1. Introduction to Java

Theory:

Java is a programming language created by Sun Microsystems, which is now owned by Oracle.

Java is an object-oriented language, that is, it is object-based and uses objects to accomplish the task.

Java is platform-independent, that is, Java programs can be run on any system having a Java Virtual Machine (JVM).

The JVM translates Java code into machine code so it can be run on various operating systems.

Java is used to develop web applications, mobile apps, and large enterprise systems.

It has a light syntax similar to C, thus easy to learn for those who know other languages.

Java offers automatic garbage collection, that is, memory management so memory usage is made efficient.

Java is secure because it has a security manager that specifies the access rules for the resources.

Java is multithreaded, that is, Java can execute multiple tasks concurrently, which can be useful while developing responsive apps.

Java's extensive standard library offers a lot of functions bundled into Java, thus faster and easier to develop.

Lab:

java

Copy

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

2. Data Types, Variables, and Operators

Theory:

Java supports primitive data types such as int, float, and char to store different types of values.

Variables hold data and are declared with a particular data type.

Java offers arithmetic, relational, and logical operators to operate on variables and accomplish different tasks.

Lab:

Develop a program that demonstrates the usage of different data types:

java

Copy

```
public class DataTypes {
    public static void main(String[] args) {
        int a = 10;
float b = 5.5f;
        char c = 'A';
        System.out.println(a + " " + b + " " + c);
    }
}
```

Create a calculator using arithmetic and relational operators:

java

Copy

```
public class Calculator {
    public static void main(String[] args) {
        int x = 10, y = 5;
        System.out.println("Sum: " + (x + y));
        System.out.println("Difference: " + (x - y));
        System.out.println("Multiplication: " + (x * y));
        System.out.println("Division: " + (x / y));
        System.out.println("Is x > y? " + (x > y));
    }
}
```

Demonstrate type casting (explicit and implicit):

java

Copy

```
public class TypeCasting {
    public static void main(String[] args) {
        int i = 10;
        double d = 5.5;
        int casted = (int) d;
        System.out.println(i + " " + d + " " + casted);
    }
}
```

3. Control Flow Statements

Theory:

Control flow statements such as if-else and switch-case assist us in determining which block of code needs to be executed depending on certain conditions.

Loops (for, while, do-while) assist us in executing actions repeatedly.

The break and continue keywords manage the flow of loops.

Lab:

Write a program to check whether a number is even or odd using an if-else statement:

java

Copy

```
public class EvenOdd {
    public static void main(String[] args) {
        int num = 3;
        if (num % 2 == 0)
            System.out.println(num + " is even");
        else
            System.out.println(num + " is odd");
    }
}
```

Design a simple menu program using a switch-case:

java

Copy

```
import java.util.Scanner;

public class Menu {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int choice;
        System.out.println("Enter choice (1-3): ");
        System.out.println("1. Add\n2. Subtract\n3. Exit");
        choice = sc.nextInt();

        switch (choice) {
            case 1:
                System.out.println("Addition");
                break;
            case 2:
                System.out.println("Subtraction");
break;
        case 3:
            System.out.println("Exit");
            break;
        default:
            System.out.println("Invalid choice");
    }
}
```

Write a program to display the Fibonacci series using a loop:

java

Copy

```
public class Fibonacci {
    public static void main(String[] args) {
        int n = 10, a = 0, b = 1;
        System.out.println("Fibonacci Series: ");
        for (int i = 1; i <= n; i++) {
            System.out.print(a + " ");
            int c = a + b;
            a = b;
            b = c;
        }
    }
}
```

4. Classes and Objects

Theory:

A class is a template for making objects. An object is a representation of a class.

Constructors are employed to initialize an object with initial values when it is created.

Constructor overloading enables seeral constructors with varying parameters.

Lab:

Develop a class Student with fields name and age, and a method to show the details:

java

Copy

```
public class Student {
    String name;
    int age;

    Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    void display() {
        System.out.println("Name: " + name + ", Age: " + age);
    }

    public static void main(String[] args) {
        Student s1 = new Student("Alice", 20);
        s1.display();
    }
}
```

Develop multiple constructors in a clas and illustrate constructor overload:

java

Copy

```
public class Person {
    String name;
    int age;

    Person(String name) {
        this.name = name;
    }

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    void display() {
        System.out.println("Name: " + name + ", Age: " + age)
    }

    public static void main(String[] args) {
        Person p1 = new Person("John");
Person p2 = new Person("Alice", 25);
        p1.display();
        p2.display();
    }
}
```

Implement a simple class with getters and setters for encapsulation:

java

Copy

```
public class Encapsulation {
    private String name;
    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public static void main(String[] args) {
        EncapsulationExample obj = new EncapsulationExample();
        obj.setName("Bob");
        System.out.println("Name: " + obj.getName());
    }
}
```

5. Methods in Java

Theory:

Methods enable us to group code into small, reusable blocks.

Methods can be armed with parameters and return types in order to conduct specific operations.

Static methods are invoked using the class nameand they are owned by the class, not by instances of the class.

Lab:

Develop a program that calculates the maximum of three numbers using a method:

java

Copy

```
public class MaxNumber {
    public static int findMax(int a, int b, int c) {
        return (a > b)? (a > c? a : c) : (b > c? b : c);
    }

    public static void main(String[] args) {
        System.out.println("Maximum: " + findMax(10, 20, 15));
    }
}
```

Write methods which can accept variables of various kinds:

java

Copy

```
public class OverloadExample {
    public static int add(int a, int b) {
        return a + b;
    }

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

    public static void main(String[] args) {
        System.out.println("Int addition: " + add(5, 10));
        System.out.println("Double addition: " + add(5.5, 10.5));
    }
}
```

Write a class with static members and methods and demonstrate how these work:

java

Copy

```
public class StaticExample {
    static int count = 0;

    static void increment() {
        count++;
    }

    public static void main(String[] args) {
        increment();
        increment();
        System.out.println("Count: " + count);
    }
}
```

6. Object-Oriented Programming (OOP) Concepts

Theory:

Object-Oriented Programming (OOP) has the concepts of Encapsulation, Inheritance, Polymorphism, and Abstraction.

Inheritance allows a class to inherit the properties and methods of another class. It may be single, multilevel, or hierarchical.

Method Overriding permits a subclass to provide its own implementation of a method already available in the superclass.

Lab:

Create a program to demonstrate single inheritance:

java

Copy

```
public class A {
    void display() {
        System.out.println("Class A");
    }
}

public class B extends A {
    public static void main(String[] args) {
        B obj = new B();
        obj.display();
    }
}
```

Create a class hierarchy and demonstrate multilevel inheritance:

java

Copy

```
public class X {
    void show() {
        System.out.println("Class X");
    }
}

public class Y extends X {
    void display() {
        System.out.println("Class Y");
    }
}

public class Z extends Y {
    public static void main(String[] args) {
        Z obj = new Z();
        obj.show();
        obj.display();
    }
}
```

Utilize method overriding to demonstrate polymorphism in action:

java

Copy

```
public class Animal {
    void sound() {
        System.out.println("Animal makes sound");
    }
}

public class Dog extends Animal {
    void sound() {
        System.out.println("Dog barks");
    }
public static void main(String[] args) {
        Animal obj = new Dog();
        obj.sound();
    }
}
```

7. Constructors and Destructors

Theory:

A constructor assists in initializing an object during creation, and it can be either parameterized or default.

Java does not support destructors, but it employs garbage collection to release memory automatically.

Constructor overloading provides for multiple constructors having different parameters to be defined.

Lab:

Create a program to create and initialize an object using a parameterized constructor:

java

Copy

```
public class Car {
    String model;

    Car(String model) {
        this.model = model;
    }

    void display() {
        System.out.println("Car model: " + model);
    }

    public static void main(String[] args) {
        Car obj = new Car("Tesla");
        obj.display();
```

~~~~
    }
}
~~~~

Describe how constructor overloading is implemented using different parameter types:

java

Copy

~~~~
public class Book {
    String title;
    int pages;

    Book(String title) {
        this.title = title;
    }

    Book(String title, int pages) {
        this.title = title;
        this.pages = pages;
    }

    void display() {
        System.out.println("Title: " + title + ", Pages: " + pages);
```
public static void main(String[] args) {
    Book obj1 = new Book("Java Basics");
    Book obj2 = new Book("Java Advanced", 500);
    obj1.display();
    obj2.display();
}
~~~~

8. Arrays and Strings

Theory:

Arrays hold multiple values of the same type in a single variable. They can be one-dimensional or multi-dimensional.

Strings in Java are objects. We have String, StringBuffer, and StringBuilder classes to handle strings.

String methods like length(), charAt(), and substring() help us in handling and manipulating string data.

Lab:

Make a program to add and subtract matrices using 2D arrays:

~~~~
public class Matrix {
    public static void main(String[] args) {
        int[][] matrix1 = {{1, 2}, {3, 4}};
        int[][] matrix2 = {{5, 6}, {7, 8}};
        int[][] result = new int[2][2];

        for (int i = 0; i < 2; i++) {
            for (int j = 0; j < 2; j++) {
                result[i][j] = matrix1[i][j] + matrix2[i][j];
            }
        }

        for (int i = 0; i < 2; i__)
```
for (int j = 0; j < 2; j++) {
                System.out.print(result[i][j] + " ");
            }
            System.out.println();
        }
    }
}
~~~~

Create a program to reverse a string and check for palindromes:

java

Copy

```
public class StringManipulation {
    public static void main(String[] args) {
        String str = "madam";
        String reverse = new StringBuilder(str).reverse().toString();
        if (str.equals(reverse)) {
            System.out.println(str + " is a palindrome");
        } else {
            System.out.println(str + " is not a palindrome");
        }
    }
}
```

Implement string comparison using equals() and compareTo() methods:

java

Copy

```
public class StringComparison {
    public static void main(String[] args) {
        String str1 = "hello";
        String str2 = "world";
        System.out.println(str1.equals(str2));
        System.out.println(str1.compareTo(str2));
    }
}
```

9. Inheritance and Polymorphism

Theory:

Inheritance enables one class to inherit members and methods from another class. It enables reusability and nested class structures.

Polymorphism enables methods to respond differently to the object type, which in turn allows method overriding.

Dynamic Binding (or Run-Time Polymorphism) is when a method call is resolved at run-time.

Time to do some lab work.

Write a program illustrating inheritance using the extends keyword:

java

Copy

```
public class Animal {
    void speak() {
        System.out.println("Animal speaks");
    }
}

public class Dog extends Animal {
    public static void main(String[] args) {
        Dog obj = new Dog();
        obj.speak();
    }
}
```

Implement runtime polymorphism by overriding methods in the child class:

java

Copy

```
public class Animal {
    void sound() {
        System.out.println("Animal sound");
    }
}

public class Cat extends Animal {
    void sound() {
        System.out.println("Cat meows");
    }

    public static void main(String[] args) {
        Animal obj = new Cat();
        obj.sound();
    }
}
```

Use the super keyword to invoke the parent class constructor and methods:

java

Copy

```
public class Parent {
    Parent() {
        System.out.println("Parent class constructor");
    }
}

public class Child extends Parent {
    Child() {
        super();
        System.out.println("Child class constructor");
    }

````
```java
public static void main(String[] args) {
    Child obj = new Child();
}
````

10. Interfaces and Abstract Classes

Theory:

Abstract classes allow you to declare methods to be called by subclasses, and they can also contain methods that are fully declared. Interfaces allow classes to acquire multiple behaviors (inherit from many sources) and define a rule that classes are to abide. Java permits a class to implement multiple interfaces in one go.

Lab:

}

Define an abstract class and call its methods in a subclass:

java

Copy

```
abstract class Animal {
    abstract void sound();
}

class Dog extends Animal {
    void sound() {
        System.out.println("Dog barks");
    }

    public static void main(String[] args) {
        Dog obj = new Dog();
        obj.sound();
    }
}
```

Define a program that employs multiple interfaces within one class:

java

Copy

```
interface Playable {
    void play();
}

interface Watchable {
    void watch();
}

public class Video implements Playable, Watchable {
    public void play() {
        System.out.println("Playing video");
    }

    public void watch() {
        System.out.println("Watching video");
    }

    public static void main(String[] args) {
        Video obj = new Video();
        obj.play();
        obj.watch();
    }
}
```

Apply a straightforward interface to a real-world situation, such as a payment system:

java

Copy

```
interface Payment {
    void processPayment();
}

class CreditCardPayment implements Payment {
    public void processPayment() {
        System.out.println("Processing credit card payment");
    }

    public static void main(String[] args) {
        CreditCardPayment obj = new CreditCardPayment();
        obj.processPayment();
    }
}
```

11. Packages and Access Modifiers

- **Theory**:
  - A package in Java is a container for related classes and interfaces. Java has built-in packages, but you can also create user-defined packages.
  - Access modifiers control the visibility of classes, methods, and variables: `private`, `default`, `protected`, and `public`.
  - The `import` statement allows classes to be accessed from other packages.
- **Lab**:
  - **Create a user-defined package and import it into another program:**

```
java
Copy
package mypackage;

public class MyClass {
    public void display() {
        System.out.println("Inside MyClass in mypackage");
    }
}

import mypackage.MyClass;

public class Test {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        obj.display();
    }
}
```

- **Demonstrate the use of different access modifiers within the same package and across different packages:**

```
java
Copy
package mypackage;

class A {
    private int x = 5;
    public int y = 10;

    public void show() {
        System.out.println("x: " + x + " y: " + y);
    }
}

public class B {
    public static void main(String[] args) {
        A obj = new A();
        obj.show();
        System.out.println("y: " + obj.y);
    }
}
```

12. Exception Handling

- **Theory**:

    - Exceptions in Java can be checked or unchecked. Checked exceptions are explicitly declared, while unchecked exceptions can be handled during runtime.
    - The `try`, `catch`, and `finally` blocks are used for exception handling. `throw` and `throws` are used for custom exceptions.
    - Custom exceptions can be created by extending the `Exception` class.
- **Lab**:
    - **Write a program to demonstrate exception handling using `try-catch-finally`:**

```
java
Copy
public class Test {
    public static void main(String[] args) {
        try {
            int result = 10 / 0;
        } catch (ArithmeticException e) {
            System.out.println("Error: " + e);
        } finally {
            System.out.println("Finally block executed");
        }
    }
}
```

- **Implement multiple `catch` blocks for different types of exceptions:**

```
java
Copy
public class Test {
    public static void main(String[] args) {
        try {
            int[] arr = new int[3];
            arr[5] = 10;
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array Index Error: " + e);
        } catch (Exception e) {
            System.out.println("General Error: " + e);
        }
    }
}
```

- **Create a custom exception class and use it in your program:**

```
java
Copy
class CustomException extends Exception {
    public CustomException(String message) {
        super(message);
    }
}

public class Test {
    public static void main(String[] args) {
        try {
            throw new CustomException("This is a custom exception");
        } catch (CustomException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

13. Multithreading

- **Theory**:
  - A thread is a lightweight process, and multithreading allows a program to perform multiple tasks simultaneously.
  - Threads can be created by extending the `Thread` class or implementing the `Runnable` interface.
  - Synchronization ensures that only one thread can access a resource at a time.
- **Lab**:
  - **Write a program to create and run multiple threads using the `Thread` class:**

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

    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start();
    }
}
```

- **Implement thread synchronization using synchronized blocks or methods:**

```
java
Copy
class Counter {
    synchronized void increment() {
        System.out.println("Incremented");
    }
}

public class Test {
    public static void main(String[] args) {
        Counter counter = new Counter();
        Thread t1 = new Thread(() -> counter.increment());
        Thread t2 = new Thread(() -> counter.increment());
        t1.start();
        t2.start();
    }
}
```

- **Use inter-thread communication methods like `wait()`, `notify()`, and `notifyAll()`:**

```
java
Copy
class Producer extends Thread {
    public void run() {
        synchronized (this) {
            try {
                System.out.println("Producer is waiting");
                wait();
                System.out.println("Producer resumed");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public class Test {
    public static void main(String[] args) throws InterruptedException {
        Producer p = new Producer();
        p.start();
        Thread.sleep(1000);
        synchronized (p) {
            p.notify();
        }
    }
}
```

14. File Handling

- **Theory**:
  - Java provides several classes for file handling such as `FileReader`, `FileWriter`, `BufferedReader`, and `BufferedWriter`.
  - These classes allow you to read from and write to files.
  - Serialization and deserialization are used to save and load objects to and from a file.
- **Lab**:
  - **Write a program to read and write content to a file using `FileReader` and `FileWriter`:**

```
java
Copy
import java.io.*;

public class Test {
    public static void main(String[] args) throws IOException {
        FileWriter fw = new FileWriter("test.txt");
        fw.write("Hello, World!");
        fw.close();

        FileReader fr = new FileReader("test.txt");
        int i;
        while ((i = fr.read()) != -1) {
            System.out.print((char) i);
        }
        fr.close();
    }
}
```

- **Implement a program that reads a file line by line using `BufferedReader`:**

```
java
Copy
import java.io.*;

public class Test {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new FileReader("test.txt"));
        String line;
        while ((line = br.readLine()) != null) {
            System.out.println(line);
        }
        br.close();
    }
}
```

- **Create a program that demonstrates object serialization and deserialization:**

```
java
Copy
import java.io.*;

class Person implements Serializable {
    String name;
    int age;

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}

public class Test {
    public static void main(String[] args) throws IOException,
ClassNotFoundException {
        Person p = new Person("John", 25);
        ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("person.txt"));
        out.writeObject(p);
        out.close();

        ObjectInputStream in = new ObjectInputStream(new
FileInputStream("person.txt"));
        Person p2 = (Person) in.readObject();
        System.out.println(p2.name + " " + p2.age);
        in.close();
    }
}
```

15. Collections Framework

- **Theory**:
  - The Collections Framework provides classes like `ArrayList`, `LinkedList`, `HashSet`, `TreeSet`, `HashMap`, and `TreeMap`.
  - It allows you to store and manipulate data in dynamic data structures like lists, sets, and maps.
  - Iterators and ListIterators help to traverse through the elements of these collections.
- **Lab**:
  - **Write a program that demonstrates the use of an `ArrayList` and `LinkedList`:**

```
java
Copy
import java.util.*;

public class Test {
    public static void main(String[] args) {
        ArrayList<String> list1 = new ArrayList<>();
        list1.add("Apple");
        list1.add("Banana");

        LinkedList<String> list2 = new LinkedList<>();
        list2.add("Cat");
        list2.add("Dog");

        System.out.println(list1);
        System.out.println(list2);
    }
}
```

- **Implement a program using `HashSet` to remove duplicate elements from a list:**

```
java
Copy
import java.util.*;

public class Test {
    public static void main(String[] args) {
        Set<String> set = new HashSet<>();
        set.add("Apple");
        set.add("Banana");
        set.add("Apple");

        System.out.println(set);
    }
}
```

- **Create a `HashMap` to store and retrieve key-value pairs:**

```
java
Copy
import java.util.*;

public class Test {
    public static void main(String[] args) {
        HashMap<String, Integer> map = new HashMap<>();
        map.put("John", 25);
        map.put("Alice", 30);

        System.out.println(map.get("John"));
    }
}
```

16. Java Input/Output (I/O)

- **Theory**:
  - Java I/O (Input/Output) provides classes for reading from and writing to files, memory, and devices using streams.
  - `InputStream` and `OutputStream` classes are used for byte-based I/O, while `Reader` and `Writer` are used for character-based I/O.
  - File handling operations allow you to read and write data to files.
- **Lab**:
  - **Write a program to read input from the console using `Scanner`:**

```
java
Copy
import java.util.*;
```

```
public class Test {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter your name: ");
        String name = sc.nextLine();
        System.out.println("Hello, " + name);
    }
}
```

- **Implement a file copy program using `FileInputStream` and `FileOutputStream`:**

```
java
Copy
import java.io.*;

public class Test {
    public static void main(String[] args) throws IOException {
        FileInputStream in = new FileInputStream("source.txt");
        FileOutputStream out = new FileOutputStream("destination.txt");

        int i;
        while ((i = in.read()) != -1) {
            out.write(i);
        }

        in.close();
        out.close();
    }
}
```

- **Create a program that reads from one file and writes the content to another file:**

```
java
Copy
import java.io.*;

public class Test {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new FileReader("source.txt"));
        BufferedWriter bw = new BufferedWriter(new
FileWriter("destination.txt"));

        String line;
        while ((line = br.readLine()) != null) {
            bw.write(line);
            bw.newLine();
        }

        br.close();
        bw.close();
    }
}
```