🎓 Core Java Hands On Assignments

---

1. Classes and Objects

1. **Immutable Value Object Design:** Design an **immutable** class, Money, that safely stores an amount (using BigDecimal for precision) and a currency (e.g., ISO 4217 code). Implement custom equals(), hashCode(), and toString(). The class must prevent any modification after instantiation and handle potential null/invalid inputs gracefully in its constructor, throwing an appropriate unchecked exception.
2. **Deep Cloning with Custom Logic:** Implement the Cloneable interface in a class hierarchy (Department contains a Manager object). Override the clone() method to perform a **deep clone** for all mutable objects. Furthermore, modify the cloned Manager object's salary in the Department's clone() method by adding a 10% bonus, ensuring the original objects remain untouched.
3. **Advanced Instance Control (Singleton/Object Pool):** Implement an **Object Pool** for a resourceheavy class, DatabaseConnection. The pool should limit the number of active instances (e.g., 5). Implement a borrowConnection() method that returns an instance if available, or blocks until one is returned. Implement a returnConnection() method, ensuring connection objects are properly reset before being returned to the pool. Use synchronized blocks for thread safety.
4. **Static Initialization Block and Class Loading:** Create a class, ConfigurationLoader, that reads critical system properties from a file (config.properties) only once. Use a **static initialization block** to handle the file I/O and potential exceptions (IOException, etc.). Ensure the properties are loaded *before* the first instance of the class is created or any static methods are called. Access the properties through a static method.
5. **Enum as State Machine:** Use a **Java Enum** (TaskState) to represent the states of a project task (e.g., PENDING, IN_PROGRESS, COMPLETED, BLOCKED). Define an abstract method within the enum, canTransitionTo(TaskState nextState), and provide concrete implementations for each enum constant to enforce valid state transitions (e.g., PENDING can transition to IN_PROGRESS but not COMPLETED).

---

2. Polymorphism and Interfaces

1. **Service Provider Framework (SPI Simulation):** Design a service interface, DataProcessor, and two concrete implementations: XMLProcessor and JSONProcessor. Create a static **Factory** class, ProcessorFactory, that uses **ServiceLoader** principles (without using the actual java.util.ServiceLoader API) to locate and instantiate all available implementations of DataProcessor from a list of class names configured in a properties file.

2. **Abstract Class vs. Interface Dilemma:** Create a hierarchy for financial instruments. Use an **abstract class** Instrument for common fields and non-functional methods (like getSymbol()). Use an **interface** Tradable with a **default method** getRiskLevel() that provides a basic risk score. Implement two concrete classes, Stock and Bond, where Stock overrides the default risk calculation and Bond uses it. Demonstrate how to call the default method through the interface reference.

3. **Marker Interface for Security:** Create a **marker interface** called SecureResource. Implement a utility method, accessResource(Object resource), that checks if the object implements SecureResource. If it does, print "Access Granted"; otherwise, throw a custom SecurityException (unchecked). Demonstrate the compile-time flexibility and runtime enforcement of the marker.

4. **Covariant Return Types in Overriding:** Create a hierarchy: AbstractUserFactory (returns User object) and AdminUserFactory (extends AbstractUserFactory). Ensure the AdminUserFactory's object creation method returns an AdminUser object (a subclass of User) while maintaining the same method signature in the parent abstract class, leveraging **covariant return types**.

5. **Interface Segregation Principle (ISP) Enforcement:** Design three interfaces for a complex media player: Playable, Skippable, and Recordable. Implement a class BasicMP3Player that only implements Playable. Implement another class DVRMediaCenter that implements all three. In a test scenario, demonstrate how enforcing ISP prevents the BasicMP3Player from being forced to implement unnecessary recording methods.

---

3. Working with Strings

1. **Lexicographical Sorter with Unicode:** Implement a custom string comparator using java.text.Collator to perform language-sensitive, **case-insensitive** lexicographical sorting of a list of country names that includes characters outside the basic ASCII set (e.g., "Åland", "España", "Austria").

2. **Efficient Substring Search (Knuth-Morris-Pratt Simulation):** Implement a method to find all occurrences of a small pattern string (e.g., "java") within a large text document. The solution must demonstrate a complexity better than $O(N \cdot M)$ (where N is text length, M is pattern length), perhaps by simulating the logic of the **KMP algorithm**'s pre-computation step or using String.indexOf(String, int) with smart skipping logic.

3. **Regex for Semantic Validation:** Write a regular expression and corresponding Java code to validate a complex data entry string that must meet **three independent criteria** simultaneously: 1. Contains at least one uppercase letter. 2. Contains a number followed by a hyphen. 3. Does not contain any consecutive whitespace characters. Use Matcher.matches() for full-string validation.

4. **String Interning and Performance:** Write a program that demonstrates the effect of **String interning** on memory usage and comparison performance. Create two large lists of identical, long strings: one created normally and one where every string is explicitly intern()-ed. Measure and compare the total memory footprint and the time taken to compare all strings in both lists using ==.

5. **JSON String Parser (Manual):** Write a method that takes a raw JSON string (e.g., {"name":"value",

"count":10}) and manually extracts the values of all **string-type keys** without using any external JSON library. Use methods like indexOf(), substring(), and split() while correctly handling escaped quotes and nested structures (to a limited depth).

---

4. Exceptions Handling

1. **Exception Chaining and Root Cause Analysis:** Implement a three-layer system: DataAccessor $\rightarrow$ BusinessService $\rightarrow$ Controller. The DataAccessor throws a checked SQLException. The BusinessService catches it, wraps it into a checked ServiceException (passing the original exception as the cause), and throws the new exception. The Controller catches the ServiceException and uses getCause() to log the **root cause** (SQLException) details before displaying a generic error message to the user.
2. **Resource Cleanup with Multiple Failures:** Write a method that opens two different resources (simulated with classes that implement AutoCloseable). The method should attempt to use both resources inside a single **try-with-resources** block. Force an exception during the *use* of the first resource, and another exception during the *close* operation of the second resource. Demonstrate how to retrieve all suppressed exceptions using the primary exception's getSuppressed() method.
   3. **Custom Unchecked Exception for Business Rules:** Create a custom **unchecked** exception, AccountOverdraftException, that holds the required amount and the current balance. Write a DebitService method that throws this exception if a debit operation violates a zero-balance rule. In the main method, catch this exception and use the exception's properties to print a highly detailed, user-friendly error message.
4. **Defensive Programming with Assertions:** Write a utility method, validateCredentials(String user, String pass), that performs input validation. Use the Java **assert** keyword (must be enabled via JVM arguments) to check for conditions that should *never* be false in a production environment (e.g., ensuring an internal list isn't empty *after* an operation). Contrast this with using standard exception throwing for external input errors.
5. **Re-throwing Exceptions without Declaring:** Write a generic utility method, rethrow(TException exception), that takes any Exception and **re-throws it** without declaring it in the method signature, leveraging the subtle rules of generic type inference and exception handling introduced in Java 7 to bypass the need for a checked exception declaration (a theoretical exercise to understand compiler behavior).

---

5. Date and Time API

1. **Temporal Adjuster for Business Days:** Implement a **custom TemporalAdjuster** called NextWorkingDayAdjuster. This adjuster should take a LocalDate and return the date of the next

working day (Monday to Friday). If the input date is a Friday, it should return the following Monday. If the input is a weekend, it should return the following Monday.

2. **Cross-Timezone Scheduling:** Write a program that takes a meeting time in UTC (e.g., "2025-1225T10:00:00Z") as a ZonedDateTime. Convert and print this time for three different time zones: "Europe/London", "Asia/Kolkata", and "America/Los_Angeles". Use a **Map<String, ZoneId>** to manage the time zones efficiently.

3. **Periodic Data Aggregation:** Use the Period class to simulate a financial report generation that runs every **3 months and 7 days**. Write a loop that starts from a fixed date (LocalDate) and iteratively adds this Period 5 times, printing each resulting date to demonstrate complex periodic date arithmetic.

4. **Duration Formatting and Conversion:** Calculate the Duration between two LocalDateTime points (one 3 days, 5 hours, and 30 minutes in the future). Format and print the duration in two ways: 1. As a standard ISO 8601 string (e.g., "PT77H30M"). 2. As a **human-readable string** ("3 days, 5 hours, and 30 minutes"), performing the conversion manually using Duration.to...() methods.

5. **Epoch Time and Timestamp Reconciliation:** Write a method that takes a **long** representing milliseconds since the Unix epoch. Convert this to a ZonedDateTime in the system's default time zone. Then, convert that ZonedDateTime back to a long timestamp and verify that it matches the original input, demonstrating accurate handling of the epoch.

---

6. Arrays, Lists, Set, Maps

1. **Concurrent Map and Atomic Updates:** Use a **ConcurrentHashMap** to maintain a score counter for a set of users. Implement a method, incrementScore(String userId, int points), that uses the atomic operation merge() or compute() to safely and efficiently increment the score for a given user, handling the case where the user key may not exist initially.

2. **LinkedHashSet as LRU Cache Base:** Use a **LinkedHashSet** to maintain a small cache of recently accessed items (e.g., max size 5). Implement a method, accessItem(String item), that adds the item to the set. If the set size exceeds 5, the **oldest** item (the one at the beginning of the set) must be removed. This simulates a basic Least Recently Used (LRU) behavior.

3. **Multidimensional Array for Sparse Matrix:** Model a sparse matrix (mostly zero values) using a **Map<Point, Double>**, where Point is a custom immutable class storing row and column indices. Implement a method multiply(Map<Point, Double> matrixA, Map<Point, Double> matrixB) that performs matrix multiplication, taking advantage of the sparse representation to only iterate over non-zero elements.

4. **Priority Queue as a K-Smallest Finder:** Use a **PriorityQueue** to efficiently find the **K smallest elements** (e.g., $K=10$) from a large, unsorted array of one million integers. The solution must ensure that the PriorityQueue size never exceeds K, demonstrating the optimal use of a maxheap approach for this problem.

5. **Bi-directional Map Implementation:** Implement a simple class, BiMap<K, V>, using two internal HashMap instances (one for K to V, one for V to K). Implement put(K key, V value), getKey(V value),

and getValue(K key). Ensure that when a key/value pair is added, it is correctly mapped in *both* directions, and that removing a key removes both entries.

---

7. Functional Interfaces, Lambda Expressions, Method References

1. **Custom Generic Functional Interface (TriFunction):** Define a generic functional interface, TriFunction<T, U, V, R>, which takes three arguments of types T, U, V and returns a result of type R. Write a lambda expression that implements this interface to calculate the net price of a product, taking basePrice (Double), taxRate (Double), and discount (Double) as inputs.
2. **Constructor Reference for Factory:** Define an interface Supplier<T>. Create a class Product with a single-argument constructor (String name). Use a **constructor reference** (Product::new) to implement the Supplier interface within a factory method, createProductFactory(String name), which returns a function that creates a new Product with the given name upon invocation.
3. **Method Reference to an Instance Method of an Arbitrary Object:** Given a List<String> of names, use a **method reference to an instance method of an arbitrary object of a particular type** (String::length) within a Stream operation to map the list of strings to a list of their lengths.
4. **Currying with Lambdas:** Implement a **curried function** using nested lambda expressions to calculate compound interest. The first lambda should accept the principal, the second should accept the rate, and the third should accept the time, returning the final amount. Demonstrate calling the curried function step-by-step.
5. **Lambda for Custom Comparator Chain:** Given a List<Person> (fields: lastName, firstName, age). Use **chained lambda expressions** with Comparator.comparing() and thenComparing() to sort the list. The primary sort must be by **last name** (ascending), followed by **age** (descending), and finally by **first name** (ascending).

---

8. Stream API

1. **Custom Collector (Averaging with Error):** Implement a **custom Collector<T, A, R>** to calculate the **average** of a stream of Double values. The collector must be implemented from scratch (i.e., not using Collectors.averagingDouble). The final result should be wrapped in an Optional<Double> to handle an empty stream gracefully.
2. **Stream of Streams (FlatMapping for Dependencies):** Given a Map<String, List<String>> representing software module dependencies (Key: Module, Value: List of dependencies). Use the **flatMap** operation to create a single Set<String> of **all unique dependency module names** across the entire system.
3. **Reducing for Complex Aggregation:** Use the **reduce()** operation to find the combined total profit from a List<Sale> objects (fields: productName, quantity, unitProfit). The reduction must

simultaneously track the total profit and the product name that contributed the **single highest profit** amount in a custom immutable result object.

4. **Parallel Stream Performance Measurement:** Write a program to compare the performance of a sequential stream versus a **parallel stream** for a CPU-intensive task (e.g., calculating the square root of 10 million random numbers). Measure the execution time for both approaches using System.nanoTime() and provide a conclusion about the parallel speedup.

5. **Infinite Stream Generation and Limiting:** Use Stream.iterate() or Stream.generate() to create an **infinite stream** of pseudo-random but predictable numbers (e.g., a Fibonacci sequence or a simple arithmetic progression). Apply skip() and **limit()** operations to extract a specific sub-sequence of 10 elements starting from the 100th element.

---

9.   Implementing Popularly Used Patterns (Builder, Factory, Strategy, Observer)

1. **Fluent Builder with Inheritance:** Implement the **Builder Pattern** for a complex class hierarchy: Vehicle (base) and Car (extends Vehicle). Design the Builders (VehicleBuilder and CarBuilder) to support a **fluent interface** where the CarBuilder inherits the build methods from VehicleBuilder and correctly returns the specific CarBuilder type after calling inherited methods (using generics and self-type idiom).

2. **Abstract Factory for Cross-Platform UI:** Implement the **Abstract Factory Pattern** to create families of related UI components: Button and Checkbox. Define two concrete factories: WindowsFactory and MacOSFactory. The client code should only interact with the abstract factory interface, allowing the entire UI theme to be swapped by changing a single factory instance.

3. **Context-Agnostic Strategy:** Implement the **Strategy Pattern** for data validation. Create a ValidationStrategy interface (validate(String data)) and strategies like EmailValidation and PasswordValidation. The context class, UserValidator, should hold and execute the strategy. Critically, ensure the strategies are **stateless** and can be reused globally, passed into the validator as a functional interface instance if possible.

4. **Event-Driven Observer with Loose Coupling:** Implement the **Observer Pattern** for a news feed system. The NewsAgency (Subject) notifies multiple NewsSubscriber (Observer) objects. Enforce extremely loose coupling by passing the update message as a **generic Object** to the observer's update method, forcing observers to check the type and cast, simulating a highly flexible (though less type-safe) event notification mechanism.

5. **Factory Method with Dependency Injection:** Implement the **Factory Method Pattern** where an abstract DocumentFactory defines the createDocument() method. Concrete factories (WordFactory, PdfFactory) override this method. The Document object being created has a mandatory dependency (Logger). Ensure the Factory is responsible for **injecting** the required Logger dependency into the newly created Document object before returning it.

---

10.   Implementing Popularly Used Patterns (Decorator, Command, Adapter and Façade)

1. **Layered Decorator for Data Encryption/Compression:** Implement the **Decorator Pattern** for a DataSource interface (writeData(String data)). Implement a FileDataSource (base component). Then, create two decorators: CompressionDecorator and EncryptionDecorator. Demonstrate the ability to stack these decorators (e.g., **new CompressionDecorator(new EncryptionDecorator(new FileDataSource())))**, ensuring the order of operations is correctly applied (e.g., Encrypt then Compress).
2. **Undoable Command Pattern:** Implement the **Command Pattern** for text editor operations (InsertCommand, DeleteCommand). The TextEditorInvoker should maintain a history of commands. Implement an undo() method for each command that reverts the last action. The Invoker should use a stack to manage and execute the undo() operations in reverse order.
3. **Adapter for Third-Party Library Integration:** You are required to use a legacy third-party logging library, LegacyLogger (method: logMessage(String msg, int level)). Your current system uses a modern interface, Appender (method: append(String message)). Implement the **Adapter Pattern** (LegacyLoggerAdapter) to wrap the LegacyLogger and make it compatible with the Appender interface, mapping the simple append() call to the more complex logMessage() call with a fixed default level.
4. **Façade for Subsystem Orchestration:** Design a complex OrderFulfillmentSystem consisting of three subsystems: InventoryService, PaymentGateway, and ShippingService. Create a **FulfillmentFacade** with a single method, placeOrder(Order order). This method must orchestrate all three subsystem calls (check inventory, process payment, schedule shipping) and handle the overall success/failure, hiding the complexity from the client code.
5. **Two-Way Adapter:** Implement a **Two-Way Adapter** between two incompatible interfaces: EuropeanSocket (method: getVolt230()) and AmericanOutlet (method: getVolt110()). Create an adapter class that can be used *both* as a EuropeanSocket (converting 110V to 230V) and as an AmericanOutlet (converting 230V to 110V), using internal logic to perform the voltage conversion in both directions.

---

11. JUnit Testing

1. Simple Calculator Class Testing

Create a class named SimpleCalculator with four basic methods: add(int a, int b), subtract(int a, int b), multiply(int a, int b), and divide(int a, int b).

- **Assignment:** Write a **JUnit test class** for SimpleCalculator.
    - Include at least one test method for each of the four operations.
    - For the divide method, include a test case that specifically checks if dividing by **zero** throws the expected ArithmeticException.

2. String Utility Method Tests

Create a class named StringUtility with the following methods:

- isPalindrome(String s): Returns true if the string is a palindrome (reads the same forwards and backward), ignoring case.

- reverse(String s): Returns the reverse of the input string.

- **Assignment:** Write a **JUnit test class** for StringUtility.

  o For isPalindrome, test cases should include an even-length palindrome, an odd-length palindrome, a non-palindrome, and an empty string.

  o For reverse, test cases should include a normal string and an empty string.

3. Basic Account Class Transaction Testing

Create a class named BankAccount with an instance variable for the balance (a double). Include the following methods:

- deposit(double amount): Adds the amount to the balance.

- withdraw(double amount): Subtracts the amount from the balance, but only if the balance is sufficient. Returns true on success, false otherwise.

- getBalance(): Returns the current balance.

- **Assignment:** Write a **JUnit test class** for BankAccount.

  o Use the @BeforeEach annotation to ensure a new BankAccount instance is created with an initial balance of $100.00 before every test.

  o Write tests for successful deposits and withdrawals, and a test for a failed withdrawal (insufficient funds).

  o Use assertions to check the **final balance** after each transaction.

4. Array Operations Class Testing

Create a class named ArrayUtils with a static method:

- findMax(int[] array): Returns the largest integer in the array.

- **Assignment:** Write a **JUnit test class** for ArrayUtils.

  o Write test cases for an array with positive numbers, an array with negative numbers, and an array containing both.

  o Include a specific test case that checks for the expected behavior (e.g., throwing a NullPointerException or returning a defined value like Integer.MIN_VALUE) when the input array is **null** or **empty**. (The student must decide and implement the expected behavior first).

5. Email Validator Format Test

Create a class named EmailValidator with a static method:

- isValid(String email): Returns true if the string follows a very basic email format (contains exactly one '@' symbol and at least one '.' after the '@').

- **Assignment:** Write a **JUnit test class** for EmailValidator.

  - Test cases should include a **valid** email format.

  - Test cases should include several **invalid** formats, such as: no '@' symbol, multiple '@' symbols, no '.' after the '@', and an empty string.