CODES  
<https://github.com/JasimAbdul-rahman/EXAMINATION->

QUESTION 1

a)

1. Class: Passenger Request

Primary Responsibility: To encapsulate the details of a transportation request, including the pickup location, destination, and number of passengers.

Collaborator Class: Booking Request (receives the request)

1. Class: Vehicle

Primary Responsibility: To represent a taxi or shuttle and manage its current status (available, and route to pick up, transporting passengers).

Collaborator Class: Booking Request (assigned to requests), Driver (updates status)

1. Class: Booking Request

Primary Responsibility: To receive passenger requests and assign available vehicles to them.

Collaborator Class: Passenger Request (receives from), Vehicle (assigns to)

1. Class: Driver

Primary Responsibility: To operate the vehicle, notify the company of status changes (arrival at pickup, drop-off), and execute the transportation.

Collaborator Class: Vehicle (updates status of), Company (notifies)

1. Class: Company ( "BALEXTRANIT (U) LTD")

Primary Responsibility: To manage the overall transportation operations, track vehicle time, and monitor lost fares.

Collaborator Class: Booking Request (receives reports from), Vehicle (tracks status of), Lost Fare Tracker (collaborates to track lost fares)

b) and c)

d)

Encapsulation is one of the fundamental principles of Object-Oriented Programming (OOP). It refers to the bundling of data (attributes) and the methods (functions) that operate on that data into a single unit, or class. More importantly, it involves restricting direct access to some of an object's components and providing a controlled interface for interaction. This is typically achieved through access modifiers like private and public.

Vehicle class:

1. Data Hiding (Using private access modifier):

All the attributes of the Vehicle class – vehicleId, type, capacity, status, and currentLocation – are declared as private.

This means that these attributes cannot be directly accessed or modified from outside the Vehicle class. For example, you cannot write myVehicle.status = VehicleStatus.BROKEN; directly from another class like Company or PassengerSource. This protects the internal state of the Vehicle object from unintended or invalid changes.

1. Controlled Access (Using public getter and setter methods):

Instead of direct access, the Vehicle class provides public methods (getters and setters) to interact with its private data.

Getters (for example ; getVehicleId(), getStatus()) allow other classes to *read* the value of an attribute.

Setters (for example ; setStatus(), setCurrentLocation()) allow other classes to *modify* the value of an attribute, but this modification can be controlled. For instance, setStatus allows changing the status, but the methods like notifyArrivedAtPickup() and notifyDroppedOffPassengers() demonstrate how internal logic can dictate *when* and *how* a status change occurs (for example ; a vehicle can only notifyArrivedAtPickup if it was EN\_ROUTE\_TO\_PICKUP). This ensures that the object remains in a valid state.

1. Bundling (Combining Data and Behavior):

The Vehicle class bundles all the data related to a vehicle (vehicleId, type, status, etc.) together with all the operations that can be performed on that data (for example ; setStatus, notifyArrivedAtPickup, notifyDroppedOffPassengers). This creates a self-contained unit.

Benefits of Encapsulation in Vehicle:

Data Integrity: By making attributes private, we prevent external code from putting the Vehicle object into an inconsistent or invalid state (for example ; setting a negative capacity, or setting a status that doesn't make sense in the operational flow).

Flexibility and Maintainability: If we decide to change how a vehicle's status is internally represented (for example ; from an enum to an integer code), we only need to modify the Vehicle class. Other classes that interact with Vehicle via its public methods (getStatus(), setStatus()) don't need to change, as their interface remains the same. This reduces coupling between classes.

* Reduced Complexity: External classes don't need to know the intricate details of how Vehicle manages its internal state; they just need to know how to use its public methods. This simplifies the overall system design.

QUESTION 2

a)

In Java, String immutability means that once a String object is created, its content cannot be changed. Any operation that appears to modify a String (like toUpperCase() or concatenation) actually results in the creation of a *new* String object, leaving the original unchanged.

This concept significantly aids the National Library of Uganda's (NLU) system in two main ways:

1. Efficiency:

String Pool: For frequently queried book titles and author names (for example ; "Things Fall Apart"), Java reuses existing String objects from a special "String Pool" instead of creating new ones every time. This reduces memory consumption and improves performance by avoiding redundant object creation.

Cached Hash Code: Since a String's content never changes, its hashCode() is computed only once and cached. This ensures fast lookups when titles or author names are used as keys in data structures like HashMap for quick retrieval.

1. Security:

Thread Safety: As String objects cannot be modified, multiple parts of the NLU system (for example ; concurrent patron queries) can safely access and read the same book title or author name without any risk of data corruption or needing complex synchronization.

Data Integrity: Immutability guarantees that once a book title or author name is validated and accepted into the system, its content cannot be altered by malicious injection attempts (like SQL injection) or accidental modification later in the processing pipeline, ensuring the reliability and trustworthiness of the metadata.

b)

Method 1: Using a String Literal ""

This is the most common, concise, and generally preferred way to create an empty string. When you use "", Java often reuses an existing empty string object from the String pool, making it very efficient.

Syntax:

Java

String emptyString = "";

Method 2: Using the new String() Constructor

You can explicitly create a new String object using its constructor without any arguments. This will create a new, distinct String object in memory that represents an empty sequence of characters. While it works, it's generally less efficient than using the literal "" because it bypasses the String pool, always creating a new object.

Syntax:

Java

String emptyString = new String();

c)

1. equalsIgnoreCase() Method

What it does: This method compares two strings for equality, ignoring case differences. It returns true if the two strings are identical in terms of characters, regardless of whether they are uppercase or lowercase.

Syntax Example:

Java

String userQuery = "the lord of the rings";

String bookTitleInCatalog = "The Lord Of The Rings";

boolean match = userQuery.equalsIgnoreCase(bookTitleInCatalog); // This will be true

How it improves NLU search:

Direct Comparison: When a patron submits a query, the system can directly compare the patron's input with the stored book titles and author names using equalsIgnoreCase().

User Convenience: This immediately solves the problem of case sensitivity. The patron doesn't have to worry about typing the exact capitalization. "harry potter" will match "Harry Potter", "HARRY POTTER", etc.

Simplicity: It's a very straightforward and readable way to perform case-insensitive comparisons without needing to modify either string beforehand.

2. toLowerCase() Method

What it does: This method converts all the characters in a string to their lowercase equivalents. It returns a *new* string with all characters in lowercase. The original string remains unchanged (due to immutability).

Syntax Example:

Java

String userQuery = "THE LORD OF THE RINGS";

String bookTitleInCatalog = "The Lord Of The Rings";

String normalizedUserQuery = userQuery.toLowerCase(); // "the lord of the rings"

String normalizedBookTitle = bookTitleInCatalog.toLowerCase(); // "the lord of the rings"

boolean match = normalizedUserQuery.equals(normalizedBookTitle); // This will be true

How it improves NLU search:

Normalization for Indexing and Broader Searches: While equalsIgnoreCase() is great for direct comparisons, toLowerCase() is powerful for normalizing data, especially when building search indexes or performing more complex, multi-word searches.

Consistent Data: The NLU system can store all book titles and author names in a consistent lowercase format in its search index. When a patron queries, their input is also converted to lowercase. This creates a uniform basis for comparison.

Partial Matches/Tokenization: If the system breaks down queries into individual words (tokens), converting both the query and the indexed data to lowercase ensures that "Potter" will match "potter" regardless of how it was originally capitalized in the catalog or the query. This is crucial for matching keywords.

Efficiency for Repeated Comparisons: If a single user query needs to be compared against *many* book titles (for example ; searching the entire catalog), converting the user query to lowercase *once* at the beginning and then using regular equals() for subsequent comparisons might be slightly more efficient than calling equalsIgnoreCase() repeatedly, especially if the underlying implementation of equalsIgnoreCase() internally performs a conversion for each comparison.

d)

Line 1: System.out.println(author1 == author3);

author1 = "Ainebyoona"; : This creates a String literal. Java's String Pool mechanism checks if "Ainebyoona" already exists in the pool. If not, it creates it in the pool, and author1 points to this object.

author3 = new String("Ainebyoona"); : This explicitly uses the new keyword. The new keyword *always* creates a brand-new String object in the heap memory, even if an identical string ("Ainebyoona") already exists in the String Pool. So, author3 points to a *different* memory location than author1.

author1 == author3:

The == operator in Java is used for reference comparison (or "identity comparison") for objects. It checks if two object references point to the *exact same memory location*.

Since author1 points to the object in the String Pool and author3 points to a newly created object in the heap, they are at different memory addresses.

Output: false

Line 2: System.out.println(author1.equalsIgnoreCase(author2));

author1 = "Ainebyoona”; Points to the string literal "Ainebyoona".

author2 = "ainebyoona”; Points to the string literal "ainebyoona". This is a different literal, so it's a different object in the String Pool from "Ainebyoona".

author1.equalsIgnoreCase(author2):

The equalsIgnoreCase() method is a String class method used for content comparison (or "value comparison"). It checks if the sequence of characters in one string is the same as in another string, ignoring case differences.

"Ainebyoona" and "ainebyoona" have the same characters in the same order, only differing in case.

Output: true

How Object Comparison Works (Key Takeaways):

1. == Operator (Reference Comparison):

For primitive types (like int, char, boolean), == compares their actual values.

For objects (like String, Book, Location), == compares the memory addresses (references) where the objects are stored. It tells you if two variables are pointing to the *exact same object* in memory. It does not look at the content of the objects.

1. .equals() Method (Content Comparison):

For objects, the equals() method (which is inherited from the Object class but often overridden by specific classes like String) is used to compare the content or value of two objects.

The String class specifically overrides equals() to compare the sequence of characters.

equalsIgnoreCase() is a specialized version of equals() that performs content comparison while ignoring case.

In essence:

Use == when you need to know if two variables refer to the *very same instance* of an object.

Use .equals() (or equalsIgnoreCase() for strings) when you need to know if two objects have the *same value or content*, regardless of whether they are the same instance in memory.

For comparing String objects, you should almost always use .equals() or equalsIgnoreCase() to compare their content, as comparing references with == often leads to unexpected results due to String Pool behavior and explicit object creation.

SECTION B

QUESTION 5

a)

1. Encapsulation

Encapsulation is the bundling of data (attributes) and the methods (behaviors) that operate on that data within a single unit (a class). It also involves data hiding, meaning the internal state of an object is kept private and cannot be directly accessed from outside the class. Access is only allowed through public methods (getters and setters).

Application in EFRIS:

Transaction Class: A Transaction class would encapsulate details like transactionId, amount, taxRate, timestamp, and status. These attributes would be private. Public methods like getAmount(), calculateTax(), and updateStatus() would provide controlled access and operations. This prevents direct manipulation of a transaction's amount or tax rate from outside, ensuring data integrity and preventing fraud.

TaxPayer Class: Encapsulates TIN (Tax Identification Number), name, address, and taxObligations. The TIN would be private, accessible only via a getter, ensuring it's not accidentally changed.

2. Abstraction

Abstraction focuses on showing only the essential features of an object or system while hiding the complex implementation details. It allows you to define "what" an object does rather than "how" it does it. This is often achieved using abstract classes and interfaces.

Application in EFRIS:

TaxableEntity Interface: URA deals with various types of taxable entities (for example ; IndividualTaxPayer, CorporateTaxPayer, Partnership). An ITaxableEntity interface could define common methods like calculateAnnualTax(), submitReturn(), or generateTaxStatement().

Different types of taxpayers would implement this interface, each with their own specific tax calculation logic, but the EFRIS system interacts with them uniformly through the ITaxableEntity interface, without needing to know the complex tax rules for each specific entity type. This simplifies the system's core logic.

3. Inheritance

Inheritance is a mechanism where one class (the "child" or "subclass") acquires the properties (attributes) and behaviors (methods) of another class (the "parent" or "superclass"). It promotes code reusability and establishes an "is-a" relationship.

Application in EFRIS:

Invoice and Receipt: Both invoices and receipts share common properties (for example ; documentId, date, totalAmount, items). A base FiscalDocument class could contain these common attributes and methods.

Invoice and Receipt classes could then extend FiscalDocument, inheriting the common features and adding their specific functionalities (for example ; Invoice might have paymentTerms, Receipt might have paymentMethod). This avoids code duplication and ensures consistency across different fiscal documents.

4. Polymorphism

Polymorphism means "many forms." It allows objects of different classes to be treated as objects of a common type. This is often achieved through method overriding (subclasses providing their own implementation of a method defined in a superclass/interface) and method overloading (multiple methods with the same name but different parameters).

Application in EFRIS:

calculateTax() Method: Building on the ITaxableEntity interface, the EFRIS system might call taxableEntity.calculateAnnualTax() for various taxpayers.

An IndividualTaxPayer object would execute its specific calculateAnnualTax() logic, while a CorporateTaxPayer object would execute *its* specific calculateAnnualTax() logic, even though the system calls the same method name. This allows the system to process different types of taxpayers uniformly through a single interface, making the code flexible and extensible for new tax categories without modifying existing code.

5. Composition (or Association/Aggregation, as an alternative to Inheritance for "has-a" relationships)

While not always listed as one of the "core five" alongside the classic four (Encapsulation, Abstraction, Inheritance, Polymorphism), Composition is crucial for building robust systems. It's a "has-a" relationship where one class contains an object of another class as a part of its state. It promotes loose coupling.

Application in EFRIS:

Invoice and LineItem: An Invoice doesn't *inherit* from LineItem; rather, an Invoice *has* multiple LineItem objects (for example ; a List<LineItem>). Each LineItem would encapsulate details like productName, quantity, unitPrice, and taxApplicable.

This allows for flexible invoice structures where an invoice can contain any number of line items. It also means changes to how a LineItem is structured don't necessarily impact the core Invoice class, promoting modularity and maintainability. Similarly, a business object might *have* a List<TaxPayer>, or a TaxPayer might *have* a List<Transaction>.

By applying these OOP principles, the URA's EFRIS system can manage complexity, ensure data security, easily adapt to changing tax laws (flexibility), and handle a growing number of transactions and taxpayers (scalability).

b)

Encapsulation is a core Object-Oriented Programming (OOP) principle that involves bundling data (attributes) and the methods (behaviors) that operate on that data into a single unit, a class. Crucially, it also involves data hiding, meaning the internal state of an object (its attributes) is kept private and cannot be directly accessed or modified from outside the class. Instead, interaction with the object's data is only allowed through a controlled public interface, typically using getter and setter methods.

In the URA's EFRIS system, which deals with highly sensitive financial and taxpayer information, encapsulation is vital for protecting data integrity, preventing unauthorized access, and ensuring the system's security and reliability.

Explanation of Protection through Encapsulation:

1. Data Hiding (private fields):

Sensitive attributes like buyerTIN, sellerTIN, invoiceAmount, and transactionTimestamp are declared as private.

This means no other class in the EFRIS system (for example ; a reporting module, a user interface component) can directly access or modify these fields using dot notation (for example ; myRecord.invoiceAmount = -100; is forbidden).

This prevents accidental corruption, unauthorized alteration, or malicious manipulation of critical transaction data. A developer cannot bypass the defined rules by directly changing the internal state.

1. Controlled Initialization (Constructor):

The TransactionRecord constructor is the primary, controlled way to create a new transaction. It requires all essential sensitive details to be provided at creation time.

This ensures that a TransactionRecord object is always created in a valid and complete state. Important data like TINs cannot be omitted, and basic validation (for example ; invoiceAmount must be positive) can be enforced immediately.

1. Controlled Read Access (public getter methods):

Public getter methods (for example ; getBuyerTIN(), getInvoiceAmount()) are provided to allow other parts of the EFRIS system to *read* the sensitive data.

While allowing read access, the getters prevent external code from knowing or relying on the *internal representation* of the data. For example, if invoiceAmount was later stored as BigDecimal instead of double, only the getInvoiceAmount() method would need to change, not every piece of code that reads the amount. This provides flexibility for future changes without breaking external dependencies.

1. Controlled Write Access (public setter methods or lack thereof):

For fields that might legitimately change (like status or potentially invoiceAmount if adjustments are allowed), public setter methods (for example ; setInvoiceAmount(), setStatus()) are provided.

These setters are not just direct assignments. They can (and in a real system, *would*) include validation logic. For instance, setInvoiceAmount() checks if the new amount is positive. setStatus() could enforce valid state transitions (for example ; a transaction cannot go directly from "PENDING" to "COMPLETED" without passing through "APPROVED").

Crucially, for highly sensitive and immutable fields like buyerTIN, sellerTIN, and transactionTimestamp, no public setter methods are provided at all. This ensures that once these details are set during object creation, they cannot be modified, guaranteeing their immutability and preventing any tampering after the transaction is recorded.

In summary, encapsulation in the TransactionRecord class (and throughout the EFRIS system) creates a robust barrier around sensitive data. It dictates *how* and *when* data can be accessed and modified, thereby safeguarding the integrity and confidentiality of critical transaction details, which is paramount for a tax authority like URA.

c)

In the URA's EFRIS system, polymorphism is incredibly powerful for handling different VAT calculation logic across various taxpayer categories like Retailers, Wholesalers, Manufacturers, and Importers. Here's how it works:

1. Establishing a Common Contract (The "What"):

The EFRIS system would first define a standard "contract" for VAT calculation. This is typically an interface or an abstract class. This contract would declare a method, let's say calculateVAT(), which specifies what needs to be done (calculate the VAT for a given transaction amount), but it doesn't provide any specific instructions on how to do it. It's like saying, "Every taxpayer category must know how to calculate VAT," without dictating the exact formula.

1. Specialized Implementations (The "How"):

Each distinct taxpayer category (Retailer, Wholesaler, Manufacturer, Importer) then becomes a concrete class in the EFRIS system. Each of these classes implements that common calculateVAT() contract, but they provide their own unique, specific logic for how they perform the calculation.

A Retailer class would implement calculateVAT() based on the standard retail VAT rate.

A Wholesaler class would implement calculateVAT() according to wholesale-specific VAT rules, which might differ.

A Manufacturer class would have its calculateVAT() method incorporate rules for input tax credits or different manufacturing-specific rates.

An Importer class would implement calculateVAT() based on import duties and specific import VAT regulations.

So, while they all have a calculateVAT() method, the internal steps and formulas within each method are tailored to that specific category.

1. Uniform Interaction (The "One Way to Talk"):

When the EFRIS system needs to calculate VAT for a transaction, it doesn't need to know the precise type of taxpayer involved (for example ; whether it's a Retailer object or an Importer object). Instead, the system simply holds a reference to the common "contract" type. It can then call the calculateVAT() method on that common type.

Because of polymorphism, the Java system automatically determines the actual type of the object at runtime and executes the specific calculateVAT() implementation belonging to that particular taxpayer category. It's like having a "Calculate VAT" button that, when pressed, automatically knows whether to apply the Retailer's formula, the Wholesaler's formula, and so on, without the main system needing to check.

Benefits for the EFRIS System:

Flexibility and Extensibility: If URA introduces a new taxpayer category in the future, a new class for that category can be added, implementing the same calculateVAT() contract. The existing core EFRIS processing logic does not need to be modified to accommodate the new type, making the system highly adaptable to changing tax laws.

Simplified Code: It eliminates the need for complex if-else if or switch statements that would otherwise be required to check the taxpayer's type and manually select the correct calculation logic. This results in cleaner, more readable, and less error-prone code.

Maintainability: Changes to a specific taxpayer's VAT calculation rules are isolated within that taxpayer's dedicated class. This makes updates easier and reduces the risk of introducing bugs into other parts of the system.

QUESTION 3

a)

In Java, an Exception is an event that disrupts the normal flow of a program's instructions. When an exceptional event occurs, it means something unexpected or erroneous has happened during the execution of your code. If not handled, an exception will typically cause the program to terminate abruptly, which is undesirable for any stable application.

Think of it like this: You're driving a car (your program is running). Normally, you follow the road. An exception is like hitting a sudden, unexpected roadblock (for example ; a tire blows out, you run out of gas, or the road simply ends). If you don't have a plan for these roadblocks, your journey (program) stops right there.

Examples of Exceptions:

Null Pointer Exception: Trying to use an object reference that points to nothing (null).

Array Index Out Of Bounds Exception: Trying to access an array element at an index that doesn't exist.

File Not Found Exception: Trying to open a file that doesn't exist on the disk.

Arithmetic Exception: Performing an illegal arithmetic operation, like dividing by zero.

Exception Handling is the mechanism in Java that allows a program to gracefully deal with these exceptional events. Instead of crashing, the program can "catch" the exception, execute specific code to recover or respond to the error, and then potentially continue its execution.

The core constructs for exception handling in Java are:

1. try block: This block contains the code that might throw an exception.
2. catch block: This block immediately follows a try block. It contains the code that will be executed if a specific type of exception (declared in the catch clause) is thrown within the preceding try block.
3. finally block (Optional): This block always executes, regardless of whether an exception was thrown or caught. It's typically used for cleanup operations, like closing files or network connections.
4. throw keyword: Used to explicitly throw an exception from within your code.
5. throws keyword: Used in a method signature to declare that a method *might* throw a certain type of exception, indicating to callers that they need to handle it.

How it Aids the AirQO Dashboard:

The AirQO dashboard for "Kampala - Smart City" would heavily rely on exception handling to be robust:

Data Retrieval: When fetching air quality data from sensors or a server, network issues (IO Exception), server timeouts (Socket Timeout Exception), or malformed data (Parse Exception) can occur. Exception handling ensures the dashboard doesn't crash but can display an "Error fetching data" message or retry.

User Input: If a user enters invalid criteria for a search (for example ; non-numeric input where a number is expected), Input Mismatch Exception could be thrown. Handling this allows the dashboard to prompt the user to re-enter valid data instead of terminating.

File Operations: If the dashboard needs to save logs or configuration to a file, FileNotFoundException or IO Exception might occur. finally blocks ensure files are closed even if errors happen.

b)

1. Mathematical Impossibility: Division by zero is a mathematically undefined operation. There is no finite number that, when multiplied by zero, results in a non-zero dividend.
2. JVM Detection: The Java Virtual Machine (JVM) detects this illegal arithmetic operation at runtime.
3. Runtime Error: Since it's an error that occurs during program execution (not during compilation), it's a runtime exception.
4. Arithmetic Exception Class: Java provides the Arithmetic Exception class specifically for this type of error and other arithmetic problems (though division by zero is the most common trigger for it). When the JVM encounters a division by zero, it creates an instance of Arithmetic Exception and "throws" it.

If this Arithmetic Exception is not caught and handled by the program using try-catch blocks, the program will terminate abruptly, which would be a significant issue for a critical system like the AirQO air quality monitoring dashboard.

c)

Both while and for loops are used to execute a block of code repeatedly. The primary distinction lies in their structure and typical use cases, particularly regarding when the number of iterations is known.

1. for Loop:

It's designed for situations where the number of iterations is known or easily determinable at the beginning of the loop. It explicitly provides sections for initialization, condition checking, and iteration (increment/decrement) all in one line.

Syntax:

Java

for (initialization; condition; increment/decrement) {

}

Flow:

Executed once at the very beginning.

Evaluated before each iteration. If true, the loop body executes; otherwise, the loop terminates.

Executed at the end of each iteration, after the loop body.

You need to iterate a fixed number of times, or over a range of numbers, or through elements in an array/collection where the count is clear.

1. while Loop:

It's designed for situations where the number of iterations is unknown and depends on a certain condition being met. It only explicitly provides a condition. Initialization and iteration steps must be managed outside and inside the loop body, respectively.

Syntax:

Java

initialization;

while (condition) {

increment/decrement;

}

Flow:

Evaluated before each iteration. If true, the loop body executes; otherwise, the loop terminates.

You need to repeat an action until a specific condition becomes false, such as reading input until a certain value is entered, or processing data until a flag changes.