JAVA

HAND - WRITTEN NOTES

Happy Learning !!!

MODULES

Module 1 : Introduction to Java

1.1 What is Java ... ?
1.2 History and Development of Java
1.3 Java Virtual Machine (JVM)
1.4 Setting up the Java ----------

Module 2 : Getting Started with Java

2.1 Basic Structure of a Java Program
2.2 Variables and Data Types
2.3 Input and Output in Java
2.4 Operators in Java

Module 3 : Control Structures

3.1 Conditional Statements
3.2 Loops
3.3 Break and Continue statements
3.4 Handling Exceptions.

Module 4 : Functions and Methods.

4.1 Function Declaration and Definition
4.2 Function Parameters and return value
4.3 Function overloading
4.4 Recursion
4.5 Lambda Fexpression

Module 5 : Object - Oriented in Java

5.1 Classes and Objects
5.2 Inheritance and Polymorphism
5.3 Encapsulation and Access Modifiers
5.4 Abstraction and Interfaces
5.5 Exception handling

Module 6 : Collections and Generics

6.1 Array and ArrayList
6.2 Maps and Set
6.3 Generic classes and methods
6.4 Iterators and foreach loop

Module 7 : Multithreading and Concurrency

7.1 Creating and Managing threads
7.2 Synchronization and locks

7.3 Threads pools and Executors
7.4 Asynchronous programming

Module 8: File I/O and Data Persistence

8.1 Reading and writing Files
8.2 Working with directories
8.3 Serialization and deserialization
8.4 Working with database using JDBC

Module 9:- User Interfaces with JavaFX

9.1 Introduction to JavaFX
9.2 Building uses interfaces with FXML
9.3 Event handling and uses interactions
9.4 JavaFX charts and Multimedia

Module 10:- Advanced Topics

10.1 Reflection and Annotations
10.2 Networking and Socket programming
10.3 Security and cryptography
10.4 Working with external ____
10.5 Design Pattern and Best Practices.

Module - 1 : Intro To Java

1.1 What is Java ???

Java is a versatile, high-level programming language with a wide range of applications. It was created by James Gosling and his team at Sun Microsystems (now Oracle Cor-poration) and First released in 1995.

Java's design philosophy revolves around the idea of "write once, run anywhere," which means that Java programs can be develop on one platform and run on any other platform without modification. This is achieved through the use of Java Virtual Machine (JVM).

Java is known for its simplicity, readability, and strong support for object-oriented prog-ramming. It has a rich ecosystem of libraries and framework, making it a popular choice for a variety of development tasks, including web development, mobile app development, desktop application, and more. Below, we will explore the key features and characterstic of Java.

Key Features of Java :-

1. Platform Independence :- Java Code is Compiled into an intermediate Form as bytecode, which can be executed on any Platform with a compatiable JVM. This ensures that Java applications can run on different operating System without modification.

2. Object-Oriented :- Java is purely object-oriented programming language. It encouroges the use of objects, classes, and inheritance, making it easier to model real-world concepts in code.

3. Robust and Secure :- Java's strict compile-time and runtime checks help catch errors early, reducing the likelihood of crashes and security vulnerabilities.

4. Multi-threading :- Java provide built-in support For multithreading, allowing developers to create concurrent and responsive applications.

5. Rich Standard Library :- Java Comes with a vast standard Library that provides pre-built classes and Functions fora wide range of tasks, Form file I/O to network communication.

1.2 History of Java ...

Java's history dates back to the early 1990s when a team of engineers at Sun Microsystems, led by James Gosling, initiated the development of new programming language. Their goal was to create a language suitable for embedded systems, a domain that was "Oak" but was later renamed to "Java".

1.3 Java Virtual Machine (JVM) ...

The Java Virtual Machine (JVM) is important component of Java's platform independence. It is virtualized execution environment that interprets and run Java bytecode, allowing Java programs to run on any platform that has a compatible JVM. Here's how it works :-

1. Compilation :- Java Source Code is First compiled by the Java Compiler (javac) into bytecode. Bytecode is a low-level representation of the code that is independent of the target platform.

2. Execution:- The bytecode is then executed by the JVM. The JVM takes care of various tasks, including memory management, garbage collection, and just-in-time (JIT) compilation to native code for improved performance.

÷ The JVM also provide a range of services to Java Comilation Applications, including security, class loading, and runtime libraries. It abstracts away the underlying hardware and operating system, ensuring that a Java program behaves consistently on different platform.

1.4 Setting up the Java Environment...

To start developing Java applications, you need to set up a Java development environment. Here are the steps to get you started:-

Step 1: Install Java Development Kit (JDK)

The JDK is a package that includes the Java Compiler (javac), the Java virtual

Machine (JVM), and other development tools. Follow these steps to install the JDK.

- Visit the Oracle or OpenJDK website to download the latest JDK version compatiable with your operating system.
- Follow the installation instruction for your specific platform.
- After installation, open a command prompt or terminal and run "java -version" to verify that the JDK is installed correctly.

Step 2: Install an Integrated Environment (IDE)

While you can write Java Code using a simple text editor and compile it from the command line, using an itegrated Development Environment (IDE) can significantly improve your productivity. Popular Java IDEs include Eclipse, IntelliJ IDEA, and NetBeans.

- Download and install you preferred IDE from the official website
- Launch the IDE and configure it to use the installed JDK.

Step 3: Create an Run First Java Program

Now that your development environment is set up, you can create a simple Java program. Here's an example "Hello, World" program.

- java

```
public class HelloWorld {
    public static void main(String[] args){
        System.out.println("Hello, World!");
    }
}
```

÷ Create a new Java project or class in IDE.

÷ Paste the code into your project.

÷ Build a run the program to see "Hello, World!" printed to the console.

Module = 2 : Getting Started - Java

2.1 Basic Structure of a Java Program

Java programs follow a specific structure, which includes a combination of packages, classes, methods, and statements. Let's break down the basic structure of a Java program with example :-

Packages :- Packages are used to organize classes and prevent naming conflicts. you should define the package name at the beginning of your Java file. For example :-

-java

```
package com.example.myprogram;
```

Classes :- A Java program typically consists of one or more classes. A "public" class in the file should have the same name as the file and contain the "main" method, which is the entry point of the program. For example :-

-java

```
public class MyProgram.{
```

```
public static void main(String[]args) {
        # your code goes here
    }
}
```

Methods :- Java programs are executed by invoking methods. The "main" method is the starting - point for your program. You can define other methods within your class to perform various tasks. For example :-

```
- java
public class MyProgram {
    public static void main (String[]args){
        # This is the main method
        # your code goes here
    }

public void anotherMethod () {
        # This is another method
        # your code goes here
    }
}
```

Statements :- Statements are individual instruct-ions that make up your code. They end with a semicolon (;). Here's an example of a statement within the "main" method :-

```
-java
System.out.println("Hello, World");
```

2.2 Variables and Data Types

In Java, variables are used to store data, and each variables has a data type associated with it. Java supports various data types, including primitive and reference types.

Primitive Data Types :- These are the most basic data types in Java. They include :-

- int :- Represent integer value.
- double :- Represent floating-point value.
- char :- Represent a single character.
- boolean :- Represent a true or false value.
- byte, short, long :- Other integer types with different size constraints.

Here's an example of variable declarations and assignments using primitive data types :-

-java

```
int age = 30;
double price = 19.99;
char grade = 'A';
boolean isStudent = true;
```

Reference Data Types :- These data types store references to objects. Some common reference data type are :-

- String :- Represent a sequence of characters.
- Scanner :- Used for reading input from the user.
- User-defined classes :- You can creat your own classes to define custom data types.

-.java

```
String name = "Pawan";
Scanner scanner = new Scanner (System.in);
```

2.3 Input and Output in Java

Java provides libraries For performing input and output operations. The "System.out" object is commonly used for output, and the "Scanner" class is widely for input. Here's how to perform basic input and output in Java.

Output (Printing to Console) :-

you can use the "System.out.printIn()" method to print text to the console.

-.java

```
System.out.printIn("Hello, World");
```

Input (Reading From Console) :-

To read input From the console, you can use the "Scanner" object and then use its methods to read different data types.

-.java

```
import java.util.Scanner;

public class InputExample {
    public static void main(String[]args){
```

```
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter your name:");
    String name = scanner.nextLine();

        System.out.print("Enter your age:");

        int age = scanner.nextInt();

        System.out.println("Name:" + name);
        System.out.println("Age:" + age);
    }
}
```

- In the above example, we import the "java.util.Scanner" class and use it to read the user's name and age. We used "nextLine()" to read a line of text and "nextLine()" to read an integer.

2.4 Operators

Operators in Java are symbols or characters used to perform operations on variables and values. They are Fundamental building blocks

in programming and serve various purposes, including arithmetic calculations, comparisons, logical operations, and more. Here, we will explore the most commonly used operators in Java.

1. Arithmetic Operators :-

Arithmetic operators are used to perform mathematical calculations. They include :-

- + (Addition) :- Adds two values together.
- - (Subtraction) :- Subtracts the right operand from the left operand.
- * (Multiplication) :- Multiplies two values.
- / (Division) :- Divides the left operand by the right operand.
- % (Modulus) :- Returns the remainder when the left operand is divided by the right operand.

Example :-

-java

```
int a = 10;
int b = 3;

int sum=a+b ;    # sum = 13
```

```
int difference = a-b;  # difference = 7
int product = a*b;     # product = 30
int quotient = a/b;    # quotient = 3
int remainder = a%b;   # remainder = 1
```

2. Comparison Operators :-

Comparison operators are used to compare values and return a boolean result (true or False). They include :-

- == (Equal to) :- Checks if two values are equal.
- != (Not equal to) :- Checks if two values are not equal.
- < (Less than) :- Checks if the left operand is less that the right operand.
- <= (Less than or equal to) :- Checks if the left operand is less than or equal to the right operand.
- > (Greater than) :- Checks if the left operand is greater than the right operand
- >= (Greater than or equal to) :- Checks if the left operand is greater than or equal to the right operand.

Example :-

-java

```
int x = 5;
int y = 10;
boolean isEqual = (x==y);          # isEqual = False
boolean isNotEqual = (x != y);     # isNotEqual = true
boolean isGreaterThan = (x>y);     # isGreaterThan = False
```

3. Logical operators :-

Logical operators are used to perform logical operations on boolean values. They include :-

- && (Logical AND) :- Returns true if both the left and right operand are true.
- || (Logical OR) :- Return true if either the left or right operand is true.
- ! (Logical NOT) :- Negates a boolean value.

Example :-

-java

```
boolean condition1 = true;
boolean condition2 = False;
```

```
boolean result1 = condition1 && condition2;
                                # result1 = false

boolean result2 = condition1 || condition2;
                                # result2 = true

boolean result3 = !condition1;  # result3 = false
```

4. Assignment Operators :-

Assignment operators are used to assign values to variables. The most basic one is the "=" operator, which assigns the value on the right to the variable on the left.

Example :-

```
- java

int x = 5;  # Assigns the value 5 to the
              variable x.
```

5. Increment and Decrement Operators :-

These operators are used to increase or decrease the value of a variable by 1. They include :-

- ++ (Increment) : Increase the value by 1.
- -- (Decrement) ; Decrease the value by 1.

Example :—

```java
int count = 5;
count++; # Increase count by 1, count is now 6.
count--; # Decrease count by 1, count is now 5 again.
```

6. Conditional (Ternary) Operator :—

The conditional operator "?" is a shorthand way of writing an "if-else" statement. It returns one of two values based on a condition.

Example :—

```java
int age = 18;
String status = (age >= 18) ? "Adult" : "Minor";
# if age is greater than or equal to 18, status is "Adult"; otherwise, it's "Minor"
```

Module - 3 : Control Structures

3.1 Conditional Statements

Conditional statements allow you to execute different blocks on specified conditions. In Java, you can use "if", "elseif", and "else" for simple conditions, and "switch" for multiple conditions.

- The "if" Statement ...

  The "if" Statement is the most basic form of conditional Statements. It allows you to execute a block of code if a condition is true.

Example :-

— java

```
int age = 25;
if (age >=18) {
    System.out.println("You are an Adult");
}
```

- In this example, if the condition "age >=18" is true, the message "You are an Adult." will be printed.

- The "if-else" Statement...

The "if-else" statement extends the "if" Statement by providing an alternative block of code to execute when the condition is false.

Example :-

-java

```
int age = 15;
if (age >= 18) {
    System.out.println("Adult");
}else {
    System.out.println("Not Adult");
}
```

- In this case, if the condition "age >=18" is false, the message "Not Adult" will be printed.

- The "else if" Statement...

The "else if" statement allows you to test multiple conditions in sequence.

Example :-

-java

```
int score = 85;
if (score >= 90) {
    System.out.println("Excellent");
} else if (score >= 80) {
    System.out.println("Very Good");
} else if (score >= 70) {
    System.out.println("Good");
} else {
    System.out.println("Need Improvement");
}
```

- In this example, the code will determine the appropriate message based on the value of "score".

- The "switch" Statement ...

The "switch" Statement is used when you have a single expression with multiple possible values, and you want to execute different block of code based on the expression's value.

Example :-

- java

```
int day = 3;
    String dayName;
switch (day) {
    case 1:
        dayName = "Sunday";
        break;

    case 2:
        dayName = "Monday";
        break;

    Case 3:
        dayName = "Tuesday";
        break;
    # Add More Cases as needed
```

```
        default :
            dayName = "Invalid day";
        }

        System.out.println("Today is " + dayName");
```

- The "switch" statement evaluates the value of "day" and executes the code block associated with the matching "case". if no match is found, the "default" block is executed.

3.2 Loops

Loops in Java allow you to repeatedly execute a block of code. There are three main types of loops: "For", "While", and "do-while".

- The "For" Loop

The "For" loop is commonly used when you know the number of iteration in advance.

Example :-

```java
for (int i=0; i<5 ; i++) {
    System.out.println ("Iteration" + i);
}
```

- In this example, the loop will run five times, printing "Iteration 0" to "Iteration 4".

- The "while" Loop

The "while" loop is used when you want to repeat a block of code as long as a certain condition is true.

Example :-

```java
int count = 0
while ( count < 5) {
    System.out.println("Count:" + count);
    count++ ;
}
```

- This loop will run until "count" is no longer less than 5, printing "Count : 0" to "Count : 4".

- The "do-while" Loop

The "do-while" Loop is similar to 'while" loop but guarantees that the loop block will execute at least once, as the condition is checked after the loop body.

Example :-

```
-java
    int number = 5;
do {
    System.out.println("Number:" + number);
    number--;
} while (number > 0);
```

- The loop will run and print "Number: 5" to "Number: 1".

3.3 Break and Continue Statement

In Java, you can use the "break" and "continue" statements to control the flow of loops and switch statements.

- The "break" Statement

The "break" statement is used to exit a loop prematurely. It is often used to terminate a loop based on a certain condition.

Example :-

-java

```
For (int i=0; i<10; i++) {
    if (i==5) {
        break; #Exits the loop
               When i equal 5.
    }
    System.out.println ("Iteration" + i);
}
```

- In this example, the loop will stop when "i" reaches **5**.

- The "continue" Statement

The "continue" Statement is used to skip the current iteration of a loop and move to the next one.

Example:-

-java

```
for(int i=0; i<10; i++) {
```

```
        if (i % 2 == 0) {
            continue;
        }
        System.out.println("Odd number: " + i);
    }
```

- In this code, even numbers are skipped, and only odd numbers are printed.

3.4 Handling Exceptions

Exception Handling is important for dealing with unexpected errors and maintaining the stability of your code of Java Programs. Java provides a mechanism to catch and handle exceptions using "try", "catch", "finally", and the "throw" statement.

- The "try-catch" block

You can use the "try" and "catch" blocks to handle exceptions. Code that might throw an exceptions is placed in the "try" block, and

you specify how to handle the exception in the "catch" block.

Example :-

```
-java

try {
    # Code that might throw an exception
    int result = 10/0; # Attempting to divide by 0.

} catch (ArithmeticException e) {
        # Handle the exception
System.err.println("Arithmetic exception: " + e.getMessage());
}
```

- In this example, the code attempts to divide by zero, which would throw an "ArithmeticException". The "catch" block handles this exceptions by printing an error message.

- The "finally" block

The "finally" block is used to execute code that should always run, whether or not an exception is thrown. It is typically used for cleanup operations.

Example :-

-java

```
try {
    # Code that might throw an exception
    int result = 10/0; # Attempting to divide by 0.
} catch (ArithmeticException e) {
    # Handle the exception
System.err.println("Arithmetic Exception: " + e.getMessage());
} finally {
    # Cleanup code
System.out.println("Finally Block Executed");
}
```

- The "finally" block will be executed even if an exception occurs.

- Throwing Exception

You can manually throw exceptions using the "throw" statement. This is useful when you want to create custom exceptions or handle specific cases.

Example :-

-java

```
int age = -5;
if (age<0) {
    throw new illegalArgumentException("Age not -ve.");
}
```

- In this example, an "illegalArgumentException" is thrown when the age is negative.

- **Catching Multiple Exceptions**

You can use multiple "catch" blocks to handle different types of exceptions. Catch blocks are evaluated from top to bottom, so place more specific exceptions types before more general ones.

Example :-

-java

```
try {
    int result = 10/0; # Attempting to divide by 0.
} catch (ArithmeticException e) {
System.err.println("ArithmeticException: " + e.getMessage());
```

```
} catch (Exception e) {
System.err.println("Exception occured:" + e.getMessage());
}
```

- In this example, the first "catch" block handles the specific "ArithmeticException", and the second "catch" block handles more general exceptions.

Module =4 : Functions And Methods

Functions and methods are important building blocks of any programming language, and in Java, the play a very important role in code organization, reusability, and modularity.

4.1 Functions Declaration and Definition

In Java, Functions are called "methods" when defined within a class. A method is a block of code that perform a specific task and can be called or invoked from other parts of the Program. A method typically consists of a "method signature" and a "method body".

- Method Signature :-
  - Method Name :- A unique name for the method.
  - Return Type :- The data type of the value that the method return. Use "void" if the method doesn't return a value.
  - ParameterList :- A list of Parameters (if any) that the method accepts.

- Method Body :—
  - The actual code that performs the desired task.

Example :- Simple method in Java

```
-java
public class Example {
        #Method declaration
    public void sayHello() {
            #Method body
        System.out.printIn("Hello, World!");
    }
}
```

- In this example, "sayHello" is a method with no parameters, and it has a return type of "void", meaning it doesn't return any value. When this method is called, it prints "Hello, World!" to the Console.

4.2 Parameters And Return Value

Functions can accept parameters (input) and return values (output). Parameters allow you to

pass data into a function, and return values allow you to get results from a function.

- **Passing Parameters :-**

You can specify parameters in the method declaration. These parameters act as variables that store the values passed to the method when it is called.

Example :-

-java

```
public class Calculator {
    #Method with parameters
    public int add (int a, int b) {
        return a+b;
    }
}
```

- In this example, the "add" method takes two integer parameter, "a" and "b". When you call this method with values, it returns the sum of "a" and "b".

- Returning Values :-

Methods can return values using the "return" statement. The return type in the method signature indicates the data type of the value that the method will return.

Example :-

```java
public class Calculator {
    //Method with a return value
    public int add(int a, int b) {
        return a+b;
    }
}
```

- In this case, the 'add' method returns the result of adding "a" and "b" which is of type "int".

- Calling Methods :-

To call a method, you use the method's name, passing in the required arguments if it has parameters. Here's how you would call the

"add" method from the "Calculator" class :-

Example :-

-java

```
public static void main(String[] args){
    Calculator calculator = new Calculator();
    int result = calculator.add(5,3);
    System.out.println("Result: " + result);
}
```

- In this example, we create an instance of the "Calculator" class, call the "add" method with the arguments "5", and "3", and store the result in the "result" variable.

4.3 Function Overloading

Function overloading allows you to define multiple methods with the same name within the same class, but with different parameter lists. Java determines which method to call based on the

number and types of arguments provides. Overloading is a way to provide flexibility and convenience to the user of a class or API.

Example :-

```java
public class MathUtils {
    public int add(int a, int b) {
        return a+b;
    }
    public double add(double a, double b) {
        return a+b;
    }
}
```

- In this example, the "MathUtils" class has two "add" methods. One takes two integers, and the other takes two doubles doubles. The Java compiler will determine which "add" method to call based on the argument types.

.....

-java

```
MathUtils mathUtils = new MathUtils();
int intSum = mathUtils.add(5,3);
double doubleSum = mathUtils.add(4.2, 2.3);
```

- In this code, we can call the "add" method with either integer or double arguments, and the appropriate version of the method will be invoked.

4.4 Recursion

Recursion is a programming technique where a method calls itself to solve a problem. It's particularly useful for solving problems that can be broken down into smaller, similar subproblems. To use recursion effectively, you need to define a base case that stops the recursion and ensure that the problem gets closer to the base case with each recursive call.

Example :-

-java

```
public class FactorialCalculator {
    public int calculateFactorial(int n) {
        if (n==0) {
            return 1; # Base case : factorial of 0 is 1.
        } else {
            return n * calculateFactorial(n-1);
                                # Recursive call
        }
    }
}
```

- In this, code, the "calculateFactorial" method calculates the factorial of a number "n". If "n" is 0, it returns 1 (the base case). Otherwise, it makes a recursive call to calculate the factorial of "n-1". The recursion continues until it reaches the base case.

-java

```
FactorialCalculator calculator = new FactorialCalculator();
int result = calculator.calculatorFactorial(5);
                        #Computes 5! = 120
```

- In this example, we create an instance of the "FactorialCalculator" class and use it to calculate the factorial of 5. The recursion eventually reaches the base case and returns the result.

4.5 Lambda Expressions

Lambda expressions are a feature introduced in Java 8 that allows you to define and use small, inline, anonymous functions. They are commonly used with functional interfaces (interfaces with a single abstract method), such as Java's built-in "Runable" and "Comparator" interfaces.

Example :-

Lambda function (expression) used with a Runnable,

-java

```
public class LambdaExample {
    public static void main(String[] args){
        #using a lambda expression to
        define a Runable.
```

```
Runnable task = () -> {
    for(int i=0; i<5 ; i++) {
      System.out.println("Task:" + i);
    }
};
# Start a new thread to run the task
Thread thread = new Thread(task);
thread.start();
    }
}
```

- In this code, we define a "Runnable" using a lambda expression and pass it to a new thread. When the thread is started, it executes the code defined in the lambda expression.

Module - 5 : Object-Oriented Programming

Object-Oriented Programming (OOP) is a fundamental paradigm in Java, providing a powerful way to structure and organize code.

5.1 Classes and Objects

In OOP, classes and objects are the building blocks of a paradigm or program. A class is a blueprint or template for creating objects, while an object is an instance of a class. Let's explore these concepts in more details with examples:-

- **Class Declaration :-**

To define a class, use the "class" Keyword followed by the class name and the class body enclosed in curly braces. The class body can includes fields (attributes) and methods (functions).

Example :-

-java

```
class Dog {
        String name;
        int age;
    void bark() {
        System.out.println(name + "says woof!");
    }
}
```

- In this example, we have defined a "Dog" class with fields "name" and "age" and a method "bark".

- Object Creation :-

To create an object of a class, you use the "new" keyword followed by the class constructor.

Example :-

-java

```
public class Main {
    public static void main(String[] args){
        Dog MyDog = new Dog();
                    # Create a Dog object
```

```
        myDog.name = "Buddy";
        myDog.age = 2;
        myDog.bark();
    }
}
```

- Here, we have created a "Dog" object named "myDog" and sets its "name" and "age" fields. We then called the "bark" method on the object.

5.2 Inheritance And Polymorphism

Inheritance is a key OOP concept that allows a class to inherit properties and behaviors from another class. Polymorphism is the ability of different objects to respond to the same method or function in a way that is specific to their individual classes. Let's see how these concepts work with examples.

- **Inheritance :-**

Inheritance is achieved using the "extends" keyword. A subclass (child class) can inherit fields and methods from a superclass (parent class).

Example :-

-java

```
class Animal {
    void eat() {
        System.out.println("This Animal is eating.");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("The Dog is Barking");
    }
}

class Cat extends Animal {
    void meow() {
        System.out.println("The cat is meowing");
    }
}
```

- In this example, the "Dog" and "Cat" classes inherit the "eat" method from the "Animal" class.

• Polymorphism :-

Polymorphism allows object of different classes to be treated as objects of a common superclass. This enables you to create more flexible and reusable code.

Example :-

-java

```
public class Main {
    public static void main (String [] args){
        Animal myDog = new Dog ();
        Animal myCat = new Cat ();

        myDog.eat ();
        myCat.eat ();
    }
}
```

- In this code, "myDog" and "myCat" are both of type "Animal", but they point to objects of the "Dog" and "Cat" classes. When we call the "eat" method on these objects, it executes the specific implementation of the method in their respective classes.

5.3 Encapsulation And Access Modifiers

Encapsulation is the concept of restricting access to the internal details of an object. Access Modifiers are used to control the visibility and accessibility of fields and methods in a class. Java provides four main access modifiers: "public", "private", "protected", and default (no modifiers). Here's how they work :—

- "public" : Accessible from anywhere.
- "private" : Accessible only within the same class.
- "protected": Accessible within the same class and subclass.
- "Default (no modifier): Accessible within the same Package.

Example :-

Let's see how access modifiers and encapsulation work in practice :—

-java

```
public class Student {
        # Public field
    Public String name;
```

Private field
```
private int age;
```

Protected field
```
protected String rollNumber;
```

Default (package-private) field
```
String address;
```

Public constructor
```
public Student(String name, int age, String rollNumber, String Address){
    this.name = name;
    this.age = age;
    this.rollNumber = rollNumber;
    this.address = address;
}
```

Public method to get age
```
public int getAge () {
    return Age;
}
```

Public method to set age
```
Public void setAge(int age) {
    this.age = age;
}
}
```

- In this example, we have a "Student" class with fidds and methods of different access levels. The "name" fields is public and can be accessed from anywhese, while the "age" field is private and can only be accessed within the some class. The "rollNumber" field is protected, and the "address" field is package-private (default).

You can also see the "getAge" and "setAge" methods that provide controlled access to the private "age" field. This is an example of encapsulation, as we have hidden the internal details of the class while providing public access to specific aspects.

5.4 Abstraction And Interfaces

Abstraction is the process of simplifying complex systems by modeling classes based on their important attributes and behaviors. In Java, interfaces are used to acheieve abstraction and define contracts for classes to implement. An interfaces is a collection of abstract methods that any class implementing the interface must provide concrete implementations for:-

Example :-

Here's an example of abstraction and interfaces :-

-java

```
interface Shape {
    double calculateArea();
    double calculatePerimeter();
}
class Circle implements Shape {
    private double radius;

    public Circle (double radius) {
        this.radius = radius;
    }
    @Override
public double calculateArea() {
    return Math.PI * radius * radius;
    }
    @Override
public double calculatePerimeter() {
    return 2 * Math.PI * radius;
    }
}
```

```
Class Rectangle implements Shape {
        private double width;
        private double height;

public Rectangle (double width, double height) {
            this.width = width;
            this.height = height;
        }

    @Override
public double calculateArea () {
            return width * height;
        }

    @Override
public double calculate Perimeter () {
            return 2 * (width + height)
        }
    }
}
```

- In this example, we have an "Shape" interfaces with two abstract methods, "calculateArea" and "calculatePerimeter". The "Circle" and "Rectangle" classes implement the "Shape" interfaces and provide concrete implementations for these methods.

- Using interfaces and abstraction, we can work with shapes without worrying about the specific details of how each shape calculates its area and perimeter.

5.5 Exception Handling

Exception handling is a important aspect of writing robust Java code. Exceptions are events that disrupt the normal flow of a program and must be handled to prevent program crashes. Java provides a comprehensive exception-handling mechanism with the "try", "catch", "finally" and "throw" statements.

Example :-

Here's how exception handling works in Java....

-java

```
public class ExceptionHandling {
    public static void main (String [] args){
        try {
            int result = divide (10,0);
            System.out.println ("Result:" + result);
```

```
    } catch (ArithmeticException e) {
        System.err.println("Error: " + e.getMessage());
    } finally {
        System.out.println("Execution completed.");
    }
}

public static int divide(int dividend, int divisor){
    if (divisor == 0) {
        throw new ArithmeticException("Can't divide by zero");
    }
    return dividend / divisor;
    }
}
```

- In this code, we have "divide" method that throws an "ArithmeticException" when attempting to divide by zero. The "main" method calls this method within a "try" block. If an exception occurs,

the progeam jumps to the corresponding "catch" block, which handles the exception. The "finally" block, is executed regardless of whether an exception occurred or not.

Module - 6 : Collections And Generics

Collection and generics are integral parts of Java, offering powerful tools for working with groups of data and creating more type-safe code.

6.1 Arrays And ArrayList

Arrays and ArrayList are used to store and manipulate collections of elements in Java. However, they have different characteristics and use cases.

- **Arrays :-**

Arrays are fixed in size, meaning you must specify their size when declaring them. Once the size is set, it cannot be changed.

Arrays can store elements of any data type, including primitive types and objects.

To access an element in an assay, you see or use an index, which starts at **0**.

Example :-

-java

```
int[] numbers = new int[5];
```

Creating an array of integers with a size of 5.

```
numbers[0] = 10;
```

Assigning a value to the first element

```
int value = numbers[0];
```

Accessing the first element

- ArrayList :-

ArrayList are dynamic in size, meaning they can grow or shrink as needed.

They can only store objects, not primitive types (e.g., int, double). For primitive types, you would typically use their corresponding wrapper classes (e.g., Integer, Double).

Elements is an ArrayList are accessed using methods like "get" and "set".

Example :-

```java
import java.util.ArrayList;

ArrayList<Integer> list = new ArrayList<>();
# Creating an ArrayList of integers

list.add(10);
# Adding an element to the ArrayList

int value = list.get(0);
# Accessing the first element
```

6.2 Maps And Sets

Maps and Sets are collection types designed for different use cases.

- **Maps :-**

Maps store key-value pairs. Each key is associated with a value.

Keys in a map must be unique. If you try to add a duplicate key, it will replace the

existing Key-value pair.

Common implementations include "HashMap", "TreeMap", and "LinkedHashMap".

Example :-

-java

```
import java.util.HashMap;
import java.util.Map;

Map<String, Integer> scores = new HashMap<>();
scores.put("Pawan", 21); # Adding a key-value pair
int pawanScore = scores.get("Pawan");
```

Retrieving a value by key

- Sets :-

Sets store a collection of unique elements. Duplicates are not allowed.

Common implementations include "HashSet", "TreeSet", and "LinkedHashSet".

Example :-

-java

```
import java.util.HashSet;
import java.util.Set;

Set<String> uniqueNames = new HashSet<>();
    uniqueNames.add("Pawan");
    # Adding an element
    uniqueNames.add("Preet");

boolean containsPawan = uniqueNames.contains("Pawan");
    # Checking if an element exists
```

6.3 Generic Classes And Methods

Generic provide a way to create classes and methods that can work with different data types while maintaining type safety.

- Generic Classes :-

You can define classes with generic type parameter. These parameters are specified when creating an instance of the class. Generics are commonly used

for collections like ArrayList and LinkeList.

Example :-

-java

```
public class Box <T> {
        private T value ;

public Box (T value ) {
        this.value =value;
    }
public T getValue () {
        return value;
    }
}
```

- In this example, "Box" is a generic class that can hold any type of object. You specify the type when creating a "Box" object.

-java

```
Box<Integer> intBox = new Box<>(42);
```
Creating a Box for Integers

```
int value = intBox.getValue ();
```
Accessing the value (no casting needed)

- Generic Methods :-

You can also create generic methods within non-generic classes. These methods can work with different data types based on the type parameter specified when calling the method.

Example :-

-Java

```
public static <T> T findMax(T[] array){
    if (array ==null || array.length ==0) {
            return null;
        }
    T max = array[0];
      for (T item : array) {
    if (item.compareTo(max) > 0){
            max = item;
        }
    }
    return max;
}
```

- In this example, "FindMax" is a generic Method

that can find the maximum element in an array of any type that implements the "Comparable" interfaces.

- Java

```
Integer [] intArray = {3,1,4,1,5,9,2,6,5,3};
    Integer max = findMax(intArray);
    # Calling the generic Method.
```

6.4 Iterators and foreach loop

Iterators and the enhanced for loop (foreach loop) are used for iterating over collections in a clean and concise way.

- **Iterators :-**

An iterators is an interfaces that provides methods for sequentially accessing elements in a collection.

You can use an iterator to traverse the elements in a collection and perform operations on them.

Example :- -java

```
import java.util.ArrayList;
import java.util.Iterator;

ArrayList<String> Fruits = new ArrayList<>();
        Fruits.add("Apple");
        Fruits.add("Banana");
        Fruits.add("Cherry");

Iterator<String> iterator = Fruits.iterator();
    while (iterator.hasNext()) {
        String Fruit = iterator.next();
        System.out.println(Fruit);
    }
```

⋮

→ In this example, we use an "iterator" to iterate through the "fruits" ArrayList and print each fruit.

- Enhanced For Loop (Foreach Loop) :-

The Enhanced for loop provides a more concise way to iterate over elements in a collection. It works with array, ArrayList, and other iterable collections.

Example :-

-java

```
String [] colors = {"Red", "Green", "Blue", "Yellow");
    For (String color : colors){
        System.out.println (color);
    }
```

-> In this code, the enhanced for loop iterates through the "colors" array and prints each color.

Module -7 : Multithreading - Concurrency

Multithreading and concurrency are fundamental concepts in Java that enable you to create responsive and efficient applications.

7.1 Creating - Managing Threads

In Java, a thread is a lightweight process that runs in the context of a program. Multithreading allows a program to execute multiple threads concurrently, which can improve performance and responsiveness. Let's start by understanding how to create and (~~manage~~) manage threads.

- **Creating Threads :-**

In Java, you can create threads by extending the "Thread" class or by implementing the "Runnable" interface. Extending the "Thread" class provides more flexibility, but implementing "Runable" is preferred because it separates the thread's behavior from the class it extends.

Example :- -java

Extending the Thread class

```
class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println("Thread is running");
    }
}
```

Implementing the Runnable Interface

```
class MyRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println("Runnable is running");
    }
}
```

- To start a thread, you create an instance of it and call the "start()" method :

-java

```
MyThread thread1 = new MyThread();
thread1.start(); # Start the thread

MyRunnable runnable = new MyRunnable();
Thread thread2 = new Thread(runnable);
thread2.start(); # Starts the thread
```

- **Thread States :-**

Threads in Java can be in various states, including :-

- New :- The thread has been created but hasn't stated yet.
- Runnable :- The thread is running or ready to run.
- Blocked :- The thread is waiting for a monitor lock.
- Waiting :- The thread is waiting indefinitely for another thread.
- Time Waiting :- The thread is waiting for a specified time period.
- Terminated :- The thread has finished its execution.

• Thread Priorities :-

Java allows you to set thread priorities using integers ranging from 1 (lowest) to 10 (highest). The default priority is 5. Higher-priority threads are given preference by the scheduler.

Example :-

```
-java

thread1.setPriority(7);
# Set the priority of thread1 to 7.
```

• Daemon Threads :-

Threads can be marked as daemon threads, which are background threads that don't prevent the JVM from exiting. Non-daemon threads, on the other hand, keep the JVM alive as long as they are running.

Example :-

```
-java

thread1.setDaemon(true);
#Mark thread1 as a daemon thread
```

7.2 Synchronization And Locks

Concurrency can introduce race conditions when multiple threads access shared resources simu-
-ltaneously. Synchronization is a technique to prevent data corruption and ensure thread safety.

- **Synchronized Methods :—**

You can make methods synchronized by adding the "Synchronized" keyword to the method declaration. This ensures that only one thread can execute a synchronized method at a time.

Example :— -java

```
class Counter {
    private int count = 0;

public synchronized void increment(){
        count++;
    }
}
```

- In this example, the "increment" method is synchronized, preventing multiple thread from simultaneously modifying the "count" variable.

- **Synchronized Blocks :-**

you can also create synchronized blocks within methods to limit the scope of synchronization. This can be more efficient that Synchronizing the entire method.

Example :-

-java

```
class Counter {
    private int count = 0;
    private Object lock = new Object();

public void increment(){
        Synchronized (lock) {
            count++;
        }
    }
}
```

- Here, the "synchronized" block with the "lock" object ensures that only one thread at a time can execute the increment operation.

- **Locks :-**

Java provides more advanced synchronization mechanism using explicit lock objects. The "ReentrantLock" class is commonly used lock implementation.

Example :-

```
import java.util.concurrent.locks.ReentrantLock;

    Class Counter {
        private int count = 0;
    private ReentrantLock lock = new ReentrantLock();

    public void increment() {
                lock.lock();
        try {
            count++;
        } finally {
```

```
            lock.unlock();
        }
    }
}
```

- Using a "ReentrantLock" allows for more fine-gained control over locking and unlocking, including the ability to specify timeouts and handle exceptions.

7.3 Threads Pools and Executors

Managing threads manually can be complex. Java provides thread pools and the 'Executor' framework to simplify concurrent task execution.

• Thread Pools :-

A thread pool is a group of worker threads that can be reused for executing tasks. Java's "ExecutorService" interface provides a high-level API for working with thread pools.

.....

Example :- -java

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

#Create a thread pool with 2 threads! -
ExecutorService executor = Executor.newFixedThreadPool(2);

    executor.submit(() -> System.out.println("Task 1"));
executor.submit(() -> System.out.println("Task 2"));
executor.shutdown(); #Shut down the executor when done
```

- In this example, we create a fixed-size thread pool with two threads and submit tasks for execution. The "ExecutorService" manages thread allocation and recycling.

- Executor Frameworks :-

The "Executor" Framework provides a way to decouple task submission from task execution. It offers various implementations, including :-

- "Executors.newFixedThreadPool(n)"
  - Create a fixed-size thread pool with "n" threads.
- "Executors.newCachedThreadPool();"
  - Creates an unbounded thread pool that creates threads on-demand.
- "Executors.newSingleThreadExecutor()"
  - Creates a single-threaded executor.

7.4 Asynchronous Programming

Asynchronous programming allows you to perform tasks concurrently, improving the responsiveness of your application. Java's "CompleteFuture" class makes it easier to work with asynchronous operations.

- Create a CompletableFuture :-

  You can create a "CompletableFuture" to represent a future result of a computation :-

Example :-

```java
import java.util.concurrent.CompletableFuture;

CompletableFuture<Integer> future = new CompletableFuture<>();
```

- Supplying a Result :-

You can complete a "CompletableFuture" with a result using the "complete" method :-

Example :-

```java
future.complete(42);
```

- Handling Asynchronous Tasks :-

"CompletableFuture" is often used for running tasks asynchronously. You can use the "supplyAsync" Method to run a task and return a "CompleteFuture" representing its result :-

Example :-

```java
import java.util.concurrent.CompletableFuture;
```

```
import java.util.concurrent.ExecutionException;

CompletableFuture<Integer> future =
CompletableFuture.supplyAsync(() -> 42);

try {
    int result = future.get(); # Get the result (blocking)
}catch(InterruptedException | ExecutionException e){
        e.printStackTrace();
    }
```

- In this example: the "supplyAsync" method runs a task that return the value 42, and we retrieve the result using "get". Note that "get" is a blocking operation that waits for the result.

- **Chaining CompletableFuture:-**

You can chain multiple asynchronous tasks using methods like "thenApply", "thenCompose" and "thenCombine".

Example :-

-java

```
import java.util.concurrent.CompletableFuture;

CompletableFuture<Integer> future =
CompleteableFuture.supplyAsync(() -> 42);

CompleteableFuture<String> result =
future.thenApplyAsync(value -> "Result: " + value);
```

- In this example, we chain the "future" with a transformation using "thenApplyAsync".

- Exception Handling :-

"CompletableFuture" allows you to handle exceptions using methods like "exceptionally" and "handle".

Example :-

-java

```
CompletableFuture<Integer> future =
Completable future.supplyAsync(() -> {
```

```
    throw new RuntimeException("Something went Wrong");
});

    Future.exceptionally(ex -> {
        System.err.println("Exception: "+ex.getMessage());
        return 0;
    });
```

- In this example, the "exceptionally" method handles exceptions by providing a fallback value.

- **Combining CompletableFutures :-**

You can combine multiple "CompletableFuture" instances using methods like "thenCombine" and "thenCompose".

Example ;

-java

```
CompletableFuture<Integer> future1 =
CompletableFuture.supplyAsync(() -> 42);
```

```
CompletableFuture<Integer> future2 =
CompletableFuture.supplyAsync(() -> 20);

CompletableFuture<Integer> combined =
future1.thenCombine(future2, (result1, result2) ->
                    result1 + result2);
```

- In this code, "thenCombine" combines the results of "future1" and "future2" using a provided function.

Module - 8 : I/O and Data Persistence

File I/O (Input/Output) and data presistence are important aspects of many software applications.

8.1 Reading and Writing Files

File I/O is important for handling data persistence in Java applications. You can read data from files and write data to files using various classes provided by the Java Standard Library.

• Reading Files :-

To read from data from a file, you can use the "FileInputStream" and "BufferedReader" classes. Here's how to read text from a file :-

Example :- -java

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
```

```
public class FileReaderExample {
    public static void main(String[] args) {
try ( BufferedReader reader =
    new BufferedReader(new FileReader("example.txt"))){
            String line;
while ((line = reader.readLine()) != null){
            System.out.println(line);
    }
    } catch (IOException e) {
            e.printStackTrace();
    }
}
}
```

- In this example, we use a "BufferedReader" to read lines from the "example.txt" file. The try-with-resources statement ensures that the file is properly closed after reading.

• Writing Files :-

To write data to a file, you can use the "FileOutputStream" and "BufferedWriter" classes. Here's how to write text to a file.

Example :- _java

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
public class FileWriteExample {
    public static void main(String[] args){
try (BufferedWriter writer = new BufferedWriter
    (new FileWriter("Output.txt"))) {
        writer.write("Hello, World!");
    }catch (IOException e){
        e.printStackTrace();
    }
}
}
```

- In this example, we use a "BufferedWriter" to write the text "Hello, World!" to the "output.tex" file. Like in the previous example, we use a try-with-resources statement to ensure proper resource management.

8.2 Working with Directories

In addition to reading and writing files, you may need to work with directories, such as creating, listing, or deleting them. Java provides classes like "File" and "Files" for these operations.

• Creating Directories :-

You can create directories using the "File" class's "mkdir()" or "mkdirs()" methods. The "mkdirs()" method creates parent directories if they don't exist.

Example :-

```java
import java.io.File;
public class CreateDirectoryExample {
    public static void main (String[] args) {
        File directory = new File ("my_directory");
```

```
        if (directory.mkdir()) {
            System.out.println("Directory Created Successfully");
        } else {
            System.out.println("Directory Creation Failed");
        }
    }
}
```

- In this example, we create a directory named "my_directory".

- **Listing Files and Directories :-**

To list files and subdirectories within a directory, you can use the "listFiles()" method.

Example :- _java

```
import java.io.File;
public class ListFilesExample {
    public static void main(String[] args) {
        File directory = new File("my directory");
        File[] files = directory.listFiles();
        if (files != null) {
            for (File file : files) {
```

```
            System.out.println(file.getName());
        }
    }
}
}
```

- This code lists the files and subdirectories within the "my-directory" directory"

- **Deleting Directories :-**

You can delete a directory and its contents using the "File" class's "delete()" method. Make suse the directory is empty before deleting it.

Example :-

-java

```
import java.io.File;
public class DeleteDirectoryExample {
    public static void main(String[]args){
        File directory = new File("my-directory");
if(directory.exists() && directory.isDirectory()){
```

```
        if (directory.delete()) {
            System.out.println("Directory Deleted Successfully");
        } else {
            System.out.println("Directory Deletion Failed");
        }
    } else {
        System.out.println("Directory does not exist");
    }
}
```

- This code attempts to delete the "my-directory" directory.

8.3 Serialization And DeSerialization

Serialization is the process of converting an object into a byte stream, while deserialization is the reverse process of reconstructing an object from a byte stream. Java provides built-in support for serialization and deserialization through the "Serializable" interface and the "ObjectInputStream" and "ObjectOutputStream" classes.

• Serialization :-

To make a class serializable, it must implement the "Serializable" interface. Here's an example of serial-izing an object to file:-

Example :- -java

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.io.Serializable;

class Person implements Serializable {
        private String name;
        private int age;

public Person (String name, int age){
        this.name = name;
        this.age = age;
    }
public String toString(){
    return "Person [name=" + name + ", age=" + age + "]";
    }
}
```

```
public class SerializationExample {
    public static void main (String [] args) {

try (FileOutputStream FileOut =
        new FileOutputStream ("person.ser");
ObjectOutputStream objectOut =
    new ObjectOutputStream (FileOut)){
Person person = new Person ("Pawon", 20);
        ObjectOut.writeObject (person);
  System.out.println ("Object has been Serialized");
} catch (IoException e){
            e.printStackTrace ();
        }
    }
}
```

- In this example, the "Person" class is marked as serializable, and an instance of "Person" is serialized to a file named "person.ser".

• Deserialization :-

To deserialize an object, you can use the "ObjectOutput--Stream" class. Here's how to deserialize an object from a file.

Example :-

-java

```
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

public class DeserializationExample {
    public static void main(String[] args){

try(FileInputStream fileIn = new FileInputStream("person.ser");
  ObjectInputStream objectIn = new ObjectInputStream(fileIn))
 {
    Person person = (Person) objectIn.readObject();
    System.out.println("Object has been deserialized:");
    System.out.println(person);
} catch(IOException | ClassNotFoundException e) {
```

```
            e.printStackTrace();
        }
    }
}
```

- In this example, we read the Serialized "Person" object from "person.ser" and deserialize it.

8.4 Working with Databases using JDBC :

Java Database connectivity (JDBC) is a standard Java API for connectivity to and interacting with relational databases. You can use JDBC to perform database operations, such as querying, inserting, updating, and deleting data.

• Database Connection :-

To work with a database, you first need to establish a database connection. You can use JDBC to connect to various databases, such as MySQL, PostgreeSQL, or Oracle. Here's how to connect to a database using JDBC.

Example :- -java

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class DatabaseConnectionExample {
    public static void main (String[] args){
String url = "jdbc:mysql://localhost:3306/mydb";
String username = "user";
String password = "password";
try {
Connection connection = DriverManager.getConnection
                        (url, username, password);
    System.out.println("Connected to the database");
        # Perform database operations here
        connection.close();
} catch (SQLException e) {
            e.printStackTrace();
    }
}
}
```

- In this example, we connect to a MySQL database using the "DriverManager" class.

• Executing SQL Queries :-

JDBC allows you to execute SQL queries and statements. Here's how to perform a simple SQL query :-

Example :-

-java

```
import java.sql.connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class JdbcQueryExample {
    public static void main (String[] args){
String url = "jdbc:mysql://localhost:3306/mydb";
String username = "user";
String password = "password";

try {
```

```
Connection connection = DriverManager.getConnection
                        (url, username, password);
    Statement statement = connection.creatStatement();
String sql = "SELECT* FROM users";
ResultSet resultSet = statement.executionQuery(sql);
While (resultSet.next()){
        int id = resultSet.getInt("id");
        String name = resultSet.getString("name");
        System.out.println("ID:" +id+ ", Name:" +name);
    }
     resultSet.close();
     Statement.close();
     connection.close();
} catch (SQLExceptione){
        e.printStackTrace();
    }
}
}
```

- In this example, we connect to a database, execute a "SELET" query, and print the result.

- Executing Updates :-

You can use JDBC to execute SQL statements that modify the database, such as "Insert", "UPDATE", or "DELETE". Here's an example of inserting data into a database :-

Example :-

- java

```
import java.sql.connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class JdbcUpdateExample {
    public static void main (String [] args) {
    String url = "jdbc:mysql://localhost:3306/mydb";
        String username = "user";
        String password = "password";
    try{
```

```
Connection connection = DriverManges.getConnection
                        (url, usesname, password);
    Statement statement = connection.createStatement();
String sql = "INSERT INTO users (name, age) VALUES ("Pawon", "20");
    int rowsAffected = statement.executeUpdate(sql);
    System.out.println("Rows affected: " + rowsAffected);
            statement.close();
            connection.close();
    } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

- This example inserts a new record into the "users" table.

Module - 9: User Interfaces With JavaFX

JavaFx is robust framework for building modern, feature-rich, and platform-independent graphical user interfaces (GUIs) in Java.

9.1 Introduction to JavaFX

JavaFx is a powerful framework for creating desktop applications with rich, interactive user interfaces. It provides aset of libraries and tools for building GUI applications, and it's part of the Java Development Kit (JDK) starting from Java 8.

• Setting up a JavaFx Project :-

To start a JavaFX project, you will need to have and a Java IDE (e.g. IntelliJ IDEA or Eclipse) installed. Here are the steps to set up a simple JavaFX project.

1. Create a new Java Project in your IDE.

2. Add JavaFX as a library to your project. In most IDEs, then this can be done by configuring the projects libraries and adding the JavaFX SDK.

3. Set up the main application class and make it extend "javafx.application.Application".

4. Implement the "start" method in your application class, which serves as the entry point for your JavaFX application.

- Here's basic structure for JavaFX application :-

Example :- -java

```
import javafx.application.Application;
import javafx.stage.Stage;

public class MyJavaFXApp extends Application {
    public static void main(String[]args){
        launch(args);
    }
}
```

```
    @Override
    public void start (Stage primaryStage) {
            # Set up your UI components and scences here
        primaryStage.setTitle("Hello JavaFX");
        primaryStage.show();
    }
}
```

- This sets up a basic JavaFX application that opens a window with the title "Hello JavaFX".

9.2 Building User Interfaces With FXML

FXML is ax XML-based markup language used in JavaFX to define the structure and layout of the user interfaces. It separtes the UI design from the application logic, promoting a cleaner and more maintainable codebase. FXML is commonly used in combination with Java Code, allowing you to create dynamic UIs.

- **FXML Structure :-**

An FXML is {an XML-based} file describes the UI

layout and components in a structured format. Here's a basic FXML structure for a simple JavaFX application :-

Example :- -xml

```
<?xml version = "1.0" encoding "UTF-8" ?>
<? import javafx.scence.contral.* ?>
<? import javafx.scence.layout.* ?>
<BorderPane xmlns = 'http://javafx.com/javafx/16"
            xmlns = "http:// javafx.com/fxml/1" >
<center>
<Button text = "Click Me!" fx:id = "myButton"
        onAction = '#handleButtonClicks" />
</center>
</BorderPane>
```

- In this FXML file, we define a "BorderPane" as the root element and place a button in the center. The 'fx:id" attribute gives the button an ID for later access, and the

"onAction" attribute specifies the method to invoke when the button is clicked.

- Loading FXML in java Code :-

To load an FXML file in your Java Code, you can use the "FXMLLoader" class. Here's an example of loading the FXML file defined above :-

Example :- -java

```
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class FXMLExample extends Application {
    public static void main(String[] args) {
        launch(args);
    }
```

```
@Override
public void start(Stage primaryStage) throws Exception {
    FXMLLoader loader = new
FXMLLoader(getClass().getResource("sample.fxml"));
    Parent root = loader.load();

    Scence scence = new Scence(root);
    primaryStage.setScence(scence);
    primaryStage.setTitle("FXML Example");
    primaryStage.show();
}
}
```

- In this code, we use the "FXMLLoader" to load the FXML file, and we set it as the root of the scence in the "start" method.

- Event Handling in FXML :-

In FXML, you can specify which method should handle events (e.g. button clicks) usin the "onAction" attribute.

Here's how you can handle a button click event in Java code :-

Example :-

- java

```
import javafx.event.ActionEvent;
import java.fxml.FXML;
import javafx.scence.control.Button;

public class FXMLController {
    @FXML
    private Button myButton;

    @FXML
    public void handleButtonClick(ActionEvent event){
        myButton.setText("Clicked");
    }
}
```

- In this example, we create "FXMLController" class that defines a method "handleButtonClick" to handle the button's action event. We link this method to the FXML file using the "onAction" attribute.

9.3 Event Handling and User Interactions

JavaFX provides a comprehensive set of event-handling mechanisms for user interactions. You can handle events like button clicks, key presses, mouse movements, and more.

• Handling Button Clicks :-

To handle a button click event, you can use an "ActionEvent" handler. Here's an Example :-

Example :-

-java

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scence.Scence;
import javafx.scene.control.Button;
import javafx.scence.layout.StackPane;
import javafx.stage.Stage;

public class ButtonClickExample extends Application {
```

```
public static void main(String[] args){
        launch(args);
    }

    @Override
    public void start(Stage primaryStage){
        primaryStage.setTitle("Button Click Example");
        Button btn = new Button("Click me!");

    # Create an event handler for the button click
    EventHandler<ActionEvent> eventHandler = e -> {
        System.out.println("Button Clicked");
    }

    # Assign the event handler to the button
        btn.setOnAction(eventHandler);

    StackPane root = new StackPane();
    root.getChildren().add(btn);
    primaryStage.setScence(new Scene(root, 300, 200));
    primaryStage.show();
}
}
```

- In this example, we create a button and attach a "EventHandler" to it to handle the click event.

- Handling key Presses :-

To handle key presses in JavaFX, you can use the "setOnKeyPressed" method. Here's an example :-

Example :-

-java

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;
import javafx.scence.input.KeyEvent;
public class KeyPressExample extends Application{
    public static void main (String[]
args){
        launch(args);
    }
@Override
public void start(Stage primaryStage) {
```

```
        primaryStage.setTitle("Key Press Example");
        StackPane root = new StackPane();
        Scene scene = new Scene(root, 300, 200);

    scene.setOnKeyPressed(event -> {
        String key = event.getCode().toString();
        System.out.println("Key pressed: " + key);
    });
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

- In this code, we set an event handler for key presses on the scene.

- Handling Mouse Events :-

You can handle various mouse events, such as mouse clicks and mouse movements, in JavaFX. Here's an example of handling a mouse click event :-

Example :- -java

```
import javafx.application.Application;
```

```
import javafx.scene.Scene;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;
import javafx.scene.input.MouseEvent;

public class MouseClickExample extends Application {
    public static void main (String[] args) {
        launch (args);
    }

    @Override
public void start (Stage primaryStage) {
    primaryStage.setTitle("Mouse Click Example");
    StackPane root = new StackPane();
    Scene scene = new Scene(root, 300, 200);

Scene.setOnMouseClicked (event -> {
        double x = event.getSceneX();
        double y = event.getSceneY();
    System.out.println("Mouse clicked at co-ordinates ("+x+", "+y+")");
});
```

```
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

- In this code, we set an event handler for mouse clicks one the scene.

9.4 JavaFX Charts and Multimedia

JavaFx provides built-in support for creating various types of charts, such as line charts, bar charts, pie charts, and more. It also offers multimedia capabilities for playing audio and video content.

• Creating a Line Chart :-

To create a line chart in JavaFX, you can use the "LineChart", "NumberAxis", and "XYChartSeries" classes. Here's an example :-

Example :-

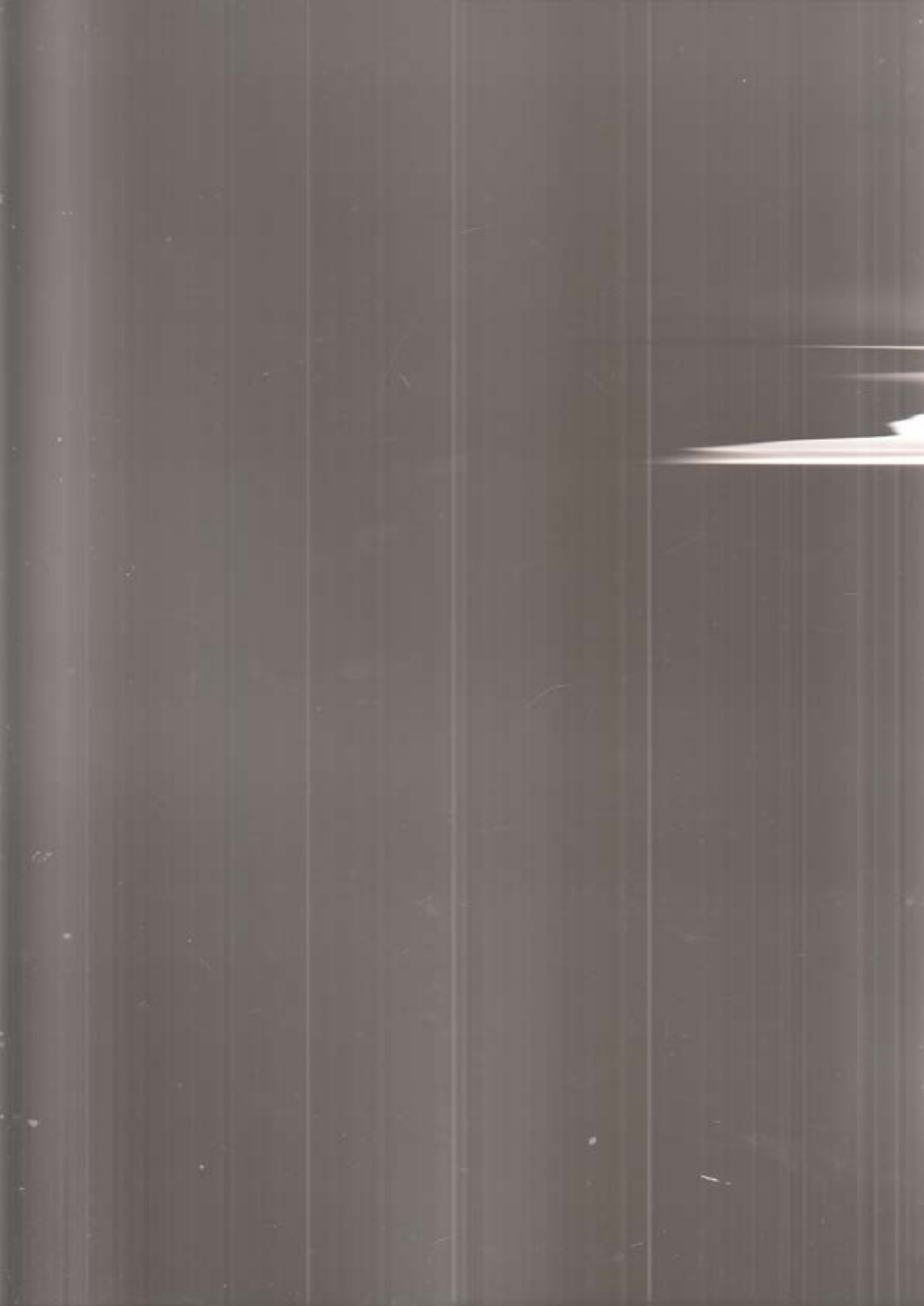
```java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.chart.LineChart;
```

```
import javafx.scene.chart.NumberAxis;
import javafx.scene.chart.XYChart;
import javafx.stage.Stage;

public class LineChartExample extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
public void start(Stage primaryStage) {
    primaryStage.setTitle("Line Chart Example");
    NumberAxis xAxis = new NumberAxis();
    NumberAxis yAxis = new NumberAxis();

LineChart<Number, Number> lineChart = new LineChart
                                        <>(xAxis, yAxis);
 lineChart.setTitle("Sample Line Chart");
XYChart.Series<Number, Number> series =
                    new XYChart.Series<>();
```



```
series.setName ("Data Series");

# Add data points to the series

series.getData().add(new XyChart.Data<>(1,2));
series.getData().add(new XyChart.Data<>(2,4));
series.getData().add(new XyChart.Data<>(3,8));
series.getData().add(new XyChart.Data<>(4,16));

lineChart.getData().add(series);

Scene scene = new Scene(lineChart, 800, 600);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

- In this code, we create a simple line chart with data series.

- **Playing Audio and Video :-**

Javafx allows you to play audio and video content with the "MediaPlayer" class. Here's an example of playing an audio file :-

Example :— -java

```
import javafx.application.Application;
import java.fx.scene.Scene;
import javafx.scene.media.Media;
import javafx.scene.media.MediaPlayer;
import javafx.scene.media.MediaView;
import javafx.stage.Stage;
public class MediaExample extends Application{
    public static void main(String[] args){
        launch(args);
    @Override
public void start(Stage primaryStage){
    primaryStage.setTitle("Media Example");
# Create a media file and a media player
Media media = new Media("file:///path/to/your/audio.mp3");
MediaPlayer mediaPlayer = new MediaPlayer(media);
```

Create a media view to display the media content

```
MediaView mediaView = new MediaView(mediaPlayer);
Scene scene = new Scene(mediaView, 800, 600);
primaryStage.setScene(scene);
```

Play the media

```
        mediaPlayer.play();
        primaryStage.show();
    }
}
```

- In this example, we create a 'MediaPlayer" and display the audio content using the "MediaView".

Module - 10 : Advanced Topics

In this module, we will will explore advanced topics that can take your Java skills to the next level. These topics include reflection and annotations, net-working and socket programming, security and cryptography, working with external libraries and frameworks, and design pattern and best practices.

10.1 Reflection and Annotations

Reflection is a powerful feature in Java that allows you to inspect and manipulate classes, methods, fields, and other program components at runtime. Annotations are a form of metdata that can be added to Java source code, classes, methods, and fields to provide additional information to the compiler, tools and runtime.

- Reflection :-

Reflection is useful when you need to perform tasks

like inspecting class metadata, accessing private fields or methods, and creating instances of classes at runtime. Here's an example of using reflection to get class information :-

Example :-

-java

```
import java.lang.reflect.Field;
import java.lang.reflect.Method;

public class ReflectionExample {
    public static void main(String[] args) {
        Class<?> clazz = String.class;

System.out.println("Class Name:" + clazz.getName());

        Field[] fields = clazz.getDeclaredFields();
        System.out.println("Fields:");

For (Field field : fields) {
    System.out.println(field.getName());
}
```

```
Method[] methods = clazz.getDeclaredMethods();
        System.out.println("Methods:");
For (Method method : methods) {
        System.out.println(method.getName());
    }
}
}
```

- In this code, we obtain class information using reflection, including class name, fields, and methods.

- Annotations :-

Annotations are metadata that can be attached to classes, methods, fields, and other program elements. you can create your custom annotations are use predefined ones, like "@Override", "@Deprecated", or "@SuppressWarnings". Here's a simple custom annotation example :-

Example :-

```
-java
```

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)

public @interface MyAnnotation {
    String value() default "Default Value";
}
```

- In this example, we define a custom annotation "@MyAnnotation" with a default value. You can use this annotation to provide additional information to methods.

Example :-

-java

```
public class AnnotationExample {
    @MyAnnotation("Custom Value")
```

```
public void myAnnotatedMethod () {
        # Method logic here
    }
}
```

- In this code, the "myAnnotatedMethod" is annotated with "@MyAnnotation", providing a custom value.

10.2 Networking and Socket Programming

Networking and socket programming are fundamental for building networked applications and services. Java provides comprehensive libraries for working with sockets, making it a powerful choice for network-related tasks.

- Creating a Server :-

Here's a simple Java program that creates a socket server and listens for incoming connections :-

Example :-

```
                    -java
import java.io.*;          → Asterisk
import java.net.*;
```

```
public class ServerExample {
    public static void main (String[] args){
try(ServerSocket serveSocket = new ServerSocket(12345)){
 System.out.println("Server is waiting for connection");
 while (true) {
      Socket clientSocket = serverSocket.accept();
 System.out.println("Client connected:" +
                    clientSocket.getInetAddress());
# Handle client communication here
# Don't forget to close the client socked when done
    }
 } catch (IOException e){
        e.printStackTrace();
    }
  }
 }
```

- This code sets up a server on port 12345 and continuously listens for incoming connections. You can perform various tasks with the client after accepting the connections.

• Creating a Client :-

Here's a Java program to create a socket client and connect to a server :-

Example :-

-java

```
import java.io.*;
import java.net.*;
public class ClientExample {
    public static void main(String[] args){
try (Socket socket = new Socket("localhost", 12345)){
        System.out.println("Connected to the Server");
# Handle client-server communication using I/O Streams
# Don't forget to close the socket when done
}
```

```
        catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- This code, connects to a server running on "localhost" at port 12345.

10.3 Security and Cryptography

Security is a important aspects of software development, and Java offers a range of features and libraries for secure applications development, including cryptography.

- **Encryption and Decryption:-**

Java provides cryptographic libraries for encryption and decryption. Here's an example of using the Java Cryptography Architecture (JCA) to encrypt and decrypt data:-

Example :-

-java

```
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import java.security.Key;
import java.util.Base64;

public class CryptoExample {
    public static void main(String[] args) throws Exception {
        # Generate a secret key
        KeyGenerator keyGen = KeyGenerator.getInstance("AES");
        SecretKey key = keyGen.generateKey();
        # Create a cipher and initialize it for encryption
        Cipher cipher = Cipher.getInstance("AES");
        cipher.init(Cipher.ENCRYPT_MODE, key);
        # Data to be encrypted
        byte[] plainText = "This is secret message.".getBytes();
```

Encrypt the Data

```
byte[] encryptedText = cipher.doFinal(plainText);
```

Print the encrypted data (base-64-encoded for readability)

```
System.out.println("Encrypted: " + Base64.getEncoder().
                    encodeToString(encryptedText));
```

Initialize the cipher for decryption

```
Cipher.init(Cipher.DECRYPT_MODE, key);
```

Decrypt the data

```
byte[] decryptedText = cipher.doFinal(encryptedText);
System.out.println("Decrypted: " + new String(
                    decryptedText));
}
}
```

- In this example, we generate a secret key, encrypt some data and then decrypt it using the same key.

10.4 : Working With External Library and Frameworks

In real-world projects, you often need to work with external libraries and frameworks to simplyfy and enhance your development process. Java has a vibrant ecosystem of libraries and frameworks, each deigned for specific purposes. Here's brief introduction to integrating external libraries in a Java project.

- **Maven :-**

Maven is a popular build automation tool and project management tool for Java projects. It simplifies the process of managing project dependencies and building projects. you can define dependencies in your project's "pom.xml" file, and Maven will download and mangge them for you.

For example, to use the Apache Commons Lang library in your project, add the following dependeny to your "pom.xml" :-

Example :-

-xml

```
<dependency>
    <groupId> org.apache.commons </groupID>
    <artifactId> commons-lang3 </artifactId>
    <version> 3.12.0 </version>
</dependency>
```

- Maven will automatically download and include the library in your project.

- Gradle :-

Gradle is another build automation tool similar to Maven. It's known for it's flexibility and support for Groovy-based build scripts. You can define dependencies in the "build.gradle" file for your proje.

- External Frameworks :-

Java has a plethora of external frameworks for various purposes, such as Spring (for enterprise application), Hibernate (for object-relational mapping), JavaFx

(for building GUIs), and many others. Integrating these frameworks typically involves adding the framework-specific dependencies to your project and configuring your application to use the framework.

10.5: Design Patterns and Best Practices

Design patterns and best practices are essential for writing clean, maintainable, and efficient code. They offer standardized solutions to common programming challenges and help you build robust software.

Here are few design Pattern:-

- Singleton Pattern
- Factory Method Pattern
- Observer Pattern
- Builder Pattern
- Decorator Pattern