Java Programming for Beginners: In-Depth Guide

# Introduction

Welcome to this comprehensive guide for learning Java, a powerful and widely-used programming language. Designed for beginners, this document provides detailed explanations, practical examples, and edge cases for core Java concepts. Topics include variables, object manipulation, strings, control structures, arrays, loops, modifiers, encapsulation, method calling, abstraction, exception handling, collections, lambda expressions, and polymorphism. Special emphasis is placed on strings and collections, with extensive coverage of types, methods, and implementations. Each topic includes multiple examples addressing various scenarios to ensure a deep understanding. By the end, you’ll be equipped to write robust Java programs and tackle real-world coding challenges.

# Differentiate Between Object Reference Variables and Primitive Variables

## Explanation

Variables in Java are either \*\*primitive\*\* or \*\*object reference\*\*, with distinct characteristics affecting their use in programs.  
  
 - \*\*Primitive Variables\*\*: These store basic data types directly in memory, including `byte` (8-bit), `short` (16-bit), `int` (32-bit), `long` (64-bit), `float` (32-bit), `double` (64-bit), `char` (16-bit Unicode), and `boolean` (true/false). They are stored on the stack, making them fast and memory-efficient. For example, `int x = 10;` stores the value `10` directly.

- \*\*Object Reference Variables\*\*: These store memory addresses pointing to objects in the heap. For example, `String s = "Hello";` stores a reference to a `String` object. Objects are complex, support methods, and can be `null`.  
  
 \*\*Key Differences\*\*:  
 - \*\*Storage\*\*: Primitives hold values; references hold addresses.  
 - \*\*Size\*\*: Primitives have fixed sizes; object sizes vary.  
 - \*\*Methods\*\*: Primitives lack methods; objects have methods.  
 - \*\*Nullability\*\*: Primitives cannot be `null`; references can.  
 - \*\*Default Values\*\*: Primitives have defaults (e.g., `0` for `int`, `false` for `boolean`); references default to `null`.

## Importance

Understanding this distinction is crucial for:  
 - \*\*Performance\*\*: Primitives are faster and use less memory, ideal for simple calculations.  
 - \*\*Method Behavior\*\*: Primitives are copied in method calls, while object references allow persistent changes to object state.  
 - \*\*Error Prevention\*\*: Object references require `null` checks to avoid `NullPointerException`.  
 - \*\*API Compatibility\*\*: Collections require objects (e.g., `Integer` vs. `int`), impacting design choices.  
 - \*\*Memory Management\*\*: Primitives avoid heap allocation, reducing garbage collection overhead.

## Examples

Below are examples demonstrating primitives and object references, including edge cases like null handling, autoboxing, and default values.

Java Code Example:

public class Main {  
 public static void main(String[] args) {  
 *// Example 1: Primitive variables* int age = 25;  
 double salary = 50000.75;  
 char grade = 'A';  
 boolean isEmployed = true;  
  
 System.out.println("Primitive int: " + age); *// Outputs 25* System.out.println("Primitive double: " + salary); *// Outputs 50000.75* System.out.println("Primitive char: " + grade); *// Outputs A* System.out.println("Primitive boolean: " + isEmployed); *// Outputs true  
  
 // Example 2: Object reference variables* String name = "Alice";  
 Integer wrappedAge = 30;  
 System.out.println("Object String: " + name); *// Outputs Alice* System.out.println("Object Integer: " + wrappedAge); *// Outputs 30  
  
 // Example 3: Null reference* String nullStr = null;  
 try {  
 System.out.println(nullStr.length()); *// Throws NullPointerException* } catch (NullPointerException e) {  
 System.out.println("Caught NullPointerException for null string");  
 }  
  
 *// Example 4: Autoboxing and unboxing  
 //int primitiveInt = 50;* Integer autoBoxed = 100; *// Autoboxing: int to Integer* int unboxed = autoBoxed; *// Unboxing: Integer to int* System.out.println("Autoboxed: " + autoBoxed); *// Outputs 100* System.out.println("Unboxed: " + unboxed); *// Outputs 100  
  
 // Example 5: Default values* new Test().print();  
  
 *// Example 6: Local variable initialization* int localInt;  
 *// System.out.println(localInt); // Compilation error: not initialized* localInt = 0;  
 System.out.println("Initialized local int: " + localInt); *// Outputs 0* }  
}  
  
class Test {  
 int **instanceInt**; *// Defaults to 0* String **instanceStr**; *// Defaults to null* void print() {  
 System.out.println("Default int: " + **instanceInt**); *// Outputs 0* System.out.println("Default String: " + **instanceStr**); *// Outputs null* }  
}

\*\*Edge Cases and Scenarios\*\*:  
 - \*\*Uninitialized Local Primitives\*\*: Must be initialized before use, or a compilation error occurs.  
 - \*\*Null References\*\*: Calling methods on `null` throws `NullPointerException`.  
 - \*\*Autoboxing Issues\*\*: Converting `null` `Integer` to `int` causes `NullPointerException`.  
 - \*\*Wrapper Overhead\*\*: Using `Integer` instead of `int` adds memory and performance costs.  
 - \*\*Primitive Overflow\*\*: Exceeding a primitive’s range (e.g., `byte b = 128;`) causes compilation or runtime errors.

# Read or Write to Object Fields

## Explanation

Object fields are variables within a class that hold an object’s state. You can read or write fields using dot notation (`object.field`) if accessible, or via getter/setter methods for controlled access. Access modifiers (`public`, `private`, `protected`, `default`) determine visibility.  
  
 - \*\*Direct Access\*\*: Suitable for `public` fields but risks data corruption.  
 - \*\*Getters/Setters\*\*: Encapsulate fields, enabling validation and abstraction.  
 - \*\*Final Fields\*\*: Can be set once (at declaration or in constructor) and are read-only thereafter.

## Importance

Field manipulation is central to object-oriented programming:  
 - \*\*State Management\*\*: Fields store data like a `Student`’s `name` or `grade`.  
 - \*\*Encapsulation\*\*: Private fields with getters/setters ensure data integrity.  
 - \*\*Validation\*\*: Setters can enforce rules (e.g., non-negative values).  
 - \*\*Flexibility\*\*: Dynamic state updates enable responsive programs.  
 - \*\*Thread Safety\*\*: Controlled access prevents concurrent modification issues.

## Examples

The following examples demonstrate reading/writing fields in various scenarios, including validation, immutability, and inheritance.

Java Code Example:  
  
public class Student {  
 private String **name**; *// Private field* private int **age**;  
 private final String **id**; *// Immutable field* public Student(String id) {  
 this.**id** = id; *// Set final field in constructor* }  
  
 *// Getters and setters* public String getName() { return **name**; }  
 public void setName(String name) {  
 if (name != null && !name.trim().isEmpty()) {  
 this.**name** = name;  
 }  
 }  
 public int getAge() { return **age**; }  
 public void setAge(int age) {  
 if (age >= 0 && age <= 120) {  
 this.**age** = age;  
 }  
 }  
  
 public String getId() { return **id**; }  
 *// No setter for id (immutable)*}  
  
class Main {  
 public static void main(String[] args) {  
 *// Example 1: Basic read/write* Student student = new Student("S123");  
 student.setName("Bob");  
 student.setAge(20);  
 System.out.println("Name: " + student.getName()); *// Outputs Bob* System.out.println("Age: " + student.getAge()); *// Outputs 20* System.out.println("ID: " + student.getId()); *// Outputs S123  
  
 // Example 2: Invalid input* student.setAge(-5); *// Ignored* System.out.println("Age after invalid input: " + student.getAge()); *// Outputs 20* student.setName(""); *// Ignored* System.out.println("Name after empty input: " + student.getName()); *// Outputs Bob  
  
 // Example 3: Inheritance and protected fields* GraduateStudent grad = new GraduateStudent("G456");  
 grad.setName("Alice");  
 grad.setThesis("Java Research at IUBAT");  
 System.out.println("Grad Name: " + grad.getName()); *// Outputs Alice* System.out.println("Thesis: " + grad.getThesis()); *// Outputs AI Research  
  
 // Example 4: Attempt to access private field  
 // student.name = "Charlie"; // Compilation error: name is private* }  
}

class GraduateStudent extends Student {  
 private String **thesis**;  
 public GraduateStudent(String id) { super(id); }  
 public void setThesis(String thesis) { this.**thesis** = thesis; }  
 public String getThesis() { return **thesis**; }  
}

\*\*Edge Cases and Scenarios\*\*:  
 - \*\*Invalid Inputs\*\*: Setters must validate inputs (e.g., `null`, empty strings, out-of-range values).  
 - \*\*Immutable Fields\*\*: `final` fields prevent reassignment, ensuring constant values.  
 - \*\*Inheritance\*\*: `protected` fields are accessible in subclasses, but `private` fields require getters/setters.  
 - \*\*Thread Safety\*\*: Uncontrolled access in multi-threaded programs risks data inconsistency; use `synchronized` methods.  
 - \*\*Performance\*\*: Excessive getter/setter calls in tight loops may impact performance; consider direct access for `public` fields in performance-critical code (with caution).

# Creating and Manipulating Strings

## Explanation

Strings in Java represent character sequences and are managed by three key classes:  
  
 - \*\*String\*\*: Immutable, thread-safe, created via literals (`"Hello"`) or `new String("Hello")`. Immutability ensures safety but creates new objects for modifications.  
 - \*\*StringBuilder\*\*: Mutable, not thread-safe, optimized for single-threaded string manipulation.  
 - \*\*StringBuffer\*\*: Mutable, thread-safe, used in multi-threaded environments but slower than `StringBuilder`.  
  
 \*\*Key String Methods\*\*:  
 - `length()`: Returns character count.  
 - `charAt(int index)`: Gets character at index.  
 - `substring(int begin, int end)`: Extracts substring.  
 - `concat(String str)`: Concatenates strings.  
 - `toUpperCase()`, `toLowerCase()`: Changes case.  
 - `trim()`: Removes leading/trailing whitespace.  
 - `replace(char old, char new)`, `replaceAll(String regex, String replacement)`: Replaces characters/patterns.  
 - `split(String regex)`: Splits into array.  
 - `indexOf(String str)`, `lastIndexOf(String str)`: Finds substring indices.  
 - `startsWith(String prefix)`, `endsWith(String suffix)`: Checks prefixes/suffixes.  
 - `equals(Object obj)`, `equalsIgnoreCase(String str)`: Compares content.  
 - `compareTo(String str)`: Lexicographical comparison.  
 - `matches(String regex)`: Checks regex match.  
 - `isEmpty()`: Checks if empty.  
 - `valueOf(type val)`: Converts to string.  
  
 \*\*StringBuilder/StringBuffer Methods\*\*:  
 - `append(type val)`: Adds data.  
 - `insert(int offset, type val)`: Inserts data.  
 - `delete(int start, int end)`: Removes substring.  
 - `replace(int start, int end, String str)`: Replaces substring.  
 - `reverse()`: Reverses content.  
 - `toString()`: Converts to `String`.  
 - `capacity()`: Returns current capacity.  
 - `ensureCapacity(int min)`: Ensures minimum capacity.

## Importance

Strings are critical for:  
 - \*\*Text Processing\*\*: Handle input, output, and data formatting.  
 - \*\*Immutability\*\*: `String` ensures safety in concurrent programs.  
 - \*\*Performance\*\*: `StringBuilder`/`StringBuffer` optimize heavy manipulation.  
 - \*\*Versatility\*\*: Extensive methods support parsing, searching, and formatting.  
 - \*\*APIs\*\*: Strings are used in databases, files, and UI components.

## Examples

The following examples cover various string operations, including parsing, formatting, performance comparisons, and edge cases.

Java Code Example:  
  
import java.util.Arrays;  
  
public class StringExample {  
 public static void main(String[] args) {  
 *// Example 1: Basic String operations* String greeting = "Hello, Java!";  
 System.out.println("Length: " + greeting.length()); *// Outputs 12* System.out.println("Char at 7: " + greeting.charAt(7)); *// Outputs J* System.out.println("Substring: " + greeting.substring(0, 5)); *// Outputs Hello* System.out.println("Concat: " + greeting.concat(" Welcome")); *// Outputs Hello, Java! Welcome* System.out.println("Uppercase: " + greeting.toUpperCase()); *// Outputs HELLO, JAVA!* System.out.println("Trim: " + " Spaces ".trim()); *// Outputs Spaces  
  
 // Example 2: String searching and comparison* System.out.println("Index of 'Java': " + greeting.indexOf("Java")); *// Outputs 7* System.out.println("Starts with 'Hello': " + greeting.startsWith("Hello")); *// Outputs true* System.out.println("Equals ignore case: " + greeting.equalsIgnoreCase("hello, java!")); *// Outputs true* System.out.println("Compare to 'Hello': " + greeting.compareTo("Hello")); *// Outputs positive  
  
 // Example 3: String splitting and replacing* String csv = "Alice,Bob,Charlie";  
 String[] names = csv.split(",");  
 System.out.println("Split: " + Arrays.*toString*(names)); *// Outputs [Alice, Bob, Charlie]* System.out.println("Replace: " + greeting.replace("Java", "World")); *// Outputs Hello, World!* System.out.println("Regex replace: " + csv.replaceAll(",", ";")); *// Outputs Alice;Bob;Charlie  
  
 // Example 4: StringBuilder for efficient concatenation* StringBuilder builder = new StringBuilder("Start");  
 builder.append(" Middle").append(" End");  
 builder.insert(5, ",");  
 builder.delete(6, 12);  
 builder.reverse();  
 System.out.println("StringBuilder: " + builder.toString()); *// Outputs dra* System.out.println("Capacity: " + builder.capacity()); *// Outputs 16 (initial) or higher  
  
 // Example 5: StringBuffer in multi-threaded context* StringBuffer buffer = new StringBuffer("Thread");  
 buffer.append(" Safe");  
 buffer.replace(6, 10, "Secure");  
 System.out.println("StringBuffer: " + buffer.toString()); *// Outputs ThreadSecure  
  
 // Example 6: Real-world parsing* String log = "ERROR: Null input at 2023-10-01";  
 if (log.matches("ERROR:.\*")) {  
 System.out.println("Error log detected");  
 String[] parts = log.split(": ");  
 System.out.println("Message: " + parts[1]); *// Outputs Null input at 2023-10-01* }  
  
 *// Example 7: Edge case - Null String* String nullStr = null;  
 try {  
 System.out.println(nullStr.length()); *// Throws NullPointerException* } catch (NullPointerException e) {  
 System.out.println("Caught NullPointerException");  
 }  
  
 *// Example 8: Edge case - Empty String* String empty = "";  
 System.out.println("Empty length: " + empty.length()); *// Outputs 0* System.out.println("Is empty: " + empty.isEmpty()); *// Outputs true  
  
 // Example 9: Edge case - Invalid index* try {  
 System.out.println(greeting.charAt(20)); *// Throws StringIndexOutOfBoundsException* } catch (StringIndexOutOfBoundsException e) {  
 System.out.println("Caught StringIndexOutOfBoundsException");  
 }  
  
 *// Example 10: Performance comparison* String concat = "";  
 StringBuilder builderConcat = new StringBuilder();  
 long start = System.*currentTimeMillis*();  
 for (int i = 0; i < 10000; i++) {  
 concat += "x";  
 }  
 long stringTime = System.*currentTimeMillis*() - start;  
 start = System.*currentTimeMillis*();  
 for (int i = 0; i < 10000; i++) {  
 builderConcat.append("x");  
 }  
 long builderTime = System.*currentTimeMillis*() - start;  
 System.out.println("String concat time: " + stringTime + "ms");  
 System.out.println("StringBuilder concat time: " + builderTime + "ms");  
 }  
}

\*\*Edge Cases and Scenarios\*\*:  
 - \*\*Null Strings\*\*: Method calls on `null` throw `NullPointerException`.  
 - \*\*Empty Strings\*\*: Valid, with length `0` and `isEmpty()` returning `true`.  
 - \*\*Index Errors\*\*: Invalid indices in `charAt()`, `substring()` throw `StringIndexOutOfBoundsException`.  
 - \*\*Regex Errors\*\*: Malformed regex in `split()` or `replaceAll()` throws `PatternSyntaxException`.  
 - \*\*Performance\*\*: `String` concatenation in loops creates many objects, slowing performance; use `StringBuilder`.  
 - \*\*Thread Safety\*\*: `StringBuffer` ensures safe concurrent modifications but is slower than `StringBuilder`.  
 - \*\*Capacity Management\*\*: `StringBuilder`/`StringBuffer` auto-expand capacity, but setting initial capacity can optimize memory.

# Create If and If/Else Constructs

## Explanation

The `if` statement evaluates a boolean condition and executes a block if `true`. The `if/else` construct provides alternatives for `false`. `else if` chains and nested `if` statements handle complex logic.  
  
 - \*\*Syntax\*\*:  
 ```java  
 if (condition) {  
 // Code  
 } else if (condition2) {  
 // Code  
 } else {  
 // Code  
 }  
 ```  
 - \*\*Operators\*\*: `==`, `!=`, `<`, `>`, `<=`, `>=`, `&&`, `||`, `!`.  
 - \*\*Ternary Operator\*\*: `condition ? valueIfTrue : valueIfFalse` for concise conditionals.

## Importance

Conditionals are foundational:  
 - \*\*Decision Making\*\*: Control program flow based on data.  
 - \*\*Validation\*\*: Check inputs or states.  
 - \*\*Error Handling\*\*: Prevent invalid operations.  
 - \*\*Dynamic Behavior\*\*: Adapt to runtime conditions.

## Examples

These examples cover various conditional scenarios, including validation, nested logic, and boundary conditions.

Java Code Example:  
  
public class IfExample {  
 public static void main(String[] args) {  
 *// Example 1: Basic if* int score = 85;  
 if (score >= 60) {  
 System.out.println("Pass");  
 }  
  
 *// Example 2: If/else* 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");  
 }  
  
 *// Example 3: Nested if* boolean isStudent = true;  
 int age = 20;  
 if (isStudent) {  
 if (age < 25) {  
 System.out.println("Eligible for student discount");  
 } else {  
 System.out.println("Not eligible: too old");  
 }  
 } else {  
 System.out.println("Not a student");  
 }  
  
 *// Example 4: Logical operators* boolean hasLicense = true;  
 if (age >= 16 && hasLicense) {  
 System.out.println("Can drive");  
 }  
  
 *// Example 5: Ternary operator* String status = score >= 60 ? "Pass" : "Fail";  
 System.out.println("Status: " + status); *// Outputs Pass  
  
 // Example 6: Edge case - Boundary condition* score = 60;  
 if (score == 60) {  
 System.out.println("Just passed");  
 }  
  
 *// Example 7: Edge case - Null check* String input = null;  
 if (input != null && input.length() > 0) {  
 System.out.println("Valid input");  
 } else {  
 System.out.println("Invalid or null input");  
 }  
  
 *// Example 8: Edge case - Short-circuiting* int x = 0;  
 if (x != 0 && (10 / x) > 2) {  
 System.out.println("Won't execute due to short-circuit");  
 } else {  
 System.out.println("Avoided division by zero");  
 }  
 }  
}

\*\*Edge Cases and Scenarios\*\*:  
 - \*\*Boundary Conditions\*\*: Exact thresholds (e.g., `score == 60`) need precise checks.  
 - \*\*Short-Circuiting\*\*: `&&` and `||` evaluate only necessary conditions, avoiding errors like division by zero.  
 - \*\*Null Checks\*\*: Always check for `null` before accessing object methods.  
 - \*\*Complex Logic\*\*: Nested `if` or long `else if` chains can reduce readability; consider `switch` or methods.  
 - \*\*Ternary Misuse\*\*: Overusing ternary operators for complex logic can obscure code.

# Use a Switch Statement

## Explanation

The `switch` statement evaluates an expression (`int`, `char`, `String`, `enum`, etc.) and executes a matching `case` block. A `default` case handles unmatched values, and `break` prevents fall-through. Java 14+ supports `switch` expressions with `->`.  
  
 - \*\*Syntax\*\*:  
 ```java  
 switch (expression) {  
 case value1:  
 // Code  
 break;  
 default:  
 // Code  
 }  
 ```  
 - \*\*Switch Expression\*\*:  
 ```java  
 result = switch (expression) {  
 case value1 -> value;  
 default -> value;  
 };  
 ```

## Importance

Switch statements are useful for:  
 - \*\*Clarity\*\*: Simplify multiple `if/else` conditions.  
 - \*\*Efficiency\*\*: Optimized for discrete value matching.  
 - \*\*Maintainability\*\*: Easy to add or modify cases.  
 - \*\*Readability\*\*: Structured format for case-based logic.

## Examples

These examples demonstrate switch usage for different types, fall-through, and switch expressions.

Java Code Example:  
  
public class SwitchExample {  
 public static void main(String[] args) {  
 *// Example 1: String switch* String day = "Wednesday";  
 switch (day) {  
 case "Monday":  
 System.out.println("Start of week");  
 break;  
 case "Friday":  
 System.out.println("End of work week");  
 break;  
 default:  
 System.out.println("Midweek or weekend");  
 }  
  
 *// Example 2: Enum switch* enum Season { WINTER, SPRING, SUMMER, FALL }  
 Season season = Season.SUMMER;  
 switch (season) {  
 case WINTER:  
 System.out.println("Cold");  
 break;  
 case SUMMER:  
 System.out.println("Hot");  
 break;  
 default:  
 System.out.println("Moderate");  
 }  
  
 *// Example 3: Switch expression (Java 14+)* String message = switch (day) {  
 case "Monday" -> "Start of week";  
 case "Friday" -> "End of work week";  
 default -> "Midweek or weekend";  
 };  
 System.out.println("Switch expression: " + message);  
  
 *// Example 4: Fall-through* int level = 2;  
 switch (level) {  
 case 1:  
 case 2:  
 System.out.println("Beginner"); *// Falls through* case 3:  
 System.out.println("Intermediate");  
 break;  
 default:  
 System.out.println("Advanced");  
 }  
  
 *// Example 5: Edge case - Null input* day = null;  
 try {  
 switch (day) {  
 case "Monday":  
 System.out.println("Won't execute");  
 break;  
 default:  
 System.out.println("Default");  
 }  
 } catch (NullPointerException e) {  
 System.out.println("Caught NullPointerException for null switch");  
 }  
  
 *// Example 6: Edge case - Missing break* int score = 85;  
 switch (score / 10) {  
 case 10:  
 case 9:  
 System.out.println("A"); *// Falls through* case 8:  
 System.out.println("B");  
 break;  
 default:  
 System.out.println("C or below");  
 }  
 }  
}

\*\*Edge Cases and Scenarios\*\*:  
 - \*\*Fall-Through\*\*: Missing `break` causes unintended execution of subsequent cases.  
 - \*\*Null Input\*\*: `null` in a `String` switch throws `NullPointerException`.  
 - \*\*Default Case\*\*: Essential for unhandled values to avoid silent failures.  
 - \*\*Type Mismatch\*\*: Case values must match the switch expression’s type.  
 - \*\*Switch Expression\*\*: Must cover all cases or include `default` to avoid compilation errors.

# Arrays

## Explanation

Arrays are fixed-size, ordered collections of elements of the same type, accessed by zero-based indices. They can store primitives (`int[]`) or objects (`String[]`). Multi-dimensional arrays (e.g., `int[][]`) are arrays of arrays.

## Importance

Arrays are essential for:  
 - \*\*Data Storage\*\*: Store multiple values efficiently.  
 - \*\*Iteration\*\*: Process elements using loops.  
 - \*\*Performance\*\*: Fixed size ensures predictable memory usage.  
 - \*\*Simplicity\*\*: Ideal for static, homogeneous data.

## Examples

These examples cover array creation, manipulation, iteration, and multi-dimensional arrays.

Java Code Example:  
  
public class ArrayExample {  
 public static void main(String[] args) {  
 *// Example 1: Basic array* int[] scores = {90, 85, 78, 92};  
 System.out.println("First score: " + scores[0]); *// Outputs 90* scores[2] = 80;  
 System.out.println("Modified score: " + scores[2]); *// Outputs 80  
  
 // Example 2: Array iteration* for (int i = 0; i < scores.**length**; i++) {  
 System.out.println("Score " + i + ": " + scores[i]);  
 }  
 for (int score : scores) {  
 System.out.println("Enhanced for: " + score);  
 }  
  
 *// Example 3: Array of objects* String[] names = new String[3];  
 names[0] = "Alice";  
 names[1] = "Bob";  
 System.out.println("First name: " + names[0]); *// Outputs Alice* System.out.println("Third name: " + names[2]); *// Outputs null  
  
 // Example 4: Multi-dimensional array* int[][] matrix = {{1, 2, 3}, {4, 5, 6}};  
 System.out.println("Matrix[1][2]: " + matrix[1][2]); *// Outputs 6* for (int[] row : matrix) {  
 for (int val : row) {  
 System.out.print(val + " ");  
 }  
 System.out.println();  
 }  
  
 *// Example 5: Edge case - Out of bounds* try {  
 System.out.println(scores[10]); *// Throws ArrayIndexOutOfBoundsException* } catch (ArrayIndexOutOfBoundsException e) {  
 System.out.println("Caught ArrayIndexOutOfBoundsException");  
 }  
  
 *// Example 6: Edge case - Jagged array* int[][] jagged = {{1, 2}, {3, 4, 5, 6}};  
 System.out.println("Jagged[1][3]: " + jagged[1][3]); *// Outputs 6  
  
 // Example by switch case* String[] days = {"Monday", "Friday", "Sunday"};  
 for (String day : days) {  
 String message = switch (day) {  
 case "Monday" -> "Start of week";  
 case "Friday" -> "End of work week";  
 default -> "Midweek or weekend";  
 };  
 System.out.println(day + ": " + message);  
 }  
 }  
}

\*\*Edge Cases and Scenarios\*\*:  
 - \*\*Out-of-Bounds\*\*: Accessing invalid indices throws `ArrayIndexOutOfBoundsException`.  
 - \*\*Null Elements\*\*: Object arrays can contain `null`, requiring checks.  
 - \*\*Jagged Arrays\*\*: Multi-dimensional arrays can have uneven lengths.  
 - \*\*Fixed Size\*\*: Arrays cannot resize; use `ArrayList` for dynamic sizing.  
 - \*\*Initialization\*\*: Uninitialized object arrays contain `null`; primitives have default values.

# Loops (Break, Continue)

## Explanation

Java supports:  
 - \*\*For Loop\*\*: Fixed iterations (`for (int i = 0; i < n; i++)`).  
 - \*\*Enhanced For Loop\*\*: Iterates arrays/collections (`for (Type var : collection)`).  
 - \*\*While Loop\*\*: Condition-based (`while (condition)`).  
 - \*\*Do-While Loop\*\*: Executes at least once (`do { } while (condition)`).  
  
 - \*\*Break\*\*: Exits the loop.  
 - \*\*Continue\*\*: Skips to the next iteration.  
 - \*\*Labeled Break/Continue\*\*: Control outer loops in nested structures.

## Importance

Loops are critical for:  
 - \*\*Iteration\*\*: Process collections or repeat tasks.  
 - \*\*Control Flow\*\*: `break` and `continue` enable precise control.  
 - \*\*Automation\*\*: Simplify repetitive operations.  
 - \*\*Efficiency\*\*: Reduce code duplication.

## Examples

These examples demonstrate different loop types, break/continue, and nested loops.

Java Code Example:  
  
public class LoopExample {  
 public static void main(String[] args) {  
 *// Example 1: For loop* for (int i = 1; i <= 5; i++) {  
 System.out.println("Number: " + i);  
 }  
  
 *// Example 2: Enhanced for loop* String[] fruits = {"Apple", "Banana", "Orange"};  
 for (String fruit : fruits) {  
 System.out.println("Fruit: " + fruit);  
 }  
  
 *// Example 3: While loop* int count = 0;  
 while (count < 3) {  
 System.out.println("Count: " + count++);  
 }  
  
 *// Example 4: Do-while loop* count = 0;  
 do {  
 System.out.println("Do-while count: " + count++);  
 } while (count < 3);  
  
 *// Example 5: Break* for (int i = 1; i <= 10; i++) {  
 if (i == 6) {  
 break;  
 }  
 System.out.println("With break: " + i);  
 }  
  
 *// Example 6: Continue* for (int i = 1; i <= 5; i++) {  
 if (i % 2 == 0) {  
 continue;  
 }  
 System.out.println("Odd number: " + i);  
 }  
  
 *// Example 7: Labeled break in nested loops* outer: for (int i = 1; i <= 3; i++) {  
 for (int j = 1; j <= 3; j++) {  
 if (i == 2 && j == 2) {  
 break outer;  
 }  
 System.out.println("i=" + i + ", j=" + j);  
 }  
 }  
  
 *// Example 8: Edge case - Infinite loop* count = 0;  
 while (true) {  
 if (count++ == 3) {  
 break;  
 }  
 System.out.println("Controlled infinite: " + count);  
 }  
 }  
}

\*\*Edge Cases and Scenarios\*\*:  
 - \*\*Infinite Loops\*\*: Missing exit conditions cause infinite loops; ensure proper termination.  
 - \*\*Nested Loops\*\*: `break`/`continue` affect only the innermost loop unless labeled.  
 - \*\*Empty Collections\*\*: Enhanced for loops execute zero times for empty arrays.  
 - \*\*Off-by-One Errors\*\*: Incorrect bounds (e.g., `i <= length`) can cause out-of-bounds errors.  
 - \*\*Do-While\*\*: Executes at least once, even if the condition is initially `false`.

# Final, Static, Access Modifiers

## Explanation

- \*\*Final\*\*: Prevents modification. `final` variables are constants, `final` methods cannot be overridden, `final` classes cannot be extended.  
 - \*\*Static\*\*: Class-level, not instance-level. Static fields/methods are shared across all instances.

## ****Overview of Access Modifiers****

1. **public**: The member is accessible from **everywhere**.
2. **protected**: The member is accessible within the **same package** and also in **subclasses** (even if they are in different packages).
3. **default** (package-private): If no access modifier is specified, the member is accessible only within the **same package**.
4. **private**: The member is accessible only within the **same class**.

### ****Access Modifier Scope Table****

| **Modifier** | **Same Class** | **Same Package** | **Subclass (Same Package)** | **Subclass (Different Package)** | **Non-Subclass (Different Package)** |
| --- | --- | --- | --- | --- | --- |
| public | Yes | Yes | Yes | Yes | Yes |
| protected | Yes | Yes | Yes | Yes (via inheritance) | No |
| default | Yes | Yes | Yes | No | No |
| private | Yes | No | No | No | No |

## Importance

These modifiers are key for:  
 - \*\*Immutability\*\*: `final` ensures constants or fixed behaviors.  
 - \*\*Shared Resources\*\*: `static` enables utility methods or shared data.  
 - \*\*Encapsulation\*\*: Access modifiers protect data.  
 - \*\*Design\*\*: Proper use ensures robust, maintainable code.

## Examples

These examples cover constants, utility methods, and access control.

Java Code Example:  
  
public class ModifierExample {  
 public static final double PI = 3.14159; *// Constant* private String **name**; *// Instance field* protected int **count**; *// Accessible in subclasses  
  
 // Static method* public static int add(int a, int b) {  
 return a + b;  
 }  
  
 *// Instance methods* public void setName(String name) { this.**name** = name; }  
 public String getName() { return **name**; }  
  
 public static void main(String[] args) {  
 *// Example 1: Static and final* System.out.println("PI: " + PI); *// Outputs 3.14159* System.out.println("Sum: " + *add*(5, 3)); *// Outputs 8  
  
 // Example 2: Instance access* ModifierExample obj = new ModifierExample();  
 obj.setName("Test");  
 System.out.println("Name: " + obj.getName()); *// Outputs Test  
  
 // Example 3: Protected access in subclass* class SubClass extends ModifierExample {  
 void increment() {  
 **count**++; *// Accessible* }  
 }  
 SubClass sub = new SubClass();  
 sub.increment();  
 System.out.println("Count: " + sub.**count**); *// Outputs 1  
  
 // Example 4: Edge case - Final modification  
 // PI = 3.14; // Compilation error* }  
}

\*\*Edge Cases and Scenarios\*\*:  
 - \*\*Final Variables\*\*: Must be initialized once; reassignment causes compilation error.  
 - \*\*Static Access\*\*: Static methods cannot access non-static fields directly.  
 - \*\*Access Violation\*\*: Accessing `private` fields outside the class causes compilation error.  
 - \*\*Inheritance\*\*: `protected` fields are accessible in subclasses, even across packages.  
 - \*\*Static Initialization\*\*: Static fields are initialized when the class is loaded.

# Encapsulation

## Explanation

Encapsulation bundles a class’s data and methods, restricting direct access to fields using `private` modifiers and providing public getters/setters. This hides implementation details and ensures controlled access.

## Importance

Encapsulation is vital for:  
 - \*\*Data Protection\*\*: Prevents invalid modifications.  
 - \*\*Maintainability\*\*: Internal changes don’t affect external code.  
 - \*\*Validation\*\*: Setters enforce data rules.  
 - \*\*Modularity\*\*: Classes become self-contained units.

## Examples

These examples demonstrate encapsulation with validation and immutability.

Java Code Example:  
  
public class Employee {  
 private String **name**;  
 private double **salary**;  
 private final int **id**;  
  
 public Employee(int id) {  
 this.**id** = id;  
 }  
  
 public String getName() { return **name**; }  
 public void setName(String name) {  
 if (name != null && !name.trim().isEmpty()) {  
 this.**name** = name;  
 }  
 }  
  
 public double getSalary() { return **salary**; }  
 public void setSalary(double salary) {  
 if (salary >= 0) {  
 this.**salary** = salary;  
 }  
 }  
  
 public int getId() { return **id**; }  
}  
  
class Main {  
 public static void main(String[] args) {  
 *// Example 1: Basic encapsulation* Employee emp = new Employee(1001);  
 emp.setName("Alice");  
 emp.setSalary(75000);  
 System.out.println("Name: " + emp.getName()); *// Outputs Alice* System.out.println("Salary: " + emp.getSalary()); *// Outputs 75000* System.out.println("ID: " + emp.getId()); *// Outputs 1001  
  
 // Example 2: Invalid input* emp.setSalary(-5000); *// Ignored* System.out.println("Salary: " + emp.getSalary()); *// Outputs 75000* emp.setName(""); *// Ignored* System.out.println("Name: " + emp.getName()); *// Outputs Alice  
  
 // Example 3: Immutable field  
 // emp.id = 1002; // Compilation error: id is final* }  
}

\*\*Edge Cases and Scenarios\*\*:  
 - \*\*Invalid Inputs\*\*: Setters must validate to maintain valid state.  
 - \*\*Immutable Fields\*\*: `final` fields ensure constant values.  
 - \*\*Direct Access\*\*: Bypassing getters/setters risks data corruption.  
 - \*\*Thread Safety\*\*: Encapsulated fields may need synchronization in concurrent programs.

# Call by Value / Call by Reference

## Explanation

Java uses \*\*call by value\*\* for all method parameters:  
 - \*\*Primitives\*\*: The value is copied; changes don’t affect the original.  
 - \*\*Objects\*\*: The reference is copied; changes to the object’s fields persist, but reassigning the reference doesn’t affect the original.

## Importance

Understanding call by value is crucial for:  
 - \*\*Behavior Prediction\*\*: Know how methods affect data.  
 - \*\*Side Effects\*\*: Control object modifications.  
 - \*\*Debugging\*\*: Clarify why changes persist or are lost.  
 - \*\*Design\*\*: Choose parameter types appropriately.

## Examples

These examples cover primitives, objects, arrays, and immutable types.

Java Code Example:  
  
public class CallExample {  
 static void modifyPrimitive(int num) {  
 num = 100;  
 }  
  
 static void modifyObject(Person person) {  
 person.setName("Changed");  
 person = new Person(); *// Local reassignment* person.setName("New");  
 }  
  
 static void modifyArray(int[] arr) {  
 arr[0] = 999;  
 }  
  
 static void modifyString(String str) {  
 str = "New String"; *// Reassignment doesn’t affect original* }  
  
 public static void main(String[] args) {  
 *// Example 1: Primitive* int value = 10;  
 *modifyPrimitive*(value);  
 System.out.println("Primitive: " + value); *// Outputs 10  
  
 // Example 2: Object* Person person = new Person();  
 person.setName("Original");  
 *modifyObject*(person);  
 System.out.println("Object: " + person.getName()); *// Outputs Changed  
  
 // Example 3: Array* int[] numbers = {1, 2, 3};  
 *modifyArray*(numbers);  
 System.out.println("Array: " + numbers[0]); *// Outputs 999  
  
 // Example 4: Immutable String* String text = "Hello";  
 *modifyString*(text);  
 System.out.println("String: " + text); *// Outputs Hello* }  
}  
  
class Person {  
 private String **name**;  
 public void setName(String name) { this.**name** = name; }  
 public String getName() { return **name**; }  
}

\*\*Edge Cases and Scenarios\*\*:  
 - \*\*Reassignment\*\*: Reassigning object parameters doesn’t change the original reference.  
 - \*\*Immutable Objects\*\*: `String` and wrappers behave like primitives due to immutability.  
 - \*\*Arrays\*\*: Changes to array elements persist, as arrays are objects.  
 - \*\*Null Parameters\*\*: Passing `null` requires checks to avoid `NullPointerException`.

# Abstract Class

## Explanation

An `abstract` class cannot be instantiated and defines a blueprint for subclasses. It may include abstract methods (no implementation), concrete methods, fields, and constructors.

## Importance

Abstract classes are key for:  
 - \*\*Code Reuse\*\*: Share common functionality.  
 - \*\*Contract Enforcement\*\*: Require subclasses to implement methods.  
 - \*\*Extensibility\*\*: Allow customization via subclassing.  
 - \*\*Hierarchy Design\*\*: Model relationships.

## Examples

These examples show abstract classes with inheritance and polymorphism.

Java Code Example:  
  
package org.example;  
  
abstract class Shape {  
 protected String **color**;  
  
 Shape(String color) {  
 this.**color** = color;  
 }  
  
 abstract double area();  
  
 String getColor() {  
 return **color**;  
 }  
}  
  
class Circle extends Shape {  
 double **radius**;  
  
 Circle(String color, double radius) {  
 super(color);  
 this.**radius** = radius;  
 }  
  
 double area() {  
 return Math.PI \* **radius** \* **radius**;  
 }  
}  
  
class Rectangle extends Shape {  
 double **width**, **height**;  
  
 Rectangle(String color, double width, double height) {  
 super(color);  
 this.**width** = width;  
 this.**height** = height;  
 }  
  
 double area() {  
 return **width** \* **height**;  
 }  
}  
  
class Main {  
 public static void main(String[] args) {  
 *// Example 1: Basic usage* Shape circle = new Circle("Red", 5);  
 Shape rectangle = new Rectangle("Blue", 4, 6);  
 System.out.println("Circle area: " + circle.area()); *// Outputs ~78.54* System.out.println("Rectangle area: " + rectangle.area()); *// Outputs 24  
  
 // Example 2: Polymorphic array* Shape[] shapes = {new Circle("Green", 3), new Rectangle("Yellow", 2, 5)};  
 for (Shape shape : shapes) {  
 System.out.println("Area: " + shape.area() + ", Color: " + shape.getColor());  
 }  
  
 *// Example 3: Edge case - Abstract instantiation  
 // Shape s = new Shape("Black"); // Compilation error* }  
}

\*\*Edge Cases and Scenarios\*\*:  
 - \*\*Instantiation\*\*: Abstract classes cannot be instantiated.  
 - \*\*Abstract Subclasses\*\*: Must implement all abstract methods or be abstract.  
 - \*\*Constructors\*\*: Used to initialize fields in subclasses.  
 - \*\*Access\*\*: Abstract methods are typically `public` or `protected`.

# Exception Handling

## Explanation

Exception handling manages runtime errors using `try`, `catch`, `finally`, `throw`, and `throws`. Exceptions are:  
 - \*\*Checked\*\*: Must be handled or declared (e.g., `IOException`).  
 - \*\*Unchecked\*\*: Optional to handle (e.g., `NullPointerException`).

## Importance

Exception handling ensures:  
 - \*\*Robustness\*\*: Prevent crashes.  
 - \*\*Resource Cleanup\*\*: Close files/connections.  
 - \*\*User Feedback\*\*: Provide meaningful errors.  
 - \*\*Debugging\*\*: Log errors systematically.

## Examples

These examples cover basic handling, custom exceptions, and resource management.

Java Code Example:  
  
public class ExceptionExample {  
 public static void main(String[] args) {  
 *// Example 1: Basic try-catch* try {  
 int[] arr = {1, 2};  
 System.out.println(arr[5]);  
 } catch (ArrayIndexOutOfBoundsException e) {  
 System.out.println("Invalid index: " + e.getMessage());  
 }  
  
 *// Example 2: Multiple catch blocks* try {  
 String str = null;  
 System.out.println(str.length());  
 } catch (NullPointerException e) {  
 System.out.println("Null string");  
 } catch (Exception e) {  
 System.out.println("General error");  
 }  
  
 *// Example 3: Finally* try {  
 int x = 10 / 0;  
 } catch (ArithmeticException e) {  
 System.out.println("Division by zero");  
 } finally {  
 System.out.println("Cleanup in finally");  
 }  
  
 *// Example 4: Custom exception* try {  
 *validateScore*(-10);  
 } catch (IllegalArgumentException e) {  
 System.out.println("Error: " + e.getMessage());  
 }  
  
 *// Example 5: Try-with-resources* try (java.util.Scanner scanner = new java.util.Scanner(System.in)) {  
 System.out.println("Enter a number: ");  
 int num = scanner.nextInt();  
 System.out.println("Number: " + num);  
 } catch (Exception e) {  
 System.out.println("Input error");  
 }  
 }  
  
 static void validateScore(int score) {  
 if (score < 0 || score > 100) {  
 throw new IllegalArgumentException("Score must be between 0 and 100");  
 }  
 }  
}

\*\*Edge Cases and Scenarios\*\*:  
 - \*\*Uncaught Exceptions\*\*: Terminate the program.  
 - \*\*Multiple Catch\*\*: Specific exceptions must precede general ones.  
 - \*\*Finally\*\*: Executes even with `return` or uncaught exceptions.  
 - \*\*Try-with-Resources\*\*: Ensures resource closure.  
 - \*\*Custom Exceptions\*\*: Provide specific error context.

# Collections

## Explanation

The Java Collections Framework provides interfaces and classes for managing object groups:  
 - \*\*List\*\*: Ordered, allows duplicates (`ArrayList`, `LinkedList`).  
 - \*\*Set\*\*: No duplicates (`HashSet`, `LinkedHashSet`, `TreeSet`).  
 - \*\*Map\*\*: Key-value pairs, unique keys (`HashMap`, `LinkedHashMap`, `TreeMap`).  
 - \*\*Queue\*\*: Ordered for processing (`PriorityQueue`, `LinkedList`).  
  
 \*\*Implementations\*\*:  
 - \*\*ArrayList\*\*: Resizable array, fast random access.  
 - \*\*LinkedList\*\*: Doubly-linked list, fast insertions/deletions.  
 - \*\*HashSet\*\*: Hash table, no order, fast lookups.  
 - \*\*LinkedHashSet\*\*: Maintains insertion order.  
 - \*\*TreeSet\*\*: Sorted, no `null`.  
 - \*\*HashMap\*\*: Hash table, fast lookups, one `null` key.  
 - \*\*LinkedHashMap\*\*: Maintains insertion order.  
 - \*\*TreeMap\*\*: Sorted, no `null` keys.  
  
 \*\*Key Methods\*\* (List):  
 - `add(E e)`, `add(int index, E e)`, `get(int index)`, `set(int index, E e)`, `remove(int index)`, `remove(Object o)`, `size()`, `isEmpty()`, `contains(Object o)`, `clear()`, `toArray()`, `subList(int from, int to)`.  
  
 \*\*Key Methods\*\* (Set):  
 - `add(E e)`, `remove(Object o)`, `contains(Object o)`, `size()`, `isEmpty()`, `clear()`, `iterator()`.  
  
 \*\*Key Methods\*\* (Map):  
 - `put(K key, V value)`, `get(Object key)`, `remove(Object key)`, `containsKey(Object key)`, `containsValue(Object value)`, `keySet()`, `values()`, `entrySet()`, `size()`, `isEmpty()`, `clear()`, `putIfAbsent(K key, V value)`, `compute(K key, BiFunction remapping)`.

## Importance

Collections are essential for:  
 - \*\*Dynamic Storage\*\*: Grow/shrink as needed.  
 - \*\*Functionality\*\*: Support searching, sorting, iteration.  
 - \*\*Performance\*\*: Optimized implementations for specific use cases.  
 - \*\*Flexibility\*\*: Diverse structures for various needs.

## Examples

These examples cover all major collection types and methods.

Java Code Example:  
  
import java.util.\*;  
  
public class CollectionExample {  
 public static void main(String[] args) {  
 *// Example 1: ArrayList  
 List*<String> names = new ArrayList<>();  
 names.add("Alice");  
 names.add(0, "Bob");  
 names.set(1, "Charlie");  
 System.out.println("ArrayList: " + names); *// Outputs [Bob, Charlie]* System.out.println("Contains Bob: " + names.contains("Bob")); *// Outputs true* names.remove("Bob");  
 System.out.println("After remove: " + names); *// Outputs [Charlie]* System.out.println("Sublist: " + names.subList(0, 1)); *// Outputs [Charlie]  
  
 // Example 2: LinkedList* LinkedList<String> queue = new LinkedList<>();  
 queue.add("First");  
 queue.addFirst("Head");  
 queue.addLast("Tail");  
 System.out.println("LinkedList: " + queue); *// Outputs [Head, First, Tail]* System.out.println("Poll: " + queue.poll()); *// Outputs Head* System.out.println("After poll: " + queue); *// Outputs [First, Tail]  
  
 // Example 3: HashSet  
 Set*<Integer> numbers = new HashSet<>();  
 numbers.add(1);  
 numbers.add(1); *// Ignored* numbers.add(null);  
 System.out.println("HashSet: " + numbers); *// Outputs [null, 1]* numbers.remove(null);  
 System.out.println("After remove: " + numbers); *// Outputs [1]  
  
 // Example 4: LinkedHashSet  
 Set*<String> orderedSet = new LinkedHashSet<>();  
 orderedSet.add("Apple");  
 orderedSet.add("Banana");  
 System.out.println("LinkedHashSet: " + orderedSet); *// Outputs [Apple, Banana]  
  
 // Example 5: TreeSet  
 Set*<Integer> sortedSet = new TreeSet<>();  
 sortedSet.add(3);  
 sortedSet.add(1);  
 System.out.println("TreeSet: " + sortedSet); *// Outputs [1, 3]  
  
 // Example 6: HashMap  
 Map*<String, Integer> ages = new HashMap<>();  
 ages.put("Alice", 25);  
 ages.put("Bob", 30);  
 ages.putIfAbsent("Alice", 26);  
 System.out.println("HashMap: " + ages); *// Outputs {Bob=30, Alice=25}* System.out.println("Keys: " + ages.keySet()); *// Outputs [Bob, Alice]* System.out.println("Values: " + ages.values()); *// Outputs [30, 25]* ages.compute("Bob", (k, v) -> v + 1);  
 System.out.println("Updated HashMap: " + ages); *// Outputs {Bob=31, Alice=25}  
  
 // Example 7: LinkedHashMap  
 Map*<String, String> orderedMap = new LinkedHashMap<>();  
 orderedMap.put("One", "1");  
 orderedMap.put("Two", "2");  
 System.out.println("LinkedHashMap: " + orderedMap); *// Outputs {One=1, Two=2}  
  
 // Example 8: TreeMap  
 Map*<Integer, String> sortedMap = new TreeMap<>();  
 sortedMap.put(2, "Two");  
 sortedMap.put(1, "One");  
 System.out.println("TreeMap: " + sortedMap); *// Outputs {1=One, 2=Two}  
  
 // Example 9: Edge case - Concurrent modification* try {  
 for (String name : names) {  
 names.add("New");  
 }  
 } catch (ConcurrentModificationException e) {  
 System.out.println("Caught ConcurrentModificationException");  
 }  
  
 *// Example 10: Edge case - Null in TreeSet* try {  
 sortedSet.add(null);  
 } catch (NullPointerException e) {  
 System.out.println("TreeSet does not allow null");  
 }  
 }  
}

\*\*Edge Cases and Scenarios\*\*:  
 - \*\*Null Handling\*\*: `HashSet`/`HashMap` allow `null`; `TreeSet`/`TreeMap` don’t.  
 - \*\*Concurrent Modification\*\*: Modifying during iteration throws `ConcurrentModificationException`.  
 - \*\*Performance\*\*: Wrong implementation (e.g., `LinkedList` for random access) degrades performance.  
 - \*\*Sorting\*\*: `TreeSet`/`TreeMap` require comparable elements or a `Comparator`.  
 - \*\*Duplicates\*\*: `Set` uses `equals()` and `hashCode()` to detect duplicates.

# Lambda Expressions

## Explanation

Lambda expressions (Java 8+) implement functional interfaces concisely: `(params) -> expression` or `(params) -> { statements; }`. Used with streams, collections, and event handling.

## Importance

Lambda expressions enable:  
 - \*\*Conciseness\*\*: Reduce boilerplate code.  
 - \*\*Functional Programming\*\*: Support filtering, mapping, etc.  
 - \*\*Readability\*\*: Simplify common tasks.  
 - \*\*Flexibility\*\*: Enhance stream operations.

## Examples

These examples cover sorting, streams, and custom functional interfaces.

Java Code Example:  
  
import java.util.\*;  
import java.util.function.*Predicate*;  
  
public class LambdaExample {  
 public static void main(String[] args) {  
 *// Example 1: Sorting with lambda  
 List*<String> names = Arrays.*asList*("Charlie", "Alice", "Bob");  
 Collections.*sort*(names, (a, b) -> a.compareTo(b));  
 System.out.println("Sorted: " + names); *// Outputs [Alice, Bob, Charlie]  
  
 // Example 2: Stream with lambda* names.stream()  
 .filter(name -> name.length() > 3)  
 .forEach(System.out::println); *// Outputs Alice, Charlie  
  
 // Example 3: Custom functional interface* interface *Operation* {  
 int compute(int a, int b);  
 }  
 *Operation* add = (a, b) -> a + b;  
 System.out.println("Add: " + add.compute(5, 3)); *// Outputs 8  
  
 // Example 4: Edge case - Empty stream  
 List*<String> empty = new ArrayList<>();  
 empty.stream()  
 .filter(name -> name.length() > 3)  
 .forEach(System.out::println); *// No output* }  
}

\*\*Edge Cases and Scenarios\*\*:  
 - \*\*Signature Mismatch\*\*: Lambda must match functional interface method.  
 - \*\*Empty Streams\*\*: No output, no errors.  
 - \*\*Variable Capture\*\*: Local variables must be effectively final.  
 - \*\*Complex Lambdas\*\*: Can reduce readability; use methods for complex logic.

# Polymorphism

## Explanation

Polymorphism allows objects of different classes to be treated as a common type:  
 - \*\*Runtime Polymorphism\*\*: Method overriding, based on actual object type.  
 - \*\*Compile-Time Polymorphism\*\*: Method overloading, based on parameter types.

## Importance

Polymorphism supports:  
 - \*\*Flexibility\*\*: Work with multiple types.  
 - \*\*Extensibility\*\*: Add subclasses without changing code.  
 - \*\*Abstraction\*\*: Focus on behavior, not implementation.  
 - \*\*Code Reuse\*\*: Leverage inheritance/interfaces.

## Examples

These examples cover overriding, overloading, and interface-based polymorphism.

Java Code Example:  
  
interface *Printable* {  
 void print();  
}  
  
class Book implements *Printable* {  
 String **title**;  
 Book(String title) { this.**title** = title; }  
 public void print() { System.out.println("Book: " + **title**); }  
}  
  
class Magazine implements *Printable* {  
 String **name**;  
 Magazine(String name) { this.**name** = name; }  
 public void print() { System.out.println("Magazine: " + **name**); }  
}  
  
public class PolymorphismExample {  
 *// Overloading* static void describe(Book b) {  
 System.out.println("Describing book: " + b.**title**);  
 }  
  
 static void describe(Magazine m) {  
 System.out.println("Describing magazine: " + m.**name**);  
 }  
  
 public static void main(String[] args) {  
 *// Example 1: Interface polymorphism  
 Printable* book = new Book("Java Guide");  
 *Printable* magazine = new Magazine("Tech Weekly");  
 book.print(); *// Outputs Book: Java Guide* magazine.print(); *// Outputs Magazine: Tech Weekly  
  
 // Example 2: Overloading  
 describe*((Book) book); *// Outputs Describing book: Java Guide  
 describe*((Magazine) magazine); *// Outputs Describing magazine: Tech Weekly  
  
 // Example 3: Array of interfaces  
 Printable*[] items = {new Book("Learn Java"), new Magazine("Code Monthly")};  
 for (*Printable* item : items) {  
 item.print();  
 }  
  
 *// Example 4: Edge case - Invalid cast* try {  
 Magazine m = (Magazine) book;  
 } catch (ClassCastException e) {  
 System.out.println("Caught ClassCastException");  
 }  
 }  
}

\*\*Edge Cases and Scenarios\*\*:  
 - \*\*Invalid Casting\*\*: Throws `ClassCastException`.  
 - \*\*Overriding Rules\*\*: Must match signatures and return types.  
 - \*\*Overloading Ambiguity\*\*: Similar signatures may cause compilation errors.  
 - \*\*Null References\*\*: Calling methods on `null` throws `NullPointerException`.

# Conclusion

This in-depth guide provides a thorough understanding of core Java concepts for beginners, with extensive examples and edge cases. Detailed sections on strings and collections cover all key types, methods, and implementations, ensuring practical knowledge. Practice these concepts, experiment with the examples, and build projects to solidify your skills.