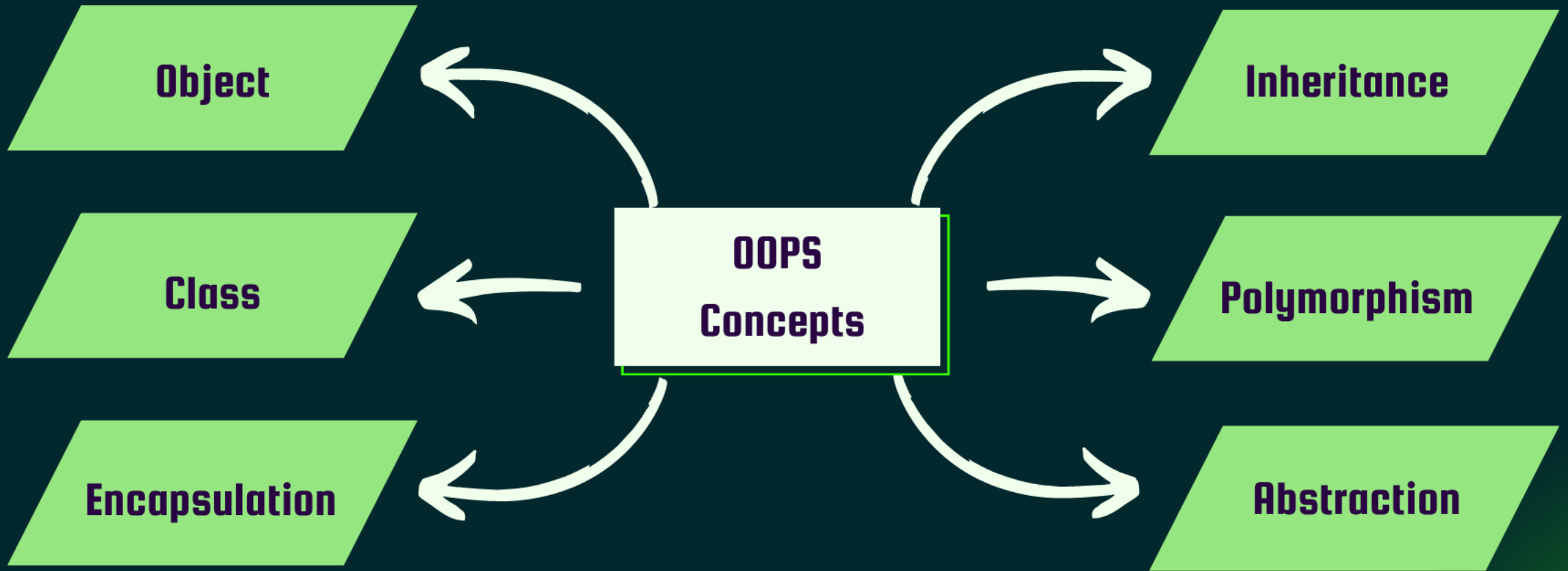
OOPs with JAVA

by

Mohammed Yahya

usn: 3PD23CS076

PRINCIPLE CONCEPTS



Object-Oriented Programming Paradigm (OOP): A New Programming Approach

Introduction to OOP as a New Paradigm

- The evolution of programming has seen significant milestones, starting from procedural programming to structured programming, and now, Object-Oriented Programming (OOP).
- OOP addresses the limitations of earlier paradigms by introducing a model that mirrors real-world objects and relationships.

Evolution of Programming Paradigms

- **Procedural Programming**: Focused on sequences of instructions, often leading to complex and rigid codebases.
- **Structured Programming**: Introduced modularity, enhancing code readability and reusability.
- **Object-Oriented Programming**: A paradigm shift that emphasizes objects and their interactions, promoting flexibility and scalability.

Structured vs. Object-Oriented Development

- **Structured Development**: Based on functions and control structures, where data and behavior are treated separately.
- **Object-Oriented Development**: Centers around **objects** and **classes**, encapsulating both data and behavior in cohesive units.

Key Features of Object-Oriented Programming

- **Objects and Classes:**
- **Objects**: Represent real-world entities with attributes (data) and methods (behavior).
- **Classes**: Blueprints that define the structure and behavior of objects.

- **Multiple Views of the Same Object:**
- Objects can be viewed differently depending on the context, enhancing flexibility.

Encapsulation and Data Abstraction:

- Encapsulation ensures data is hidden from unauthorized access.
- Abstraction focuses on essential characteristics, hiding complex details.

- **Inheritance:**
- Enables new classes to reuse and extend existing classes, promoting code reusability.

Delegation (Object Composition):

- Combines objects to achieve functionality without relying solely on inheritance.

- **Polymorphism:**
- Allows objects to be treated as instances of their parent class, supporting flexibility in method usage.

Introducing Data Types and Operators in Java

- **Java Primitive Types and Literals**
- Java provides a variety of primitive types, such as `int`, `float`, `char`, and `boolean`.
- **Literals**: Fixed values assigned to variables, like `123` for integers or `'A'` for characters.
- **Variables: Scope and Lifetime**
- Variables have **scope** (visibility) and **lifetime** (existence duration), critical for memory management.

Operators in Java

- **Arithmetic Operators**: Perform basic mathematical operations (+, -, *, /, %).
- **Relational and Logical Operators**: Compare values and manage logical conditions (>, <, &&, ||).
- **Short-Circuit Logical Operators**: Efficiently evaluate conditions using && and ||.
- **Assignment Operators**: Assign values (=, +=, -=).

Type Conversion and Operator Precedence

- **Type Conversion**: Converts variables from one type to another, using explicit casting if necessary.
- **Operator Precedence**: Determines the order in which operators are evaluated in an expression.

String Handling in Java

- **Fundamentals of Strings**
- Strings in Java are immutable sequences of characters, managed by the `String` class.
- **Constructors and Methods**
- **String Constructors**: Allow strings to be initialized in various ways.
- **Key Methods**:
- `length()`: Returns the number of characters in a string.
- `indexOf()` and `lastIndexOf()`: Locate characters or substrings.

Conclusion

- Object-Oriented Programming represents a robust paradigm shift, offering solutions to the limitations of earlier programming models.
- Java, with its strong support for OOP and efficient string handling, serves as a versatile tool for modern software development.
- Mastering these foundational concepts paves the way for building scalable and maintainable applications.

More Data Types and Operators in Java

- **Arrays and Multidimensional Arrays**
- **Arrays**:
- Arrays are collections of data elements of the same type, stored in contiguous memory locations.
- Declared using syntax: `type[] arrayName;` or `type arrayName[];` (alternative syntax).

Multidimensional Arrays:

- Arrays with two or more dimensions, often used for matrices.
- Example: `int[][] matrix = new int[3][3];`.
- **Key Features:**
- **Assigning Array References**: References can be reassigned to arrays of compatible types.
- **Using the `length` Member**: Provides the number of elements in an array.
- **For-Each Loop**: Simplifies iteration over array elements.

Strings

- Strings are immutable sequences of characters, handled using the `String` class.
- **Bitwise Operators**
- Perform operations on binary representations of integers.
- Examples: & (AND), | (OR), ^ (XOR), ~ (complement), and shift operators like <<, >>, >>>.

Introducing Classes, Objects, and Methods

- **Class Fundamentals**
- A **class** is a blueprint for objects, encapsulating data (fields) and behavior (methods).
- **Object Creation and Reference Variables**
- **Object Creation**: Done using the new operator. Example: `MyClass obj = new MyClass();`.
- **Reference Variables**: Store the address of an object and allow access to its members.

Methods

- Defined blocks of code that perform specific tasks.
- **Returning from Methods**: Methods can return values or be `void`.
- **Using Parameters**: Pass data to methods via arguments.
- **Constructors**
- **Constructors**: Special methods for initializing objects, automatically called during object creation.
- **Parameterized Constructors**: Accept arguments to initialize object attributes.

Key Concepts

- **The `new` Operator**: Revisited for memory allocation during object creation.
- **Garbage Collection**: Automatic memory management by removing unused objects.
- **The `this` Keyword**: Refers to the current object instance, often used to differentiate between instance variables and parameters.

A Closer Look at Methods and Classes

- **Controlling Access to Class Members**
- Use access modifiers (`private`, `protected`, `public`) to define visibility and encapsulation.
- **Passing and Returning Objects**
- Objects can be passed as arguments and returned from methods, enhancing modularity.
- **Method Overloading and Constructors**
- **Method Overloading**: Defining multiple methods with the same name but different parameter lists.
- **Overloading Constructors**: Providing multiple ways to initialize an object.

Understanding Static

- Static methods and variables belong to the class rather than any object, accessible without creating an instance.

- **Nested and Inner Classes**
- **Nested Classes**: Declared inside another class, can be static or non-static.
- **Inner Classes**: Non-static classes defined inside a class, have access to the outer class's members.

Conclusion

- Java’s data types, operators, classes, and methods form the foundation of efficient programming.
- Mastering advanced features like nested classes, recursion, and method overloading enables writing clean, modular, and reusable code.
- Understanding these concepts is crucial for solving real-world problems with Java.

Inheritance

- **Basics of Inheritance**
- **Definition**: Inheritance allows a class (subclass) to acquire properties and methods from another class (superclass).
- Promotes **code reusability** and supports **hierarchical classification**.
- Syntax: `class Subclass extends Superclass`.
- **Member Access and Inheritance**
- Subclasses inherit public and protected members of the superclass.
- Private members are not directly accessible, but can be accessed via public/protected methods.

Constructors and Inheritance

- **Constructors**: Not inherited but can be invoked using the `super` keyword.
- **Using `super`**:
- **To Call Superclass Constructors**: Ensures proper initialization of superclass members.
- **To Access Superclass Members**: Resolves naming conflicts between superclass and subclass.

Advanced Features

- **Abstract Classes**:
- Classes declared with `abstract` keyword cannot be instantiated.
- Used to define a common base with some methods implemented and others left abstract.
- **Using `final`**:
- Prevents a class from being extended or a method from being overridden.
- **The `Object` Class**:
- The root class of all Java classes, providing methods like `toString()`, `equals()`, and `hashCode()`.

Interfaces

- **Interface Fundamentals**
- **Definition**: Interfaces define a contract for classes to implement, specifying method signatures without implementation.
- Syntax: `interface InterfaceName {}`.
- **Creating and Implementing Interfaces**
- **Creating an Interface**: Define method signatures and constants.
- **Implementing an Interface**: A class uses `implements` to adhere to an interface's contract.

Advanced Features of Interfaces

- **Using Interface References**: Interfaces can be used to reference objects of classes that implement them.
- **Implementing Multiple Interfaces**: Java allows a class to implement multiple interfaces, promoting flexibility.
- **Constants in Interfaces**: Variables in interfaces are implicitly `public`, `static`, and `final`.
- **Extending Interfaces**: Interfaces can inherit from other interfaces.

Packages

- **Package Fundamentals**
- Packages are namespaces for organizing classes and interfaces, preventing naming conflicts.
- Syntax: `package packageName;`.
- **Packages and Member Access**
- **Default Access**: Members are accessible only within the same package.
- **Public Access**: Members are accessible across packages.

Importing Packages

- Use `import packageName.*;` to include a package.
- Example: `import java.util.*;`.

- **Static Import**
- Allows importing static members of a class directly, removing the need for class qualification.
- Syntax: `import static packageName.ClassName.staticMember;`.

Conclusion

- **Inheritance**: Enables code reuse, supports polymorphism, and introduces concepts like `super` and abstract classes.
- **Interfaces**: Provide a contract for classes, supporting abstraction and multiple inheritance.
- **Packages**: Offer an efficient way to organize and access Java classes, improving code maintainability.

Exception Handling

- **. The Exception Hierarchy**
- All exceptions in Java are derived from the `Throwable` class.
  - **Checked Exceptions**: Must be declared in a method's `throws` clause (e.g., `IOException`).
  - **Unchecked Exceptions**: Do not require declaration (e.g., `ArithmeticException`).
  - Subclasses: `Error` (fatal issues) and `Exception` (recoverable issues).

Exception Handling Fundamentals

- **Try-Catch Mechanism:**
- Use `try` to enclose code that may throw an exception.
- Use `catch` to handle specific exceptions.

- **Consequences of an Uncaught Exception**
- If an exception is not caught, it propagates up the call stack.
- The program terminates abruptly if no handler is found.

Using Multiple `catch` Clauses

- Each `catch` handles a specific exception type.
- **Example:**
- try {
- // risky code
- } catch (IOException e) {
- // handle IO exception
- } catch (Exception e) {
- // handle general exception
- }

Multithreaded Programming

- **Multithreading Fundamentals**
- **Definition**: Multithreading allows concurrent execution of two or more threads.
- **Benefits**: Improved performance, better resource utilization, and responsiveness.
- **The `Thread` Class and `Runnable` Interface**
- Two ways to create threads:
- Extend the `Thread` class.
- Implement the `Runnable` interface

Creating Threads

- Using Thread Class:
- class MyThread extends Thread {
- public void run() {
- System.out.println("Thread is running.");
- }
- }
- MyThread t = new MyThread();
- t.start();

Using Runnable Interface

- class MyRunnable implements Runnable {
- public void run() {
- System.out.println("Runnable thread is running.");
- }
- }
- Thread t = new Thread(new MyRunnable());
- t.start();

Synchronization

- Ensures thread safety when accessing shared resources.
- **Synchronization Methods**: Use `synchronized` to lock critical sections.
- **Synchronized Statement**: Synchronize specific blocks of code
- synchronized(obj) {
- // critical section
- }

Thread Communication

- **Methods**: `wait()`, `notify()`, and `notifyAll()` enable inter-thread communication.
- synchronized(obj) {
- obj.wait();
- obj.notify();
- }

Conclusion

- **Exception Handling**: Enables robust error management and graceful program recovery.
- **Multithreading**: Allows simultaneous task execution, improving application performance and responsiveness.

Applets

- Applet Basics
- **Definition**: Applets are small Java programs embedded in a web page and run inside a browser or applet viewer.
- **Execution Environment**: Requires Java-enabled browsers or an applet viewer for execution.
- **Restrictions**: Applets run in a secure sandbox with limited access to system resources for safety.

A Complete Applet Skeleton

- A basic applet structure contains the following methods:
- `init()`: Initializes the applet.
- `start()`: Starts or resumes the applet.
- `stop()`: Pauses the applet.
- `destroy()`: Cleans up resources before the applet is terminated.

A Complete Applet Skeleton

```
import java.applet.*;
import java.awt.*;

public class SimpleApplet extends Applet {
    public void init() {
        // Initialization code
    }
    public void paint(Graphics g) {
        g.drawString("Hello, Applet!", 20, 20);
    }
}
```

- **Applet Initialization and Termination**
- **`init()`**: Used for one-time initialization, like setting up UI components.
- **`destroy()`**: Called to release resources before the applet shuts down.
- **Key Aspect of Applet Architecture**
- Applets rely on lifecycle methods (`init`, `start`, `stop`, `destroy`) and a graphical user interface framework to interact with users.
- **Requesting Repainting**
- Use the `repaint()` method to request the applet to redraw its content.
- The `paint(Graphics g)` method handles the actual drawing.

Passing Parameters to Applets

- Parameters can be passed using <PARAM> tags in the HTML code.
- html
- Copy code
- <applet code="SimpleApplet.class" width="300" height="200">
- <param name="message" value="Hello, Parameterized Applet!">
- </applet>
-
- Retrieve parameters using the getParameter(String name) method.

Event Handling

- **Two Event Handling Mechanisms**
- **Old Event Model**: Pre-Java 1.1; used `handleEvent()` for event processing (now obsolete).
- **Delegation Event Model**: Introduced in Java 1.1; separates event sources and event listeners for cleaner and more modular code

Event Classes

- **`ActionEvent`**: Represents actions like button clicks.
- **`AdjustmentEvent`**: Represents adjustments to adjustable components like scrollbars.
- **`ComponentEvent`**: Represents changes in component visibility or size.
- **`ContainerEvent`**: Represents changes in a container, such as adding or removing components.
- **`FocusEvent`**: Represents focus gained or lost by a component.

- **`InputEvent`**: Base class for keyboard and mouse events.
- **`ItemEvent`**: Represents state changes in checkboxes or other items.
- **`KeyEvent`**: Represents keyboard actions.
- **`MouseEvent`**: Represents mouse clicks, movements, and drags.
- **`MouseWheelEvent`**: Represents mouse wheel rotation.
- **`TextEvent`**: Represents changes in a text area or text field.
- **`WindowEvent`**: Represents actions on windows, such as opening or closing.

Handling Events

- Implement event listeners and override methods to handle specific events.
- Example of handling an action event:

```
import java.awt.*;
import java.awt.event.*;

public class ButtonExample extends Frame implements ActionListener {
    Button b;

    public ButtonExample() {
        b = new Button("Click Me");
        b.addActionListener(this);
        add(b);
        setSize(200, 200);
        setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
        System.out.println("Button clicked!");
    }
}
```

Adapter Classes

- Simplify event handling by overriding only the required methods.
- Example:
- java
- Copy code
- class MyMouseAdapter extends MouseAdapter {
-     public void mousePressed(MouseEvent e) {
-         System.out.println("Mouse pressed!");
-     }
- }
-

```
class MyMouseAdapter extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
        System.out.println("Mouse pressed!");
    }
}
```

Inner Classes and Anonymous Inner Classes

- Inner classes can be used for event handling.
- Anonymous inner classes are useful for compact code.
- java
- Copy code
- button.addActionListener(new ActionListener() {
-     public void actionPerformed(ActionEvent e) {
-         System.out.println("Button clicked!");
-     }
- });

Conclusion

- **Applets**: Provide a platform for lightweight, browser-based Java programs with lifecycle methods.
- **Event Handling**: Offers a robust mechanism for responding to user interactions, enabling the creation of interactive applications.

THANK YOU

FOR
YOUR

ATTENTION