Advanced Programming (OOP)

Module 2: Basic Principles of OOP Development with Java

SDB

Module 2: Topics

- Introduction to the Object-Oriented Paradigm.
- Identifying classes, attributes, methods, and objects.
- Understanding class relationships (containment and association).
- Writing and instantiating classes.
- Java Basics to Intermediate

Outline

Object-Oriented Paradigm

Definition: Programming paradigm based on the concept of "objects" containing data and methods.

Key Features:

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Benefits:

- Models real-world entities.
- Enhances code reusability.
- Improves program organization.

Identifying Classes, Attributes, and Methods

Class Design Example:

- **Class:** Student
- **Attributes:**
  - name: String
  - rollNo: int
  - marks: float
- **Methods:**
  - calculateGrade()
  - displayDetails()

Real-World Example:

- A library system with classes like ‘Book‘, ‘Member‘, and ‘Librarian‘.
- Methods include issuing books, calculating fines, etc.

Containment and Association

Containment:

- "Has-a" relationship.
- Example: A ‘Car‘ class containing an ‘Engine‘ object.

Association:

- "Uses-a" relationship.
- Example: A ‘Customer‘ class associated with multiple ‘Order‘ objects.

Java Example: Containment

```java
class Engine {
    private String type;
    public Engine(String type) { this.type = type;}
    public String getType() { return type;}
}

class Car {
    private Engine engine;
    public Car(Engine engine) { this.engine = engine;}
    public void displayEngineType() {
        System.out.println("Engine: " + engine.getType());
    }
}
```

Outline

Introduction to Java Basics

Definition: Java is a high-level, class-based, object-oriented programming language designed for portability and flexibility.

Key Features:

- Platform-independent ("Write Once, Run Anywhere").
- Automatic memory management using garbage collection.
- Supports multi-threading and concurrency.
- Strongly typed with a rich set of libraries.

Java Classes

Definition: A class in Java is a blueprint for creating objects that encapsulate data and methods.

Structure:

- Fields (attributes): Variables that store object state.
- Methods: Functions that define the behavior of the object.
- Constructors: Special methods to initialize objects.

Example:

```java
class Car {
    String brand;
    int year;

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

    void displayInfo() {
        System.out.println("Brand: " + brand + ", Year: " + year);
    }
}
```

Listing 1: Basic Class Example

Overview of Java Keywords

Java Keywords: Reserved words in Java that have predefined meanings.
Categories:

- **Access Control:** ‘public‘, ‘private‘, ‘protected‘
- **Modifiers:** ‘static‘, ‘final‘, ‘abstract‘, ‘synchronized‘
- **Flow Control:** ‘if‘, ‘else‘, ‘switch‘, ‘case‘, ‘default‘
- **Loops:** ‘for‘, ‘while‘, ‘do‘
- **Object-Oriented:** ‘class‘, ‘interface‘, ‘extends‘, ‘implements‘
- **Error Handling:** ‘try‘, ‘catch‘, ‘finally‘, ‘throw‘, ‘throws‘
- **Others:** ‘new‘, ‘this‘, ‘super‘, ‘return‘

Branching in Java

Definition: Branching allows the program to make decisions and execute different paths based on conditions.

Types:

- **‘if‘ Statement:** Executes a block if the condition is true.
- **‘if-else‘ Statement:** Provides an alternative block if the condition is false.
- **‘switch‘ Statement:** Selects from multiple options.

Example:

```java
int number = 10;
if (number > 0) {
    System.out.println("Positive number");
} else {
    System.out.println("Negative number");
}
```

Listing 2: If-Else Example

Looping in Java

Definition: Loops are used to execute a block of code repeatedly.
Types:

- **For Loop:** Iterates a block a fixed number of times.
- **While Loop:** Repeats while a condition is true.
- **Do-While Loop:** Executes the block at least once.
- **Enhanced For Loop:** Iterates over arrays or collections.

Example:

```java
for (int i = 1; i <= 5; i++) {
    System.out.println("Count: " + i);
}
```

Listing 3: For Loop Example

Best Practices for Java

- Use meaningful names for classes, methods, and variables.
- Keep methods focused and concise.
- Follow consistent indentation and coding standards.
- Comment code where necessary, especially for complex logic.
- Use branching and looping constructs effectively to improve code readability.
- Practice identifying relationships between classes (e.g., inheritance, aggregation).

Outline

Access Specifiers in Java

Definition: Access specifiers define the visibility and accessibility of classes, methods, and variables in Java.

Types of Access Specifiers:

- **Public:** Accessible from anywhere.
- **Private:** Accessible only within the same class.
- **Protected:** Accessible within the same package and subclasses.
- **Default (Package-Private):** Accessible only within the same package.

Example:

```java
class Example {
    public int publicVar = 10;
    private int privateVar = 20;
    protected int protectedVar = 30;
    int defaultVar = 40; // Default access
}
```

Listing 4: Access Specifiers

The ‘this‘ and ‘super‘ Keywords

‘this‘ Keyword:

- Refers to the current instance of the class.
- Used to access current class methods, fields, and constructors.
- Resolves naming conflicts between instance variables and parameters.

‘super‘ Keyword:

- Refers to the immediate parent class instance.
- Used to call parent class methods and constructors.
- Accesses hidden fields or overridden methods in the parent class.

Example: ‘this‘

Using ‘this‘ to Resolve Naming Conflicts:

```
class Example {
    int value;

    Example(int value) {
        this.value = value; // Resolves conflict with
            parameter
    }

    void display() {
        System.out.println("Value: " + this.value); //
            Refers to instance variable
    }
}
```

Listing 5: Using this Keyword

Example: ‘super‘

Using ‘super‘ to Access Parent Class Members:

```java
class Parent {
    void show() {
        System.out.println("Parent method");
    }
}

class Child extends Parent {
    void show() {
        super.show(); // Calls parent class method
        System.out.println("Child method");
    }
}
```

Listing 6: Using super Keyword

Outline

Java Constructors

Definition: A constructor is a special method used to initialize objects in Java.

Key Features:

- Same name as the class.
- No return type.
- Automatically called when an object is created.

Types of Constructors:

- Default Constructor
- Parameterized Constructor
- Copy Constructor (not natively supported but can be implemented).

Default Constructor

Definition: A constructor with no arguments provided by the compiler if none is explicitly defined.
Example:

```java
class Example {
    int value;

    Example() {
        value = 10; // Default initialization
    }
    void display() {
        System.out.println("Value: " + value);
    }
}

public class Test {
    public static void main(String[] args) {
        Example obj = new Example();
        obj.display(); // Output: Value: 10
    }
}
```

Listing 7: Default Constructor

Parameterized Constructor

Definition: A constructor that accepts arguments to initialize fields with custom values.

Example:

```java
class Example {
    int value;
    Example(int value) {
        this.value = value; // Custom initialization
    }
    void display() {
        System.out.println("Value: " + value);
    }
}
public class Test {
    public static void main(String[] args) {
        Example obj = new Example(42);
        obj.display(); // Output: Value: 42

        //Example obj2 = new Example();
        //will the above code ^^^ still work?
        //Check for yourself.
    }
}
```

Listing 8: Parameterized Constructor

Constructor Overloading

Definition: Multiple constructors in the same class with different parameter lists.

Example:

```java
class Example {
    int value;

    // Default constructor
    Example() {value = 0;}

    // Parameterized constructor
    Example(int value) {this.value = value;}

    void display() {System.out.println("Value: " + value);}
}

public class Test {
    public static void main(String[] args) {
        Example obj1 = new Example();
        Example obj2 = new Example(99);
        obj1.display(); // Output: Value: 0
        obj2.display(); // Output: Value: 99
    }
}
```

Listing 9: Constructor Overloading

Copy Constructor

Definition: A constructor that creates a new object as a copy of an existing object (not natively supported in Java).

Example:

```java
class Example {
    int value;

    // Parameterized constructor
    Example(int value) {this.value = value;}

    // Copy constructor
    Example(Example obj) {this.value = obj.value;}

    void display() {System.out.println("Value: " + value);}
}

public class Test {
    public static void main(String[] args) {
        Example obj1 = new Example(123);
        Example obj2 = new Example(obj1); // Copy constructor

        obj1.display(); // Output: Value: 123
        obj2.display(); // Output: Value: 123
    }
}
```

Listing 10: Copy Constructor

Best Practices for Using Constructors

- Always initialize fields to meaningful default values.
- Use 'this' keyword to differentiate between class attributes and parameters.
- Avoid complex logic inside constructors to keep initialization simple.
- Use constructor chaining to avoid code duplication.
- Implement a copy constructor for creating duplicates when necessary.

Constructor Chaining

Definition: Calling one constructor from another within the same class.
Example:

```
class Example {
    int value;

    Example() {
        this(42); // Calls parameterized constructor
    }

    Example(int value) {
        this.value = value;
    }

    void display() {System.out.println("Value: " + value);}
}

public class Test {
    public static void main(String[] args) {
        Example obj = new Example();
        obj.display(); // Output: Value: 42
    }
}
```

Listing 11: Constructor Chaining

Outline

Introduction to the ‘abstract‘ Keyword in Java

Definition: The ‘abstract‘ keyword in Java is used to define a class or method that is incomplete and must be implemented or extended.

Uses:

- To create abstract classes that serve as a blueprint for subclasses.
- To declare abstract methods that subclasses must implement.

Key Points:

- Abstract classes cannot be instantiated.
- Abstract methods do not have a body.
- A class with at least one abstract method must be declared abstract.

Abstract Class Example

Example:

```
abstract class Animal {
    abstract void sound(); // Abstract method

    public void eat() {
        System.out.println("This animal eats food.");
    }
}

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

public class Test {
    public static void main(String[] args) {
        Animal dog = new Dog();
        dog.eat();
        dog.sound();
    }
}
```

Abstract Method Example

Definition: An abstract method is a method that is declared without an implementation.

Syntax:

```java
abstract class Shape {
    abstract void draw(); // Abstract method
}

class Circle extends Shape {
    @Override
    void draw() {
        System.out.println("Drawing a circle.");
    }
}
```

Key Points:

- Subclasses must override abstract methods.
- Abstract methods cannot be ‘final‘, ‘static‘, or ‘private‘.

Features of the ‘abstract‘ Keyword

For Classes:

- Can include concrete (non-abstract) methods.
- Can define fields and constructors.
- Acts as a base class for other classes.

For Methods:

- Forces subclasses to provide specific behavior.
- Declared using the ‘abstract‘ keyword without a body.

Restrictions:

- Cannot instantiate abstract classes.
- Abstract methods cannot have a body.

Advanced Example: Abstract Class with Fields and Constructor

```java
abstract class Vehicle {
    String type;
    Vehicle(String type) {
        this.type = type;
    }
    abstract void start();
    public void stop() {
        System.out.println(type + " has stopped.");
    }
}
class Car extends Vehicle {
    Car() {
        super("Car");
    }
    @Override
    void start() {
        System.out.println("Car is starting.");
    }
}

public class Test {
    public static void main(String[] args) {
        Vehicle car = new Car();
        car.start();
        car.stop();
    }
}
```

When to Use the ‘abstract‘ Keyword

Scenarios:

- To define a common interface or contract for related classes.
- When shared behavior needs to be implemented once in a base class.
- When some methods must be implemented by subclasses, enforcing consistency.

Examples:

- Abstract ‘Shape‘ class with subclasses like ‘Circle‘ and ‘Rectangle‘.
- Abstract ‘Employee‘ class for ‘Manager‘ and ‘Developer‘.

Common Mistakes with the ‘abstract‘ Keyword

1. **Instantiating Abstract Classes:**

```
abstract class AbstractClass {}

public class Test {
    public static void main(String[] args) {
        // AbstractClass obj = new AbstractClass(); // Error
    }
}
```

2. **Missing Abstract Method Implementations:**

```
abstract class Parent {
    abstract void method();
}

class Child extends Parent {
    // Error: Child must implement abstract method
}
```

Best Practices for Using the ‘abstract‘ Keyword

- Use abstract classes for shared behavior and partial implementation.
- Prefer abstract classes when common fields or methods are needed.
- Avoid making an abstract class overly specific.
- Document the purpose of abstract methods for clarity.

Outline

Introduction to the ‘final‘ Keyword in Java

Definition: The ‘final‘ keyword in Java is used to restrict the modification of classes, methods, and variables.

Key Features:

- A ‘final‘ variable becomes a constant and cannot be reassigned.
- A ‘final‘ method cannot be overridden by subclasses.
- A ‘final‘ class cannot be extended.

Use Case: To ensure immutability, prevent inheritance, or avoid method overriding.

Final Variables

Definition: A variable declared as ‘final‘ can only be assigned once.

```java
class Constants {
    final int MAX_VALUE = 100;

    void display() {
        // MAX_VALUE = 200; // Error: Cannot assign a value to final variable
        System.out.println("Max Value: " + MAX_VALUE);
    }
}

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

Key Points:

- ‘final‘ variables must be initialized at the time of declaration or in the constructor.
- Ensures immutability for constants.

Final Methods

Definition: A method declared as ‘final‘ cannot be overridden by subclasses.

```java
class Parent {
    final void display() {
        System.out.println("This is a final method.");
    }
}

class Child extends Parent {
    // void display() { // Error: Cannot override final method
    //     System.out.println("Overriding final method.");
    // }
}

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

Final Classes

Definition: A class declared as 'final' cannot be extended.

```java
final class ImmutableClass {
    void display() {
        System.out.println("This is a final class.");
    }
}

// class SubClass extends ImmutableClass { // Error: Cannot inherit from final class
// }

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

Final Keyword with Reference Variables

Key Points:

- The reference variable declared as 'final' cannot point to a different object.
- However, the object it references can be modified.

```java
class Example {
    public static void main(String[] args) {
        final StringBuilder sb = new StringBuilder("Hello");

        sb.append(" World"); // Allowed: Modifying the object
        System.out.println(sb);

        // sb = new StringBuilder("New"); // Error: Cannot reassign final reference
    }
}
```

Advantages of the ‘final‘ Keyword

- Improves security by preventing unintended modification of data or behavior.
- Enhances readability by clearly signaling immutability or fixed behavior.
- Helps in optimization by allowing the compiler to inline final methods.
- Prevents inheritance for sensitive classes, ensuring encapsulation.

Disadvantages of the ‘final‘ Keyword

- Reduces flexibility in extending or overriding behavior.
- Can lead to verbose code when alternative designs (e.g., composition) are required.
- Misuse may unnecessarily restrict extensibility.

Common Mistakes with the ‘final‘ Keyword

1. Not Initializing Final Variables:

```java
class Example {
    final int value;

    Example() {
        // value is not initialized
    }
}
```

2. Reassigning Final References:

```java
final String name = "John";
// name = "Doe"; // Error: Cannot reassign final variable
```

Best Practices for Using the ‘final‘ Keyword

- Use ‘final‘ for constants, method parameters, and variables that should not change.
- Prefer ‘final‘ classes for utility or immutable classes.
- Avoid overusing ‘final‘ where flexibility is required.
- Use descriptive naming conventions for ‘final‘ variables (e.g., ‘MAX_VALUE‘).

Outline

Introduction to the ‘static‘ Keyword in Java

Definition: The ‘static‘ keyword in Java is used to indicate that a member (field, method, block, or nested class) belongs to the class rather than to any specific instance.

Key Characteristics:

- Shared across all instances of the class.
- Can be accessed without creating an object of the class.
- Improves memory efficiency by maintaining a single copy.

Static Fields (Class Variables)

Definition: A static field is a variable that is shared among all instances of the class.

Example:

```java
class Counter {
    static int count = 0; // Static field

    Counter() {
        count++;
    }
}

public class Test {
    public static void main(String[] args) {
        new Counter();
        new Counter();
        System.out.println("Total objects created: " + Counter.count);
    }
}
```

Listing 12: Static Field Example

Output:

- Total objects created: 2

Static Methods and Variables: Overview

Static Members:

- Declared using the ‘static‘ keyword.
- Belong to the class rather than any instance.
- Shared across all instances of the class.

Static Method:

- Can be called without creating an instance of the class.
- Cannot access non-static fields or methods directly.

Static Variable:

- A single copy is created and shared among all instances.
- Useful for storing common properties or counters.

Static Methods

Definition: A static method belongs to the class and can be called without an instance.

Key Points:

- Cannot access non-static fields or methods directly.
- Can be overloaded but not overridden.

Example:

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

public class Test {
    public static void main(String[] args) {
        int result = Utility.add(5, 10);
        System.out.println("Sum: " + result);
    }
}
```

Listing 13: Static Method Example

Static Blocks

Definition: A static block is used to initialize static fields and is executed when the class is loaded into memory.

Example:

```java
class Initialization {
    static int value;

    static {
        value = 42;
        System.out.println("Static block executed.");
    }
}

public class Test {
    public static void main(String[] args) {
        System.out.println("Value: " + Initialization.value);
    }
}
```

Listing 14: Static Block Example

Output:

- Static block executed.
- Value: 42

Static Nested Classes

Definition: A static nested class is a nested class that does not require an instance of the enclosing class.

Example:

```java
class Outer {
    static class Nested {
        void display() {
            System.out.println("Inside static nested class.");
        }
    }
}

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

Listing 15: Static Nested Class Example

Combining Static and Non-Static Members

```java
class MixedExample {
    static int staticCount = 0; // Shared among all instances
    int instanceCount = 0; // Unique to each instance

    void increment() {
        staticCount++; instanceCount++;
    }
    static void displayStaticCount() {
        System.out.println("Static Count: " + staticCount);
        // System.out.println("Instance Count: " + instanceCount); // Error
    }
    void displayInstanceCount() {
        System.out.println("Instance Count: " + instanceCount);
    }
}
public class Test {
    public static void main(String[] args) {
        MixedExample obj1 = new MixedExample();
        MixedExample obj2 = new MixedExample();

        obj1.increment();  obj2.increment();

        MixedExample.displayStaticCount(); // Outputs: Static Count: 2
        obj1.displayInstanceCount(); // Outputs: Instance Count: 1
        obj2.displayInstanceCount(); // Outputs: Instance Count: 1
    }
}
```

Listing 16: Mixing Static and Non-Static Members

Advantages and Disadvantages of the ‘static‘ Keyword

Advantages:

- Reduces memory usage by sharing fields and methods. Efficient memory usage as only one copy of static variables exists.
- Can be accessed globally without creating instances. Improves performance by avoiding the need for object creation.
- Provides a mechanism for utility methods and constants.
- Simplifies code structure for nested classes.

Disadvantages:

- Limits flexibility as static members cannot access instance variables or methods.
- May lead to tight coupling if overused.
- Makes unit testing more difficult in some cases.
- Not thread-safe unless explicitly synchronized.

Common Mistakes with the 'static' Keyword I

1. Accessing Non-Static Members in Static Context: A static method cannot directly access non-static fields or methods.

```
class Example {
    int instanceVar = 10;

    static void display() {
        // System.out.println(instanceVar); // Error: Non-static field cannot be
            referenced
    }
}
```

Listing 17: Access Error

2. Overusing Static Members:

- Making everything static reduces modularity and object-oriented design principles.
- Excessive use can lead to unexpected behaviors and global state issues.

Common Mistakes with the ‘static‘ Keyword II

3. Misunderstanding Static Method Behavior: Static methods cannot be overridden; they are hidden instead.

```java
class Parent {
    static void display() { System.out.println("Parent static display");}
}
class Child extends Parent {
    static void display() { System.out.println("Child static display");}
}
public class Test {
    public static void main(String[] args) {
        Parent.display(); // Output: Parent static display
        Child.display(); // Output: Child static display
    }
}
```

Listing 18: Static Method Hiding

Best Practices for Using the ‘static‘ Keyword

- Use ‘static‘ for utility or helper methods (e.g., ‘Math.sqrt()‘).
- Declare constants as ‘static final‘ (e.g., ‘PI = 3.14‘).
- Avoid overusing static fields and methods to maintain encapsulation. Limit the use of static variables to avoid unintended global state.
- Use static blocks for complex initialization logic only when necessary.
- Prefer static nested classes for grouping related classes.
- Use static methods for utility or helper functions that do not depend on instance variables.
- Document the purpose of static members clearly to avoid confusion.
- Avoid accessing static members through instances; use the class name for clarity.
- Ensure thread safety when static variables are shared across multiple threads.

Outline

Understanding ‘public static void main‘

Definition: The ‘main‘ method is the entry point for any standalone Java application.

Components of ‘public static void main‘:

- **public:** Allows the method to be accessible from anywhere.
- **static:** Enables the method to be called without creating an instance of the class.
- **void:** Specifies that the method does not return any value.
- **main:** The name of the method recognized by the JVM as the starting point.
- **String[] args:** Represents command-line arguments passed to the program.

Using ‘main‘ Method Arguments:

```java
public class CommandLineExample {
    public static void main(String[] args) {
        System.out.println("Number of arguments: " + args.length);

        for (int i = 0; i < args.length; i++) {
            System.out.println("Argument " + i + ": " + args[i]);
        }
    }
}

// Run with: java CommandLineExample arg1 arg2 arg3
// Output:
// Number of arguments: 3
// Argument 0: arg1
// Argument 1: arg2
// Argument 2: arg3
```

Listing 19: Command-Line Arguments

Key Points to Remember

- The ‘main‘ method is mandatory for standalone Java applications.
- Only one ‘main‘ method per class is executed.
- You can overload the ‘main‘ method, but only the standard signature is used as the entry point.
- Command-line arguments allow passing data to the application at runtime.

Best Practices for Using ‘PSVM‘

- Minimize logic in the ‘main‘ method; delegate tasks to other methods.
- Validate command-line arguments before using them.
- Avoid hardcoding in the ‘main‘ method; use configuration files or arguments.
- Document the expected arguments for clarity.
- Use tools like ‘System.exit(int status)‘ for proper program termination.

Outline

Introduction to Garbage Collection in Java

Definition: Garbage collection in Java is an automatic process that identifies and removes unused or unreachable objects to free up memory.

Key Features:

- Managed by the JVM, eliminating manual memory management.
- Ensures efficient memory utilization and prevents memory leaks.
- Uses various algorithms and strategies to optimize performance.

Use Case: Reclaims memory occupied by objects no longer referenced in the application.

How Garbage Collection Works

Phases of Garbage Collection:

- **Mark:** Identifies objects that are still reachable.
- **Sweep:** Removes objects that are no longer reachable.
- **Compact:** Rearranges remaining objects to eliminate fragmentation (optional).

Roots of Reachability:

- Local variables and method parameters.
- Static fields of loaded classes.
- Active threads.

Best Practices for Managing Garbage Collection

- Use appropriate JVM options to optimize GC for your application.
- Avoid creating unnecessary objects.
- Prefer primitives over wrapper classes when possible.
- Profile and monitor GC using tools like VisualVM or JConsole.
- Use soft and weak references for cache management.

Outline

Introduction to Java Lambdas

Definition: A lambda expression is a concise way to represent an anonymous function.

Key Characteristics:

- Introduced in Java 8.
- Enables functional programming.
- Simplifies the use of functional interfaces.

Syntax: '(parameters) -¿ expression'

- **No Parameters:** '() $->$ System.out.println("Hello")'
- **Single Parameter:** 'x $->$ x * x'
- **Multiple Parameters:** '(x, y) $->$ x $+$ y'

Example: Using Lambda Expressions

Example: Iterating Over a List

```
import java.util.*;

public class LambdaExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

        names.forEach(name -> System.out.println(name));
    }
}
```

Listing 20: Lambda for Iteration

Functional Interfaces and Lambdas

Definition: A functional interface is an interface with exactly one abstract method.

Common Functional Interfaces:

- **'Predicate**$< T >$**':** Tests a condition.
- **'Function**$< T, R >$**':** Transforms an input to an output.
- **'Consumer**$< T >$**':** Consumes an input without returning a value.
- **'Supplier**$< T >$**':** Supplies a value without input.

Example:

```java
import java.util.function.*;

public class FunctionalInterfaceExample {
    public static void main(String[] args) {
        Predicate<Integer> isEven = x -> x % 2 == 0;
        System.out.println(isEven.test(4)); // true

        Function<String, Integer> length = str -> str.length();
        System.out.println(length.apply("Hello")); // 5
    }
}
```

Listing 21: Using Functional Interfaces

Advantages and Disadvantages of Lambda Expressions

Advantages:

- Reduces boilerplate code for anonymous classes.
- Enhances readability and conciseness.
- Enables functional programming paradigms.
- Simplifies working with streams and collections.

Disadvantages:

- Limited debugging due to lack of explicit names.
- Can reduce readability if overused in complex logic.
- May introduce performance overhead in some scenarios.

Common Mistakes with Lambdas

1. **Overusing Lambdas:**
   - Using lambdas for complex logic reduces code readability.
2. **Ignoring Capturing Rules:**
   - Lambda expressions cannot modify non-final local variables.

Example:

```java
public class CaptureExample {
    public static void main(String[] args) {
        int num = 10;

        Runnable r = () -> {
            // num++; // Error: Variable used in lambda should be final or effectively
                final
            System.out.println(num);
        };

        r.run();
    }
}
```

Listing 22: Variable Capture

Best Practices for Using Lambdas

- Use lambdas for short, concise logic.
- Prefer method references where applicable (e.g., ‘System.out::println‘).
- Avoid embedding lambdas in deeply nested code.
- Use functional interfaces to enhance reusability.

Outline

Introduction to Anonymous Classes

What is an Anonymous Class?

- A class that is defined without a name.
- Used to create an instance of a class or implement an interface.
- Defined inside a method or a block of code.

Syntax of Anonymous Classes:

```
new ClassName() {
  // class body
};
```

Or:

```
new InterfaceName() {
  // interface implementation
};
```

Example of Anonymous Class

Example:

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

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

Listing 23: Example of Anonymous Class

Benefits of Anonymous Classes

Benefits:

- Can be used to create an instance of a class or implement an interface.
- Can be defined inside a method or a block of code.
- Can access the variables of the surrounding scope.
- Can be used to create a one-time use class.

Use Cases of Anonymous Classes

Use Cases:

- Event handling: creating event listeners.
- GUI programming: creating GUI components.
- Multithreading: creating threads.
- Data processing: creating data processors.

Best Practices for Anonymous Classes

Best Practices:

- Keep anonymous classes short and simple.
- Use them sparingly and only when necessary.
- Avoid complex logic inside anonymous classes.
- Use them to create one-time use classes.

Outline

Exercises for Students

1. **Class Creation:**
   - Create a class ‘Employee‘ with attributes like ‘name‘, ‘id‘, and ‘salary‘.
   - Write methods to calculate annual salary and display details.
2. **Class Relationships:**
   - Implement a ‘Department‘ class that contains multiple ‘Employee‘ objects.
   - Write methods to add employees and display department details.

Exercise Scenarios for Students

Easy Level:

- Create a ‘Book‘ class with attributes ‘title‘, ‘author‘, and ‘price‘. Add a method to display the book details.
- Write a program to calculate the sum of the first 10 natural numbers using a ‘for‘ loop.

Intermediate Level:

- Implement a ‘Student‘ class with attributes ‘name‘, ‘rollNo‘, and ‘marks‘. Include methods to calculate and display the grade.
- Write a program to count the number of vowels in a given string using a ‘switch‘ statement.

Hard Level:

- Create a ‘BankAccount‘ class with methods for deposit, withdrawal, and balance inquiry. Add validations for sufficient balance.
- Write a program to find the factorial of a number using recursion.

Advanced Level:

- Design a ‘Library‘ system with classes ‘Library‘, ‘Book‘, and ‘Member‘. Implement relationships like aggregation and methods for

Discussion Questions

- How would you identify classes and objects in a real-world system?
- What are the advantages of using containment over direct attributes?
- Can you think of scenarios where association is preferred over inheritance?

Advanced Java Keywords

- final: Used to declare constants and prevent method overriding.
- static: Used to declare methods and variables that belong to a class rather than an instance.
- abstract: Used to declare methods and classes that cannot be instantiated.
- interface: Used to declare a contract that must be implemented by any class that implements it.

Java Packages

- Definition: A way of organizing related classes and interfaces into a single unit.
- Purpose: To provide a way of grouping related classes and interfaces and to prevent naming conflicts.
- How to create: Use the package keyword followed by the name of the package.

Outline

Introduction to ‘public static void main‘ (PSVM) in Java

Definition: The ‘public static void main‘ method is the entry point for any standalone Java application.

Signature:

```java
public static void main(String[] args)
```

Listing 24: Main Method Syntax

Key Components:

- **‘public‘:** The method is accessible from anywhere.
- **‘static‘:** Allows the JVM to invoke the method without instantiating the class.
- **‘void‘:** The method does not return a value.
- **‘String[] args‘:** An array to capture command-line arguments.

Role of ‘PSVM‘ in Java

Importance:

- Serves as the starting point for JVM to execute a Java program.
- Provides a way to interact with the program through command-line arguments.
- Ensures compatibility across different Java applications.

Example:

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

Listing 25: Basic Main Method

Advanced Example: Using Command-Line Arguments

Command-Line Arguments: Allows the user to pass inputs when starting the program.

Example:

```java
public class CommandLineExample {
    public static void main(String[] args) {
        System.out.println("Number of arguments: " + args.length);

        for (int i = 0; i < args.length; i++) {
            System.out.println("Argument " + i + ": " + args[i]);
        }
    }
}
```

Listing 26: Command-Line Arguments Example

Execution:

- Compile: ‘javac CommandLineExample.java‘
- Run: ‘java CommandLineExample arg1 arg2 arg3‘

Overloading the ‘main‘ Method

Key Points:

- The ‘main‘ method can be overloaded like any other method.
- The JVM only calls the ‘public static void main(String[] args)‘ method.
- Overloaded ‘main‘ methods can be invoked explicitly.

Example:

```java
public class MainOverload {
    public static void main(String[] args) {
        System.out.println("Default main method");
        main(5);
    }

    public static void main(int number) {
        System.out.println("Overloaded main method: " + number);
    }
}
```

Listing 27: Overloading Main Method

Common Mistakes with ‘PSVM‘

1. **Missing ‘static‘ Keyword:**

```
public class Test {
    public void main(String[] args) { // Error: JVM requires static
        System.out.println("Missing static");
    }
}
```

Listing 28: Error: Missing Static Keyword

2. **Incorrect Method Signature:**

```
public class Test {
    static public void Main(String[] args) { // Error: Method name is case-sensitive
        System.out.println("Incorrect signature");
    }
}
```

Listing 29: Error: Incorrect Signature

Debugging with ‘PSVM‘

Tips:

- Add debug statements to track execution flow.
- Use IDE breakpoints for step-by-step debugging.
- Log command-line arguments to verify inputs.

Example:

```
public class DebugExample {
    public static void main(String[] args) {
        System.out.println("Program started");

        if (args.length == 0) {
            System.out.println("No arguments provided");
        } else {
            for (String arg : args) {
                System.out.println("Argument: " + arg);
            }
        }

        System.out.println("Program ended");
    }
}
```

Listing 30: Debugging with Print Statements

Outline

Garbage Collection Algorithms

1. **Serial Garbage Collector:**
   - Uses a single thread for garbage collection.
   - Suitable for applications with small heaps.
2. **Parallel Garbage Collector:**
   - Uses multiple threads for garbage collection.
   - Optimized for multi-threaded applications.
3. **G1 Garbage Collector:**
   - Divides the heap into regions and prioritizes garbage collection based on regions with the most garbage.
   - Balances throughput and low-latency requirements.
4. **Z Garbage Collector:**
   - Designed for low-latency applications.
   - Handles very large heaps (up to terabytes).

Calling Garbage Collector:

- The ‘System.gc()‘ or ‘Runtime.getRuntime().gc()‘ method suggests garbage collection to the JVM.
- Garbage collection is not guaranteed.

Example:

```
public class GarbageCollectionExample {
    public static void main(String[] args) {
        GarbageCollectionExample obj = new GarbageCollectionExample();
        obj = null; // Object is now eligible for garbage collection

        System.gc();
        System.out.println("Garbage collection requested.");
    }

    @Override
    protected void finalize() throws Throwable {
        System.out.println("Finalize method called.");
    }
}
```

Listing 31: Calling Garbage Collector

Garbage Collection in Java 9+

Improvements in Java 9+:

- **Unified Logging:** Provides better logging for GC activities using the '-Xlog:gc' option.
- **G1 as Default Collector:** The G1 garbage collector is the default GC from Java 9.
- **Z Garbage Collector (Java 11):** Introduced for low-latency, scalable garbage collection.

Advanced Options:

- `-XX:+UseG1GC`: Enable G1 GC.
- `-XX:+UseZGC`: Enable Z GC (Java 11+).
- `-XX:MaxHeapFreeRatio`: Control heap resizing.

Common Mistakes with Garbage Collection

1. **Relying on Finalize Method:**
   - The ‘finalize()‘ method is deprecated and not recommended for cleanup.
   - Use ‘try-with-resources‘ or explicit resource management instead.
2. **Creating Memory Leaks:**
   - Retaining references to objects unnecessarily.
   - Example: Adding objects to collections but not removing them.
3. **Ignoring GC Logs:**
   - GC logs provide valuable insights into application performance and memory usage.

Tools for Monitoring Garbage Collection

1. **VisualVM:**
   - Monitors memory usage and garbage collection activity.
2. **JConsole:**
   - Provides real-time insights into JVM memory management.
3. **GC Logs:**
   - Use the '-Xlog:gc' flag to enable detailed logging.
4. **Java Mission Control:**
   - Advanced profiling and diagnostics tool for JVM.

Outline

Introduction to the Java Virtual Machine (JVM)

Definition: The JVM is an abstract computing machine that enables a computer to run Java programs and programs written in other languages compiled to Java bytecode.

Key Responsibilities:

- Executes Java bytecode.
- Provides runtime environment.
- Manages memory and garbage collection.

Components:

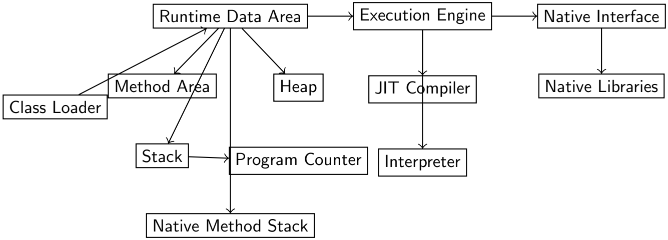
- Class Loader.
- Runtime Data Area.
- Execution Engine.
- Native Method Interface and Libraries.

Architecture of the JVM

Key Components:

- **Class Loader:** Loads, links, and initializes classes and interfaces.
- **Runtime Data Area:** Includes memory areas like the method area, heap, stack, program counter, and native method stack.
- **Execution Engine:** Executes bytecode using an interpreter or Just-In-Time (JIT) compiler.
- **Native Method Interface (JNI):** Facilitates interaction with native libraries.

Diagram:



Class Loading Mechanism

Steps:

- **Loading:** Finds and loads the class file.
- **Linking:**
  - **Verify:** Ensures bytecode complies with JVM specifications.
  - **Prepare:** Allocates memory for class variables and initializes them to default values.
  - **Resolve:** Replaces symbolic references with direct references.
- **Initialization:** Executes static initializers and static blocks.

Example:

```java
class Example {
    static {
        System.out.println("Class loaded and initialized.");
    }
}

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

Listing 32: Class Loading Example

Runtime Data Area

Memory Areas:

- **Method Area:** Stores class structures (e.g., method code, constants).
- **Heap:** Stores objects and JRE classes.
- **Stack:** Contains stack frames for method invocations (local variables, partial results).
- **Program Counter (PC):** Tracks the address of the current executing instruction.
- **Native Method Stack:** Supports native methods used by the JVM.

Execution Engine

Components:

- **Interpreter:** Executes bytecode line by line but is slower.
- **Just-In-Time (JIT) Compiler:** Compiles bytecode to native code for faster execution.
- **Garbage Collector:** Reclaims unused memory to optimize memory usage.

JIT Compilation:

- Converts frequently used bytecode into native machine code.
- Optimizes performance using techniques like inlining and loop unrolling.

Garbage Collection in JVM

Key Features:

- Automates memory management.
- Eliminates the need for manual ‘free()‘ calls.

Algorithms:

- Mark-and-Sweep.
- Generational Garbage Collection.
- G1 Garbage Collector.

Tuning GC:

- Use JVM options like ‘-Xms‘, ‘-Xmx‘, ‘-XX:+UseG1GC‘.

JVM Options for Optimization

Memory Management:

- ‘-Xms‘: Initial heap size.
- ‘-Xmx‘: Maximum heap size.
- ‘-XX:MetaspaceSize‘: Initial metaspace size.
- ‘-XX:+UseG1GC‘: Enable G1 garbage collector.

Performance Monitoring:

- ‘-Xlog:gc‘: Log garbage collection activity.
- ‘-XX:+PrintCompilation‘: Logs JIT compilation details.

Common Mistakes with JVM

1. **Misconfigured Memory Settings:**
   - Setting heap size too low or too high can degrade performance.
2. **Ignoring GC Logs:**
   - Failing to analyze garbage collection logs may lead to memory leaks.
3. **Overlooking Thread Dumps:**
   - Missing out on diagnosing deadlocks or thread contention.

Best Practices for JVM Optimization

- Profile applications to identify memory and performance bottlenecks.
- Use appropriate garbage collectors for your application needs.
- Monitor JVM metrics using tools like JConsole, VisualVM, or Java Mission Control.
- Regularly update to the latest JVM version for performance improvements.