# **Complete Java Programming Notes - Basic to Advanced**

#### **Table of Contents**

- 1. Variables and Data Types
- 2. Conditions
- 3. Loops
- 4. Arrays
- 5. 2D Arrays
- 6. Reference Types
- 7. Shallow vs Deep Copy
- 8. Pass by Value vs Pass by Reference
- 9. Classes and Objects
- 10. Annotations
- 11. Collection Framework

## 1. Variables and Data Types

#### **Variables**

Variables are containers that store data values. In Java, variables must be declared with a specific data type.

```
java

// Variable declaration and initialization
int age = 25;

String name = "John";
double salary = 50000.50;
```

## **Primitive Data Types**

#### **Numeric Types**

java		
3		

#### **Non-Primitive Data Types**

```
java

// String

String text = "Hello World";

// Arrays

int[] numbers = {1, 2, 3, 4, 5};

// Objects

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

## **Variable Scope**

```
public class VariableScope {
    static int classVariable = 10; // Class variable
    int instanceVariable = 20; // Instance variable

    public void method() {
        int localVariable = 30; // Local variable
        // Local variable scope ends here
    }
}
```

#### 2. Conditions

#### **If-Else Statements**

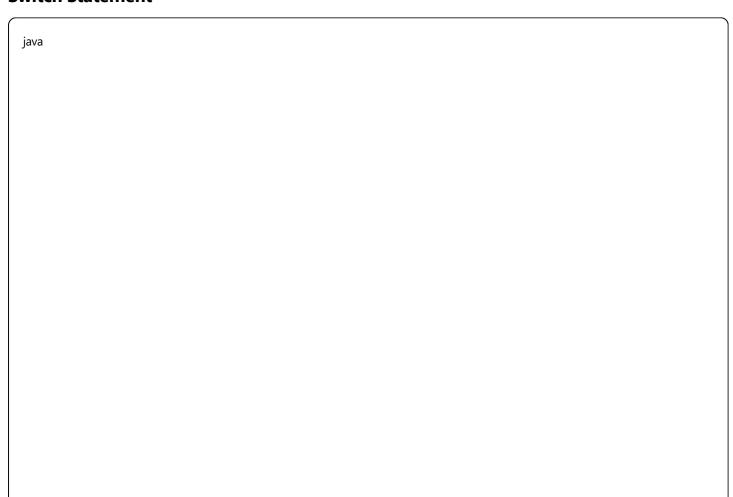
```
int score = 85;

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");
}
```

# **Ternary Operator**

```
int age = 18;
String status = (age >= 18) ? "Adult" : "Minor";
System.out.println(status);
```

#### **Switch Statement**



```
char grade = 'B';
switch (grade) {
  case 'A':
     System.out.println("Excellent!");
     break;
  case 'B':
     System.out.println("Good job!");
     break:
  case 'C':
     System.out.println("Keep trying!");
     break:
  default:
     System.out.println("Invalid grade");
// Enhanced Switch (Java 14+)
String result = switch (grade) {
  case 'A' -> "Excellent!";
  case 'B' -> "Good job!";
  case 'C' -> "Keep trying!";
  default -> "Invalid grade";
};
```

## **Logical Operators**

```
java

boolean a = true, b = false;

// AND operator
if (a && b) { /* Both must be true */ }

// OR operator
if (a || b) { /* At least one must be true */ }

// NOT operator
if (la) { /* Opposite of a */ }
```

# 3. Loops

## **For Loop**

```
java
```

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

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

# **While Loop**

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

# **Do-While Loop**

```
int num = 1;
do {
    System.out.println("Number: " + num);
    num++;
} while (num <= 5);</pre>
```

# **Loop Control Statements**

```
for (int i = 0; i < 10; i++) {
    if (i == 3) {
        continue; // Skip iteration when i = 3
    }
    if (i == 7) {
        break; // Exit loop when i = 7
    }
    System.out.println(i);
}</pre>
```

## **Nested Loops**

```
java

// Print multiplication table

for (int i = 1; i <= 5; i++) {
    for (int j = 1; j <= 5; j++) {
        System.out.print((i * j) + "\t");
    }

    System.out.println();
}</pre>
```

# 4. Arrays

## **Array Declaration and Initialization**

```
java

// Method 1: Declare then initialize
int[] numbers;
numbers = new int[5];

// Method 2: Declare and initialize
int[] scores = new int[5];

// Method 3: Declare with values
int[] grades = {85, 90, 78, 92, 88};

// Method 4: Using new keyword with values
int[] marks = new int[]{75, 80, 85, 90, 95};
```

# **Array Operations**

java		

```
int[] arr = {10, 20, 30, 40, 50};

// Access elements
System.out.println("First element: " + arr[0]);
System.out.println("Last element: " + arr[arr.length - 1]);

// Modify elements
arr[2] = 35;

// Array length
System.out.println("Array length: " + arr.length);

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

// Enhanced for loop
for (int value : arr) {
    System.out.println(value);
}</pre>
```

## **Array Utility Methods**

```
import java.util.Arrays;
int[] numbers = {5, 2, 8, 1, 9};

// Sort array
Arrays.sort(numbers);
System.out.println(Arrays.toString(numbers));

// Binary search (array must be sorted)
int index = Arrays.binarySearch(numbers, 8);

// Copy array
int[] copy = Arrays.copyOf(numbers, numbers.length);

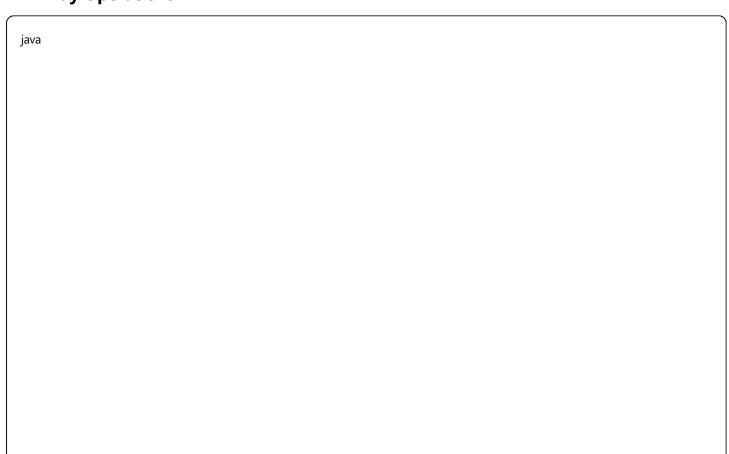
// Fill array
int[] filled = new int[5];
Arrays.fill(filled, 10);
```

# 5. 2D Arrays

#### **Declaration and Initialization**

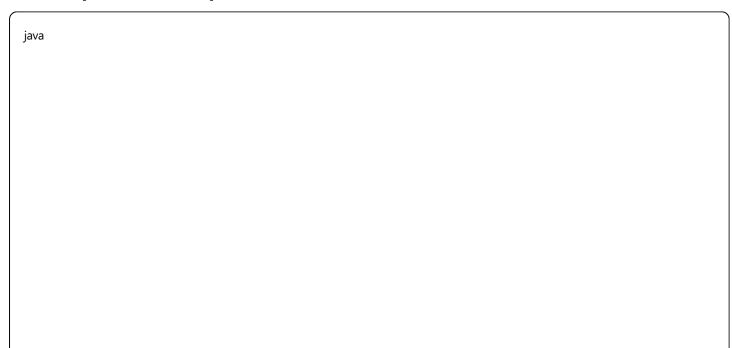
```
java
// Method 1: Declare then initialize
int[][] matrix;
matrix = new int[3][4];
// Method 2: Declare and initialize
int[][] grid = new int[3][3];
// Method 3: Initialize with values
int[][] table = {
  {1, 2, 3},
  {4, 5, 6},
  {7, 8, 9}
};
// Method 4: Jagged array (different column sizes)
int[][] jagged = {
  {1, 2},
  \{3, 4, 5\},\
  {6, 7, 8, 9}
};
```

# **2D Array Operations**



```
int[][] matrix = {
  {1, 2, 3},
  {4, 5, 6},
  {7, 8, 9}
// Access elements
System.out.println("Element at [1][2]: " + matrix[1][2]);
// Get dimensions
int rows = matrix.length;
int cols = matrix[0].length;
// Iterate through 2D array
for (int i = 0; i < matrix.length; i++) {
  for (int j = 0; j < matrix[i].length; <math>j++) {
     System.out.print(matrix[i][j] + " ");
  System.out.println();
// Enhanced for loop for 2D arrays
for (int[] row : matrix) {
  for (int value : row) {
     System.out.print(value + " ");
  System.out.println();
```

# **Matrix Operations Example**



```
public class MatrixOperations {
  public static void printMatrix(int[][] matrix) {
    for (int[] row : matrix) {
      for (int value : row) {
            System.out.print(value + "\t");
        }
        System.out.println();
    }
}

public static int[][] addMatrices(int[][] a, int[][] b) {
    int rows = a.length;
    int cols = a[0].length;
    int cols = a[0].length;
    int[][] result = new int[rows][cols];

for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            result[i][j] = a[i][j] + b[i][j];
        }
    }
    return result;
}
</pre>
```

# 6. Reference Types

## **Understanding Reference vs Primitive**

```
java

// Primitive types store actual values
int a = 10;
int b = a; // b gets a copy of a's value
a = 20; // Changing a doesn't affect b

// Reference types store memory addresses

String str1 = new String("Hello");

String str2 = str1; // str2 references same object as str1

// Both str1 and str2 point to the same object in memory
```

# **Object References**

```
java
```

```
class Person {
  String name;
  int age;
  Person(String name, int age) {
    this.name = name;
    this.age = age;
public class ReferenceExample {
  public static void main(String[] args) {
    Person person1 = new Person("John", 25);
    Person person2 = person1; // person2 references same object
    person2.age = 30; // Changes the object that both references point to
    System.out.println(person1.age); // Output: 30
    // Creating a new object
    person2 = new Person("Jane", 28); // person2 now references a new object
    System.out.println(person1.age); // Output: 30 (unchanged)
    System.out.println(person2.age); // Output: 28
```

#### **Null References**

```
String str = null; // Reference pointing to nothing
if (str == null) {
    System.out.println("String is null");
}

// Attempting to use null reference causes NullPointerException
// str.length(); // This would throw NullPointerException
```

# 7. Shallow vs Deep Copy

# **Shallow Copy**

```
class Address {
  String city;
  String country;
  Address(String city, String country) {
    this.city = city;
    this.country = country;
class Person implements Cloneable {
  String name;
  Address address;
  Person(String name, Address address) {
    this.name = name:
    this.address = address:
  // Shallow copy - only copies references
  @Override
  protected Object clone() throws CloneNotSupportedException {
    return super.clone();
// Shallow copy example
Person original = new Person("John", new Address("New York", "USA"));
Person shallowCopy = (Person) original.clone();
// Both objects share the same Address reference
shallowCopy.address.city = "Boston";
System.out.println(original.address.city); // Output: Boston
```

## **Deep Copy**

```
class Person implements Cloneable {
  String name;
  Address address:
  Person(String name, Address address) {
    this.name = name;
    this.address = address:
  // Deep copy - creates new objects for all references
  @Override
  protected Object clone() throws CloneNotSupportedException {
     Person cloned = (Person) super.clone();
    cloned.address = new Address(this.address.city, this.address.country);
    return cloned;
// Deep copy example
Person original = new Person("John", new Address("New York", "USA"));
Person deepCopy = (Person) original.clone();
// Each object has its own Address instance
deepCopy.address.city = "Boston";
System.out.println(original.address.city); // Output: New York
System.out.println(deepCopy.address.city); // Output: Boston
```

## **Copy using Constructor**

```
class Person {
    String name;
    Address address;

// Original constructor
Person(String name, Address address) {
    this.name = name;
    this.address = address;
}

// Copy constructor for deep copy
Person(Person other) {
    this.name = other.name;
    this.address = new Address(other.address.city, other.address.country);
}
```

# 8. Pass by Value vs Pass by Reference

### **Java Uses Pass by Value**

Java always passes arguments by value, but the behavior differs between primitives and objects.

### **Pass by Value with Primitives**

```
public class PassByValueExample {
   public static void modifyPrimitive(int x) {
      x = 100; // This only changes the local copy
      System.out.println("Inside method: " + x); // Output: 100
   }

   public static void main(String[] args) {
      int num = 50;
      modifyPrimitive(num);
      System.out.println("Outside method: " + num); // Output: 50
   }
}
```

## **Pass by Value with Objects**

```
java
```

```
class Student {
  String name;
  int age;
  Student(String name, int age) {
    this.name = name;
    this.age = age;
public class PassByReferenceExample {
  // The reference is passed by value
  public static void modifyObject(Student s) {
    s.name = "Modified"; // This changes the original object
    s.age = 99;
  // Reassigning the reference doesn't affect original
  public static void reassignObject(Student s) {
    s = new Student("New Student", 25); // Local reference change
  public static void main(String[] args) {
    Student student = new Student("John", 20);
    modifyObject(student);
    System.out.println(student.name); // Output: Modified
    System.out.println(student.age); // Output: 99
    reassignObject(student);
    System.out.println(student.name); // Output: Modified (unchanged)
```

### **Array Parameter Passing**

```
public class ArrayPassing {
  public static void modifyArray(int[] arr) {
    arr[0] = 999; // Modifies original array
}

public static void reassignArray(int[] arr) {
    arr = new int[]{100, 200, 300); // Doesn't affect original
}

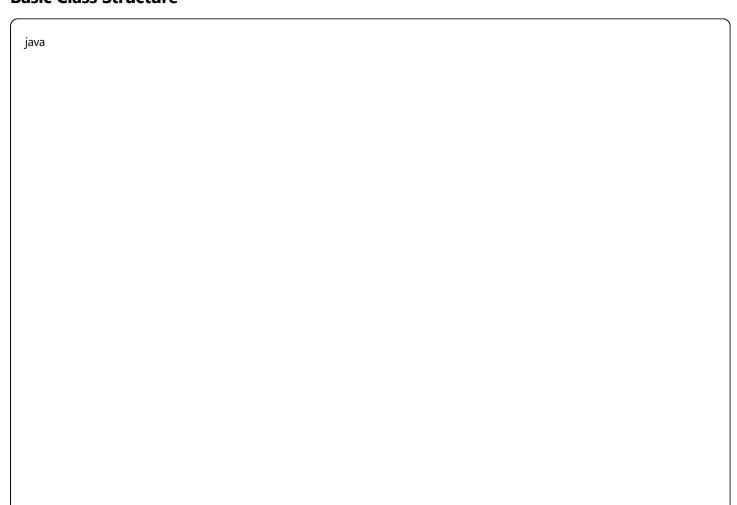
public static void main(String[] args) {
    int[] numbers = {1, 2, 3, 4, 5};

    modifyArray(numbers);
    System.out.println(numbers[0]); // Output: 999

    reassignArray(numbers);
    System.out.println(numbers[0]); // Output: 999 (unchanged)
}
}
```

# 9. Classes and Objects

## **Basic Class Structure**



```
public class Car {
  // Instance variables (attributes)
  private String brand;
  private String model;
  private int year;
  private double price;
  // Static variable (class variable)
  private static int carCount = 0;
  // Constructor
  public Car(String brand, String model, int year, double price) {
    this.brand = brand;
    this.model = model;
    this.year = year;
    this.price = price;
    carCount++; // Increment car count
 // Default constructor
  public Car() {
    this("Unknown", "Unknown", 0, 0.0);
  // Instance methods
  public void startEngine() {
    System.out.println(brand + " " + model + " engine started!");
  public void displayInfo() {
    System.out.println("Brand: " + brand);
    System.out.println("Model: " + model);
    System.out.println("Year: " + year);
    System.out.println("Price: $" + price);
  // Getter methods
  public String getBrand() { return brand; }
  public String getModel() { return model; }
  public int getYear() { return year; }
  public double getPrice() { return price; }
  // Setter methods
  public void setBrand(String brand) { this.brand = brand; }
  public void setModel(String model) { this.model = model; }
  public void setYear(int year) { this.year = year; }
```

```
public void setPrice(double price) { this.price = price; }

// Static method
public static int getCarCount() {
    return carCount;
}

// Override toString method
@Override
public String toString() {
    return year + " " + brand + " " + model + " - $" + price;
}
```

## **Object Creation and Usage**

```
java
public class CarDemo {
  public static void main(String[] args) {
     // Creating objects
     Car car1 = new Car("Toyota", "Camry", 2023, 25000.0);
     Car car2 = new Car("Honda", "Civic", 2022, 22000.0);
     Car car3 = new Car(); // Using default constructor
     // Using objects
     car1.startEngine();
     car1.displayInfo();
     // Using getters and setters
     car3.setBrand("Ford");
     car3.setModel("Mustang");
     System.out.println("Car3 brand: " + car3.getBrand());
     // Using static method
     System.out.println("Total cars created: " + Car.getCarCount());
     // Using toString
     System.out.println(car1.toString());
```

#### **Inheritance**

```
// Base class
class Vehicle {
  protected String brand;
  protected int year;
  public Vehicle(String brand, int year) {
     this.brand = brand;
     this.year = year;
  public void start() {
     System.out.println("Vehicle started");
// Derived class
class Car extends Vehicle {
  private int doors;
  public Car(String brand, int year, int doors) {
     super(brand, year); // Call parent constructor
     this.doors = doors:
  @Override
  public void start() {
     System.out.println("Car engine started");
  public void honk() {
     System.out.println("Car honking");
```

#### **Abstract Classes and Interfaces**

```
// Abstract class
abstract class Shape {
  protected String color;
  public Shape(String color) {
     this.color = color;
  // Abstract method
  public abstract double calculateArea();
  // Concrete method
  public void displayColor() {
     System.out.println("Color: " + color);
// Interface
interface Drawable {
  void draw();
  default void print() {
     System.out.println("Printing shape");
// Implementation
class Circle extends Shape implements Drawable {
  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");
```

#### 10. Annotations

#### **Built-in Annotations**

```
java
// @Override - indicates method overrides parent method
class Parent {
  public void display() {
     System.out.println("Parent display");
class Child extends Parent {
  @Override
  public void display() {
     System.out.println("Child display");
// @Deprecated - marks method as deprecated
class Calculator {
  @Deprecated
  public int add(int a, int b) {
     return a + b;
  public int sum(int a, int b) {
     return a + b;
// @SuppressWarnings - suppresses compiler warnings
class WarningExample {
  @SuppressWarnings("unchecked")
  public void method() {
     List list = new ArrayList(); // Raw type warning suppressed
     list.add("Hello");
```

#### **Custom Annotations**

```
import java.lang.annotation.*;
// Define custom annotation
@Retention(RetentionPolicy.RUNTIME) // Available at runtime
@Target(ElementType.METHOD)
                                     // Can be applied to methods
public @interface Timer {
  String value() default "default";
  int maxTime() default 1000;
// Using custom annotation
class Service {
  @Timer(value = "database", maxTime = 5000)
  public void fetchData() {
     System.out.println("Fetching data...");
  @Timer // Using default values
  public void processData() {
     System.out.println("Processing data...");
// Reading annotations using reflection
import java.lang.reflect.Method;
public class AnnotationProcessor {
  public static void main(String[] args) {
     Class < Service > clazz = Service.class;
     Method[] methods = clazz.getMethods();
     for (Method method: methods) {
       if (method.isAnnotationPresent(Timer.class)) {
         Timer timer = method.getAnnotation(Timer.class);
         System.out.println("Method: " + method.getName());
         System.out.println("Timer value: " + timer.value());
         System.out.println("Max time: " + timer.maxTime());
```

#### **Meta-Annotations**

```
// @Retention - specifies how long annotation is retained
@Retention(RetentionPolicy.SOURCE) // Discarded by compiler
@Retention(RetentionPolicy.CLASS) // Stored in class file
@Retention(RetentionPolicy.RUNTIME) // Available at runtime

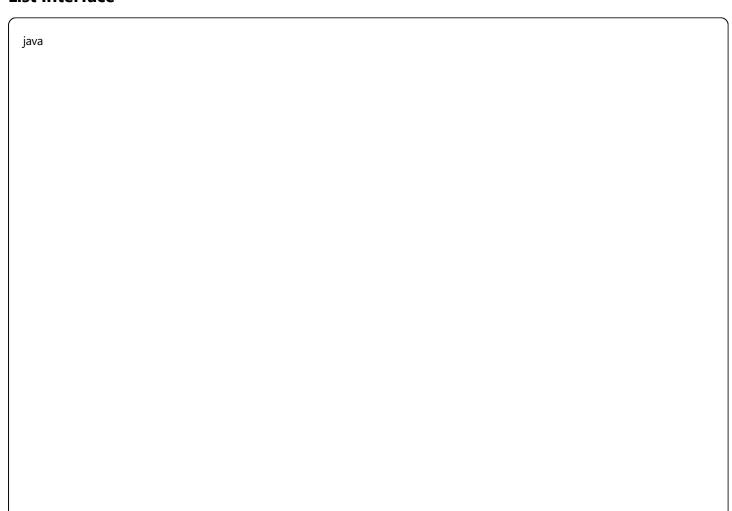
// @Target - specifies where annotation can be applied
@Target(ElementType.TYPE) // Classes, interfaces
@Target(ElementType.METHOD) // Methods
@Target(ElementType.FIELD) // Fields
@Target(ElementType.PARAMETER) // Parameters

// @Inherited - annotation is inherited by subclasses
@Inherited
@interface ParentAnnotation {}

// @Documented - annotation appears in JavaDoc
@Documented
@interface DocumentedAnnotation {}
```

# 11. Collection Framework

#### **List Interface**

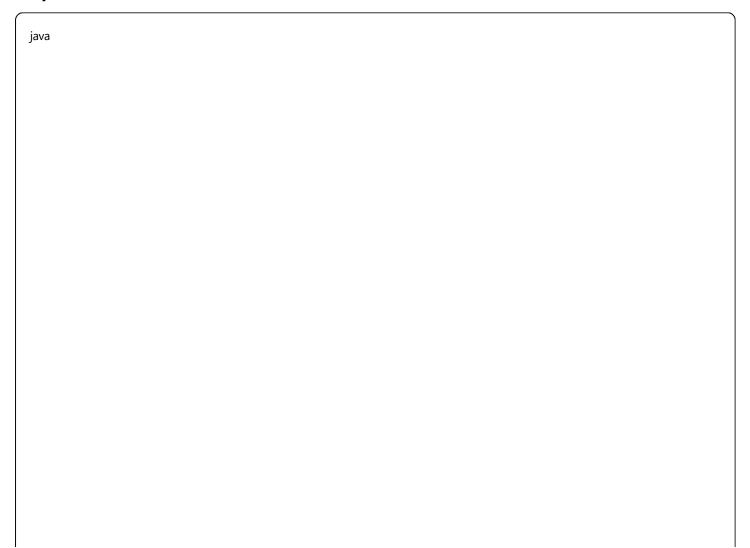


```
import java.util.*;
// ArrayList - dynamic array
List<String> arrayList = new ArrayList<>();
arrayList.add("Apple");
arrayList.add("Banana");
arrayList.add("Cherry");
arrayList.add(1, "Apricot"); // Insert at index 1
// LinkedList - doubly linked list
List<String> linkedList = new LinkedList<>();
linkedList.add("First");
linkedList.add("Second");
((LinkedList<String>) linkedList).addFirst("Zero");
// Vector - synchronized ArrayList
List < String > vector = new Vector < > ();
vector.add("Element1");
vector.add("Element2");
// Common List operations
System.out.println("Size: " + arrayList.size());
System.out.println("Element at index 0: " + arrayList.get(0));
arrayList.remove("Banana");
System.out.println("Contains Apple: " + arrayList.contains("Apple"));
// Iteration
for (String fruit : arrayList) {
  System.out.println(fruit);
```

#### **Set Interface**

```
// HashSet - no duplicates, no order guarantee
Set<Integer> hashSet = new HashSet<>();
hashSet.add(10);
hashSet.add(20);
hashSet.add(10); // Duplicate, won't be added
System.out.println("HashSet: " + hashSet);
// LinkedHashSet - maintains insertion order
Set<String> linkedHashSet = new LinkedHashSet<>();
linkedHashSet.add("Third");
linkedHashSet.add("First");
linkedHashSet.add("Second");
System.out.println("LinkedHashSet: " + linkedHashSet);
// TreeSet - sorted set
Set < String > treeSet = new TreeSet < > ();
treeSet.add("Zebra");
treeSet.add("Apple");
treeSet.add("Banana");
System.out.println("TreeSet: " + treeSet); // Sorted order
```

## **Map Interface**



```
// HashMap - key-value pairs
Map < String, Integer > hashMap = new HashMap < > ();
hashMap.put("John", 25);
hashMap.put("Jane", 30);
hashMap.put("Bob", 35);
System.out.println("John's age: " + hashMap.get("John"));
System.out.println("Contains key 'Jane': " + hashMap.containsKey("Jane"));
// Iterate through map
for (Map.Entry < String, Integer > entry : hashMap.entrySet()) {
  System.out.println(entry.getKey() + " -> " + entry.getValue());
// LinkedHashMap - maintains insertion order
Map<String> linkedHashMap = new LinkedHashMap<>>();
linkedHashMap.put("First", "A");
linkedHashMap.put("Second", "B");
linkedHashMap.put("Third", "C");
// TreeMap - sorted by keys
Map<Integer, String> treeMap = new TreeMap<>();
treeMap.put(3, "Three");
treeMap.put(1, "One");
treeMap.put(2, "Two");
System.out.println("TreeMap: " + treeMap); // Sorted by keys
```

#### **Queue Interface**

```
// LinkedList as Queue
Queue < String > queue = new LinkedList <> ();
queue.offer("First"); // Add to rear
queue.offer("Second");
queue.offer("Third");
System.out.println("Front element: " + queue.peek()); // View front
System.out.println("Removed: " + queue.poll()); // Remove from front
// PriorityQueue - heap-based priority queue
Queue < Integer > priorityQueue = new PriorityQueue < > ();
priorityQueue.offer(30);
priorityQueue.offer(10);
priorityQueue.offer(20);
while (!priorityQueue.isEmpty()) {
  System.out.println(priorityQueue.poll()); // Outputs in priority order
// Deque - double-ended queue
Deque < String > deque = new ArrayDeque < > ();
deque.addFirst("Middle");
deque.addFirst("First");
deque.addLast("Last");
System.out.println("Deque: " + deque);
```

## **Utility Classes**

```
// Collections utility class
List<Integer> numbers = Arrays.asList(5, 2, 8, 1, 9);
// Sort
Collections.sort(numbers);
System.out.println("Sorted: " + numbers);
// Reverse
Collections.reverse(numbers);
System.out.println("Reversed: " + numbers);
// Shuffle
Collections.shuffle(numbers);
System.out.println("Shuffled: " + numbers);
// Binary search (list must be sorted first)
Collections.sort(numbers);
int index = Collections.binarySearch(numbers, 5);
System.out.println("Index of 5: " + index);
// Min and Max
System.out.println("Min: " + Collections.min(numbers));
System.out.println("Max: " + Collections.max(numbers));
// Frequency
System.out.println("Frequency of 2: " + Collections.frequency(numbers, 2));
```

#### **Generic Collections**

```
// Generic class example
class Box<T> {
  private T content;
  public void set(T content) {
    this.content = content;
  public T get() {
    return content;
// Usage
Box<String> stringBox = new Box<>();
stringBox.set("Hello");
String content = stringBox.get();
Box<Integer> intBox = new Box<>();
intBox.set(42);
Integer number = intBox.get();
// Bounded generics
class NumberBox<T extends Number> {
  private T number;
  public NumberBox(T number) {
    this.number = number;
  public double getDoubleValue() {
    return number.doubleValue();
NumberBox<Integer> intNumberBox = new NumberBox<>(10);
NumberBox<Double> doubleNumberBox = new NumberBox<>(3.14);
```

# Streams API (Java 8+)

```
import java.util.stream.*;
List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David", "Eve");
// Filter and collect
List < String > filteredNames = names.stream()
  .filter(name -> name.length() > 3)
  .collect(Collectors.toList());
// Map and reduce
int totalLength = names.stream()
  .mapToInt(String::length)
  .sum();
// More complex operations
List<Person> people = Arrays.asList(
  new Person("Alice", 30),
  new Person("Bob", 25),
  new Person("Charlie", 35)
);
List < String > adultNames = people.stream()
  .filter(person -> person.getAge() >= 30)
  .map(Person::getName)
  .sorted()
  .collect(Collectors.toList());
// Group by
Map<Integer, List<Person>> peopleByAge = people.stream()
  .collect(Collectors.groupingBy(Person::getAge));
```

# **Best Practices and Tips**

## **Memory Management**

- Understand the difference between stack and heap memory
- Be aware of memory leaks with static collections
- Use appropriate collection types for your use case

#### **Performance Considerations**

- Use ArrayList for frequent random access
- Use LinkedList for frequent insertions/deletions
- Use HashMap for fast key-based lookups

• Use TreeMap when you need sorted keys

## **Code Quality**

- Follow naming conventions (camelCase for variables/methods, PascalCase for classes)
- Use meaningful variable and method names
- Write comments for complex logic
- Use access modifiers appropriately (private, protected, public)
- Implement equals() and hashCode() for custom objects used in collections

#### **Common Pitfalls**

- Avoid NullPointerException by checking for null
- Be careful with array index bounds
- Remember that Java is pass-by-value
- Understand the difference between == and .equals()
- Be aware of autoboxing/unboxing with wrapper classes

This comprehensive guide covers the fundamental to advanced concepts in Java programming.