Lecture helper

1> Java Data types:

a. Primitive Data Types:

1. **byte**
   - **Size**: 1 byte (8 bits)
   - **Range**: -128 to 127
1. **short**
   - **Size**: 2 bytes (16 bits)
   - **Range**: -32,768 to 32,767
1. **int**
   - **Size**: 4 bytes (32 bits)
   - **Range**: -2,147,483,648 to 2,147,483,647
1. **long**
   - **Size**: 8 bytes (64 bits)
   - **Range**: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
1. **float**
   - **Size**: 4 bytes (32 bits)
   - **Range**: Approximately ±3.40282347E+38F (single-precision floating point)
1. **double**
   - **Size**: 8 bytes (64 bits)
   - **Range**: Approximately ±1.79769313486231570E+308 (double-precision floating point)
1. **char**
   - **Size**: 2 bytes (16 bits)
   - **Range**: 0 to 65,535 (represents a single Unicode character)
1. **boolean**
   - **Size**: Not precisely defined (depends on the JVM, typically represented as 1 bit, but might use a byte for simplicity)
   - **Values**: true or false

b. Non-Primitive Data Types:

1. String
2. Arrays
3. Objects/Classes

Hands On:

Create variables for user details (e.g., String userName = "John Doe";, int userId = 12345;).
Create a method addTransaction(double amount) that updates the wallet balance, checks for valid transactions using boolean, and logs results using String.

c. Type Inference

Type inference in Java 10 refers to the ability of the compiler to automatically determine the type of a variable based on the context of its assignment.JAVA 10.

```
public class TypeInferenceExample {
    public static void main(String[] args) {
        var greeting = "Hello, Java 10!"; // Inferred as String
        var count = 100;                  // Inferred as int
        var list = List.of("Java", "Kotlin", "Scala"); // Inferred as List<String>

        // Iterating using var
```

```
        for (var item : list) {
            System.out.println(item); // Compiler knows item is of type String
        }
    }
}
```

d. Disadvantages of using primitive arrays and Using Lists and ArrayLists.

1. **Fixed Size**:
- **Static Length**: Once an array is created, its size is fixed and cannot be modified. This inflexibility can lead to wasted memory if the array is too large or the need to recreate and copy elements to a new array if it's too small.
2. **Limited Functionality**:
- **Basic Operations Only**: Primitive arrays do not have built-in methods for common operations such as adding, removing, or searching for elements. You have to implement these operations manually, which can be error-prone and time-consuming.

1. **Dynamic Size**:
- **Automatic Resizing**: ArrayList automatically resizes when elements are added or removed, making it more flexible and adaptable to changing data requirements.
2. **Built-in Methods**:
- **Ease of Use**: List and ArrayList come with a wide range of built-in methods like add(), remove(), contains(), and indexOf() that simplify common operations and improve productivity.

```
// Primitive Array
int[] numbers = new int[5]; // Fixed size
numbers[0] = 10; // Manual assignment

// ArrayList
ArrayList<Integer> numberList = new ArrayList<>(); // Dynamic size
numberList.add(10); // Easier to add elements
numberList.add(20);


public class WalletManager {
    public static void main(String[] args) {
        // Create an ArrayList to store wallet transactions
        ArrayList<Double> transactions = new ArrayList<>();

        // Sample transactions for demonstration
        transactions.add(150.0); // Deposit
        transactions.add(-20.0); // Withdrawal
        transactions.add(100.0); // Deposit
        transactions.add(-50.0); // Withdrawal

        // Display all transactions
        System.out.println("Initial Transactions:");
        displayTransactions(transactions);

        // Adding a new transaction (user input)
        Scanner scanner = new Scanner(System.in);
        System.out.println("\nEnter a new transaction (positive for deposit, negative for
withdrawal):");
        double newTransaction = scanner.nextDouble();
        transactions.add(newTransaction);

        // Display updated transactions
        System.out.println("Updated Transactions:");
        displayTransactions(transactions);
```

```
        // Calculate total balance
        double balance = calculateBalance(transactions);
        System.out.println("Current Balance: $" + balance);
    }

    // Method to display all transactions
    public static void displayTransactions(ArrayList<Double> transactions) {
        for (int i = 0; i < transactions.size(); i++) {
            System.out.println("Transaction " + (i + 1) + ": " + (transactions.get(i) >= 0 ? "+$" : "-$") +
Math.abs(transactions.get(i)));
        }
    }

    // Method to calculate the total balance
    public static double calculateBalance(ArrayList<Double> transactions) {
        double total = 0;
        for (double transaction : transactions) {
            total += transaction;
        }
        return total;
    }
}
```

Explain about the System inputs (Scanner, console) and disadvantages,advantaes..

e. Java Loops:

1. for Loop
- **Purpose**: Used when the number of iterations is known beforehand.

2. Enhanced for Loop (For-each Loop)
- **Purpose**: Used to iterate over arrays or collections without dealing with index variables.

3. while Loop
- **Purpose**: Used when the number of iterations is not known in advance and depends on a condition.

4. do-while Loop
- **Purpose**: Similar to the while loop, but ensures that the block of code is executed at least once before checking the condition.

5. for Loop with Multiple Variables
- **Purpose**: Allows using multiple variables in the initialisation and update sections of the loop.

-> Loop Control Statements:

1. Break: Used to exit prematurely when a specified condition is met.
2. continue: Skips the current iteration and proceeds with the next iteration of the loop.
3. Return: Exits the loop and the method in which it resides.

f. Java Conditions:

1. If-else
2. Nested if else
3. Switch

2> Java Error Handling:

a. Key Concepts of Error Handling in Java

1. **Exception**: An event that disrupts the normal flow of the program. Exceptions are objects that describe an error condition.
2. **Checked Exceptions**: Exceptions that are checked at compile-time (e.g., IOException, FileNotFoundException).
3. **Unchecked Exceptions**: Exceptions that are not checked at compile-time (e.g., ArithmeticException, NullPointerException).
4. **Error**: Indicates serious problems that a reasonable application should not try to catch (e.g., OutOfMemoryError).

b. Basic Structure of Error Handling

Java provides try, catch, finally, throw, and throws to handle errors:

- **try block**: Wraps code that might throw an exception.
- **catch block**: Catches and handles exceptions.
- **finally block**: Executes code regardless of whether an exception occurred (e.g., closing resources).
- **throw statement**: Used to explicitly throw an exception.
- **throws keyword**: Indicates that a method may throw exceptions, which must be handled by the caller.

//prime,fibonnaci,handsOn//case - implement a case where you have to take three integers as input and if any number is less than 40 return fail, and If their sum is less than 125 the also fail, else if their sum is more than or eqi

3> Java OOPS: //Give real life example

a. Encapsulation: Encapsulation is the bundling of data (fields) and methods that operate on that data within a class, and restricting access to some components using **access modifiers** (private, protected, public).

The knowledge of scopes in summarised by the knowledge of "**scope**".

Summary of Access Modifiers

Modifier	Same Class	Same Package	Subclass (Different Package)	Other Packages
public	✅	✅	✅	✅
protected	✅	✅	✅	❌
default	✅	✅	❌	❌
private	✅	❌	❌	❌

b. Inheritance: Inheritance allows one class (child) to acquire the properties and methods of another class (parent), enabling code reuse.//same code as encapsulated

c.Polymorphism: Polymorphism allows objects to take on multiple forms. It is achieved through "**method overriding** and **method overloading**."

Overriding:

```
class Wallet {
    public void transaction(double amount) {
```

```
        System.out.println("Processing a transaction of $" + amount);
    }
}

class DigitalWallet extends Wallet {
    @Override
    public void transaction(double amount) {
        System.out.println("Processing a digital wallet transaction of $" + amount);
    }
}

public class Main {
    public static void main(String[] args) {
        Wallet wallet = new Wallet();
        Wallet digitalWallet = new DigitalWallet();  // Polymorphism

        wallet.transaction(50.0);           // Calls parent method
        digitalWallet.transaction(100.0);   // Calls child method
    }
}
```

Overloading:

```
class Calculator {
    // Method Overloading: Same method name, different parameter lists
    public int add(int a, int b) {
        return a + b;
    }

    public double add(double a, double b) {
        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        Calculator calculator = new Calculator();
        System.out.println("Sum of integers: " + calculator.add(5, 10));
        System.out.println("Sum of doubles: " + calculator.add(3.5, 2.5));
    }
}
```

d.Abstraction: Abstraction is the process of hiding implementation details and exposing only the functionality through abstract classes or interfaces.They focus on what and not how. They cannot be instantiated like Payment payment = new Payment()

```
abstract class Payment {
    public abstract void makePayment(double amount);  // Abstract method
}

class CardPayment extends Payment {
    @Override
    public void makePayment(double amount) {
        System.out.println("Payment of $" + amount + " made using Card.");
    }
}

class WalletPayment extends Payment {
    @Override
```

```
    public void makePayment(double amount) {
        System.out.println("Payment of $" + amount + " made using Wallet.");
    }
}

public class Main {
    public static void main(String[] args) {
        Payment payment1 = new CardPayment();
        Payment payment2 = new WalletPayment();

        payment1.makePayment(50.0);
        payment2.makePayment(30.0);
    }
}
```

-> This is where our interface will come into feature, whenever we have a common method or data type to be used across project we define or instantiate it directly here.Concrete implementation of the method is not allowed. Methods and data are implicitly public and abstract

3> Java Classes Type: Regular,Anonymous1,Anonymous2

4> Java Beans Using Class

Hands-on participant case problem

5> Spring and Beans:

Spring implements **IoC**, which shifts the responsibility of creating and managing objects (dependencies) to the framework.
This makes the code loosely coupled, modular, and easier to test.

Create a new maven project