HTML5 Learning Guide - Overview

---

1. Introduction

Overview and Prerequisites:

Introduces what HTML5 is about and what knowledge (basic HTML, CSS, JS) is required.

What Is HTML5?:

Defines HTML5 as the latest version of HTML, designed for better structure, media handling, and APIs.

History of HTML:

Covers the evolution from HTML to HTML4 and now HTML5.

HTML5 Platform:

Describes HTML5 as a platform (not just a markup language) supporting apps, media, communication, and more.

Standards Bodies:

W3C and WHATWG are the organizations standardizing HTML5.

2. What, Why, and When

What: Structural Elements Part 1:

Introduction to new structural tags like <header>, <footer>, <section>, etc.

Structural Elements Part 2:

Deeper dive into semantic layout elements and their usage.

Elements with APIs:

HTML elements that come bundled with APIs for richer functionality (like <canvas>, <audio>, <video>).

Form Elements:

New input types (email, date, range), validation features, and attributes.

New APIs - Graphics and Typography:

New capabilities like Canvas, SVG, Web Fonts.

Interaction, Events, and Messaging:

Covers event handling, Web Workers, Cross-document messaging (postMessage).

Storage and Files:

Features like Web Storage (localStorage, sessionStorage) and File APIs.

Real Time Communications:

WebRTC, WebSockets for real-time communication (audio, video, data).

Web Components:

Reusable custom elements, Shadow DOM, Templates, HTML Imports.

Performance Analysis:

Tools and APIs like Navigation Timing, Resource Timing for performance monitoring.

Security and Privacy:

Covers HTTPS, CORS, sandboxing, and related security measures.

Miscellaneous APIs:

Other useful APIs like Battery Status, Vibration, and Gamepad API.

Why:

Explains the need for HTML5 (mobile support, offline apps, media without plugins).

When:

Timeline for HTML5 adoption and when it is suitable for use.

---

3. HTML4 vs HTML5

A side-by-side comparison showing key differences, advantages of HTML5 over HTML4.

---

4. Getting Started with HTML5 Pages

Detecting Features:

Using Modernizr or native JavaScript to detect browser feature support.

Finding Parts of the Page:

How to select and manipulate DOM elements.

---

5. Working with User Input

(Chapters 48–59)

Sample Setup:

Basic HTML page setup to work with inputs.

getElementsByClassName:

Selecting multiple elements by class name.

querySelector and querySelectorAll:

Selecting elements using CSS selectors.

querySelectorAll Returns a NodeList:

NodeList object basics and handling.

Iterating NodeList - for loop:

Looping through NodeList using traditional for loop.

Iterating NodeList - forEach loop:

Using forEach on NodeList for simpler iteration.

Live Result from getElementsByClassName:

Dynamically updated results when DOM changes.

Static Result from querySelectorAll:

Fixed results unaffected by later DOM changes.

Summary:

Wrap-up of working with user input elements using different selection methods.

---

6. New Elements Overview

New Elements Overview (continued):

More new HTML5 elements and their purposes.

Validation Rules Overview:

Native form validation features added in HTML5.

New Elements: Demo:

Demonstration of how new elements work.

New Elements: Markup:

Sample code for new HTML5 elements.

New Elements in Different Browsers:

Browser support status for different new elements.

Building Interactivity with CSS Pseudo Classes:

Using pseudo-classes like :valid, :invalid, :checked, etc.

Custom Validation Messages: Demo and Markup:

How to customize error messages for forms.

Custom Validation Messages: JavaScript:

Using JS to set custom validation messages dynamically.

Custom Validation Rules: Demo, Markup, and JavaScript:

Define and enforce your own validation rules beyond HTML5 defaults.

Bootstrap Module: Demo and Markup:

Using Bootstrap framework to style forms and inputs.

---

7. Music & Video (Without Plugins)

Audio Formats:

Supported formats (MP3, OGG, WAV) and browser compatibility.

Video Formats:

Supported video formats (MP4, WebM, OGG) and fallbacks.

Basic Controls:

Using simple <audio> and <video> elements with controls attribute.

Basic Controls (continued):

Further customization of basic media controls.

Scripted Controls:

Using JavaScript to create custom play, pause, volume buttons.

Dynamic Content:

Loading or changing media sources dynamically.

Summary:

Review of handling multimedia in HTML5 without Flash or plugins.

---

8. Drawing Shapes, Charts, and More

Introduction:

Overview of Canvas API and what it enables.

What is Canvas?:

<canvas> element and drawing with JavaScript.

Drawing Basics Part One & Two:

Drawing simple shapes like lines and circles.

Drawing Rectangles:

Filling and stroking rectangles.

Order of Operations:

How drawing commands are executed sequentially.

Arcs, Text, and Gradients:

Drawing arcs, adding text, and using gradients for colors.

Effects - Scale and Rotation:

Scaling and rotating shapes on canvas.

Effects - Translate:

Moving (translating) the origin point for easier drawing.

State Management:

Saving and restoring drawing states (like color, transformations).

Animation Basics:

Moving shapes smoothly across the canvas using JavaScript loops.

Clipping Introduction and Techniques:

Restricting drawing inside a particular region.

Demo: Magnifying Glass:

Example where canvas is used to magnify part of an image.

Demo: Static Chart:

Drawing non-changing charts with canvas.

Demo: Dynamic Chart:

Updating the chart based on user input or live data.

Demo: Video Thumbnails:

Grabbing video frames onto a canvas.

Summary:

Overall usage and possibilities with canvas in HTML5.

---

9. Drag and Drop

Introduction:

Drag-and-drop support in HTML5 for elements.

Events Overview:

Explains events like dragstart, dragover, drop.

Events in Detail:

Detailed behavior and sequence of drag-and-drop events.

Basics - Demo and Markup:

Basic drag-and-drop functionality with minimal HTML.

Basics - JavaScript (Parts 1 and 2):

Implementing drag-and-drop logic using JavaScript.

Role Selectors: Demo and Code:

Using role attributes or selectors to define draggable elements.

Events - Demo and Markup:

More examples of drag events and how to set up page elements.

Events - JavaScript:

Writing JavaScript to manage more complex drag-and-drop operations.

Effects - Demo and Markup:

Visual feedback and effects when dragging elements.

Effects - JavaScript:

JavaScript to control the visual effects and animations during drag actions.

Summary

You will be learning:

The evolution of HTML to HTML5

Core structural, media, and form enhancements

JavaScript APIs introduced by HTML5

Real-time communication techniques

Performance, security, and advanced web features

Drawing, media control, and drag-and-drop interactivity

Validation, custom form control, and responsive designs

HTML5 Mind Map

1. Introduction

→ Overview and Prerequisites

→ What is HTML5

→ History

→ HTML5 Platform

→ Standards Bodies

---

2. What, Why, and When

What:

→ Structural Elements (Part 1 & 2)

→ Elements with APIs

→ Form Elements

→ New APIs: Graphics, Typography

→ Interaction, Events, Messaging

→ Storage and Files

→ Real Time Communication (WebRTC, WebSockets)

→ Web Components

→ Performance Analysis

→ Security and Privacy

→ Miscellaneous APIs

Why:

→ Need for HTML5 (mobile, offline, richer apps)

When:

→ Timeline of adoption

---

3. HTML4 vs HTML5

→ Key differences

---

4. Getting Started

→ Detecting Features (Modernizr, JS)

→ Finding Parts of Page (DOM selection)

---

5. Working with User Input

→ getElementsByClassName

→ querySelector/querySelectorAll

→ Iterating NodeList

→ Live vs Static NodeList

→ Summary

---

6. New Elements and Validation

→ New Elements Overview

→ Validation Rules Overview

→ Custom Validation Messages

→ Custom Validation Rules

→ Bootstrap Module Integration

---

7. Media (Music & Video)

→ Audio Formats

→ Video Formats

→ Basic & Scripted Controls

→ Dynamic Content

→ Summary

---

8. Drawing and Graphics (Canvas)

→ Canvas Introduction

→ Drawing Basics (Shapes, Text, Gradients)

→ Transformations (Scale, Rotate, Translate)

→ Animation Basics

→ Clipping Techniques

→ Demos: Magnifying Glass, Charts, Video Thumbnails

---

9. Drag and Drop

→ Events Overview (dragstart, dragover, drop)

→ JavaScript Implementation

→ Role Selectors

→ Visual Effects (during drag)

XML

1. Introduction to XML

What Is XML?

Extensible Markup Language (XML) is a flexible, text-based format for representing structured data, derived from SGML and widely used for data exchange across systems .

Advantages of XML

XML is platform- and language-independent, human- and machine-readable, supports hierarchical data, and separates data from presentation .

HTML vs. XML

Unlike HTML's fixed tag set for page layout, XML allows you to define your own tags suited to your data, making XML ideal for data transport rather than presentation .

---

2. Building Blocks of an XML Document

XML Elements

Elements are the primary containers in XML, marked by start/end tags, and can nest other elements or text .

Attributes

Attributes provide metadata for elements as name–value pairs inside the start tag and must be quoted .

Entities

Entities are placeholders for special characters or strings, defined in DTD or internal subsets, enabling reuse and escaping .

PCDATA and CDATA

– PCDATA (Parsed Character Data) is text that the parser examines for markup.

– CDATA sections tell the parser to treat enclosed text as raw data, not markup .

Lend a Hand: XML Explaining the Building Blocks

Practical exercise: Annotate a sample XML file to identify elements, attributes, entities, PCDATA, and CDATA.

---

3. DTD (Document Type Definition)

What Is a DTD?

A DTD defines the legal structure, elements, and attributes of an XML document, serving as its schema .

How to Declare a DTD

– Internal DTD: placed within the XML <!DOCTYPE> declaration.

– External DTD: referenced by a system or public identifier in the <!DOCTYPE> declaration .

Why Do We Need a DTD?

DTD validation ensures that XML documents conform to a predefined structure, catching errors early and enabling robust data exchange .

Learning DTD Using a Sample XML

Practical exercise: Create a sample XML (e.g., <bookstore>) and write a DTD defining its elements and attributes.

DTD Element and Attribute Declaration

Use <!ELEMENT> to define element content models and <!ATTLIST> for attribute lists within the DTD .

Complete DTD for the Sample XML

Assemble all element and attribute declarations into a coherent DTD file and test validation.

Elements vs. Attributes

Discuss design choices:

Use elements for data that can repeat or contain complex content.

Use attributes for metadata or small simple values.

Features of Well-Formed vs. Valid XML

Well-formed: meets basic XML syntax rules (one root, proper nesting).

Valid: also conforms to its DTD or schema .

Lend a Hand: Create a DTD

Practical exercise: Develop and validate a DTD for a given XML use case.

---

4. XML Schema (XSD) and Namespaces

What Is a Schema?

XML Schema Definition (XSD) precisely describes XML document structure, data types, and namespaces, offering richer typing than DTD .

What Is a Namespace?

A namespace is a URI-based mechanism to qualify element and attribute names, avoiding name collisions in XML documents .

Elements of Schema

Core schema components include <xs:element>, <xs:complexType>, <xs:simpleType>, <xs:attribute>, and occurrence indicators (minOccurs, maxOccurs) .

Estimated Time Duration for This Topic

~2–3 hours for fundamentals and hands-on XSD creation.

Schema Element

Defines a global element in the XSD with its type and occurrence.

Simple Element

An element with text-only content, defined via <xs:simpleType> or built-in types.

Complex Element

An element that contains other elements and/or attributes, defined via <xs:complexType>.

Element Attributes

Defined within <xs:complexType> using <xs:attribute>, specifying type and use (optional/required).

Indicators: All, Choice, Sequence, Min & Max

Control element occurrence and order:

<xs:all> (all children, any order),

<xs:choice> (one of),

<xs:sequence> (ordered),

minOccurs/maxOccurs for repetition limits .

Lend a Hand: Creating an XSD

Practical exercise: Write an XSD for the earlier XML sample, using simple and complex types.

XSD vs. DTD

XSD supports namespaces, data typing (integers, dates), and richer constraints compared to DTD’s basic structural rules.

Step-By-Step Approach to Create an XSD

1. Define namespaces and schema root.

2. Declare global elements and types.

3. Add attributes and constraints.

4. Validate XML against the XSD.

Hands-On Exercise: Create an XSD

Practical exercise: Develop and test an XSD for a real-world XML dataset.

---

5. XML Parsers

What Is an XML Parser?

A parser reads XML documents, exposes structure and data to applications, and optionally validates them .

Types of XML Parsers

Tree-based (DOM): builds entire in-memory tree.

Event-based (SAX, StAX): streams through document, triggering callbacks .

What Is a SAX Parser?

Simple API for XML (SAX) is an event-driven parser where applications handle parsing events (start/end element, text) .

Styles

SAX can be used for validation, filtering, or data extraction in a streaming fashion.

Estimated Time Duration for This Topic

~1–2 hours to understand and implement SAX/DOM parsing.

SAX Parser – How It Works

1. Create a SAXParser instance.

2. Implement handler methods (startElement, endElement, characters).

3. Parse and handle events as the document streams .

More About SAX Parsers

SAX is memory-efficient but cannot navigate backwards in the document.

DOM Parser

Document Object Model (DOM) parser loads the whole XML into a tree structure in memory .

DOM Tree

A hierarchical object model with Document, Element, Text, and other node types, allowing random access and modification .

DOM vs. SAX

DOM: random access, easier navigation, higher memory use.

SAX: streaming, low memory, forward-only access.

Steps to Parse XML Using DOM Parser

1. Create DocumentBuilder.

2. Parse XML into Document.

3. Traverse nodes with getElementsByTagName, getAttribute, etc. .

Lend a Hand: Parse XML Using DOM Parser

Practical exercise: Write code to load and query an XML document with a DOM parser.

Hands-On Exercise: DOM Parser Demo

Practical exercise: Build a small Java or JavaScript app that reads an XML file and displays data in a UI.

---

6. WSDL & SOAP

WSDL Document Introduction and Example

WSDL (Web Services Description Language) is an XML vocabulary for describing web services, including their endpoints, messages, and operations .

Notational Conventions

WSDL uses <definitions>, <types>, <message>, <portType>, <binding>, <service> elements with XML namespaces.

Service Definition

Defines operations in <portType> with input/output messages.

Document Structure

Hierarchical WSDL document: <definitions> root → <types> (data schemas) → <message> → <portType> → <binding> → <service>.

Document Naming and Linking

WSDL uses name, targetNamespace, and imports/includes to modularize definitions.

Authoring Style

Best practices for readability, reusability, and namespace organization.

Language Extensibility and Binding

WSDL supports multiple protocol bindings (SOAP, HTTP, MIME) and allows extension via additional attributes.

Documentation

Use <documentation> elements within WSDL to annotate services, operations, and messages.

Types

Define data types for messages using embedded XML Schema within <types> .

Messages

Logical definitions of data communicated: <message> contains one or more <part>.

Message Parts

Each <part> references an element or type defining the message payload.

Abstract vs. Concrete Messages

– Abstract: defined in <message> and <portType>.

– Concrete: binding to protocol in <binding> and <service>.

Port Types

Interface definitions grouping operations (e.g., <portType name="MyServicePortType">).

Operation Styles

One-way: only request.

Request-response: standard call and return.

Solicit-response: service initiates response.

Notification: one-way from service to client.

Names of Elements Within an Operation

Use name attributes on <operation>, <input>, and <output>.

Parameter Order Within an Operation

Order parts in <message> to control serialization.

Bindings

Concrete protocol details in <binding>, referencing a <portType> and specifying transport (SOAP, HTTP).

Ports

Specify an address for a binding in <service> using <port>.

Services

Container for a set of related ports, exposing endpoints to clients.

SOAP Binding

Defines SOAP protocol usage in WSDL (transport, style, use — literal/encoded) .

SOAP Examples

Example <binding> and <port> definitions showing SOAP envelope and endpoint.

How the SOAP Binding Extends WSDL

SOAP adds specific <soap:binding>, <soap:operation>, <soap:body> elements to standard WSDL to enable message formatting and transmission.

Javascript and NODE JS

1. Overview of JavaScript

Definition: JavaScript is a high-level, interpreted scripting language used to create dynamic website behavior—everything from updating content without reloads to animating graphics
.

Example:

```
<script>
  document.body.innerHTML = '<h1>Hello, JavaScript!</h1>';
</script>
```

---

2. Advantages and Limitations

Advantages

Ease of Learning: Simple syntax and quick entry point for beginners .

Client-Side Execution: Runs in the browser, reducing server load and latency .

Rich Ecosystem: Vast libraries and frameworks (React, Vue, Angular) .

Limitations

Security Risks: Client code is visible and susceptible to XSS attacks .

Browser Compatibility: Some features vary across browsers .

Performance Constraints: Single-threaded nature can hinder CPU-intensive tasks .

---

3. Development Tools

Browser DevTools: Built-in inspectors for HTML, CSS, JS debugging and performance profiling .

Code Editors/IDEs: VS Code, WebStorm, Sublime Text with syntax highlighting and linting .

Linters & Formatters: ESLint, Prettier for code quality and consistency.

---

4. Estimated Time Duration for This Section

~1–2 hours

---

5. JavaScript Syntax and Code Structure

Basic Syntax

Statements: End with semicolons (optional in many cases) .

Blocks: Use {...} for scoping.

Comments: // single-line, /* multi-line */.

Example

```
function greet(name) {
  console.log('Hello, ' + name);
}
greet('World'); // Hello, World
```

---

6. Enabling JavaScript in Different Browsers

Most browsers enable JS by default, but users can disable it via settings. Always check with feature-detection (not user-agent sniffing) .

---

7. Estimated Time Duration

~15 minutes

---

8. JavaScript in Header and Body Tags

You can place <script> in <head> (executes before DOM ready) or at end of <body> (executes after HTML loads) .

Example (body):

```
<body>
 <h1 id="demo">Hi</h1>
 <script>
  document.getElementById('demo').innerText = 'Hello after load!';
 </script>
</body>
```

---

9. Accessing JavaScript from External File

Example:

```
<script src="app.js"></script>
```

Contents of app.js:

```
console.log('Loaded from external file');
```

External files enable reuse across pages .

---

10. Estimated Time Duration

~10 minutes

---

11. Data Types

JS supports:

Primitive: Number, String, Boolean, Null, Undefined, Symbol, BigInt .

Objects: Dictionaries, Arrays, Functions, Dates, RegExps.

---

12. Variables

var: Function-scoped or global .

let/const: Block-scoped; const is read-only after initialization .

---

13. Variable Scope

Global: Declared outside functions.

Local: Inside functions or blocks (let/const).

Hoisting: var declarations move to top of scope, but not their initializations.

---

14. Variable Names and Reserved Words

Must start with letter, _, or $.

Cannot use reserved words: if, for, class, etc. .

---

15. Estimated Time Duration for This Section

~20 minutes

---

16. Operators

Arithmetic

+, -, *, /, %, ** (exponent) .

Comparison

==, ===, !=, !==, >, <, >=, <= .

Logical

&&, ||, ! .

Bitwise

&, |, ^, ~, <<, >> .

Assignment

=, +=, -= ... etc. .

Miscellaneous

Ternary: condition ? a : b, typeof, instanceof, void, delete.

---

17. Estimated Time Duration

~20 minutes

---

18. Control Flow and Loops

If-Else

if (x > 0) { ... } else if (x < 0) { ... } else { ... }

` ` ` 22.

Switch...Case

```js
switch(day) {
  case 'Mon': ...; break;
  default: ...;
}
```

Loops

- **`while` / `do...while`** 24
- **`for`** 25
- **`for...in` / `for...of`** 26

Loop Controls

`break`, `continue` to alter loop flow 27.

---

19. Estimated Time Duration

~30 minutes

---

20. Hands-On Exercise (Topics 1–6)

Build: A simple webpage that uses external JS to manipulate content (e.g., change text, style elements, and log messages).

---

21. Estimated Time Duration

~1 hour

---

22. Functions

Definition & Calls

```js
function sum(a, b) {
  return a + b;
}
let total = sum(2, 3); // 5
```

28.

Parameters & Return

- **Parameters:** Inputs listed in parentheses.
- **Return:** Sends a value back to caller.

Nested Functions

Functions inside other functions with closure access.

Function Constructor & Literals

- **Constructor:** `new Function('x','y','return x+y');`

- **Literal:** `() => x + y` (arrow functions) 29.

---

23. Estimated Time Duration

~30 minutes

---

24. Hands-On Exercise (Functions)

Build: Calculator functions for add/subtract/multiply/divide, using both traditional and arrow syntax.

---

25. Estimated Time Duration

~45 minutes

---

26. Events

What Are Events?

Notifications (click, keypress, load) that let code react to user or browser actions 30.

Different Event Types

- **User Interaction:** `click`, `mouseover`, `keydown`.

- **Form Events:** `submit`, `change`.

- **Window Events:** `load`, `resize`.

Standard HTML5 Events

Cover all modern events, including drag-and-drop, media events, and custom events 31.

---

27. Estimated Time Duration

~30 minutes

---

28. Hands-On Exercise (Events)

Build: A to-do list where clicking items toggles completion, and form submission adds new items.

---

29. Estimated Time Duration

~1 hour

---

30. Cookies & Page Control

Cookies

- **Access:** `document.cookie` to read/write cookies 32.

- **Set Expiry:** `expires=Date.toUTCString()`.

- **Delete:** Set past expiry date.

Page Redirect & Refresh

- **Redirect:** `window.location.href = 'url';`

- **Refresh:** `location.reload();`.

---

31. Lend a Hand: Practice Cookies

Build: A simple login form that saves username in a cookie and greets user on reload.

---

32. Estimated Time Duration

~45 minutes

---

33. Dialogs

Alert

```js
```

alert('Hello!');

Confirmation

if (confirm('Proceed?')) { ... }

```
33.

### Prompt

```js
let name = prompt('Enter name:');
```
34.

---

34. Lend a Hand: Practice Dialogs

Build: A quiz that uses prompts for answers, confirms to submit, and alerts to show score.

---

35. Estimated Time Duration

~20 minutes

---

36. Objects & the DOM
```

Objects

- **Properties & Methods:** `{ name: 'A', greet() {...} }`.

- **User-Defined:** Constructor functions or `class` syntax.

Built-Ins

`Number`, `Boolean`, `Date`, `String`, `Array`, `Math`, `RegExp`.

DOM

Document Object Model: interface for querying (`getElementById`), traversing, and manipulating HTML.

---

37. Lend a Hand: Practice Objects & DOM

Build: A mini user directory: add/remove users (objects) and update the HTML table dynamically.

---

38. Error Handling

Error Types

- **Syntax Errors:** Invalid code, caught before runtime.

- **Runtime Errors:** Exceptions during execution.

- **Logical Errors:** Wrong results, no exception.

Handling

```js
try {
  // code
} catch (e) {
  console.error(e);
} finally {
  // cleanup
}
```

throw new Error('msg'); and window.onerror for global catching .

Debugging

Use DevTools consoles, breakpoints, and error panels in Chrome, Firefox, IE.

---

39. Estimated Time Duration

~30 minutes

---

40. Node.js Introduction

Node.js & NPM Setup

Install: from nodejs.org .

NPM: package manager for libraries (npm init, npm install).

Core Modules

File: fs.readFileSync('file.txt').

HTTP: create server http.createServer().

URL: url.parse(req.url) .

Events

EventEmitter pattern for asynchronous handling.

Full-Stack App

Combine HTTP module (backend) with static file serving for front end.

REST WEB SERVICES

You’ll learn what REST is and its architectural principles, common HTTP methods, and how to build RESTful web services with JAX-RS. You’ll install and configure your Java environment (JDK, Eclipse, Jersey, Tomcat) before defining resources, their URIs, and representations. You’ll explore HTTP messages (requests and responses), implement CRUD operations via GET/POST/PUT/DELETE/OPTIONS, and use JAX-RS annotations (@Path, @Consumes, @Produces, etc.) to create, deploy, and test services. Finally, you’ll cover REST’s statelessness, common security concerns (authentication, authorization, CSRF, caching, concurrency), HTTP status codes, and JAX-RS specification annotations.

---

1. What Is REST?

Definition

REST (Representational State Transfer) is an architectural style for distributed hypermedia systems, emphasizing scalability, statelessness, and a uniform interface .

Core Constraints

1. Client–Server: Separation of concerns improves portability and scalability.

2. Stateless: Each request contains all needed information; the server retains no client context between calls .

3. Cacheable: Responses must indicate whether they are cacheable to improve network efficiency.

4. Uniform Interface: Standardized methods (e.g., GET, POST) and resource identification via URIs.

5. Layered System: Intermediary servers (load balancers, proxies) can be inserted.

6. Code on Demand (optional): Servers can extend client functionality by transferring executable code.

REST Architecture

RESTful systems expose resources (nouns) via URIs and manipulate them using HTTP methods (verbs) .

---

2. HTTP Methods

Overview

HTTP defines methods to perform operations on resources identified by URIs .

Example: Fetching Data

GET /api/books/123 HTTP/1.1

Host: example.com

Accept: application/json

Responds with the book's JSON representation .

---

3. Introduction to RESTful Web Service

JAX-RS Overview

JAX-RS is Java's annotation-driven API for creating RESTful services, simplifying development by mapping Java classes and methods to HTTP resources and operations .

Example Skeleton:

```
@Path("/books")
public class BookResource {
  @GET
  @Produces(MediaType.APPLICATION_JSON)
  public List<Book> listBooks() { ... }
}
```

---

4. Environment Setup

4.1 Setup JDK

Download and install the Java SE Development Kit (JDK) from Oracle, choosing the appropriate installer for your OS (Windows, Linux, or macOS) .

4.2 Setup Eclipse IDE

1. Download the Eclipse Installer from eclipse.org.

2. Run the installer, select the “Eclipse IDE for Enterprise Java and Web Developers” package, install, and launch .

4.3 Setup Jersey Framework

Add Jersey (the JAX-RS RI) libraries to your project either via Maven dependencies or by downloading the Jersey bundle from EE4J's site .

4.4 Setup Apache Tomcat

Download Tomcat from the official Apache site and configure it in Eclipse as a new Server runtime; ensure the CATALINA_HOME environment variable points to your Tomcat directory .

Estimated Time Duration: ~45 minutes

---

5. Resources and Representations

5.1 Introduction to Resources

A resource is any information entity (e.g., a book, user, or image) identified by a URI .

5.2 Representation of Resources

i. URIs for Resources

URI Structure: <scheme>://<host>:<port>/<context-root>/<resource-path> .

Standard Query Parameters: ?filter=author&sort=title for filtering and sorting lists.

ii. Entities

Representations (Content Types): JSON, XML, HTML, etc., specified via Content-Type and Accept headers .

Hypertext Linking: Include links (_links in HAL, <link> in XML) within representations for discoverability.

Entity ID: Unique identifier inside the representation (e.g., "id": 123).

Expansion for Entities: Use query parameters (?expand=author) to include related sub-resources.

5.3 Characteristics of Good Representation

Self-Descriptive: Contains enough information to process it (media type, links).

Consistent Structure: Predictable across resources.

Compact: Avoid unnecessary verbosity.

Estimated Time Duration: ~1 hour

---

6. HTTP Messages

6.1 HTTP Request

Composed of a start-line (METHOD URI HTTP/1.1), headers, and an optional body (for POST/PUT) .

6.2 HTTP Response

Includes a status-line (HTTP/1.1 200 OK), headers, and an optional body with the representation .

Estimated Time Duration: ~30 minutes

---

7. HTTP Methods in Detail

Supported Methods

GET: Retrieve resource.

POST: Create sub-resource or submit data.

PUT: Update or create resource at known URI.

DELETE: Remove resource.

OPTIONS: Discover allowed methods.

Example:

PUT /api/books/123 HTTP/1.1

Host: example.com

Content-Type: application/json

{"title":"New Title","author":"Jane Doe"}

Updates book 123 .

Estimated Time Duration: ~20 minutes

---

8. Building a RESTful Service with JAX-RS

8.1 Creating a Root Resource Class

a. Application Configuration

```
@ApplicationPath("/api")
public class RestApp extends Application { }
```

b. @Path Annotation & URI Templates

@Path("/books/{id}") maps a method or class to URIs; {id} denotes a path parameter .

c. Handling HTTP Methods

Use @GET, @POST, @PUT, @DELETE on methods.

```
@GET
@Path("/{id}")
@Produces(MediaType.APPLICATION_JSON)
public Book getBook(@PathParam("id") int id) { ... }
```

d. @Consumes & @Produces

@Consumes: Declares acceptable input formats (application/json).

@Produces: Declares output formats .

e. Extracting Request Parameters

@PathParam, @QueryParam, @HeaderParam, etc., inject values from the URI or headers.

8.2 Build, Deploy, Run, Test

1. Package as a WAR (mvn package).

2. Deploy to Tomcat’s webapps folder or via Eclipse.

3. Test with cURL, Postman, or a browser.

Estimated Time Duration: ~1.5 hours

---

9. Statelessness

Definition

Statelessness requires each client request to contain all context data; the server holds no session state .

Advantages

Scalability: Servers can handle requests independently.

Reliability: No session affinity needed.

Disadvantages

Client Overhead: Clients must manage more data.

Increased Payload: Headers/bodies may carry redundant info.

Estimated Time Duration: ~20 minutes

---

10. Security Concerns

10.1 Authentication & Authorization

Basic Auth / OAuth 2.0 tokens in headers for stateless credential passing .

10.2 Cross-Site Request Forgery (CSRF)

Protect state-changing operations (POST, PUT, DELETE) using anti-CSRF tokens or same-site cookies .

10.3 Caching

Leverage HTTP cache headers (Cache-Control, ETag) to reduce load and latency .

10.4 Concurrency

Use ETags or versioning in payloads to handle concurrent updates.

10.5 HTTP Status Codes

Return appropriate codes (200 OK, 201 Created, 204 No Content, 400 Bad Request, 401 Unauthorized, 404 Not Found, 500 Internal Server Error, etc.) to signal outcomes .

Estimated Time Duration: ~1 hour

---

11. Specifications and Annotations

11.1 Core JAX-RS Annotations

@ApplicationPath

@Path

@GET, @POST, @PUT, @DELETE, @OPTIONS

@Produces, @Consumes

@PathParam, @QueryParam, @HeaderParam, @FormParam

11.2 Hands-On Exercise

Task: Build a Product service with CRUD endpoints, namespace your application at /api, secure write operations with Basic Auth, and demonstrate statelessness and caching headers.

J2EE Architecture

In this section, you will learn what software architecture is and how it applies to J2EE's layered model, the core Java EE libraries, and the role of open source in the Java EE ecosystem. You'll then see how to install and configure a web server (Apache Tomcat) in Eclipse, start/stop it both in the IDE and via the command line, and deploy, start, stop, and access web applications through the browser's Manager interface.

What Is Software Architecture?

Software architecture defines the high-level structures of a software system and the discipline of creating such structures by identifying components, their relationships, and the principles guiding their design and evolution .

J2EE Layered Architecture

J2EE applications follow a multitier (often three-tier) architecture separating concerns into layers :

1. Client Layer – User agents (browsers, Swing clients) interact with the application .

2. Presentation Layer – Servlets/JSPs render dynamic content in the web container .

3. Business Logic Layer – EJBs or POJOs encapsulate transactional, security, and business rules .

4. Integration/Resource Layer – Connectors (JCA), JDBC, JMS provide access to EIS and databases.

This separation improves maintainability, scalability, and testability.

Core Libraries Inside Java EE

Jakarta EE (formerly J2EE) specifies a suite of APIs for enterprise features :

Servlets, JSP, JSF (Web presentation)

EJB, JPA, JTA (Business and persistence)

JAX-RS, JAX-WS (REST/SOAP Web services)

JMS (Messaging), JCA (Connector Architecture)

JavaMail, JNDI, JMX (Mail, naming, management)

Bean Validation, CDI (Contexts & Dependency Injection)

Each library addresses a specific concern, enabling modular and extensible enterprise solutions.

Open Source: Definition and Examples

Open source software makes its source code freely available under licenses permitting modification and redistribution . In the Java EE ecosystem, open-source application servers include:

GlassFish / Payara (the reference implementation)

WildFly / JBoss EAP by Red Hat

Apache Tomcat (Servlet/JSP container)

Eclipse Jetty, Open Liberty, Apache Geronimo

These projects foster community collaboration, rapid bug fixes, and flexible licensing.

Installing a Web Server (Apache Tomcat)

1. Download Tomcat from the Apache site.

2. Extract the archive to a local directory, setting the CATALINA_HOME environment variable to that path .

3. Verify you have JDK installed and JAVA_HOME set appropriately.

Configuring the Web Server in Eclipse

1. In Eclipse, open Window → Preferences → Server → Installed Runtimes.

2. Add an Apache Tomcat runtime by pointing to your CATALINA_HOME directory and selecting the matching Tomcat version .

3. Eclipse will now offer a Servers view to manage Tomcat.

Starting and Stopping the Server in Eclipse

In the Servers view:

Right-click the configured Tomcat server and select Start or Stop.

Console output appears in the Console view, confirming server status .

Deploying, Starting, and Stopping Applications via Browser

Use Tomcat’s Manager App:

1. Navigate to http://localhost:8080/manager/html and log in with a user having the manager-gui role.

2. Deploy a WAR by selecting the file under “Deploy → WAR file to deploy”.

3. Start or Stop any application by clicking the corresponding link in the “Applications” table .

Accessing the Application

Once deployed and started, access your app at

http://localhost:8080/<context-path>/

where <context-path> is either the WAR name or as defined in Context configuration.

Starting and Stopping the Server via Command Window

From the Tomcat bin directory:

Windows:

catalina.bat start

catalina.bat stop

Unix/Linux:

./catalina.sh start

./catalina.sh stop

Or use the tomcat.bat start -f conf/server.xml syntax for alternate configurations .

J2EE Architecture Quiz (25 Questions)

1. Define software architecture.

Answer: High-level structure of a system, defining components and their interactions .

2. Name the three tiers in a typical J2EE application.

Answer: Client, server (presentation & business), and database .

3. Which J2EE API handles transactional, secure business logic?

Answer: Enterprise Java Beans (EJB) .

4. What is the role of the presentation layer?

Answer: Generates user interface via Servlets/JSPs .

5. List two Java EE persistence APIs.

Answer: JPA (Java Persistence API) and JDBC (Java Database Connectivity) .

6. What constraint ensures no server-side session state is stored?

Answer: REST's statelessness; every request must be self-contained.

7. Name two messaging/connectivity APIs in JEE.

Answer: JMS (Java Message Service) and JCA (Java EE Connector Architecture) .

8. What does CDI stand for?

Answer: Contexts and Dependency Injection, for managed beans lifecycle.

9. How do you install Tomcat in Eclipse?

Answer: Add a new runtime under Preferences → Server → Installed Runtimes .

10. Where do you set CATALINA_HOME?

Answer: As an environment variable pointing to the Tomcat installation directory .

11. Which file controls web application context paths?

Answer: conf/server.xml <Context> elements or webapps/<name>.xml.

12. How do you deploy a WAR via Tomcat Manager?

Answer: Use the Manager's "WAR file to deploy" form at /manager/html .

13. What URL accesses the Manager App?

Answer: http://localhost:8080/manager/html.

14. How to start Tomcat from the command line on Linux?

Answer: ./catalina.sh start in the bin directory .

15. Which role allows scripted access to the Manager?

Answer: manager-script role .

16. Name two open-source Java EE servers.

Answer: GlassFish and WildFly .

17. What is the purpose of @ApplicationPath in JAX-RS?

Answer: Defines the base URI for all REST resources in a JAX-RS app.

18. How do Servlets differ from JSPs?

Answer: Servlets are Java classes handling requests; JSPs are HTML pages with embedded Java.

19. Why use JTA?

Answer: To coordinate distributed transactions across multiple resources.

20. What does JNDI provide?

Answer: Java Naming and Directory Interface for resource lookup.

21. How do you stop a single web app via Manager?

Answer: Click its “Stop” link in the Applications table .

22. What’s the default Tomcat HTTP port?

Answer: 8080 (configurable in server.xml).

23. Define a Resource Adapter in JCA.

Answer: A pluggable component enabling EIS connectivity in JEE .

24. What’s the benefit of separating layers?

Answer: Enhances modularity, maintainability, and scalability .

UNIX and PYTHON

A quick summary:

On the Unix side, you’ll learn the OS’s origins, kernel role, file-system layout, common editors and commands, then delve into shell scripting fundamentals (variables, operators, flow control, functions), I/O redirection, job control, regular expressions, signals/traps, and best practices. On the Python side, you’ll cover core programming constructs (data types through modules), exception handling, file I/O, asynchronous programming with asyncio, and Python’s memory management model—all with clear examples.

Unix Operating System

Overview of Unix OS, Kernel, History

Unix was first developed in 1969 at AT&T's Bell Labs by Ken Thompson and Dennis Ritchie as a portable, multi-user, multitasking OS .

Its kernel is the core component managing CPU scheduling, memory, and I/O between hardware and user processes .

Linux is a Unix-like open-source OS kernel created by Linus Torvalds in 1991, extending Unix concepts under the GPL .

File System Basics

Unix organizes files in a single-rooted hierarchy beginning at / .

Directories like /bin, /etc, /home, /var, and /mnt serve standard purposes (executables, config, user homes, variable data, mount points) .

To mount a device manually:

```
mount -t iso9660 /dev/cdrom /mnt/cdrom
```

mounts an ISO9660 CD-ROM at /mnt/cdrom .

Editors

Common Unix editors include:

vi/vim: modal editor with powerful text-object commands .

nano: user-friendly, modeless editor ideal for quick edits .

emacs: extensible, Lisp-based environment for editing and more .

Unix Commands

Fundamental commands:

Navigation: ls, cd, pwd .

File Ops: cp, mv, rm, mkdir .

Text Processing: cat, grep, awk, sed .

Process Management: ps, kill, top .

Network: ssh, scp, ftp, curl .

More Unix Commands

Advanced or less-common tools:

find for locating files:

find /var -name '*.log'

xargs to build argument lists.

ncdu, htop for disk and process visualization .

rsync, tar, zip for backups and archives.

Estimated Time Duration for Unix Basics

~3–4 hours total (OS overview, file system, editors, commands)

Introduction to Shell Scripting

What Is Shell Scripting?

A shell script is a text file of commands interpreted by a Unix shell (bash, sh, csh, ksh) to automate tasks like file manipulation, program execution, and text output .

Shell Variables and Operators

Variables:

NAME="Alice"

echo "Hello, $NAME"

assign and reference simple strings or numbers .

Operators:

Arithmetic: expr $a + $b or ((a + b)) .

Relational: -lt, -gt, -eq in [ ].

Logical: &&, || .

Program Flow Controls and Functions

if...then...else:

```
if [ $age -ge 18 ]; then
  echo "Adult"
else
  echo "Minor"
fi
```

Loops:

```
for file in *.txt; do
  echo "Processing $file"
done
```

Functions:

```
greet() {
  echo "Hello, $1"
}
greet "World"
```

Estimated Time Duration for Shell Scripting Fundamentals

~2–3 hours

I/O Redirection, Job Control, Embedded Scripts

Command Redirection

Output: > overwrite, >> append

Input: < file.txt

Pipes: ls | grep '.log' .

Job Control

Background: append &: sleep 60 & .

jobs: list background/suspended jobs .

fg / bg: bring jobs to foreground or background .

disown/nohup: detach jobs from shell .

Embedded Scripts

You can embed shell code in other scripts or config files using heredocs:

cat <<EOF >my.txt

Line1

Line2

EOF

``` 27.

---

## Regular Expressions, Signals, Traps, Other Useful Commands

### Regular Expressions

Used with `grep`, `sed`, `awk` to match patterns, e.g., `grep '^error' syslog` finds lines starting with “error” 28.

### Signals & Traps

- **Signals**: `SIGINT` (Ctrl+C), `SIGTERM` to terminate processes.

- **trap**: handle signals in scripts:

  ```bash

  trap 'echo "Interrupted!"' SIGINT

```  29.

### Other Commands

- **`cron`** for scheduling.

- **`ssh-keygen`**, **`scp`** for secure access.

- **`chmod`**, **`chown`** for permissions.

---

## Best Practices

- Use **`set -euo pipefail`** for strict error handling.

- Quote variables: `"$var"` to prevent word splitting 30.

- Modularize scripts into functions.

- Comment liberally and use meaningful names.

- Validate inputs before use.

---

## Python Programming

### Basic Programming Constructs

#### Data Types & Variables

Python has built-in types: `int`, `float`, `str`, `bool`, `list`, `tuple`, `dict`, `set` 31.

Variables are names bound to objects:

```python
x = 42      # int
```
```

name = "Bob" # str

``` 32.

#### Operators

- Arithmetic: `+`, `-`, `*`, `/`, `**`
- Comparison: `==`, `!=`, `<`, `>`
- Logical: `and`, `or`, `not` 33.

#### Control Structures

- **`if` ** / **`elif` ** / **`else` **
- **Loops**: `for`, `while`
- **Comprehensions**: `[x*2 for x in nums]`

#### Data Structures

- **Lists**: ordered, mutable.
- **Dicts**: key–value maps.
- **Sets**: unordered unique elements 34.

#### Functions

```python
def greet(name):
    return f"Hello, {name}"
```

First-class citizens, support default/keyword arguments.

Classes & Objects
```

class Person:

  def __init__(self, name):

self.name = name

  def greet(self):

print(f"Hi, {self.name}")

``` 35.

#### Modules & Packages

Logical code grouping:

```python

# mymodule.py

def foo(): pass

# usage

import mymodule

Supports namespaces and reusability.

---

Exception Handling

try:

  1/0

except ZeroDivisionError as e:

```
    print("Cannot divide by zero")
else:
    print("No error")
finally:
    print("Cleanup")
```

Define custom exceptions by subclassing Exception .

File Handling

```
with open('data.txt','r') as f:
contents = f.read()
with open('out.txt','w') as f:
    f.write("Hello")
```

Supports text and binary modes; ensures automatic closure via with .

Async Programming

```
import asyncio
```

```
async def say_after(delay, msg):
    await asyncio.sleep(delay)
    print(msg)

async def main():
    await asyncio.gather(say_after(1, "Hello"), say_after(2, "World"))

asyncio.run(main())
```

Uses async/await coroutines for cooperative multitasking .

Memory Management

Python’s memory manager allocates objects on a private heap and reclaims via reference counting plus cyclic-garbage collection .

The gc module exposes functions to tune collection thresholds and inspect objects.

PYTHON

1. Basic Programming Constructs

1.1 Data Types and Variables

Python supports various data types:

Integers (int): Whole numbers.

age = 25

Floating-point numbers (float): Numbers with decimal points.

price = 19.99

Strings (str): Textual data.

name = "Alice"

Booleans (bool): Logical values.

is_active = True

Variables are created by assigning values using the = operator.

1.2 Operators

Python provides several operators:

Arithmetic Operators: +, -, *, /, //, %, **

```
result = 10 + 5  # Addition
```

Comparison Operators: ==, !=, >, <, >=, <=

```
is_equal = (10 == 5)
```

Logical Operators: and, or, not

```
is_valid = True and False
```

1.3 Control Structures

Control structures manage the flow of execution:

Conditional Statements:

```
if age >= 18:
   print("Adult")
else:
   print("Minor")
```

Loops:

For Loop:

```
for i in range(5):
    print(i)
```

While Loop:

```
count = 0
while count < 5:
    print(count)
    count += 1
```

1.4 Data Structures

Python offers built-in data structures:

Lists: Ordered, mutable collections.

```
fruits = ["apple", "banana", "cherry"]
fruits.append("orange")
```

Tuples: Ordered, immutable collections.

coordinates = (10, 20)

Sets: Unordered collections of unique elements.

unique_numbers = {1, 2, 3}

Dictionaries: Key-value pairs.

person = {"name": "Alice", "age": 25}

1.5 Functions

Functions encapsulate reusable code blocks:

```
def greet(name):
    return f"Hello, {name}!"

message = greet("Bob")
print(message)
```

1.6 Classes and Objects

Python supports object-oriented programming:

```
class Person:
    def __init__(self, name):
self.name = name

    def greet(self):
print(f"Hi, I'm {self.name}")

p = Person("Alice")
p.greet()
```

1.7 Modules and Packages

Modules are Python files containing functions and classes. Packages are directories containing multiple modules.

Creating a Module (mymodule.py):

```
def add(a, b):
    return a + b
```

Using a Module:

```
import mymodule
result = mymodule.add(5, 3)
```

---

2. Exception Handling

Python uses exceptions to handle errors gracefully.

2.1 Try-Except Block

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero.")
```

2.2 Else and Finally Clauses

Else: Executes if no exception occurs.

Finally: Executes regardless of exceptions.

```
try:
    value = int("10")
except ValueError:
    print("Invalid input.")
else:
```

```
    print("Conversion successful.")
finally:
    print("Execution completed.")
```

---

3. File Handling

Python provides functions to work with files.

3.1 Opening and Reading Files

```
with open("example.txt", "r") as file:
content = file.read()
    print(content)
```

3.2 Writing to Files

```
with open("example.txt", "w") as file:
    file.write("Hello, World!")
```

3.3 Appending to Files

```
with open("example.txt", "a") as file:
    file.write("\nAppended line.")
```

---

4. Asynchronous Programming

Python's asyncio library enables asynchronous programming.

4.1 Async Functions

```
import asyncio

async def say_hello():
    await asyncio.sleep(1)
    print("Hello")

asyncio.run(say_hello())
```

4.2 Running Multiple Tasks

```
import asyncio

async def task(name):
    await asyncio.sleep(1)
    print(f"Task {name} completed")
```

```
async def main():
    await asyncio.gather(
        task("A"),
        task("B"),
        task("C")
    )

asyncio.run(main())
```

---

5. Memory Management

Python manages memory automatically using reference counting and garbage collection.

5.1 Reference Counting

Each object keeps track of the number of references to it. When the count reaches zero, the memory is deallocated.

5.2 Garbage Collection

Python's garbage collector handles cyclic references.

```
import gc
```

```
# Enable automatic garbage collection
gc.enable()

# Manually trigger garbage collection
gc.collect()
```

PLSQL

PL/SQL is Oracle’s powerful procedural extension to SQL, enabling you to build modular, high-performance database programs with loops, conditionals, exception handling, and reusable units such as procedures, functions, and packages—all executing inside the database engine to minimize network traffic and improve security. A PL/SQL program is organized into blocks with declaration, execution, and exception sections, and supports all SQL operations alongside procedural constructs for controlling flow, managing errors, and iterating over query results. You’ll learn how to declare variables of various scalar and composite data types, use operators, write selection and iteration statements, define and invoke procedures/functions, group code into packages, handle runtime errors via predefined and user-defined exceptions, work with explicit and REF cursors (including FOR UPDATE), create DML triggers at statement and row levels, manipulate RECORD and COLLECTION types, and finally apply these concepts through hands-on “Lend a Hand” examples and practice problems.

Introduction to PL/SQL

PL/SQL (Procedural Language/SQL) is Oracle's proprietary extension of SQL designed for seamless integration of data access and procedural logic, running server-side within the Oracle database for optimal performance and security. It supports binding SQL statements within procedural blocks, enabling transaction control, error handling, and modular programming through subprograms and packages.

Block Structure & Types of SQL Statements

PL/SQL Block Structure

A PL/SQL block comprises three sections:

1. Declaration: where you define variables, constants, cursors, and exceptions.

2. Execution: contains the procedural and SQL statements to be executed.

3. Exception: holds handlers that run when runtime errors occur.

DECLARE

```
 v_count NUMBER;
BEGIN
 SELECT COUNT(*) INTO v_count FROM employees;
 DBMS_OUTPUT.PUT_LINE('Total Employees: ' || v_count);
EXCEPTION
 WHEN NO_DATA_FOUND THEN
  DBMS_OUTPUT.PUT_LINE('No records found.');
END;
/
```

Types of SQL Statements

DDL (Data Definition Language): CREATE, ALTER, DROP.

DML (Data Manipulation Language): SELECT, INSERT, UPDATE, DELETE.

TCL (Transaction Control): COMMIT, ROLLBACK, SAVEPOINT.

DCL (Data Control): GRANT, REVOKE.

PL/SQL Building Elements & Data Types

Scalar Data Types

NUMBER: numeric types, e.g., NUMBER(8,2).

VARCHAR2: variable-length strings, e.g., VARCHAR2(100).

DATE: dates and times.

BOOLEAN: logical TRUE/FALSE values.

Composite & LOB Types

RECORD: user-defined composite types (see Records section).

TABLE/VARRAY: PL/SQL collections (see Collections section).

CLOB/BLOB: large object types for text/binary data.

Operators & Variables in PL/SQL

Operators

Arithmetic: +, -, *, /, **.

Relational: =, !=, <, >, <=, >=.

Logical: AND, OR, NOT.

String: concatenation via ||.

Variable Declaration

```
DECLARE
  v_salary  NUMBER(7,2) := 4500.50;
  v_name    VARCHAR2(30) := 'Alice';
  c_limit   CONSTANT NUMBER := 100;
BEGIN
  NULL;
END;
/
```

Uninitialized variables default to NULL.

Lend a Hand: Simple Example

Task: List employees in department 10.

```
DECLARE
  CURSOR c_dept10 IS
    SELECT employee_id, first_name, salary
      FROM employees
     WHERE department_id = 10;
  v_emp_rec c_dept10%ROWTYPE;
BEGIN
  FOR v_emp_rec IN c_dept10 LOOP
    DBMS_OUTPUT.PUT_LINE(
      v_emp_rec.employee_id || ' - ' ||
      v_emp_rec.first_name || ' : ' ||
      v_emp_rec.salary
    );
  END LOOP;
END;
/
```

This uses an explicit cursor and a cursor FOR loop to iterate results.

Estimated Time: 30 minutes

PL/SQL Statements and Types

Selection Statements

IF...THEN...ELSIF...ELSE: conditional execution of code blocks.

CASE: multi-way branch based on an expression.

```
IF v_salary > 5000 THEN
  DBMS_OUTPUT.PUT_LINE('High');
ELSIF v_salary > 3000 THEN
  DBMS_OUTPUT.PUT_LINE('Medium');
ELSE
  DBMS_OUTPUT.PUT_LINE('Low');
END IF;
```

CASE Statement

```
CASE v_job_id
  WHEN 'IT_PROG' THEN v_level := 'A';
  WHEN 'ST_CLERK' THEN v_level := 'B';
  ELSE v_level := 'C';
END CASE;
```

Iteration Statements

LOOP...END LOOP: infinite loop with explicit exit.

WHILE...LOOP: pre-test conditional loop.

FOR i IN ... LOOP: count-controlled loop.

```
FOR i IN 1..3 LOOP
  DBMS_OUTPUT.PUT_LINE('Iteration ' || i);
END LOOP;
```

Sequential Statements & Nesting

PL/SQL blocks can nest arbitrarily:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE('Start');
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Nested');
  END;
  DBMS_OUTPUT.PUT_LINE('End');
END;
/
```

Estimated Time: 1 hour

Subprograms: Procedures and Functions

Procedures

```
CREATE OR REPLACE PROCEDURE raise_salary (
  p_emp_id  IN employees.employee_id%TYPE,
  p_pct     IN NUMBER
) AS
BEGIN
  UPDATE employees
     SET salary = salary * (1 + p_pct/100)
   WHERE employee_id = p_emp_id;
  COMMIT;
END raise_salary;
/
```

Procedures perform actions but do not return values.

Lend a Hand on Procedures

Task: Add a department.

```
CREATE OR REPLACE PROCEDURE add_dept (
  p_dept_id   IN departments.department_id%TYPE,
  p_dept_name IN departments.department_name%TYPE
) AS
BEGIN
  INSERT INTO departments (department_id, department_name)
  VALUES (p_dept_id, p_dept_name);
  COMMIT;
END add_dept;
/
```

Functions

```
CREATE OR REPLACE FUNCTION calc_bonus (
  p_salary IN NUMBER
) RETURN NUMBER AS
BEGIN
  RETURN p_salary * 0.10;
END calc_bonus;
/
```

Functions return a value and can be used in SQL statements.

Estimated Time: 1.5 hours

Packages

Packages group related procedures, functions, types, and variables into a single schema object with a specification and body.

```
-- Specification
CREATE OR REPLACE PACKAGE emp_pkg AS
  PROCEDURE list_dept(p_dept_id IN NUMBER);
  FUNCTION avg_salary(p_dept_id IN NUMBER) RETURN NUMBER;
END emp_pkg;
/

-- Body
CREATE OR REPLACE PACKAGE BODY emp_pkg AS
  PROCEDURE list_dept(p_dept_id IN NUMBER) IS
  BEGIN
    FOR r IN (SELECT first_name FROM employees WHERE department_id=p_dept_id) LOOP
      DBMS_OUTPUT.PUT_LINE(r.first_name);
    END LOOP;
  END;
  FUNCTION avg_salary(p_dept_id IN NUMBER) RETURN NUMBER IS
    v_avg NUMBER;
```

```
 BEGIN
   SELECT AVG(salary) INTO v_avg FROM employees WHERE department_id=p_dept_id;
   RETURN v_avg;
 END;
END emp_pkg;
/
```

Estimated Time: 1 hour

Exception Handling

Predefined Exceptions

Common ones include NO_DATA_FOUND, TOO_MANY_ROWS, ZERO_DIVIDE.

User-Defined Exceptions

```
DECLARE
 e_low_balance EXCEPTION;
BEGIN
 IF v_balance < v_withdraw THEN
   RAISE e_low_balance;
 END IF;
```

```
EXCEPTION
  WHEN e_low_balance THEN
    DBMS_OUTPUT.PUT_LINE('Insufficient funds');
END;
/
```

```
RAISE_APPLICATION_ERROR
```

```
IF v_age < 18 THEN
  RAISE_APPLICATION_ERROR(-20001, 'Age must be >= 18');
END IF;
```

Estimated Time: 45 minutes

Cursors

Explicit Cursors

```
DECLARE
  CURSOR c_emp(p_dept NUMBER) IS
    SELECT employee_id, salary FROM employees WHERE department_id = p_dept;
  v_rec c_emp%ROWTYPE;
BEGIN
```

```
 OPEN c_emp(30);
 LOOP
  FETCH c_emp INTO v_rec;
  EXIT WHEN c_emp%NOTFOUND;
  DBMS_OUTPUT.PUT_LINE(v_rec.employee_id || ': ' || v_rec.salary);
 END LOOP;
 CLOSE c_emp;
END;
/
```

Cursor FOR Loop

```
FOR v_rec IN (SELECT * FROM employees WHERE department_id=20) LOOP
 DBMS_OUTPUT.PUT_LINE(v_rec.first_name);
END LOOP;
```

FOR UPDATE & WHERE CURRENT OF

```
FOR r IN (SELECT * FROM employees WHERE department_id=20 FOR UPDATE) LOOP
 UPDATE employees SET salary = salary*1.05 WHERE CURRENT OF r;
END LOOP;
```

REF Cursors

```
DECLARE
 TYPE emp_ref IS REF CURSOR;
```

```
 v_cur emp_ref;
 v_emp employees%ROWTYPE;
BEGIN
 OPEN v_cur FOR 'SELECT * FROM employees WHERE department_id=10';
 LOOP
  FETCH v_cur INTO v_emp;
  EXIT WHEN v_cur%NOTFOUND;
  DBMS_OUTPUT.PUT_LINE(v_emp.first_name);
 END LOOP;
 CLOSE v_cur;
END;
/
```

Estimated Time: 1 hour

Triggers

Statement-Level vs. Row-Level

Statement-Level: fires once per SQL statement.

Row-Level: fires once per row affected.

Row-Level Trigger Example

```
CREATE OR REPLACE TRIGGER trg_sal_check
 BEFORE UPDATE OF salary ON employees
 FOR EACH ROW
BEGIN
 IF :NEW.salary > :OLD.salary * 1.20 THEN
  RAISE_APPLICATION_ERROR(-20002, 'Raise exceeds 20%');
 END IF;
END;
/
```

Lend a Hand on Triggers

Task: Log department inserts.

```
CREATE OR REPLACE TRIGGER trg_dept_audit
 AFTER INSERT ON departments
 FOR EACH ROW
BEGIN
 INSERT INTO dept_log(dept_id, action_date)
 VALUES(:NEW.department_id, SYSDATE);
END;
/
```

Estimated Time: 45 minutes

Records & Collections

Records

Table-Based Record: employees%ROWTYPE.

Custom Record:

TYPE rec_emp IS RECORD (id NUMBER, name VARCHAR2(50));

v_emp_rec rec_emp;

Cursor-Based Record: %ROWTYPE of an explicit cursor.

Collections

Three types:

Associative Arrays (index-by tables).

Nested Tables (store unbounded lists).

VARRAYs (fixed-size arrays).

```
DECLARE
  TYPE num_tab IS TABLE OF NUMBER INDEX BY PLS_INTEGER;
  v_nums num_tab;
BEGIN
  v_nums(1) := 100;
  v_nums(2) := 200;
  DBMS_OUTPUT.PUT_LINE(v_nums(1));
END;
/
```