Java Programming Course

Welcome to the Java Programming course! This course is structured to provide a comprehensive understanding of Java from basic to advanced topics, covering core concepts, object-oriented principles, and data structures and algorithms.

Module 1: Introduction to Java

Lesson 1: Introduction to Java Programming

- Java is an object-oriented, platform-independent, and high-level programming language.
- It follows the Write Once, Run Anywhere (WORA) principle.
- Key features of Java include:
  - Object-Oriented Programming (OOP)
  - Platform Independence (Java Virtual Machine)
  - Rich Standard Library
  - Automatic Memory Management (Garbage Collection)

Installing Java:

1. Download and install JDK from the official Oracle website.
2. Set up environment variables (`JAVA_HOME`, `PATH`).
3. Verify the installation by running `java -version` and `javac -version`.

Lesson 2: Java Basics

- **Syntax and Structure**: Java programs consist of classes, methods, and statements.

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

Explanation:

- **public class HelloWorld**: Declares a public class named **HelloWorld**.

- **public static void main(String[] args)**: This is the entry point of any Java application.
- **System.out.println("Hello, World!");**: This prints the string "Hello, World!" to the console.

Lesson 2: Java Syntax Basics

Variables:

- In Java, variables store data values.

Example:

```
int x = 5; // Integer variable
double y = 3.14; // Floating point number
char c = 'A'; // Character variable
boolean isJavaFun = true; // Boolean variable
String message = "Welcome to Java!"; // String variable
```

Operators:

- Arithmetic Operators:

```
int a = 10, b = 20;
int sum = a + b; // sum = 30
int diff = b - a; // diff = 10
int product = a \* b; // product = 200
int quotient = b / a; // quotient = 2
int remainder = b % a; // remainder = 0
```

Control Flow (if-else, loops):

- If-Else Example:

```
int age = 20;
if (age >= 18) {
System.out.println("Adult");
} else {
System.out.println("Minor");
}
```

- For Loop Example:

```
for (int i = 0; i < 5; i++) {
System.out.println(i); // Prints 0 to 4
}
```

Lesson 3: Functions/Methods in Java

Methods are blocks of code designed to perform a specific task. You can call them whenever needed.

```
public class Example {
// Method with parameters
public static int add(int a, int b) {
return a + b;
}

    public static void main(String[] args) {
        int result = add(10, 5); // Calling the add method
        System.out.println(result); // Prints 15
    }

}
```

Module 2: Object-Oriented Programming (OOP) in Java

Lesson 4: Classes and Objects

- Class: A blueprint for creating objects. An object is an instance of a class.

```
public class Car {
// Attributes (fields)
String model;
int year;

    // Method (behavior)
    public void start() {
        System.out.println(model + " is starting.");
    }

}

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

```
Car myCar = new Car(); // Creating an object of Car
myCar.model = "Toyota";
myCar.year = 2020;
myCar.start(); // Calls the method in Car class
}
}
```

Lesson 5: Inheritance

- Inheritance allows a class to inherit methods and fields from another class. The class that inherits is called the subclass, and the class being inherited from is the superclass.

```
class Animal {
void sound() {
System.out.println("Animals make sounds");
}
}

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

public class Main {
public static void main(String[] args) {
Animal animal = new Animal();
animal.sound(); // Prints "Animals make sounds"

        Dog dog = new Dog();
        dog.sound();  // Prints "Dog barks"
    }

}
```

Lesson 6: Polymorphism

- Polymorphism means "many shapes." It allows one object to take many forms. This can be achieved via method overloading (compile-time polymorphism) and method overriding (runtime polymorphism).

- Method Overloading (same method name, different parameters):

```
class Display {
void show(int a) {
System.out.println("Integer: " + a);
```

```
}

    void show(String a) {
        System.out.println("String: " + a);
    }

}

public class Main {
public static void main(String[] args) {
Display display = new Display();
display.show(5); // Calls show(int)
display.show("Hello"); // Calls show(String)
}
}
```

- Method Overriding:

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

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

public class Main {
public static void main(String[] args) {
Animal myAnimal = new Animal();
myAnimal.sound(); // Prints "Animal sound"

        Animal myDog = new Dog();
        myDog.sound();  // Prints "Bark"
    }

}
```

Lesson 7: Encapsulation

- Encapsulation means bundling data (attributes) and methods (functions) that operate on the data into a single unit called a class.

- It restricts access to some of the object's components by making fields private and providing public getter and setter methods.

```
public class Person {
private String name; // Private field

    // Public getter method
    public String getName() {
        return name;
    }

    // Public setter method
    public void setName(String name) {
        this.name = name;
    }

}

public class Main {
public static void main(String[] args) {
Person person = new Person();
person.setName("John");
System.out.println(person.getName()); // Prints "John"
}
}
```

Module 3: Advanced Java Features

Lesson 9: Collections Framework

- The Java Collections Framework provides classes and interfaces to handle groups of objects. It contains:
  - List: Ordered collection (e.g., ArrayList, LinkedList)
  - Set: Unordered collection with unique elements (e.g., HashSet)
  - Map: Collection of key-value pairs (e.g., HashMap)

Example: Using ArrayList

```
import java.util.ArrayList;

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

```
list.add("Cherry");

        for (String fruit : list) {
            System.out.println(fruit);  // Prints all elements of
the list
        }
    }

}
```

Lesson 10: Exception Handling

- Exception: An event that disrupts the normal flow of the program.
- Try-Catch: Used to handle exceptions.

```
public class Main {
public static void main(String[] args) {
try {
int result = 10 / 0; // Division by zero will throw
ArithmeticException
} catch (ArithmeticException e) {
System.out.println("Error: " + e.getMessage()); // Catches the
exception
} finally {
System.out.println("This will always be executed.");
}
}
}
```

Module 4: Data Structures in Java

Lesson 13: Arrays

- Arrays: Arrays are used to store multiple values in a single variable. They are fixed in size and allow easy access to elements by index.

```
public class Main {
public static void main(String[] args) {
int[] numbers = {1, 2, 3, 4, 5};

        for (int i = 0; i < numbers.length; i++) {
            System.out.println(numbers[i]);  // Prints elements
of the array
        }
```

```
    }

}
```

Lesson 14: Linked Lists

A Linked List is a linear data structure where each element is a separate object called a node. Each node contains a reference (link) to the next node in the sequence.

```
class Node {
int data;
Node next;

    public Node(int data) {
        this.data = data;
        this.next = null;
    }

}

public class LinkedList {
Node head;

    public void add(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
        } else {
            Node temp = head;
            while (temp.next != null) {
                temp = temp.next;
            }
            temp.next = newNode;
        }
    }

    public void display() {
        Node temp = head;
        while (temp != null) {
            System.out.print(temp.data + " ");
            temp = temp.next;
        }
    }

}

public class Main {
public static void main(String[] args) {
LinkedList list = new LinkedList();
list.add(10);
```

```
list.add(20);
list.add(30);
list.display(); // Prints 10 20 30
}
}
```

Lesson 15: Stacks and Queues

1. Stack Implementation: Using Stack class in Java

A **Stack** is a data structure that follows the Last In First Out (LIFO) principle. The last element added to the stack is the first one to be removed.

In Java, the `Stack` class is a part of the `java.util` package. It provides methods to push, pop, peek, and check if the stack is empty.

- **Push**: Adds an element to the top of the stack.
- **Pop**: Removes the top element from the stack.
- **Peek**: Returns the top element without removing it.
- **isEmpty**: Checks if the stack is empty.

Example:

```
import java.util.Stack;

public class StackExample {
    public static void main(String[] args) {
        Stack<Integer> stack = new Stack<>();

        stack.push(10);  // Push element to the stack
        stack.push(20);
        stack.push(30);

        System.out.println("Top element: " + stack.peek()); //
Prints 30

        stack.pop();  // Removes the top element
        System.out.println("Top element after pop: " +
stack.peek()); // Prints 20

        System.out.println("Is stack empty? " + stack.isEmpty());
// false
    }
}
```

2. Queue Implementation: Using Queue interface and LinkedList

A Queue is a data structure that follows the First In First Out (FIFO) principle. The first element added to the queue is the first one to be removed.

In Java, the Queue interface is a part of the java.util package. The most commonly used implementation of the Queue interface is LinkedList. It provides methods like offer(), poll(), peek(), and isEmpty().

- Offer: Adds an element to the end of the queue.
- Poll: Removes and returns the front element of the queue.
- Peek: Returns the front element without removing it.
- isEmpty: Checks if the queue is empty.

```
import java.util.LinkedList;
import java.util.Queue;

public class QueueExample {
    public static void main(String[] args) {
        Queue<Integer> queue = new LinkedList<>();

        queue.offer(10);  // Add elements to the queue
        queue.offer(20);
        queue.offer(30);

        System.out.println("Front element: " + queue.peek()); //
Prints 10

        queue.poll();  // Removes the front element
        System.out.println("Front element after poll: " +
queue.peek()); // Prints 20

        System.out.println("Is queue empty? " + queue.isEmpty());
// false
    }
}
```

3. Applications of Stack and Queue

- a. Balanced Parentheses (Using Stack) One classic application of stacks is checking if the parentheses in a string are balanced. The idea is to traverse the string and use a stack to store opening parentheses. Each time a closing parenthesis is encountered, check if it matches the most recent opening parenthesis by popping from the stack.

Example of balanced parentheses:

```
import java.util.Stack;

public class BalancedParentheses {
    public static boolean isBalanced(String str) {
        Stack<Character> stack = new Stack<>();

        for (char ch : str.toCharArray()) {
            if (ch == '(') {
                stack.push(ch);  // Push opening parentheses onto
stack
            } else if (ch == ')') {
                if (stack.isEmpty()) {
                    return false;  // No matching opening
parenthesis
                }
                stack.pop();  // Pop matching opening parenthesis
            }
        }

        return stack.isEmpty();  // Stack should be empty if
balanced
    }

    public static void main(String[] args) {
        System.out.println(isBalanced("(()())"));  // true
        System.out.println(isBalanced("(()"));     // false
    }
}
```

- b. Queue Management (Using Queue) Queues are commonly used in scheduling tasks or managing resources like print jobs or requests in a server. A simple real-life example is a queue at a bank where customers are served in the order they arrive.

Lesson 16: Trees

1. Binary Trees: Creating and Traversing Binary Trees

A Binary Tree is a tree in which each node has at most two children: left and right. It's a fundamental data structure used in many applications.

Creating a Binary Tree: A binary tree consists of nodes, where each node has a value and two pointers (or references) to its left and right children.

```
class Node {
    int value;
    Node left, right;
```

```
    Node(int value) {
        this.value = value;
        left = right = null;
    }
}

public class BinaryTree {
    Node root;

    public BinaryTree(int value) {
        root = new Node(value);
    }

    // Pre-order Traversal (Root, Left, Right)
    void preOrder(Node node) {
        if (node != null) {
            System.out.print(node.value + " ");
            preOrder(node.left);
            preOrder(node.right);
        }
    }

    // In-order Traversal (Left, Root, Right)
    void inOrder(Node node) {
        if (node != null) {
            inOrder(node.left);
            System.out.print(node.value + " ");
            inOrder(node.right);
        }
    }

    // Post-order Traversal (Left, Right, Root)
    void postOrder(Node node) {
        if (node != null) {
            postOrder(node.left);
            postOrder(node.right);
            System.out.print(node.value + " ");
        }
    }

    public static void main(String[] args) {
        BinaryTree tree = new BinaryTree(1);
        tree.root.left = new Node(2);
        tree.root.right = new Node(3);
        tree.root.left.left = new Node(4);
        tree.root.left.right = new Node(5);

        System.out.print("Pre-order: ");
        tree.preOrder(tree.root);

        System.out.print("\nIn-order: ");
        tree.inOrder(tree.root);

        System.out.print("\nPost-order: ");
```

```
        tree.postOrder(tree.root);
    }
}
```

Output:

```
Pre-order: 1 2 4 5 3
In-order: 4 2 5 1 3
Post-order: 4 5 2 3 1
```

2. Binary Search Tree (BST): Insertion, Deletion, and Searching

A Binary Search Tree (BST) is a binary tree where the left child is smaller than the parent node, and the right child is greater than the parent node. This property allows for efficient searching, insertion, and deletion operations.

- Insertion in a BST: To insert a new node in a BST, start from the root and follow the left or right pointers depending on whether the new node's value is smaller or larger than the current node.

```
class BST {
    Node root;

    BST() {
        root = null;
    }

    // Insert a node in the BST
    public void insert(int value) {
        root = insertRec(root, value);
    }

    private Node insertRec(Node root, int value) {
        if (root == null) {
            root = new Node(value);
            return root;
        }

        if (value < root.value) {
            root.left = insertRec(root.left, value);
        } else if (value > root.value) {
            root.right = insertRec(root.right, value);
        }

        return root;
    }
```

```
    // Search a node in the BST
    public boolean search(int value) {
        return searchRec(root, value);
    }

    private boolean searchRec(Node root, int value) {
        if (root == null) {
            return false;
        }
        if (value == root.value) {
            return true;
        }
        return value < root.value ? searchRec(root.left, value) :
searchRec(root.right, value);
    }

    public static void main(String[] args) {
        BST bst = new BST();
        bst.insert(50);
        bst.insert(30);
        bst.insert(70);
        bst.insert(20);
        bst.insert(40);
        bst.insert(60);
        bst.insert(80);

        System.out.println("Is 40 present in the tree? " +
bst.search(40));  // true
        System.out.println("Is 100 present in the tree? " +
bst.search(100)); // false
    }
}
```

3. AVL Trees: Self-balancing Binary Search Trees

An AVL Tree is a self-balancing binary search tree. It maintains its balance by ensuring that the heights of the two child subtrees of any node differ by no more than one.

Rotations (Left, Right) are used to maintain balance when the height difference between left and right subtrees exceeds 1. The key advantage of AVL trees is that they provide O(log n) time complexity for search, insert, and delete operations due to their self-balancing nature.

4. Heaps: Introduction to Min-Heap and Max-Heap

A Heap is a special tree-based data structure that satisfies the heap property. There are two types of heaps:

- Min-Heap: The value of each parent node is less than or equal to the values of its children. The root contains the smallest element.
- Max-Heap: The value of each parent node is greater than or equal to the values of its children. The root contains the largest element.

Min-Heap Example:

```
import java.util.PriorityQueue;

public class MinHeapExample {
    public static void main(String[] args) {
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();

        minHeap.add(10);
        minHeap.add(20);
        minHeap.add(5);
        minHeap.add(15);

        System.out.println("Min-Heap root: " + minHeap.peek());
// Smallest element (5)
    }
}
```

In Java, a PriorityQueue is often used to implement a Min-Heap. For a Max-Heap, you can use a comparator to invert the order.

Module 5: Algorithms in Java

Lesson 17: Sorting Algorithms

- Bubble Sort: Repeatedly swaps adjacent elements if they are in the wrong order.

```
public class BubbleSort {
public static void bubbleSort(int[] arr) {
int n = arr.length;
for (int i = 0; i < n-1; i++) {
for (int j = 0; j < n-i-1; j++) {
if (arr[j] > arr[j+1]) {
// Swap arr[j] and arr[j+1]
int temp = arr[j];
arr[j] = arr[j+1];
arr[j+1] = temp;
}
}
}
```

```
}

    public static void main(String[] args) {
        int[] arr = {64, 25, 12, 22, 11};
        bubbleSort(arr);

        for (int num : arr) {
            System.out.print(num + " "); // Prints sorted array
        }
    }

}
```

Module 6: Advanced Java Features

Lesson 18: Threads and Concurrency

- Threads are the smallest unit of execution. Java provides built-in support for multithreading, which allows multiple operations to run concurrently, improving performance for tasks like I/O operations, calculations, etc.

Creating a Thread using Thread class:

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

    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start();  // Starts the thread
    }

}
```

Creating a Thread using Runnable interface:

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

    public static void main(String[] args) {
        MyRunnable r = new MyRunnable();
        Thread t = new Thread(r);
        t.start();  // Starts the thread
```

```
    }

}
```

Synchronization in Java: Synchronization is used to control access to resources when multiple threads are involved. Use the synchronized keyword to ensure only one thread accesses a block of code at a time.

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

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

    public synchronized int getCount() {
        return count;
    }

}

public class Main {
public static void main(String[] args) throws
InterruptedException {
Counter counter = new Counter();

        // Thread 1
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });

        // Thread 2
        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });

        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println("Final Count: " + counter.getCount());
    }

}
```

Module 7: Data Structures in Depth

Lesson 19: Stacks

Stack is a linear data structure that follows the Last In First Out (LIFO) principle. Java provides a Stack class, but it's typically recommended to use Deque or LinkedList for stack operations.

- Basic Stack Operations:

```
import java.util.Stack;

public class StackExample {
public static void main(String[] args) {
Stack<Integer> stack = new Stack<>();

        stack.push(10);  // Push elements onto the stack
        stack.push(20);
        stack.push(30);

        System.out.println("Top element: " + stack.peek()); //
Prints 30

        stack.pop();  // Removes the top element
        System.out.println("Top element after pop: " +
stack.peek()); // Prints 20

        System.out.println("Is stack empty? " + stack.isEmpty());
// false
    }

}
```

Lesson 20: Queues

Queue is a linear data structure that follows the First In First Out (FIFO) principle. It is used in scenarios like scheduling, buffering, etc.

- Using Queue interface (LinkedList as implementation):

```
import java.util.LinkedList;
import java.util.Queue;

public class QueueExample {
    public static void main(String[] args) {
        Queue<Integer> queue = new LinkedList<>();
```

```
        queue.offer(10);  // Add elements to the queue
        queue.offer(20);
        queue.offer(30);

        System.out.println("Front element: " + queue.peek()); //
Prints 10

        queue.poll();  // Removes the front element
        System.out.println("Front element after poll: " +
queue.peek()); // Prints 20

        System.out.println("Is queue empty? " + queue.isEmpty());
// false
    }
}
```

Lesson 21: Deque (Double-Ended Queue)

A Deque allows elements to be added or removed from both ends. You can use ArrayDeque or LinkedList in Java to implement Deques.

```
import java.util.ArrayDeque;
import java.util.Deque;

public class DequeExample {
    public static void main(String[] args) {
        Deque<Integer> deque = new ArrayDeque<>();

        deque.addFirst(10);  // Adds to the front
        deque.addLast(20);   // Adds to the end
        deque.addFirst(30);

        System.out.println("Front element: " +
deque.peekFirst()); // 30
        System.out.println("Last element: " + deque.peekLast());
// 20

        deque.removeFirst(); // Removes from the front
        deque.removeLast();  // Removes from the end

        System.out.println("Front element after removal: " +
deque.peekFirst()); // 10
    }
}
```

Module 8: Algorithms and Problem Solving

Lesson 22: Sorting Algorithms (Continued)

Quick Sort: A divide-and-conquer algorithm that divides the array into subarrays and sorts them recursively.

```
class QuickSort {
    public static void quickSort(int[] arr, int low, int high) {
        if (low < high) {
            int pi = partition(arr, low, high);

            quickSort(arr, low, pi - 1);  // Left part
            quickSort(arr, pi + 1, high); // Right part
        }
    }

    public static int partition(int[] arr, int low, int high) {
        int pivot = arr[high];
        int i = (low - 1);  // Index of smaller element

        for (int j = low; j < high; j++) {
            if (arr[j] <= pivot) {
                i++;
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }

        int temp = arr[i + 1];
        arr[i + 1] = arr[high];
        arr[high] = temp;

        return i + 1;
    }

    public static void main(String[] args) {
        int[] arr = {64, 34, 25, 12, 22, 11, 90};
        quickSort(arr, 0, arr.length - 1);

        for (int num : arr) {
            System.out.print(num + " "); // Prints sorted array
        }
    }
}
```

Lesson 23: Searching Algorithms

- Binary Search: A faster way to search an element in a sorted array. It works by repeatedly dividing the search interval in half.

```
class BinarySearch {
    public static int binarySearch(int[] arr, int left, int
right, int target) {
        if (right >= left) {
            int mid = left + (right - left) / 2;

            if (arr[mid] == target) {
                return mid; // Target found
            }

            if (arr[mid] > target) {
                return binarySearch(arr, left, mid - 1, target);
// Search left half
            }

            return binarySearch(arr, mid + 1, right, target); //
Search right half
        }
        return -1; // Target not found
    }

    public static void main(String[] args) {
        int[] arr = {2, 3, 4, 10, 40};
        int target = 10;

        int result = binarySearch(arr, 0, arr.length - 1,
target);

        if (result == -1) {
            System.out.println("Element not found");
        } else {
            System.out.println("Element found at index: " +
result); // Prints 3
        }
    }
}
```

Module 9: Miscellaneous Java Topics

Lesson 24: Lambda Expressions and Functional Programming

Lambda Expressions provide a concise way to represent an instance of a functional interface (interface with a single method).

```
interface MathOperation {
    int operate(int a, int b);
}
```

```
public class Main {
    public static void main(String[] args) {
        // Lambda expression for addition
        MathOperation addition = (a, b) -> a + b;

        // Lambda expression for subtraction
        MathOperation subtraction = (a, b) -> a - b;

        System.out.println("10 + 5 = " + addition.operate(10,
5));
        System.out.println("10 - 5 = " + subtraction.operate(10,
5));
    }
}
```

Lesson 25: Streams API

Streams API: Allows you to process collections of objects in a functional style.

```
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6,
7, 8, 9);

        // Filtering even numbers and printing them
        numbers.stream()
               .filter(n -> n % 2 == 0)
               .forEach(System.out::println);
    }
}
```
