookie("name", "value");` `[PYQ]`
  - `c.setMaxAge(seconds);` (Positive: persistent; -1: session; 0: delete). `[PYQ]`
  - `response.addCookie(c);` (Sends to browser). `[PYQ]`
  - `Cookie[] cookies = request.getCookies();` (Gets from browser). `[PYQ]`
- **How Cookies Work for Session Tracking (Simplified):**
  1. Client requests.
  2. Server creates cookie (e.g., session ID), `response.addCookie()`.
  3. Browser stores cookie.
  4. Client subsequent request.
  5. Browser auto-sends cookie in request header.

6. Server `request.getCookies()`, reads ID, retrieves user state.

- **Example (Setting & Getting):**

```
// Servlet1 (Setter)
Cookie userCookie = new Cookie("user", "john");
userCookie.setMaxAge(3600); response.addCookie(userCookie);
// Servlet2 (Getter)
Cookie[] cks = request.getCookies(); if(cks!=null) for(Cookie c:cks)
if(c.getName().equals("user")) out.print(c.getValue());
```

JAVA UNIT 3 NOTES - ULTRA CONCISE

Part 1: What is JSP and Why Use It? `[PYQ]`

- **JSP (JavaServer Pages):** HTML page with embedded Java code; creates dynamic web pages. Files: `.jsp`. Combines static (HTML/XML) and dynamic (JSP tags, Java) content.
- **Why Use JSP (vs. HTML):** HTML is static. JSP shows different content based on input, time, DB, etc.
- **Why Use JSP (vs. Servlets):** `[PYQ]`
  - Servlets mix presentation (HTML via `out.println`) with Java logic = messy for complex layouts.
  - JSP is easier for page layout (HTML-like); Java added via tags. Clearer for designers.
- **Key Idea:** Separates presentation (HTML) from logic (Java). (MVC: JSP as View).
- **Relationship to Servlets:** JSP is extension/wrapper over Servlet tech. JSP page translated/compiled into a Java Servlet. `[PYQ]`

Part 2: How JSP Works (Magic Behind the Scenes) `[PYQ]`

1. **Translation Phase (First request or modified `.jsp`):**
   - JSP engine reads `.jsp` file.
   - Translates to Java Servlet source (`.java`). Static HTML -> `out.println()`; dynamic elements -> Java code.
2. **Request Processing Phase (Every user request, after translation/compilation):**
   - **Compilation (First request/after modification):** `.java` to `.class` (Servlet bytecode).
   - **Loading:** Servlet `.class` file loaded.
   - **Instantiation:** Servlet object created (typically once).
   - **Execution:** Compiled Servlet's `_jspService()` method (auto-generated) runs, contains logic from JSP, generates HTML/XML.
   - **Sending Response:** Container sends Servlet output to browser (user sees HTML, not JSP source).

- **Faster Subsequent Requests:** Skips translation/compilation if `.jsp` unchanged.

Part 3: JSP Life Cycle (Birth, Life, Death of a JSP) `[PYQ]`
Based on Servlet lifecycle + translation/compilation.

1. **Translation:** `.jsp` to `.java`.
2. **Compilation:** `.java` to `.class`.
3. **Loading:** Servlet `.class` loaded.
4. **Instantiation:** Servlet instance created (once).
5. **Initialization (`jspInit()`):** `[PYQ]`
   - User-defined: `<%! public void jspInit() { /* one-time setup */ } %>`.
   - Called ONCE by container after instantiation. Analogous to Servlet `init()`.
6. **Request Processing (`_jspService()`):** `[PYQ]`
   - Auto-generated by container from JSP content. NOT user-written.
   - Called for EVERY request. Analogous to Servlet `service()` / `doGet()` / `doPost()`.
   - Processes request, executes scriptlets/actions, outputs HTML. Receives implicit `request`, `response`.
7. **Destruction (`jspDestroy()`):** `[PYQ]`
   - User-defined: `<%! public void jspDestroy() { /* cleanup */ } %>`.
   - Called ONCE by container before servlet instance destroyed (app undeploy/shutdown). Analogous to Servlet `destroy()`.

- Arrow Text Diagram: `[PYQ]`
  `Request -> (Translate*.java -> Compile*.class -> Load -> Instantiate -> jspInit())* -> _jspService() -> Response --(Shutdown)--> jspDestroy()`
  `(* only on first request/change)`

Part 4: JSP Scripting Elements (Putting Java in JSP)

- **Warning:** Old style; EL & JSTL preferred for clean separation.

1. **Scriptlets:** `<% ... Java code ... %>` `[PYQ]`
   - Embeds Java blocks (statements, loops, if/else).
   - Executed every request. Code goes into `_jspService()`.
   - Access implicit objects. Ex: `<% if(user!=null) out.print(user); %>`
2. **Expressions:** `<%= ... Java expression ... %>` `[PYQ]`
   - Prints result of a single Java expression to HTML.
   - Evaluated every request. Auto-converted to String, written to `out`.
   - NO semicolon inside. Ex: `<p>Time: <%= new java.util.Date() %></p>`

3. **Declarations:** `<%! ... Java declarations ... %>` `[PYQ]`
    - Declares member variables/methods for generated Servlet class (outside `_jspService()`).
    - Code executed at Servlet init (for methods) or available as fields.
    - Shared among all requests (thread-safety concern for vars).
    - Used for `jspInit()`, `jspDestroy()`. Ex: `<%! private int count=0; %>`
4. **JSP Comments:** `<%-- ... comment ... --%>`
    - For developers; NOT in final HTML or generated Servlet. Ignored by container.

Part 5: Implicit Objects (Built-in Variables in JSP) `[PYQ]`
Predefined, auto-available Java objects in JSP scripting.

1. **`request`**: (`HttpServletRequest`) Client's HTTP request data. Scope: request. `[PYQ]`
    - Syntax: `request.getParameter("name")`
2. **`response`**: (`HttpServletResponse`) Server's HTTP response. Scope: request. `[PYQ]`
    - Syntax: `response.setContentType("text/html")`, `response.sendRedirect("url")`
3. **`out`**: (`JspWriter`) Writes char data to response stream. Scope: page. `[PYQ]`
    - Syntax: `out.println("text")`
4. **`session`**: (`HttpSession`) User's session on server; stores user-specific data across requests. Scope: session. Default: `session="true"` page directive. `[PYQ]`
    - Syntax: `session.setAttribute("key", value)`, `session.getAttribute("key")`
5. **`application`**: (`ServletContext`) Entire web app; shared data for all users. Scope: application. `[PYQ]`
    - Syntax: `application.setAttribute("key", value)`, `application.getRealPath("/")`
6. **`pageContext`**: (`PageContext`) "God object"; access to all scopes & other implicit objects. Scope: page.
    - Syntax: `pageContext.findAttribute("key")`
7. **`config`**: (`ServletConfig`) Servlet config object for generated Servlet. Scope: page.
    - Syntax: `config.getInitParameter("paramName")`
8. **`page`**: (`Object`, `this` in Servlet) The generated Servlet instance itself. Scope: page.
    - Syntax: `((HttpServlet)page).getServletName()`
9. **`exception`**: (`Throwable`) Uncaught exception. Only in error pages (`isErrorPage="true"`). Scope: page.
    - Syntax: `exception.getMessage()`

Part 6: JSP Directives (Instructions for JSP Engine) `[PYQ]`
Tags `<%@ ... %>`; instruct container on translation/execution. Processed at translation.

1. **`page` Directive:** `<%@ page attribute="value" ... %>` (Most common, page-specific). `[PYQ]`
    - `import="java.util.*, java.text.*"`: Makes Java classes available.
    - `language="java"`: Default.
    - `contentType="text/html; charset=UTF-8"`: Sets response Content-Type header. **Very Important!**
    - `pageEncoding="UTF-8"`: Encoding of `.jsp` file itself. Match with `contentType`.
    - `isErrorPage="true|false"`: If true, `exception` object available. Default: false.
    - `errorPage="url"`: URL to forward to if uncaught Throwable.
    - `session="true|false"`: If true, `session` object available. Default: true.
    - `buffer="none|sizekb"`: Output buffer size for `out`.
    - `autoFlush="true|false"`: If true, buffer flushes when full.
2. **`include` Directive:** `<%@ include file="path/to/resource" %>` `[PYQ]`
    - Includes content of another resource (HTML, JSP, text) AT TRANSLATION TIME (static include).
    - Content literally copy-pasted into JSP source before servlet compilation.
    - Path relative to current JSP. Good for static headers/footers.
3. **`taglib` Directive:** `<%@ taglib uri="taglibURI" prefix="prefix" %>` `[PYQ]`
    - Declares use of a Custom Tag Library (e.g., JSTL).
    - `uri`: Unique ID for TLD (Tag Library Descriptor) file.
    - `prefix`: Short nickname for tags from this library (e.g., `c` for JSTL core).

Part 7: JSP Standard Actions (XML-like Tags for Common Tasks) `[PYQ]`
Predefined XML-like tags (`<jsp:... />`), processed at REQUEST TIME. Cleaner than scriptlets.

1. **`<jsp:useBean id="beanName" class="pkg.BeanClass" scope="request|page|session|application" />`**: `[PYQ]`
    - Locates/creates a JavaBean instance. `id`: scripting var name. `class`: FQCN. `scope`: where to find/store.
2. **`<jsp:setProperty name="beanName" property="propName" value="val" />`** or `param="reqParam"` or `property="*"`: `[PYQ]`
    - Sets bean property. `name` matches `id` from `useBean`. `property` is bean field. `value` is literal. `param` uses request parameter. `*` matches all request params to bean props.
3. **`<jsp:getProperty name="beanName" property="propName" />`**: `[PYQ]`
    - Gets bean property value and prints to output.
4. **`<jsp:include page="url" flush="true|false" />`**: `[PYQ]`
    - Includes output of another resource (JSP, Servlet, HTML) AT REQUEST TIME (dynamic include).

- `page`: URL of resource. `flush`: if true, flushes buffer before include.
- Difference vs. `@include` directive: dynamic, can pass params.

5. **`<jsp:forward page="url" />`**: `[PYQ]`
   - Terminates current JSP, forwards request & response to another resource on server. Browser URL doesn't change.
6. **`<jsp:param name="paramName" value="paramValue" />`**: `[PYQ]`
   - Used *within* `<jsp:include>` or `<jsp:forward>` to add/modify request parameters.
7. **`<jsp:plugin>`**: Embeds applet/plugin (largely deprecated).

Part 8: Custom Tag Libraries (Making JSP Cleaner!) `[PYQ]`

- **Problem:** Scriptlets for complex logic are messy.
- **Solution:** Create reusable XML-like tags via Custom Tag Libraries.
- **Why:** Cleaner JSPs, Code Reusability, Better Separation of Concerns, Abstraction.
- **Key Concepts:**
  - **Tag Library:** Collection of custom tags.
  - **Tag Library Descriptor (TLD):** XML file (`.tld`) in `WEB-INF/tlds` or JAR; defines tags, attributes, maps to Tag Handler Class.
  - **Tag Handler Class:** Java class (implements `javax.servlet.jsp.tagext.SimpleTag` or `Tag`, etc.) containing logic for the tag. `doTag()` method.
- **Using Custom Tag:**
  1. Add JAR (if external, e.g., JSTL) to `WEB-INF/lib`.
  2. Declare: `<%@ taglib uri="tldUri" prefix="pfx" %>` in JSP.
  3. Use: `<pfx:tagName attr="val" />`.
- **JSTL (JSP Standard Tag Library):** `[PYQ]` Most important custom tag library. Modern best practice with EL.
  - **How to Use:** Add JSTL JARs. Declare with `<%@ taglib ... %>`. Use tags.
  - **Expression Language (EL): `${...}`**: `[PYQ]` Standard part of JSP (2.0+). Used with JSTL to access data without scriptlets. Simpler, safer. Accesses bean props (`${myBean.prop}`), map/list elements, implicit objects (`${param.name}`, `${sessionScope.user}`). Handles nulls gracefully.
  - **Common JSTL Libraries (Prefixes conventional):**
    - **Core (`c`):** `<c:set var="x" value="${y}"/>`, `<c:if test="${cond}">...</c:if>`, `<c:forEach items="${list}" var="i">...</c:forEach>`, `<c:out value="${val}" escapeXml="true"/>` (XSS prevention!), `<c:url value="/path"/>`. `[PYQ]`
    - **Formatting (`fmt`):** `<fmt:formatNumber value="${num}" type="currency"/>`, `<fmt:formatDate value="${date}" pattern="yyyy-MM-dd"/>`.

- **SQL (`sql`):** Avoid! Direct DB access from JSP is bad practice.
- **XML (`x`):** For XML processing.
- **Functions (`fn`):** `${fn:length(str)}`, `${fn:toUpperCase(str)}`.

JAVA UNIT 4 NOTES - ULTRA CONCISE

1. The Roles of Client and Server `[PYQ]`

- **Client-Server Model:** Fundamental distributed computing pattern. Programs on different computers interact over a network.
- **Client:** Initiates requests for services/data. Interacts with user. Connects to server. (Ex: Web browser).
- **Server:** Listens for and responds to client requests. Runs continuously. Serves multiple clients concurrently. Manages shared resources (DB, app logic).
- **Interaction:** Via network using protocols (HTTP, RMI custom protocols).
- **Characteristics:** Request-response; Common protocol; Server handles limited concurrent requests (priority/queuing); Vulnerable to DoS.
- **Advantages:** Data centralization (mgmt, security); Efficient access; Easy client/server updates.
- **Disadvantages:** Server overload; Single point of failure; High setup/maintenance cost.
- **vs. P2P:** P2P: nodes equal, share directly. Client-Server: central server.

2. Remote Method Invocation (RMI) - What is it? `[PYQ]`

- **RMI:** Java's native API for creating distributed Java apps. Object in one JVM invokes methods on object in another JVM (potentially different machine).
- **Goal:** Make remote calls look like local calls, hiding network complexity.
- **Why Important:** Distribute app logic; Multi-tiered enterprise apps.
- **Key Concepts:**
  - **Distributed Computing Model:** RMI is Java's object-to-object communication in this model.
  - **Remote Object:** Object whose methods can be invoked from a different JVM. `[PYQ]`
  - **Remote Interface:** Java `interface` extending `java.rmi.Remote` (marker). Defines methods callable remotely. All methods `throw java.rmi.RemoteException`. `[PYQ]`
  - **Remote Object Implementation:** Class implementing remote interface. Extends `java.rmi.server.UnicastRemoteObject` (or use `exportObject()`). Constructor `throws RemoteException`. `[PYQ]`
  - **Stub (Client-side Proxy):** Resides in Client's JVM. Implements remote interface. Client calls methods on Stub.
    - Stub: Connects to server, marshals params, sends request, waits, unmarshals result/exception, returns to client. `[PYQ]`

- **Skeleton (Server-side Helper - largely eliminated in Java 2+ SDK):** Resided on server. Received request from Stub, unmarshalled params, invoked method on real object, marshalled result, sent response to Stub. RMI framework now handles this server-side. `[PYQ]`
- **RMI Registry (`java.rmi.registry.Registry`):** Naming service ("phonebook"). Default port 1099. `[PYQ]`
  - Server: Registers (binds) remote object instance with Registry (`Naming.rebind("rmi://host:port/service", obj)`). `[PYQ]`
  - Client: Looks up remote object in Registry (`Naming.lookup("rmi://host:port/service")`) to get Stub. `[PYQ]`

3. Setup for Remote Method Invocation (RMI Example Steps) `[PYQ]`

1. **Define Remote Interface:** `interface MyService extends Remote { String getData() throws RemoteException; }`
2. **Implement Remote Interface:** `class MyServiceImpl extends UnicastRemoteObject implements MyService { ... }`
3. **Compile & Create Stub/Skeleton (Older RMI):** `javac *.java`; (Older: `rmic MyServiceImpl`). Modern RMI often handles stubs dynamically.
4. **Start RMI Registry:** `rmiregistry` (cmd) or `LocateRegistry.createRegistry(1099);` (programmatic).
5. **Write & Start Server App:** Create impl instance, `Naming.rebind(...)`.
6. **Write & Start Client App:** `Naming.lookup(...)`, cast to interface, call methods.

- **Example (Calculator RMI):** `[PYQ]`
  - Interface: `Calculator extends Remote { int add(int a,int b) throws RE; }`
  - Impl: `CalculatorImpl extends URO implements Calculator { public int add(...) {return a+b;} }`
  - Server: `Calculator c = new CalculatorImpl(); Naming.rebind("CalcService", c);`
  - Client: `Calculator stub = (Calculator)Naming.lookup("CalcService"); stub.add(5,3);`

4. Parameter Passing in Remote Methods `[PYQ]`

- **What:** How arguments (client->server) and return values (server->client) are transferred.
- **Serializable Objects:** Arguments/return values must be serializable (primitives, or implement `java.io.Serializable`).
  - **Serialization:** Object state to byte stream.
  - **Deserialization:** Byte stream to object.
- **Pass By Value (Non-Remote Objects):** `[PYQ]`
  - For primitives and serializable objects (not themselves Remote Objects).

    - Object serialized, sent, new distinct copy deserialized on receiving side. Changes on receiving side copy DON'T affect original. (Like photocopy).
- **Pass By Reference (Remote Objects):** `[PYQ]`
    - For objects that are themselves Remote Objects (implement Remote interface, exported).
    - RMI sends a Stub (reference to original remote object) over network.
    - Receiving side gets Stub. Method calls on received Stub execute on the single, original remote object instance. (Like remote control).
- **Exception Handling:** Remote methods `throw RemoteException`. Clients must handle. `[PYQ]`
- **Class Loading:** If class of arg/return value not local, JVM may try dynamic loading from sender's codebase (via RMIClassLoader).

5. Introduction of Hibernate (HB) / Object-Relational Mapping (ORM) `[PYQ]`

- **Persistence:** Object state survives process that created it (e.g., saving to DB).
- **Database Interaction (Traditional JDBC):** Manual SQL, mapping object fields to table columns (INSERT/UPDATE), rows to fields (SELECT), handling relationships.
- **Object-Relational Impedance Mismatch:** Differences between OO structure (objects, inheritance) and Relational DB structure (tables, rows, flat data).
- **ORM (Object-Relational Mapping):** Programming technique mapping OO objects to relational DB data. `[PYQ]`
- **Hibernate (HB):** Popular, open-source, lightweight Java ORM framework. Implements JPA. `[PYQ]`
- **Why Hibernate Important:**
    - **Simplifies Data Persistence:** Automates object-DB mapping, less manual JDBC/SQL. "Less Code". `[PYQ]`
    - **Increased Productivity:** Focus on business logic, less on DB plumbing.
    - **Improved Maintainability:** Cleaner, less error-prone code.
    - **Database Independence:** Switch DBs with config changes; HB handles SQL dialects (HQL/JPQL). `[PYQ]`
- **Key Concepts (Hibernate/JPA):**
    - **Entities (Persistent Objects):** POJOs mapped to DB tables (e.g., via `@Entity`). `[PYQ]`
    - **Mapping:** Config (annotations or XML `.hbm.xml`) telling HB how entities/fields map to tables/columns/relationships. `[PYQ]`
    - **JPA (Java Persistence API):** Standard Java spec for ORM. Hibernate is a JPA provider. `javax.persistence` / `jakarta.persistence` package. `[PYQ]`
- **Advantages of Hibernate:** Open Source, Lightweight, Fast Performance (caching: 1st/2nd level), DB Independent Queries (HQL/JPQL), Automatic Table Creation, Simplifies Complex Joins. `[PYQ]`

6. Hibernate Architecture `[PYQ]`

- **What:** Main components (objects, config files) and their interaction.
- **Key Components & Objects:**
  1. **Configuration (`org.hibernate.cfg.Configuration` / `persistence.xml`):** `[PYQ]`
     - Settings for DB connection (driver, URL, user, pass), dialect, mappings, caching, etc. Loaded once at startup.
  2. **SessionFactory (`org.hibernate.SessionFactory` / `EntityManagerFactory`):** `[PYQ]`
     - Factory for `Session` objects. Thread-safe. Created once per app (expensive). Holds mappings, manages 2nd-Level Cache.
  3. **Session (`org.hibernate.Session` / `EntityManager`):** `[PYQ]`
     - Single-threaded unit of work/conversation with DB. NOT thread-safe. Short-lived.
     - Primary API for CRUD ops, queries. Manages 1st-Level Cache (session-specific).
  4. **Transaction (`org.hibernate.Transaction` / `EntityTransaction`):** `[PYQ]`
     - Atomic unit of work. From `Session`. `commit()` or `rollback()`.
  5. **Persistent Objects (Entities):** `[PYQ]` Mapped POJOs.
  6. **ConnectionProvider:** `[PYQ]` Factory for JDBC connections (often a pool). Abstracts connection source.
  7. **Query (HQL, Criteria, Native SQL):** `[PYQ]` Objects for data retrieval/manipulation.
- **Hibernate Layers (Conceptual):** `[PYQ]`
  1. **Java Application Layer:** Uses Hibernate APIs.
  2. **Hibernate Framework Layer:** Core engine (SessionFactory, Session, caches, etc.).
  3. **Backend API Layer:** Uses JDBC, JTA (opt), JNDI (opt).
  4. **Database Layer:** Actual RDBMS.
- **Basic Workflow (Simplified):** `[PYQ]`
  Load Config -> Create SessionFactory (once) -> Open Session -> Begin Transaction -> Execute Logic (save/load/query via Session) -> Commit/Rollback Transaction -> Close Session.
- **Arrow Text Diagram:** `[PYQ]`
  `AppCode -> Config -> SessionFactory -> Session -> Transaction -> (CRUD/Query) -> JDBC -> DB`
