Comprehensive Java Learning Guide

Table of Contents

Java Basic Details

1. Introduction to Java

Java is a high-level, object-oriented programming language developed by Sun Microsystems (now Oracle) in 1995. It follows the principle of "Write Once, Run Anywhere" (WORA), meaning Java code can run on any platform that has a Java Virtual Machine (JVM).

Key Features:

- **Platform Independent**: Java bytecode runs on any system with JVM
- **Object-Oriented**: Everything in Java is an object (except primitives)
- **Secure**: Built-in security features and sandbox environment
- **Robust**: Strong memory management and exception handling
- **Multithreaded**: Built-in support for concurrent programming
- **Portable**: Java code can run on different platforms without modification

2. Java Architecture

JVM (Java Virtual Machine): Runtime environment that executes Java bytecode **JRE (Java Runtime Environment)**: JVM + libraries needed to run Java applications **JDK (Java Development Kit)**: JRE + development tools (compiler, debugger, etc.)

Compilation Process:

1. Source Code (.java) → Java Compiler (javac) → Bytecode (.class)
2. Bytecode → JVM → Machine Code → Execution

3. Data Types

Primitive Data Types

java

```
// Integer types
byte    myByte = 127;        // 8-bit, range: -128 to 127
short   myShort = 32767;     // 16-bit, range: -32,768 to 32,767
int     myInt = 2147483647;  // 32-bit, range: -2^31 to 2^31-1
long    myLong = 9223372036854775807L; // 64-bit, range: -2^63 to 2^63-1

// Floating point types
float   myFloat = 3.14f;     // 32-bit IEEE 754
double  myDouble = 3.14159;  // 64-bit IEEE 754

// Character and boolean
char    myChar = 'A';        // 16-bit Unicode character
boolean myBoolean = true;    // true or false
```

Non-Primitive Data Types

java

```
// Strings
String name = "John Doe";
String greeting = new String("Hello World");

// Arrays
int[] numbers = {1, 2, 3, 4, 5};
String[] names = new String[5];

// Classes and Objects
Person person = new Person("Alice", 25);
```

4. Variables and Constants

Variable Types

java

```java
public class VariableExample {
    // Instance variables (non-static fields)
    private String name;
    private int age;

    // Class variables (static fields)
    private static int totalPersons = 0;

    // Local variables (method variables)
    public void displayInfo() {
        String message = "Person Info"; // Local variable
        System.out.println(message);
    }

    // Parameters
    public void setAge(int newAge) { // newAge is a parameter
        this.age = newAge;
    }
}
```

Constants

java

```java
// Final variables (constants)
public class Constants {
    public static final double PI = 3.14159;
    public static final String COMPANY_NAME = "Tech Corp";
    private final int MAX_SIZE = 100;
}
```

5. Operators

Arithmetic Operators

java

```java
int a = 10, b = 3;
int sum = a + b;        // Addition: 13
int difference = a - b; // Subtraction: 7
int product = a * b;    // Multiplication: 30
int quotient = a / b;   // Division: 3
int remainder = a % b;  // Modulus: 1

// Increment/Decrement
int x = 5;
x++;    // Post-increment: x becomes 6
++x;    // Pre-increment: x becomes 7
x--;    // Post-decrement: x becomes 6
--x;    // Pre-decrement: x becomes 5
```

Comparison Operators

java

```java
int x = 5, y = 10;
boolean isEqual = (x == y);     // false
boolean isNotEqual = (x != y);  // true
boolean isGreater = (x > y);    // false
boolean isLess = (x < y);       // true
boolean isGreaterEqual = (x >= y); // false
boolean isLessEqual = (x <= y);   // true
```

Logical Operators

java

```java
boolean a = true, b = false;
boolean andResult = a && b;  // Logical AND: false
boolean orResult = a || b;   // Logical OR: true
boolean notResult = !a;      // Logical NOT: false
```

Bitwise Operators

java

```java
int a = 5;  // Binary: 101
int b = 3;  // Binary: 011

int bitwiseAnd = a & b;  // 001 = 1
int bitwiseOr = a | b;   // 111 = 7
int bitwiseXor = a ^ b;  // 110 = 6
int bitwiseNot = ~a;     // ...11111010 = -6
int leftShift = a << 1;  // 1010 = 10
int rightShift = a >> 1; // 010 = 2
```

6. Control Flow Statements

Conditional Statements

java

```java
// if-else statement
int score = 85;
if (score >= 90) {
    System.out.println("Grade A");
} else if (score >= 80) {
    System.out.println("Grade B");
} else if (score >= 70) {
    System.out.println("Grade C");
} else {
    System.out.println("Grade F");
}

// switch statement
int day = 3;
switch (day) {
    case 1:
        System.out.println("Monday");
        break;
    case 2:
        System.out.println("Tuesday");
        break;
    case 3:
        System.out.println("Wednesday");
        break;
    default:
        System.out.println("Invalid day");
}

// Ternary operator
String result = (score >= 60) ? "Pass" : "Fail";
```

Loop Statements

java

```
// for loop
for (int i = 0; i < 5; i++) {
    System.out.println("Count: " + i);
}

// Enhanced for loop (for-each)
int[] numbers = {1, 2, 3, 4, 5};
for (int num : numbers) {
    System.out.println(num);
}

// while loop
int i = 0;
while (i < 5) {
    System.out.println("While count: " + i);
    i++;
}

// do-while loop
int j = 0;
do {
    System.out.println("Do-while count: " + j);
    j++;
} while (j < 5);
```

7. Methods

Method Structure

java

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

    // Method without return value (void)
    public void printMessage(String message) {
        System.out.println(message);
    }

    // Method with multiple parameters
    public double calculateArea(double length, double width) {
        return length * width;
    }

    // Method overloading
    public int multiply(int a, int b) {
        return a * b;
    }

    public double multiply(double a, double b) {
        return a * b;
    }

    public int multiply(int a, int b, int c) {
        return a * b * c;
    }
}
```

Static Methods

java

```java
public class MathUtils {
    // Static method - can be called without creating an instance
    public static int max(int a, int b) {
        return (a > b) ? a : b;
    }

    // Usage: MathUtils.max(5, 10);
}
```

8. Arrays

Array Declaration and Initialization

java

```
// Declaration and initialization
int[] numbers = {1, 2, 3, 4, 5};
String[] names = new String[5];

// Array initialization
int[] scores = new int[10];
for (int i = 0; i < scores.length; i++) {
    scores[i] = i * 10;
}

// Multidimensional arrays
int[][] matrix = new int[3][4];
int[][] table = {{1, 2}, {3, 4}, {5, 6}};
```

Array Operations

java

```
public class ArrayExample {
    public static void main(String[] args) {
        int[] arr = {5, 2, 8, 1, 9};

        // Accessing elements
        System.out.println("First element: " + arr[0]);
        System.out.println("Array length: " + arr.length);

        // Iterating through array
        for (int i = 0; i < arr.length; i++) {
            System.out.println("Element " + i + ": " + arr[i]);
        }

        // Enhanced for loop
        for (int element : arr) {
            System.out.println("Element: " + element);
        }
    }
}
```

9. Strings

String Operations

java

```java
public class StringExample {
    public static void main(String[] args) {
        String str1 = "Hello";
        String str2 = "World";
        String str3 = new String("Hello");

        // String concatenation
        String greeting = str1 + " " + str2;
        String greeting2 = str1.concat(" ").concat(str2);

        // String methods
        System.out.println("Length: " + str1.length());
        System.out.println("Uppercase: " + str1.toUpperCase());
        System.out.println("Lowercase: " + str1.toLowerCase());
        System.out.println("Character at index 1: " + str1.charAt(1));
        System.out.println("Substring: " + str1.substring(1, 4));
        System.out.println("Index of 'l': " + str1.indexOf('l'));

        // String comparison
        System.out.println("str1 equals str3: " + str1.equals(str3));
        System.out.println("str1 == str3: " + (str1 == str3));

        // String formatting
        String formatted = String.format("Hello %s, you are %d years old", "John", 25);
    }
}
```

Object-Oriented Programming (OOP)

1. Classes and Objects

Class Definition

java

```java
public class Person {
    // Fields (instance variables)
    private String name;
    private int age;
    private String email;

    // Static field (class variable)
    private static int totalPersons = 0;

    // Constructor
    public Person(String name, int age, String email) {
        this.name = name;
        this.age = age;
        this.email = email;
        totalPersons++;
    }

    // Default constructor
    public Person() {
        this("Unknown", 0, "");
    }

    // Getter methods
    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public String getEmail() {
        return email;
    }

    // Setter methods
    public void setName(String name) {
        this.name = name;
    }

    public void setAge(int age) {
        if (age >= 0) {
            this.age = age;
        }
    }
    public void setEmail(String email) {
        this.email = email;
    }

    // Instance method
    public void displayInfo() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("Email: " + email);
    }

    // Static method
    public static int getTotalPersons() {
        return totalPersons;
    }

    // toString method
    @Override
    public String toString() {
        return "Person{name='" + name + "', age=" + age + ", email='" + email + "'}";
    }
}
```

Object Creation and Usage

java

```
public class PersonExample {
    public static void main(String[] args) {
        // Creating objects
        Person person1 = new Person("Alice", 25, "alice@email.com");
        Person person2 = new Person("Bob", 30, "bob@email.com");
        Person person3 = new Person();

        // Using object methods
        person1.displayInfo();
        person3.setName("Charlie");
        person3.setAge(35);

        // Accessing static members
        System.out.println("Total persons: " + Person.getTotalPersons());
    }
}
```

2. Encapsulation

Encapsulation is the bundling of data and methods that operate on that data within a single unit (class), and restricting direct access to the internal state of an object.

Access Modifiers

```java
public class EncapsulationExample {
    public String publicField;       // Accessible from anywhere
    protected String protectedField; // Accessible within package and subclasses
    String packageField;             // Accessible within package (default)
    private String privateField;     // Accessible only within this class

    // Private fields with public getter/setter methods
    private String name;
    private int age;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        // Validation logic
        if (name != null && !name.trim().isEmpty()) {
            this.name = name;
        }
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        // Validation logic
        if (age >= 0 && age <= 120) {
            this.age = age;
        }
    }
}
```

3. Inheritance

Inheritance allows a class to inherit properties and methods from another class.

Basic Inheritance

java

```java
// Base class (Parent/Super class)
public class Animal {
    protected String name;
    protected int age;

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

    public void eat() {
        System.out.println(name + " is eating");
    }

    public void sleep() {
        System.out.println(name + " is sleeping");
    }

    public void makeSound() {
        System.out.println(name + " makes a sound");
    }
}

// Derived class (Child/Sub class)
public class Dog extends Animal {
    private String breed;

    public Dog(String name, int age, String breed) {
        super(name, age); // Call parent constructor
        this.breed = breed;
    }

    // Method overriding
    @Override
    public void makeSound() {
        System.out.println(name + " barks");
    }

    // New method specific to Dog
    public void wagTail() {
        System.out.println(name + " is wagging tail");
    }

    public String getBreed() {
        return breed;
    }
}

// Another derived class
public class Cat extends Animal {
    private boolean isIndoor;

    public Cat(String name, int age, boolean isIndoor) {
        super(name, age);
        this.isIndoor = isIndoor;
    }

    @Override
    public void makeSound() {
        System.out.println(name + " meows");
    }

    public void climb() {
        System.out.println(name + " is climbing");
    }
}
```

Inheritance Usage

java

```java
public class InheritanceExample {
    public static void main(String[] args) {
        Dog dog = new Dog("Buddy", 3, "Golden Retriever");
        Cat cat = new Cat("Whiskers", 2, true);

        dog.eat();       // Inherited method
        dog.makeSound(); // Overridden method
        dog.wagTail();   // Dog-specific method

        cat.eat();       // Inherited method
        cat.makeSound(); // Overridden method
        cat.climb();     // Cat-specific method
    }
}
```

4. Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common base class.

Method Overriding (Runtime Polymorphism)

java

```java
public class Shape {
    protected String color;

    public Shape(String color) {
        this.color = color;
    }

    public double calculateArea() {
        return 0.0;
    }

    public void draw() {
        System.out.println("Drawing a shape");
    }
}

public class Circle extends Shape {
    private double radius;

    public Circle(String color, double radius) {
        super(color);
        this.radius = radius;
    }

    @Override
    public double calculateArea() {
        return Math.PI * radius * radius;
    }

    @Override
    public void draw() {
        System.out.println("Drawing a circle");
    }
}

public class Rectangle extends Shape {
    private double length;
    private double width;

    public Rectangle(String color, double length, double width) {
        super(color);
        this.length = length;
        this.width = width;
    }

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

    @Override
    public void draw() {
        System.out.println("Drawing a rectangle");
    }
}
```

Polymorphism in Action

java

```java
public class PolymorphismExample {
    public static void main(String[] args) {
        Shape[] shapes = {
            new Circle("Red", 5.0),
            new Rectangle("Blue", 4.0, 6.0),
            new Circle("Green", 3.0)
        };

        // Polymorphic behavior
        for (Shape shape : shapes) {
            shape.draw();                    // Calls appropriate draw method
            System.out.println("Area: " + shape.calculateArea());
        }

        // Method overloading example
        Calculator calc = new Calculator();
        System.out.println(calc.add(5, 3));          // int version
        System.out.println(calc.add(5.5, 3.2));      // double version
        System.out.println(calc.add(1, 2, 3));       // three parameter version
    }
}

class Calculator {
    // Method overloading (Compile-time polymorphism)
    public int add(int a, int b) {
        return a + b;
    }

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

    public int add(int a, int b, int c) {
        return a + b + c;
    }
}
```

5. Abstraction

Abstraction hides complex implementation details and shows only essential features.

Abstract Classes

java

```java
public abstract class Vehicle {
    protected String brand;
    protected int year;

    public Vehicle(String brand, int year) {
        this.brand = brand;
        this.year = year;
    }

    // Abstract method - must be implemented by subclasses
    public abstract void start();
    public abstract void stop();
    public abstract double calculateFuelEfficiency();

    // Concrete method - can be inherited as-is
    public void displayInfo() {
        System.out.println("Brand: " + brand + ", Year: " + year);
    }
}

public class Car extends Vehicle {
    private int numberOfDoors;

    public Car(String brand, int year, int numberOfDoors) {
        super(brand, year);
        this.numberOfDoors = numberOfDoors;
    }

    @Override
    public void start() {
        System.out.println("Car is starting with key ignition");
    }

    @Override
    public void stop() {
        System.out.println("Car is stopping");
    }

    @Override
    public double calculateFuelEfficiency() {
        return 25.5; // Miles per gallon
    }
}

public class Motorcycle extends Vehicle {
    private boolean hasSidecar;
    public Motorcycle(String brand, int year, boolean hasSidecar) {
        super(brand, year);
        this.hasSidecar = hasSidecar;
    }

    @Override
    public void start() {
        System.out.println("Motorcycle is starting with kick start");
    }

    @Override
    public void stop() {
        System.out.println("Motorcycle is stopping");
    }

    @Override
    public double calculateFuelEfficiency() {
        return 45.0; // Miles per gallon
    }
}
```

Interfaces

java

```
public interface Drawable {
    // Abstract method (implicitly public and abstract)
    void draw();

    // Default method (Java 8+)
    default void print() {
        System.out.println("Printing the drawable object");
    }

    // Static method (Java 8+)
    static void showInfo() {
        System.out.println("This is a drawable interface");
    }

    // Constants (implicitly public, static, final)
    String DEFAULT_COLOR = "BLACK";
    int MAX_SIZE = 100;
}

public interface Colorable {
    void setColor(String color);
    String getColor();
}

// Multiple interface implementation
public class Square implements Drawable, Colorable {
    private double side;
    private String color;

    public Square(double side, String color) {
        this.side = side;
        this.color = color;
    }

    @Override
    public void draw() {
        System.out.println("Drawing a square with side " + side);
    }

    @Override
    public void setColor(String color) {
        this.color = color;
    }

    @Override
    public String getColor() {
        return color;
    }
}
```

Collection Framework

The Java Collections Framework provides a set of interfaces and classes to handle groups of objects efficiently.

1. Collection Framework Overview

Collection Hierarchy

```
Collection (I)
├── List (I)
│   ├── ArrayList (C)
│   ├── LinkedList (C)
│   └── Vector (C)
├── Set (I)
│   ├── HashSet (C)
│   ├── LinkedHashSet (C)
│   └── TreeSet (C)
└── Queue (I)
    ├── PriorityQueue (C)
    └── ArrayDeque (C)

Map (I)
├── HashMap (C)
├── LinkedHashMap (C)
├── TreeMap (C)
└── Hashtable (C)
```

2. List Interface

Lists are ordered collections that allow duplicate elements.

ArrayList

java

```java
import java.util.*;

public class ArrayListExample {
    public static void main(String[] args) {
        // Creating ArrayList
        ArrayList<String> fruits = new ArrayList<>();

        // Adding elements
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Orange");
        fruits.add("Apple"); // Duplicates allowed

        // Adding at specific index
        fruits.add(1, "Mango");

        // Accessing elements
        System.out.println("First fruit: " + fruits.get(0));
        System.out.println("Size: " + fruits.size());

        // Modifying elements
        fruits.set(2, "Grapes");

        // Removing elements
        fruits.remove("Banana");
        fruits.remove(0); // Remove by index

        // Iterating through ArrayList
        for (String fruit : fruits) {
            System.out.println(fruit);
        }

        // Using Iterator
        Iterator<String> iterator = fruits.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }

        // Searching
        if (fruits.contains("Apple")) {
            System.out.println("Apple found at index: " + fruits.indexOf("Apple"));
        }

        // Converting to array
        String[] fruitArray = fruits.toArray(new String[0]);
        // Sorting
        Collections.sort(fruits);
        System.out.println("Sorted fruits: " + fruits);
    }
}
```

LinkedList

java

```
import java.util.*;

public class LinkedListExample {
    public static void main(String[] args) {
        LinkedList<Integer> numbers = new LinkedList<>();

        // Adding elements
        numbers.add(10);
        numbers.add(20);
        numbers.add(30);

        // Adding at beginning and end
        numbers.addFirst(5);
        numbers.addLast(40);

        // Accessing elements
        System.out.println("First: " + numbers.getFirst());
        System.out.println("Last: " + numbers.getLast());

        // Removing elements
        numbers.removeFirst();
        numbers.removeLast();

        // Using as Queue
        numbers.offer(50); // Add to end
        int head = numbers.poll(); // Remove from beginning

        // Using as Stack
        numbers.push(60); // Add to beginning
        int top = numbers.pop(); // Remove from beginning

        System.out.println("LinkedList: " + numbers);
    }
}
```

Vector (Legacy, synchronized)

java

```java
import java.util.*;

public class VectorExample {
    public static void main(String[] args) {
        Vector<String> vector = new Vector<>();

        // Adding elements
        vector.add("A");
        vector.add("B");
        vector.add("C");

        // Vector is synchronized (thread-safe)
        System.out.println("Vector: " + vector);
        System.out.println("Capacity: " + vector.capacity());
    }
}
```

3. Set Interface

Sets are collections that do not allow duplicate elements.

HashSet

java

```java
import java.util.*;

public class HashSetExample {
    public static void main(String[] args) {
        HashSet<String> colors = new HashSet<>();

        // Adding elements
        colors.add("Red");
        colors.add("Blue");
        colors.add("Green");
        colors.add("Red"); // Duplicate - won't be added

        System.out.println("HashSet: " + colors);
        System.out.println("Size: " + colors.size());

        // Checking existence
        if (colors.contains("Blue")) {
            System.out.println("Blue is present");
        }

        // Removing elements
        colors.remove("Green");

        // Iterating
        for (String color : colors) {
            System.out.println(color);
        }

        // Set operations
        HashSet<String> moreColors = new HashSet<>();
        moreColors.add("Yellow");
        moreColors.add("Purple");
        moreColors.add("Red");

        // Union
        HashSet<String> union = new HashSet<>(colors);
        union.addAll(moreColors);
        System.out.println("Union: " + union);

        // Intersection
        HashSet<String> intersection = new HashSet<>(colors);
        intersection.retainAll(moreColors);
        System.out.println("Intersection: " + intersection);

        // Difference
        HashSet<String> difference = new HashSet<>(colors);
        difference.removeAll(moreColors);
        System.out.println("Difference: " + difference);
    }
}
```

LinkedHashSet

java

```java
import java.util.*;

public class LinkedHashSetExample {
    public static void main(String[] args) {
        LinkedHashSet<String> orderedSet = new LinkedHashSet<>();

        // Maintains insertion order
        orderedSet.add("First");
        orderedSet.add("Second");
        orderedSet.add("Third");

        System.out.println("LinkedHashSet: " + orderedSet);
        // Output maintains insertion order
    }
}
```

TreeSet

java

```
import java.util.*;

public class TreeSetExample {
    public static void main(String[] args) {
        TreeSet<Integer> sortedNumbers = new TreeSet<>();

        // Adding elements
        sortedNumbers.add(50);
        sortedNumbers.add(20);
        sortedNumbers.add(80);
        sortedNumbers.add(10);

        System.out.println("TreeSet (sorted): " + sortedNumbers);

        // NavigableSet methods
        System.out.println("First: " + sortedNumbers.first());
        System.out.println("Last: " + sortedNumbers.last());
        System.out.println("Lower than 50: " + sortedNumbers.lower(50));
        System.out.println("Higher than 50: " + sortedNumbers.higher(50));

        // Range operations
        System.out.println("HeadSet (< 50): " + sortedNumbers.headSet(50));
        System.out.println("TailSet (>= 50): " + sortedNumbers.tailSet(50));
        System.out.println("SubSet [20, 80): " + sortedNumbers.subSet(20, 80));
    }
}
```

4. Queue Interface

Queues are collections designed for holding elements prior to processing.

PriorityQueue

java

```
import java.util.*;

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

        // Adding elements
        pq.offer(30);
        pq.offer(10);
        pq.offer(50);
        pq.offer(20);

        // Elements are ordered by priority (natural ordering)
        System.out.println("Priority Queue: " + pq);

        // Removing elements (always removes the smallest)
        while (!pq.isEmpty()) {
            System.out.println("Poll: " + pq.poll());
        }

        // Custom comparator
        PriorityQueue<String> stringPQ = new PriorityQueue<>(
            (a, b) -> b.compareTo(a) // Reverse order
        );

        stringPQ.offer("Banana");
        stringPQ.offer("Apple");
        stringPQ.offer("Cherry");

        while (!stringPQ.isEmpty()) {
            System.out.println("Poll: " + stringPQ.poll());
        }
    }
}
```

ArrayDeque

java

```java
import java.util.*;

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

        // Adding elements
        deque.addFirst("First");
        deque.addLast("Last");
        deque.offerFirst("Before First");
        deque.offerLast("After Last");

        System.out.println("Deque: " + deque);

        // Accessing elements
        System.out.println("First element: " + deque.peekFirst());
        System.out.println("Last element: " + deque.peekLast());

        // Removing elements
        System.out.println("Remove first: " + deque.removeFirst());
        System.out.println("Remove last: " + deque.removeLast());

        // Using as Stack
        deque.push("Top");
        System.out.println("Pop: " + deque.pop());

        // Using as Queue
        deque.offer("Queue Element");
        System.out.println("Poll: " + deque.poll());
    }
}
```

5. Map Interface

Maps store key-value pairs and do not allow duplicate keys.

HashMap

java

```java
import java.util.*;

public class HashMapExample {
    public static void main(String[] args) {
        HashMap<String, Integer> ageMap = new HashMap<>();

        // Adding key-value pairs
        ageMap.put("Alice", 25);
        ageMap.put("Bob", 30);
        ageMap.put("Charlie", 35);
        ageMap.put("Alice", 26); // Updates existing key

        // Accessing values
        System.out.println("Alice's age: " + ageMap.get("Alice"));
        System.out.println("David's age: " + ageMap.get("David")); // null
        System.out.println("David's age (with default): " + ageMap.getOrDefault("David", 0));

        // Checking existence
        if (ageMap.containsKey("Bob")) {
            System.out.println("Bob exists in map");
        }

        if (ageMap.containsValue(30)) {
            System.out.println("Age 30 exists in map");
        }

        // Iterating through map
        // Method 1: Key set
        for (String name : ageMap.keySet()) {
            System.out.println(name + " -> " + ageMap.get(name));
        }

        // Method 2: Entry set
        for (Map.Entry<String, Integer> entry : ageMap.entrySet()) {
            System.out.println(entry.getKey() + " -> " + entry.getValue());
        }

        // Method 3: Values
        for (Integer age : ageMap.values()) {
            System.out.println("Age: " + age);
        }

        // Method 4: forEach (Java 8+)
        ageMap.forEach((name, age) ->
            System.out.println(name + " is " + age + " years old")
        );
        // Removing elements
        ageMap.remove("Charlie");
        ageMap.remove("Alice", 26); // Remove only if value matches

        // Useful methods
        System.out.println("Size: " + ageMap.size());
        System.out.println("Is empty: " + ageMap.isEmpty());

        // Advanced operations (Java 8+)
        ageMap.putIfAbsent("Eve", 28);
        ageMap.compute("Bob", (key, val) -> val + 1);
        ageMap.computeIfPresent("Alice", (key, val) -> val + 1);
        ageMap.computeIfAbsent("Frank", key -> key.length() * 10);

        System.out.println("Final map: " + ageMap);
    }
}
```

LinkedHashMap

```java
import java.util.*;

public class LinkedHashMapExample {
    public static void main(String[] args) {
        // Maintains insertion order
        LinkedHashMap<String, String> orderedMap = new LinkedHashMap<>();

        orderedMap.put("First", "1st");
        orderedMap.put("Second", "2nd");
        orderedMap.put("Third", "3rd");

        System.out.println("LinkedHashMap: " + orderedMap);

        // Access order LinkedHashMap
        LinkedHashMap<String, String> accessOrderMap = new LinkedHashMap<>(16, 0.75f, true);
        accessOrderMap.put("A", "Apple");
        accessOrderMap.put("B", "Banana");
        accessOrderMap.put("C", "Cherry");

        accessOrderMap.get("A"); // This moves A to the end
        System.out.println("Access order map: " + accessOrderMap);
    }
}
```

TreeMap

java

```
import java.util.*;

public class TreeMapExample {
    public static void main(String[] args) {
        TreeMap<String, Integer> sortedMap = new TreeMap<>();

        // Adding elements (sorted by key)
        sortedMap.put("Charlie", 35);
        sortedMap.put("Alice", 25);
        sortedMap.put("Bob", 30);

        System.out.println("TreeMap (sorted): " + sortedMap);

        // NavigableMap methods
        System.out.println("First key: " + sortedMap.firstKey());
        System.out.println("Last key: " + sortedMap.lastKey());
        System.out.println("Lower key than 'Bob': " + sortedMap.lowerKey("Bob"));
        System.out.println("Higher key than 'Bob': " + sortedMap.higherKey("Bob"));

        // Range operations
        System.out.println("Head map (< 'Charlie'): " + sortedMap.headMap("Charlie"));
        System.out.println("Tail map (>= 'Bob'): " + sortedMap.tailMap("Bob"));
        System.out.println("Sub map ['Alice', 'Charlie'): " + sortedMap.subMap("Alice", "Charlie"));

        // Descending order
        NavigableMap<String, Integer> descendingMap = sortedMap.descendingMap();
        System.out.println("Descending map: " + descendingMap);
    }
}
```

6. Collection Utility Classes

Collections Class

java

```java
import java.util.*;

public class CollectionsExample {
    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>(Arrays.asList(5, 2, 8, 1, 9));

        // Sorting
        Collections.sort(numbers);
        System.out.println("Sorted: " + numbers);

        Collections.sort(numbers, Collections.reverseOrder());
        System.out.println("Reverse sorted: " + numbers);

        // Searching
        Collections.sort(numbers);
        int index = Collections.binarySearch(numbers, 5);
        System.out.println("Index of 5: " + index);

        // Shuffling
        Collections.shuffle(numbers);
        System.out.println("Shuffled: " + numbers);

        // Min and Max
        System.out.println("Min: " + Collections.min(numbers));
        System.out.println("Max: " + Collections.max(numbers));

        // Frequency
        List<String> words = Arrays.asList("apple", "banana", "apple", "cherry", "apple");
        System.out.println("Frequency of 'apple': " + Collections.frequency(words, "apple"));

        // Replacing
        Collections.replaceAll(words, "apple", "orange");
        System.out.println("After replace: " + words);

        // Rotating
        List<Integer> list = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));
        Collections.rotate(list, 2);
        System.out.println("After rotate by 2: " + list);

        // Immutable collections
        List<String> immutableList = Collections.unmodifiableList(words);
        Set<String> immutableSet = Collections.unmodifiableSet(new HashSet<>(words));

        // Synchronized collections
        List<String> syncList = Collections.synchronizedList(new ArrayList<>());
        Map<String, Integer> syncMap = Collections.synchronizedMap(new HashMap<>());
        // Empty collections
        List<String> emptyList = Collections.emptyList();
        Set<String> emptySet = Collections.emptySet();
        Map<String, String> emptyMap = Collections.emptyMap();

        // Singleton collections
        List<String> singletonList = Collections.singletonList("only");
        Set<String> singletonSet = Collections.singleton("only");
        Map<String, String> singletonMap = Collections.singletonMap("key", "value");
    }
}
```

Arrays Class

java

```java
import java.util.*;

public class ArraysExample {
    public static void main(String[] args) {
        int[] numbers = {5, 2, 8, 1, 9};

        // Sorting
        Arrays.sort(numbers);
        System.out.println("Sorted array: " + Arrays.toString(numbers));

        // Binary search
        int index = Arrays.binarySearch(numbers, 5);
        System.out.println("Index of 5: " + index);

        // Filling
        int[] filled = new int[5];
        Arrays.fill(filled, 42);
        System.out.println("Filled array: " + Arrays.toString(filled));

        // Copying
        int[] copy = Arrays.copyOf(numbers, numbers.length);
        int[] partialCopy = Arrays.copyOfRange(numbers, 1, 4);
        System.out.println("Copy: " + Arrays.toString(copy));
        System.out.println("Partial copy: " + Arrays.toString(partialCopy));

        // Comparing
        int[] another = {1, 2, 5, 8, 9};
        System.out.println("Arrays equal: " + Arrays.equals(numbers, another));

        // Converting to List
        String[] stringArray = {"a", "b", "c"};
        List<String> stringList = Arrays.asList(stringArray);
        System.out.println("List from array: " + stringList);

        // Multi-dimensional arrays
        int[][] matrix = {{1, 2}, {3, 4}};
        System.out.println("2D array: " + Arrays.deepToString(matrix));

        // Parallel operations (Java 8+)
        int[] largeArray = new int[1000];
        Arrays.parallelSetAll(largeArray, index -> index * 2);
        Arrays.parallelSort(largeArray);
    }
}
```

7. Generics in Collections

Generic Collections

java

```
import java.util.*;

public class GenericsExample {
    public static void main(String[] args) {
        // Generic List
        List<String> stringList = new ArrayList<>();
        stringList.add("Hello");
        // stringList.add(42); // Compile-time error

        // Generic Map
        Map<String, List<Integer>> complexMap = new HashMap<>();
        complexMap.put("numbers", Arrays.asList(1, 2, 3));

        // Bounded type parameters
        List<? extends Number> numbers = new ArrayList<Integer>();
        // numbers.add(42); // Cannot add - unknown type

        List<? super Integer> integers = new ArrayList<Number>();
        integers.add(42); // Can add Integer or subtype

        // Wildcard usage
        printList(Arrays.asList("a", "b", "c"));
        printList(Arrays.asList(1, 2, 3));
    }

    // Generic method with wildcard
    public static void printList(List<?> list) {
        for (Object item : list) {
            System.out.print(item + " ");
        }
        System.out.println();
    }

    // Generic method with bounded type parameter
    public static <T extends Comparable<T>> T findMax(List<T> list) {
        if (list.isEmpty()) {
            return null;
        }

        T max = list.get(0);
        for (T item : list) {
            if (item.compareTo(max) > 0) {
                max = item;
            }
        }
        return max;
    }
  }
```

8. Iterator and Enhanced For Loop

Iterator Usage

java

```java
import java.util.*;

public class IteratorExample {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>(Arrays.asList("A", "B", "C", "D"));

        // Iterator
        Iterator<String> iterator = list.iterator();
        while (iterator.hasNext()) {
            String element = iterator.next();
            if (element.equals("B")) {
                iterator.remove(); // Safe removal during iteration
            }
        }

        // ListIterator (bidirectional)
        ListIterator<String> listIterator = list.listIterator();
        while (listIterator.hasNext()) {
            String element = listIterator.next();
            if (element.equals("C")) {
                listIterator.set("Modified C"); // Modify during iteration
                listIterator.add("Added after C"); // Add during iteration
            }
        }

        // Backward iteration
        while (listIterator.hasPrevious()) {
            System.out.println("Previous: " + listIterator.previous());
        }

        // Enhanced for loop (for-each)
        for (String element : list) {
            System.out.println(element);
            // Cannot modify collection during for-each loop
        }

        // Streams (Java 8+)
        list.stream()
            .filter(s -> s.startsWith("A"))
            .forEach(System.out::println);
    }
}
```

9. Concurrent Collections

Thread-Safe Collections

java

```java
import java.util.concurrent.*;
import java.util.*;

public class ConcurrentCollectionsExample {
    public static void main(String[] args) {
        // ConcurrentHashMap
        ConcurrentHashMap<String, Integer> concurrentMap = new ConcurrentHashMap<>();
        concurrentMap.put("key1", 1);
        concurrentMap.put("key2", 2);

        // Atomic operations
        concurrentMap.putIfAbsent("key3", 3);
        concurrentMap.compute("key1", (key, val) -> val + 1);

        // CopyOnWriteArrayList
        CopyOnWriteArrayList<String> cowList = new CopyOnWriteArrayList<>();
        cowList.add("A");
        cowList.add("B");

        // BlockingQueue
        BlockingQueue<String> queue = new LinkedBlockingQueue<>();
        queue.offer("item1");
        queue.offer("item2");

        try {
            String item = queue.take(); // Blocks if empty
            System.out.println("Taken: " + item);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }

        // ConcurrentSkipListMap (sorted concurrent map)
        ConcurrentSkipListMap<String, Integer> skipListMap = new ConcurrentSkipListMap<>();
        skipListMap.put("c", 3);
        skipListMap.put("a", 1);
        skipListMap.put("b", 2);

        System.out.println("Sorted concurrent map: " + skipListMap);
    }
}
```

Decorator Pattern

The Decorator Pattern allows behavior to be added to objects dynamically without altering their structure. It's a structural design pattern that provides a flexible alternative to subclassing for extending functionality.

1. Basic Decorator Pattern Structure

Component Interface

java

```java
// Base component interface
public interface Coffee {
    String getDescription();
    double getCost();
}
```

Concrete Component

java

```java
// Basic coffee implementation
public class SimpleCoffee implements Coffee {
    @Override
    public String getDescription() {
        return "Simple Coffee";
    }

    @Override
    public double getCost() {
        return 2.00;
    }
}
```

Base Decorator

java

```java
// Abstract decorator class
public abstract class CoffeeDecorator implements Coffee {
    protected Coffee coffee;

    public CoffeeDecorator(Coffee coffee) {
        this.coffee = coffee;
    }

    @Override
    public String getDescription() {
        return coffee.getDescription();
    }

    @Override
    public double getCost() {
        return coffee.getCost();
    }
}
```

Concrete Decorators

java

```java
// Milk decorator
public class MilkDecorator extends CoffeeDecorator {
    public MilkDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public String getDescription() {
        return coffee.getDescription() + ", Milk";
    }

    @Override
    public double getCost() {
        return coffee.getCost() + 0.50;
    }
}

// Sugar decorator
public class SugarDecorator extends CoffeeDecorator {
    public SugarDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public String getDescription() {
        return coffee.getDescription() + ", Sugar";
    }

    @Override
    public double getCost() {
        return coffee.getCost() + 0.25;
    }
}

// Whipped cream decorator
public class WhippedCreamDecorator extends CoffeeDecorator {
    public WhippedCreamDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public String getDescription() {
        return coffee.getDescription() + ", Whipped Cream";
    }

    @Override
    public double getCost() {
        return coffee.getCost() + 0.75;
    }
}

// Vanilla decorator
public class VanillaDecorator extends CoffeeDecorator {
    public VanillaDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public String getDescription() {
        return coffee.getDescription() + ", Vanilla";
    }

    @Override
    public double getCost() {
        return coffee.getCost() + 0.60;
    }
}
```

Usage Example

java

```java
public class DecoratorPatternExample {
    public static void main(String[] args) {
        // Start with simple coffee
        Coffee coffee = new SimpleCoffee();
        System.out.println(coffee.getDescription() + " - $" + coffee.getCost());

        // Add milk
        coffee = new MilkDecorator(coffee);
        System.out.println(coffee.getDescription() + " - $" + coffee.getCost());

        // Add sugar
        coffee = new SugarDecorator(coffee);
        System.out.println(coffee.getDescription() + " - $" + coffee.getCost());

        // Add whipped cream
        coffee = new WhippedCreamDecorator(coffee);
        System.out.println(coffee.getDescription() + " - $" + coffee.getCost());

        // Add vanilla
        coffee = new VanillaDecorator(coffee);
        System.out.println(coffee.getDescription() + " - $" + coffee.getCost());

        // Create another combination
        Coffee anotherCoffee = new VanillaDecorator(
            new MilkDecorator(
                new SimpleCoffee()
            )
        );
        System.out.println("\nAnother combination:");
        System.out.println(anotherCoffee.getDescription() + " - $" + anotherCoffee.getCost());
    }
}
```

2. Advanced Decorator Pattern Example

Text Processing System

java

```
// Base text interface
public interface Text {
    String getContent();
    int getLength();
}

// Plain text implementation
public class PlainText implements Text {
    private String content;

    public PlainText(String content) {
        this.content = content;
    }

    @Override
    public String getContent() {
        return content;
    }

    @Override
    public int getLength() {
        return content.length();
    }
}

// Abstract text decorator
public abstract class TextDecorator implements Text {
    protected Text text;

    public TextDecorator(Text text) {
        this.text = text;
    }

    @Override
    public String getContent() {
        return text.getContent();
    }

    @Override
    public int getLength() {
        return text.getLength();
    }
}

// Bold decorator
public class BoldDecorator extends TextDecorator {
    public BoldDecorator(Text text) {
        super(text);
    }

    @Override
    public String getContent() {
        return "<b>" + text.getContent() + "</b>";
    }

    @Override
    public int getLength() {
        return text.getLength() + 7; // Adding <b></b>
    }
}

// Italic decorator
public class ItalicDecorator extends TextDecorator {
    public ItalicDecorator(Text text) {
        super(text);
    }

    @Override
    public String getContent() {
        return "<i>" + text.getContent() + "</i>";
    }

    @Override
    public int getLength() {
        return text.getLength() + 7; // Adding <i></i>
    }
}

// Underline decorator
public class UnderlineDecorator extends TextDecorator {
    public UnderlineDecorator(Text text) {
        super(text);
    }

    @Override
    public String getContent() {
        return "<u>" + text.getContent() + "</u>";
    }

    @Override
    public int getLength() {
        return text.getLength() + 7; // Adding <u></u>
    }
```

```
}

// Color decorator
public class ColorDecorator extends TextDecorator {
    private String color;

    public ColorDecorator(Text text, String color) {
        super(text);
        this.color = color;
    }

    @Override
    public String getContent() {
        return "<span style='color:" + color + "'>" + text.getContent() + "</span>";
    }

    @Override
    public int getLength() {
        return text.getLength() + 27 + color.length(); // Adding span tags
    }
}
```

3. Decorator Pattern with Functional Interface

Functional Decorator Approach

java

```java
import java.util.function.Function;

// Functional approach to decorator pattern
public class FunctionalDecoratorExample {

    // Base interface using functional approach
    @FunctionalInterface
    public interface TextProcessor extends Function<String, String> {
    }

    // Decorator functions
    public static TextProcessor bold() {
        return text -> "<b>" + text + "</b>";
    }

    public static TextProcessor italic() {
        return text -> "<i>" + text + "</i>";
    }

    public static TextProcessor underline() {
        return text -> "<u>" + text + "</u>";
    }

    public static TextProcessor color(String color) {
        return text -> "<span style='color:" + color + "'>" + text + "</span>";
    }

    public static TextProcessor uppercase() {
        return String::toUpperCase;
    }

    public static TextProcessor addPrefix(String prefix) {
        return text -> prefix + text;
    }

    public static TextProcessor addSuffix(String suffix) {
        return text -> text + suffix;
    }

    public static void main(String[] args) {
        String originalText = "Hello World";

        // Compose decorators
        TextProcessor processor = bold()
            .andThen(italic())
            .andThen(color("red"))
            .andThen(addPrefix("Welcome: "))
            .andThen(addSuffix("!!!"));

        String decoratedText = processor.apply(originalText);
        System.out.println("Original: " + originalText);
        System.out.println("Decorated: " + decoratedText);

        // Another combination
        TextProcessor anotherProcessor = underline()
            .andThen(uppercase())
            .andThen(color("blue"));

        String anotherDecorated = anotherProcessor.apply("Java Programming");
        System.out.println("Another decorated: " + anotherDecorated);
    }
}
```

4. I/O Decorator Pattern Example

Stream Decoration

java

```java
import java.io.*;
import java.util.zip.GZIPOutputStream;

public class IODecoratorExample {

    // Custom output stream decorator
    public static class UpperCaseOutputStream extends FilterOutputStream {
        public UpperCaseOutputStream(OutputStream out) {
            super(out);
        }

        @Override
        public void write(int b) throws IOException {
            super.write(Character.toUpperCase((char) b));
        }

        @Override
        public void write(byte[] b, int off, int len) throws IOException {
            for (int i = off; i < off + len; i++) {
                write(b[i]);
            }
        }
    }

    // Encryption decorator (simple Caesar cipher)
    public static class EncryptionOutputStream extends FilterOutputStream {
        private int shift;

        public EncryptionOutputStream(OutputStream out, int shift) {
            super(out);
            this.shift = shift;
        }

        @Override
        public void write(int b) throws IOException {
            if (Character.isLetter(b)) {
                char base = Character.isUpperCase(b) ? 'A' : 'a';
                b = (char) ((b - base + shift) % 26 + base);
            }
            super.write(b);
        }
    }

    public static void main(String[] args) {
        try {
            // Layered stream decorators
            FileOutputStream fileOut = new FileOutputStream("decorated_output.txt");
            BufferedOutputStream bufferedOut = new BufferedOutputStream(fileOut);
            UpperCaseOutputStream upperOut = new UpperCaseOutputStream(bufferedOut);
            EncryptionOutputStream encryptedOut = new EncryptionOutputStream(upperOut, 3);

            // Write to decorated stream
            String message = "Hello World! This is a test message.";
            encryptedOut.write(message.getBytes());
            encryptedOut.close();

            System.out.println("Message written to file with decorations applied.");

            // Read and display the result
            FileInputStream fileIn = new FileInputStream("decorated_output.txt");
            BufferedInputStream bufferedIn = new BufferedInputStream(fileIn);

            int data;
            System.out.print("File content: ");
            while ((data = bufferedIn.read()) != -1) {
                System.out.print((char) data);
            }
            bufferedIn.close();

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

5. Real-world Decorator Pattern Applications

Web Component Decoration

java

```java
// HTML component decorator pattern
public interface HtmlComponent {
    String render();
}

// Basic HTML elements
public class DivComponent implements HtmlComponent {
    private String content;

    public DivComponent(String content) {
        this.content = content;
    }

    @Override
    public String render() {
        return "<div>" + content + "</div>";
    }
}

// CSS class decorator
public class CssClassDecorator implements HtmlComponent {
    private HtmlComponent component;
    private String cssClass;

    public CssClassDecorator(HtmlComponent component, String cssClass) {
        this.component = component;
        this.cssClass = cssClass;
    }

    @Override
    public String render() {
        String html = component.render();
        // Insert class attribute
        return html.replaceFirst(">", " class='" + cssClass + "'>");
    }
}

// ID decorator
public class IdDecorator implements HtmlComponent {
    private HtmlComponent component;
    private String id;

    public IdDecorator(HtmlComponent component, String id) {
        this.component = component;
        this.id = id;
    }
    @Override
    public String render() {
        String html = component.render();
        return html.replaceFirst(">", " id='" + id + "'>");
    }
}

// Style decorator
public class StyleDecorator implements HtmlComponent {
    private HtmlComponent component;
    private String style;

    public StyleDecorator(HtmlComponent component, String style) {
        this.component = component;
        this.style = style;
    }

    @Override
    public String render() {
        String html = component.render();
        return html.replaceFirst(">", " style='" + style + "'>");
    }
}

// Usage example
public class WebComponentExample {
    public static void main(String[] args) {
        // Create basic component
        HtmlComponent div = new DivComponent("Hello World");
        System.out.println("Basic: " + div.render());

        // Add decorations
        HtmlComponent styledDiv = new StyleDecorator(
            new CssClassDecorator(
                new IdDecorator(div, "main-content"),
                "container"
            ),
            "color: red; font-size: 16px;"
        );

        System.out.println("Decorated: " + styledDiv.render());
    }
}
```

6. Decorator Pattern Benefits and Considerations

Benefits:

1. **Flexibility**: Add or remove behaviors at runtime
2. **Composition over Inheritance**: Avoid class explosion
3. **Single Responsibility**: Each decorator has one responsibility
4. **Open/Closed Principle**: Open for extension, closed for modification

Considerations:

1. **Complexity**: Can make code harder to understand
2. **Performance**: Multiple layers may impact performance
3. **Debugging**: Stack traces can be deep and complex
4. **Interface Consistency**: All decorators must implement the same interface

Best Practices:

java

```java
// Use builder pattern with decorators for complex configurations
public class CoffeeBuilder {
    private Coffee coffee;

    public CoffeeBuilder() {
        this.coffee = new SimpleCoffee();
    }

    public CoffeeBuilder withMilk() {
        this.coffee = new MilkDecorator(this.coffee);
        return this;
    }

    public CoffeeBuilder withSugar() {
        this.coffee = new SugarDecorator(this.coffee);
        return this;
    }

    public CoffeeBuilder withWhippedCream() {
        this.coffee = new WhippedCreamDecorator(this.coffee);
        return this;
    }

    public CoffeeBuilder withVanilla() {
        this.coffee = new VanillaDecorator(this.coffee);
        return this;
    }

    public Coffee build() {
        return this.coffee;
    }
}

// Usage with builder
public class CoffeeBuilderExample {
    public static void main(String[] args) {
        Coffee coffee = new CoffeeBuilder()
            .withMilk()
            .withSugar()
            .withVanilla()
            .build();

        System.out.println(coffee.getDescription() + " - $" + coffee.getCost());
    }
}
```

Summary

This comprehensive guide covers:

1. **Java Basics**: Data types, variables, operators, control flow, methods, arrays, and strings
2. **Object-Oriented Programming**: Classes, objects, encapsulation, inheritance, polymorphism, and abstraction
3. **Collection Framework**: Lists, Sets, Maps, Queues