CORE JAVA SYLLABUS

1. FUNDAMENTALS

1.1 JVM

A runtime environment that executes Java bytecode. It provides memory management, garbage collection, and security features. The JVM is platform-specific but allows Java programs to run on any operating system.

1.2 - Bytecode

Intermediate code generated by the Java compiler (.class files). It's platform-independent and executed by the JVM. Bytecode is more efficient than interpreted languages but slower than native machine code.

1.3 - Platform Independence

Java's "write once, run anywhere" principle. Source code compiles to bytecode that runs on any JVM, regardless of the underlying operating system or hardware architecture.

1.4 - Data types, variables, control flow, arrays

1.5 - Packages

Namespace mechanism to organize classes and avoid naming conflicts. Use package declarations at the top of source files. Import classes using import statements or fully qualified names.

1.6 - Access Modifiers

Control visibility of classes, methods, and fields. Four levels: public (everywhere), protected (package + subclasses), default/package-private (same package), private (same class only).

Java runs on the JVM using bytecode for cross-platform execution. Master basic syntax including types, variables, control structures, and arrays. Organize code with packages and control visibility using access modifiers.

2. OBJECT-ORIENTED PROGRAMMING

2.1 - Classes

Blueprints or templates that define the structure and behavior of objects. Contain fields (data) and methods (functions). Use PascalCase naming convention.

2.2 - Objects

Instances of classes created using the new keyword. Each object has its own state (field values) and can invoke methods. Objects are reference types in Java.

2.3 - **Constructors**

Special methods that initialize objects when created. Must have the same name as the class. Default constructor is provided if none is defined. Use this() to call other constructors.

2.4 - **Encapsulation**

Data hiding principle that bundles data and methods together. Use private fields with public getter/setter methods. Protects internal state from external modification.

2.5 - **Inheritance**

Mechanism where a class (subclass) inherits properties and methods from another class (superclass). Use extends keyword. Supports code reuse and "is-a" relationships.

2.6 - **Polymorphism**

Ability of objects to take multiple forms. Method overriding (runtime) and method overloading (compile-time). Enables flexible and extensible code design.

2.7 - **Interfaces**

Contracts that define what methods a class must implement. Use implements keyword. Support multiple inheritance. All methods are public and abstract by default.

2.8 - **Abstract Classes**

Classes that cannot be instantiated directly. Can contain both abstract and concrete methods. Use abstract keyword. Support single inheritance only.

Design software using classes and objects with proper constructors. Apply OOP principles: encapsulation (data hiding), inheritance (code reuse), and polymorphism (flexible behavior). Understand when to use interfaces versus abstract classes.

3. STRINGS

3.1 - **String Immutability**

Strings cannot be changed after creation. Any modification creates a new String object. Ensures thread-safety and prevents accidental modifications. Example: String s = "Hello"; s += " World"; creates a new object.

3.2 - **String Pooling**

JVM maintains a pool of unique string literals to save memory. String literals with same content share the same memory location. Use intern() method to add strings to pool. Example: "Hello" == "Hello" returns true.

3.3 - **StringBuilder**

Mutable sequence of characters for efficient string building. Not thread-safe but faster than StringBuffer. Use for single-threaded string concatenation. Example: StringBuilder sb = new StringBuilder(); sb.append("Hello").append(" World");

3.4 - StringBuffer

Thread-safe mutable sequence of characters. Synchronized methods ensure thread safety but slower than StringBuilder. Use for multi-threaded string operations. Example: StringBuffer sb = new StringBuffer(); sb.append("Hello").append(" World");

3.5 - Common APIs

substring(int start, int end): Extract portion of string. split(String regex): Split string into array. replace(String old, String new): Replace occurrences. format(String format, Object... args): Format string with placeholders. Example: String.format("Hello %s, you are %d years old", "John", 25);

Strings are immutable and pooled for memory efficiency. Use StringBuilder for efficient concatenation. Master essential APIs like substring, split, replace, and format.

4. EXCEPTIONS AND ERROR HANDLING

4.1 - Checked Exceptions

Exceptions that must be handled at compile time. Compiler forces you to either catch or declare in method signature using throws. Examples: IOException, SQLException, ClassNotFoundException. Must be handled or propagated.

4.2 - Unchecked Exceptions

Runtime exceptions that don't need to be declared or caught. Extend RuntimeException. Examples: NullPointerException, ArrayIndexOutOfBoundsException, IllegalArgumentException. Can be caught but not required.

4.3 - try/catch/finally

try block contains code that might throw exceptions. catch blocks handle specific exception types. finally block always executes regardless of exceptions. Use for cleanup operations. Example: try { riskyCode(); } catch (Exception e) { handle(e); } finally { cleanup(); }

4.4 - try-with-resources

Automatic resource management for classes implementing AutoCloseable. Resources are automatically closed even if exceptions occur. Introduced in Java 7. Example: try (FileInputStream fis = new FileInputStream("file.txt")) { /* use fis */ }

4.5 - Custom Exceptions

User-defined exception classes extending Exception or RuntimeException. Should include meaningful constructors and messages. Use for domain-specific error conditions. Example: class CustomException extends Exception { public CustomException(String message) { super(message); } }

Distinguish between checked exceptions (must handle) and unchecked exceptions (runtime errors). Use try/catch/finally and try-with-resources for proper resource management. Create custom exceptions with meaningful context.

5. GENERICS

5.1 - Type Parameters

Placeholder types that make classes, interfaces, and methods type-safe. Declared in angle brackets <T>. Can be used for any type. Example: List<String> list = new ArrayList<>(); or public class Box<T> { private T item; }

5.2 - Bounds

Constraints on type parameters to restrict what types can be used. extends keyword for upper bounds. Example: public class NumberBox<T extends Number> means T must be Number or its subclass. Multiple bounds: T extends Number & Comparable<T>

5.3 - Wildcards ? extends

Upper bounded wildcard for covariance. Accepts the specified type and its subclasses. Read-only access. Example: List<? extends Number> can hold List<Integer>, List<Double>, etc. Cannot add elements (except null).

5.4 - Wildcards ? super

Lower bounded wildcard for contravariance. Accepts the specified type and its superclasses. Write access allowed. Example: List<? super Integer> can hold List<Integer>, List<Number>, List<Object>. Can add Integer and its subclasses.

5.5 - Type Erasure

Process where generic type information is removed at compile time. All generic types become Object or their upper bound. Enables backward compatibility but limits runtime type checking. Example: List<String> becomes List at runtime.

Generics provide compile-time type safety through type parameters and bounds. Use wildcards for flexible type relationships. Understand type erasure limitations at runtime.

6. COLLECTIONS FRAMEWORK

6.1 - Core Interfaces

6.1.1 **Collection**: Root interface for all collections. Defines basic operations: add, remove, contains, size, iterator. Example: Collection<String> coll = new ArrayList<>();

6.1.2 **List**: Ordered collection allowing duplicates. Index-based access. Examples: ArrayList, LinkedList, Vector. Example: List<String> list = new ArrayList<>(); list.get(0);

6.1.3 **Set**: Collection with no duplicates. No index-based access. Examples: HashSet, LinkedHashSet, TreeSet. Example: Set<String> set = new HashSet<>(); set.add("item");

6.1.4 **Map**: Key-value pairs, no duplicate keys. Examples: HashMap, LinkedHashMap, TreeMap. Example: Map<String, Integer> map = new HashMap<>(); map.put("key", 1);

6.2 - **Key Implementations**

6.2.1 ArrayList: Resizable array, fast random access, slow insertion/deletion. O(1) get, O(n) add/remove. Example: List<String> list = new ArrayList<>();

6.2.2 LinkedList: Doubly-linked list, fast insertion/deletion, slow random access. O(n) get, O(1) add/remove. Example: List<String> list = new LinkedList<>();

6.2.3 HashMap: Hash table, O(1) average case, no ordering. Example: Map<String, Integer> map = new HashMap<>();

6.2.4 TreeMap: Red-black tree, O(log n) operations, sorted by keys. Example: Map<String, Integer> map = new TreeMap<>();

6.3 - **equals**() **Contract**

Reflexive: x.equals(x) must be true. Symmetric: x.equals(y) iff y.equals(x). Transitive: if x.equals(y) and y.equals(z), then x.equals(z). Consistent: multiple calls return same result. Non-null: x.equals(null) returns false.

6.4 - hashCode() Contract

If two objects are equal according to equals(), they must have same hashCode(). If objects have same hashCode(), they may or may not be equal. Must be consistent across multiple calls. Example: public int hashCode() { return Objects.hash(name, age); }

Use Collection, List, Set, and Map interfaces with appropriate implementations. Choose based on performance characteristics and ordering needs. Always implement equals/hashCode correctly.

7. FUNCTIONAL PROGRAMMING (JAVA 8+)

7.1 - **Functional Interfaces**

Interfaces with exactly one abstract method. Can be annotated with @FunctionalInterface. Built-in examples: Predicate<T>, Function<T,R>, Consumer<T>, Supplier<T>. Example: Predicate<String> isEmpty = s -> s.isEmpty();

7.2 - **Lambdas**

Anonymous functions that implement functional interfaces. Syntax: (parameters) -> expression or (parameters) -> { statements }. Type inference allows omitting parameter types. Example: list.forEach(item -> System.out.println(item));

7.3 - **Streams API**

Declarative way to process collections. Create with collection.stream(). Intermediate operations: filter, map, sorted, distinct. Terminal operations: collect, forEach, reduce, findFirst. Example: list.stream().filter(x -> x > 5).map(x -> x * 2).collect(Collectors.toList());

7.4 - **Optional**

Container that may or may not contain a non-null value. Prevents NullPointerException. Methods: of(), empty(), ofNullable(), isPresent(), get(), orElse(). Example: Optional<String> name = Optional.ofNullable(getName()); String result = name.orElse("Unknown");

Write concise code using functional interfaces and lambda expressions. Process data declaratively with Streams. Use Optional to handle null values safely.

8. DATE AND TIME (JAVA.TIME)

8.1 - **LocalDate**

Represents a date without time or timezone (year-month-day). Immutable and thread-safe. Methods: now(), of(), parse(), plusDays(), minusMonths(). Example: LocalDate today = LocalDate.now(); LocalDate birthday = LocalDate.of(1990, 5, 15);

8.2 - **LocalTime**

Represents time without date or timezone (hour-minute-second-nanosecond). Immutable and thread-safe. Methods: now(), of(), parse(), plusHours(), minusMinutes(). Example: LocalTime now = LocalTime.now(); LocalTime meeting = LocalTime.of(14, 30);

8.3 - **LocalDateTime**

Combines LocalDate and LocalTime without timezone information. Immutable and thread-safe. Methods: now(), of(), parse(), plusDays(), minusHours(). Example: LocalDateTime now = LocalDateTime.now(); LocalDateTime event = LocalDateTime.of(2024, 12, 25, 10, 30);

8.4 - **Instant**

Represents a point in time on the UTC timeline. Used for timestamps and logging. Methods: now(), ofEpochSecond(), toEpochMilli(). Example: Instant now = Instant.now(); long timestamp = now.toEpochMilli();

8.5 - **ZoneId**

Represents a timezone identifier. Use with ZonedDateTime for timezone-aware operations. Common zones: "UTC", "America/New_York", "Europe/London". Example: ZoneId zone = ZoneId.of("America/New_York"); ZonedDateTime zoned = LocalDateTime.now().atZone(zone);

8.6 - **DateTimeFormatter**

Formats and parses date-time objects. Predefined patterns: ISO_LOCAL_DATE, ISO_LOCAL_TIME. Custom patterns: "yyyy-MM-dd HH:mm:ss". Example: DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/yyyy"); String formatted = date.format(formatter);

Use immutable java.time types instead of legacy Date/Calendar. Handle time zones properly with ZoneId. Format and parse dates with DateTimeFormatter.

9. I/O AND NIO.2

9.1 - java.io Streams

Byte-oriented streams for binary data. InputStream/OutputStream are abstract base classes. Examples: FileInputStream, FileOutputStream, BufferedInputStream. Example: FileInputStream fis = new FileInputStream("file.txt"); int data = fis.read();

9.2 - Readers

Character-oriented streams for text data. Reader/Writer are abstract base classes. Examples: FileReader, BufferedReader, StringReader. Example: BufferedReader reader = new BufferedReader(new FileReader("file.txt")); String line = reader.readLine();

9.3 - Writers

Character-oriented streams for writing text data. Examples: FileWriter, BufferedWriter, PrintWriter. Example: BufferedWriter writer = new BufferedWriter(new FileWriter("file.txt")); writer.write("Hello World");

9.4 - java.nio.file Paths

Modern way to represent file/directory paths. Path interface is platform-independent. Methods: get(), resolve(), normalize(), toAbsolutePath(). Example: Path path = Paths.get("folder", "file.txt"); Path absolute = path.toAbsolutePath();

9.5 - java.nio.file Files

Utility class for file operations. Methods: exists(), createFile(), delete(), copy(), move(), readAllLines(), write(). Example: Files.copy(source, destination); List<String> lines = Files.readAllLines(path);

9.6 - try-with-resources

Automatic resource management for AutoCloseable resources. Resources are closed automatically even if exceptions occur. Introduced in Java 7. Example: try (FileInputStream fis = new FileInputStream("file.txt")) { /* use fis */ }

Use java.io for basic I/O operations with proper resource management. Prefer java.nio.file for modern file operations. Always use try-with-resources for automatic cleanup.

10. CONCURRENCY

10.1 Threads

Lightweight processes that run concurrently. Create with Thread class or Runnable interface. Methods: start(), run(), sleep(), join(). Example: Thread thread = new Thread(() -> System.out.println("Hello")); thread.start();

10.2 - Executors

Framework for managing thread pools and asynchronous execution. ExecutorService interface provides lifecycle management. Examples: newFixedThreadPool(), newCachedThreadPool(), newSingleThreadExecutor(). Example: ExecutorService executor = Executors.newFixedThreadPool(4);

10.3 - Runnable/Callable

Runnable: Interface for tasks that don't return values. Single method: run(). Callable: Interface for tasks that return values. Single method: call() throws Exception. Example: Callable<String> task = () -> "Result"; Future<String> future = executor.submit(task);

10.4 - Synchronization

Mechanism to control access to shared resources. Use synchronized keyword on methods or blocks. Ensures only one thread can access synchronized code at a time. Example: synchronized(this) { sharedResource++; }

10.5 - volatile

Keyword that ensures variable visibility across threads. Prevents compiler optimizations that could cause visibility issues. Doesn't provide atomicity. Example: private volatile boolean flag = false;

10.6 - atomics

Thread-safe classes for atomic operations. Examples: AtomicInteger, AtomicLong, AtomicReference. Provide compare-and-swap operations. Example: AtomicInteger counter = new AtomicInteger(0); counter.incrementAndGet();

10.7 - CompletableFuture

Asynchronous programming with composable futures. Methods: supplyAsync(), thenApply(), thenCompose(), thenCombine(). Handles exceptions and cancellation. Example: CompletableFuture.supplyAsync(() -> "Hello").thenApply(s -> s + " World");

Create and manage threads using Executors. Ensure thread-safety with synchronization, volatile, and atomic classes. Compose asynchronous operations with CompletableFuture.

11. JVM INTERNALS AND PERFORMANCE

11.1 Memory Model

11.1.1 Heap: Runtime data area for objects and arrays. Divided into Young Generation (Eden, Survivor spaces) and Old Generation. Shared among all threads. Example: Object obj = new Object(); // stored in heap

11.1.2 Stack: Thread-specific memory for method calls, local variables, and return addresses. Each thread has its own stack. LIFO structure. Example: int localVar = 10; // stored in stack

11.1.3 Metaspace: Memory area for class metadata (Java 8+). Replaces PermGen. Stores class definitions, method bytecode, constant pool. Example: Class<?> clazz = MyClass.class; // metadata in metaspace

11.2 Garbage Collection

Automatic memory management that reclaims unused objects. Algorithms: Serial, Parallel, CMS, G1, ZGC, Shenandoah. Young GC (frequent, fast) vs Full GC (infrequent, slow). Example: System.gc(); // suggests GC (not guaranteed)

11.3 - Profiling Tools

11.3.1 JFR (Java Flight Recorder): Low-overhead profiling built into JVM. Records events, performance metrics, and memory usage. Example: java -XX:+FlightRecorder -XX:StartFlightRecording=duration=60s,filename=profile.jfr MyApp

11.3.2 VisualVM: GUI tool for monitoring and profiling. Shows memory usage, thread activity, CPU usage. Example: jvisualvm command to launch GUI

jcmd, jmap, jstack: Command-line tools for diagnostics. jcmd for general commands, jmap for heap dumps, jstack for thread dumps. Example: jcmd <pid> GC.run_finalization

Understand JVM memory organization and garbage collection algorithms. Profile applications using JFR, VisualVM, or async-profiler. Monitor performance with jcmd, jmap, jstack.

12. REFLECTION, ANNOTATIONS, SPI

12.1 - java.lang.reflect

Runtime inspection and manipulation of classes, methods, fields. Classes: Class, Method, Field, Constructor. Methods: getDeclaredMethods(), getField(), invoke(). Example: Class<?> clazz = MyClass.class; Method method = clazz.getDeclaredMethod("methodName", String.class); method.invoke(instance, "arg");

12.2 - Custom Annotations

User-defined metadata for classes, methods, fields. Define with @interface. Use @Retention, @Target, @Documented. Example: @Retention(RetentionPolicy.RUNTIME) @Target(ElementType.METHOD) public @interface MyAnnotation { String value() default ""; }

12.3 - ServiceLoader

Pluggable architecture for loading service implementations. Create META-INF/services/interfaceName file. Load implementations dynamically. Example: ServiceLoader<MyService> loader = ServiceLoader.load(MyService.class); for (MyService service : loader) { service.doSomething(); }

Use reflection for runtime inspection and dynamic invocation. Create custom annotations with proper retention and targets. Implement pluggable architectures with ServiceLoader.

13. MODULES (JAVA 9+)

13.1 - module-info.java

Module descriptor file that defines module boundaries and dependencies. Contains module name, exports, requires, provides, uses clauses. Example: module com.example.app { requires java.base; exports com.example.api; }

13.2 - exports/requires

exports: Makes packages available to other modules. requires: Declares dependency on another module. Control module boundaries and encapsulation. Example: exports com.example.api; requires java.sql;

13.3 - Migration from classpath

Transition from classpath-based to module-path-based applications. Identify dependencies, create module-info.java files, handle split packages. Use --module-path instead of -cp. Example: java --module-path mods --module com.example.app/com.example.Main

Organize code into modules with module-info.java. Control module boundaries with exports and requires. Plan migration from classpath to module-path carefully.

14. NETWORKING

14.1 - Sockets

Low-level network communication using TCP/UDP. ServerSocket for servers, Socket for clients. Example: ServerSocket server = new ServerSocket(8080); Socket client = server.accept();

14.2 - URL/HttpURLConnection

Basic HTTP client for simple requests. Synchronous, blocking operations. Example: URL url = new URL("http://example.com"); HttpURLConnection conn = (HttpURLConnection) url.openConnection();

14.3 - java.net.http.HttpClient

Modern HTTP client (Java 11+). Supports HTTP/2, async operations, timeouts. Example: HttpClient client = HttpClient.newHttpClient(); HttpRequest request = HttpRequest.newBuilder().uri(URI.create("http://example.com")).build();

14.4 - Error handling and timeouts

Implement proper exception handling for network operations. Set timeouts to prevent hanging. Use retry logic for transient failures. Example: request.timeout(Duration.ofSeconds(30));

Use sockets for low-level networking. Prefer HttpClient for HTTP operations with proper error handling. Implement timeouts and retry logic.

15. SECURITY BASICS

15.1 - JCA/JCE (hashing, encryption)

Java Cryptography Architecture for hashing, encryption, digital signatures. Classes: MessageDigest, Cipher, Signature. Example: MessageDigest md = MessageDigest.getInstance("SHA-256"); byte[] hash = md.digest(data);

15.2 - KeyStore, SSL/TLS

KeyStore: Secure storage for cryptographic keys and certificates. SSL/TLS: Secure communication protocols. Example: KeyStore ks = KeyStore.getInstance("JKS"); SSLContext sslContext = SSLContext.getInstance("TLS");

15.3 - SecureRandom

Cryptographically secure random number generator. Use for generating keys, salts, nonces. Example: SecureRandom random = new SecureRandom(); byte[] salt = new byte[16]; random.nextBytes(salt);

Use Java Cryptography Architecture for hashing and encryption. Manage keys with KeyStore and secure communications with TLS. Use SecureRandom for cryptographic operations.

16. DATABASE ACCESS (JDBC)

16.1 - Connection, Statement, ResultSet

Connection: Database connection. Statement: SQL execution. ResultSet: Query results. Example: Connection conn = DriverManager.getConnection(url); Statement stmt = conn.createStatement(); ResultSet rs = stmt.executeQuery("SELECT * FROM users");

16.2 - Connection pooling

Reuse database connections for better performance. Libraries: HikariCP, Apache DBCP. Example: HikariConfig config = new HikariConfig(); config.setJdbcUrl("jdbc:mysql://localhost/db"); HikariDataSource ds = new HikariDataSource(config);

16.3 - SQL injection prevention

Use PreparedStatement with parameterized queries. Never concatenate user input into SQL strings. Example: PreparedStatement ps = conn.prepareStatement("SELECT * FROM users WHERE id = ?"); ps.setInt(1, userId);

Access databases using JDBC with proper resource management. Use connection pools like HikariCP for performance. Prevent SQL injection with PreparedStatement.

17. JSON/XML

17.1 - Jackson, Gson for JSON

17.1.1 **Jackson**: High-performance JSON library with streaming and tree models. \
17.1.2 **Gson**: Google's JSON library with simple API. Example: ObjectMapper mapper = new ObjectMapper(); String json = mapper.writeValueAsString(object);

17.2 - **DOM, SAX, StAX for XML**

17.2.1 DOM: In-memory tree representation.

17.2.2 SAX: Event-driven parsing. StAX: Pull-parsing API. Choose based on memory and performance needs. Example: DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance(); Document doc = factory.newDocumentBuilder().parse(file);

17.3 - **Serialization best practices**

Handle null values, circular references, and type information. Use annotations for customization. Consider performance implications. Example: @JsonIgnore private String password; @JsonProperty("user_id") private Long userId;

Use Jackson or Gson for JSON processing. Choose appropriate XML parsing model (DOM/SAX/StAX). Handle serialization edge cases properly.

18. LOGGING

18.1 - **slf4j facade**

Simple Logging Facade for Java. Provides abstraction over logging implementations. Example: Logger logger = LoggerFactory.getLogger(MyClass.class); logger.info("Application started");

18.2 - **Logback**, **Log4j2**

Logback: Successor to Log4j, faster and more flexible. Log4j2: High-performance logging with async capabilities. Example: logback.xml configuration file.

18.3 - **Structured logging**

Use consistent format with key-value pairs. Include correlation IDs, timestamps, log levels. Example: logger.info("User login: userId={}, timestamp={}", userId, Instant.now());

Use slf4j as a logging facade with Logback or Log4j2 backend. Implement structured logging with MDC for correlation. Avoid logging sensitive information.

19. INTERNATIONALIZATION (I18N/L10N)

19.1 - **ResourceBundle**

Mechanism for loading locale-specific resources. Properties files with locale suffixes. Example: ResourceBundle bundle = ResourceBundle.getBundle("messages", Locale.FRENCH); String message = bundle.getString("welcome");

19.2 - **Locale-aware formatting**

Format dates, numbers, currency according to user's locale. Use NumberFormat, DateFormat, DecimalFormat. Example: NumberFormat nf = NumberFormat.getCurrencyInstance(Locale.US); String price = nf.format(19.99);

19.3 - Message localization

Externalize user-facing strings to properties files. Support pluralization and parameter substitution. Example: messages.properties: welcome=Welcome {0}! messages_fr.properties: welcome=Bienvenue {0}!

Localize applications using ResourceBundle and Locale. Format dates, numbers, and currency per user locale. Support pluralization and right-to-left languages.

20. TESTING

20.1 - JUnit 5

Modern testing framework with annotations: @Test, @BeforeEach, @AfterEach, @ParameterizedTest. Example: @Test void testMethod() { assertEquals(expected, actual); }

20.2 - Mockito for mocking

Mocking framework for creating test doubles. Annotations: @Mock, @InjectMocks, @Spy. Example: @Mock private UserService userService; when(userService.findById(1L)).thenReturn(user);

20.3 - Testcontainers for integration tests

Integration testing with real databases and services in Docker containers. Example: @Testcontainers class DatabaseTest { @Container static PostgreSQLContainer<?> postgres = new PostgreSQLContainer<>("postgres:13"); }

Write unit tests with JUnit 5 including parameterized tests. Mock dependencies with Mockito. Use Testcontainers for integration testing with real databases.

21. BUILD AND DEPENDENCY MANAGEMENT

21.1 - Maven, Gradle

21.1.1 **Maven**: XML-based build tool with lifecycle phases.

21.1.2 **Gradle**: Groovy/Kotlin DSL with incremental builds. Example: pom.xml for Maven, build.gradle for Gradle.

21.2 - Dependency scopes

21.2.1 **compile**: Default scope, available in all phases.

21.2.2 **test**: Only for testing.

21.2.3 **provided:** Provided by runtime.

21.2.4 **runtime:** Not needed for compilation. Example: <scope>test</scope>

21.3 - **Multi-module projects**

Organize large projects into multiple modules. Parent-child relationships, dependency management. Example: parent/pom.xml with <modules> section.

Use Maven or Gradle for build automation and dependency management. Understand dependency scopes and version alignment. Structure multi-module projects effectively.

22. CODING PRACTICES

22.1 - **Immutability**

Create objects that cannot be modified after creation. Use final fields, defensive copying. Example: public final class Person { private final String name; private final int age; }

22.2 - **equals/hashCode/compareTo**

Implement these methods consistently. Use Objects.equals(), Objects.hash(). Example: public boolean equals(Object obj) { if (this == obj) return true; if (!(obj instanceof Person)) return false; Person other = (Person) obj; return Objects.equals(name, other.name); }

22.3 - **API design**

Design clear, consistent, and intuitive APIs. Use meaningful names, proper error handling, documentation. Example: public Optional<User> findUserById(Long id) throws IllegalArgumentException { if (id == null) throw new IllegalArgumentException("ID cannot be null"); return userRepository.findById(id); }

Favor immutable objects and defensive copying. Implement equals/hashCode/compareTo correctly. Design clear, consistent APIs with proper error handling.

23. LANGUAGE FEATURES BY VERSION

23.1 - **var, switch expressions, text blocks**

23.1.1 var: Local variable type inference (Java 10).

23.1.2 switch expressions: Expression-based switch (Java 14).

23.1.3 text blocks: Multi-line strings (Java 15). Example: var list = new ArrayList<String>(); String result = switch (day) { case MONDAY -> "Start of week"; default -> "Other day"; };

23.2 - **records**, **sealed classes**

23.2.1 **records**: Data classes with automatic equals/hashCode/toString (Java 16).

23.2.2 **sealed** classes: Restricted inheritance (Java 17). Example: public record Person(String name, int age) {} public sealed class Shape permits Circle, Rectangle {}