# oops in java

Target Audience: Undergraduate students

## Module Outline

```markdown

# OOPs in Java Curriculum Outline (8 Weeks)

**Course Title:** Object-Oriented Programming in Java

**Target Audience:** Undergraduate Students

**Course Duration:** 8 Weeks

---

### Module 1: Foundations of OOP & Java Basics

#### Week 1: Introduction to Object-Oriented Programming and Java Setup

* \*\*Title:\*Introduction to OOP and Java Environment
* \*\*Topics:\*\*
* \*\*Paradigm Shift:\*Understanding the transition from Procedural to Object-Oriented Programming (OOP).
* \*\*Key OOP Concepts Overview:\*High-level introduction to Abstraction, Encapsulation, Inheritance, and Polymorphism.
* \*\*Java Fundamentals:\*History, Features (Platform Independence - WORA), Java Virtual Machine (JVM), Java Runtime Environment (JRE), Java Development Kit (JDK).
* \*\*Development Environment Setup:\*Installing JDK, configuring PATH, introduction to Integrated Development Environments (IDEs) like IntelliJ IDEA or Eclipse.
* \*\*Basic Java Syntax:\*Data types (primitives), Variables, Operators, Input/Output (`System.out.println`), Control Flow Statements (`if-else`, `switch`, `for`, `while`, `do-while`).
* \*\*First Program:\*Writing and executing "Hello World!".
* \*\*Learning Outcomes:\*\*
* Differentiate between procedural and object-oriented programming paradigms.
* Identify and briefly explain the four core pillars of OOP.
* Successfully set up a Java development environment.
* Write, compile, and execute basic Java programs using fundamental syntax and control flow.

#### Week 2: Building Blocks: Classes, Objects, and Basic Encapsulation

* \*\*Title:\*Classes, Objects, and Basic Encapsulation
* \*\*Topics:\*\*
* \*\*Classes and Objects:\*Defining classes (blueprints) with attributes (fields/instance variables) and behaviors (methods).
* \*\*Object Instantiation:\*Creating objects from classes using the `new` keyword.
* \*\*Instance vs. Static Members:\*Understanding the difference and appropriate usage of instance variables/methods and static variables/methods.
* \*\*The `this` Keyword:\*Referencing the current object's instance members.
* \*\*Basic Encapsulation:\*Introduction to access modifiers (`public`, `private`) and their role in data hiding.
* \*\*Getters and Setters:\*Implementing accessor (getter) and mutator (setter) methods for controlled access to private data.
* \*\*Packages:\*Organizing classes and managing namespaces.
* \*\*Learning Outcomes:\*\*
* Design and implement classes with appropriate attributes and methods.
* Create and manipulate objects effectively.
* Distinguish between instance and static members and use them correctly.
* Apply basic encapsulation principles using `private` fields and `public` getters/setters.
* Structure Java code logically using packages.

---

### Module 2: Core OOP Principles I - Encapsulation & Inheritance

#### Week 3: Mastering Encapsulation and Object Initialization

* \*\*Title:\*Deep Dive into Encapsulation and Object Initialization
* \*\*Topics:\*\*
* \*\*Constructors:\*Default constructors, parameterized constructors, and constructor overloading.
* \*\*Constructor Chaining:\*Using `this()` to call other constructors within the same class.
* \*\*The `final` Keyword:\*Applying `final` to variables (constants), methods (preventing overriding), and classes (preventing inheritance).
* \*\*Garbage Collection:\*Overview of automatic memory management in Java and the role of the Garbage Collector.
* \*\*Object Class Methods (Basic):\*Overriding `toString()` for meaningful object representation. Introduction to `equals()` and `hashCode()`.
* \*\*Learning Outcomes:\*\*
* Implement various types of constructors for robust object initialization.
* Utilize constructor chaining to reduce code duplication.
* Apply the `final` keyword appropriately for immutability and design control.
* Understand the fundamental concept of Java's automatic garbage collection.
* Override `toString()` to provide custom string representations of objects.

#### Week 4: Inheritance: Reusability and Specialization

* \*\*Title:\*Inheritance: Reusability and Specialization
* \*\*Topics:\*\*
* \*\*Introduction to Inheritance:\*The `extends` keyword and establishing "is-a" relationships.
* \*\*Parent (Super) Class and Child (Sub) Class:\*Understanding the hierarchy.
* \*\*Types of Inheritance (Java Support):\*Single, Multilevel, and Hierarchical inheritance through classes.
* \*\*The `super` Keyword:\*Accessing parent class members and invoking parent class constructors.
* \*\*Method Overriding:\*Redefining an inherited method in the child class; rules and the `@Override` annotation.
* \*\*Preventing Inheritance:\*Using `final` with classes.
* \*\*Learning Outcomes:\*\*
* Apply inheritance to achieve code reusability and model hierarchical relationships.
* Implement single, multilevel, and hierarchical inheritance structures.
* Correctly use the `super` keyword to interact with superclass members and constructors.
* Override methods effectively to provide specialized behavior in subclasses.

---

### Module 3: Core OOP Principles II - Polymorphism & Abstraction

#### Week 5: Polymorphism: Many Forms, One Interface

* \*\*Title:\*Polymorphism: Many Forms, One Interface
* \*\*Topics:\*\*
* \*\*Introduction to Polymorphism:\*The concept of "many forms."
* \*\*Compile-time Polymorphism (Static Polymorphism):\*Method Overloading – defining multiple methods with the same name but different parameters.
* \*\*Run-time Polymorphism (Dynamic Polymorphism):\*Method Overriding – dynamic method dispatch.
* \*\*Upcasting and Downcasting:\*Safely casting objects (with `instanceof` operator).
* \*\*Dynamic Method Dispatch:\*How the JVM determines which overridden method to call at runtime.
* \*\*Covariant Return Types:\*Allowing overridden methods to return a more specific type than the overridden method.
* \*\*Learning Outcomes:\*\*
* Explain the principle of polymorphism in OOP.
* Implement method overloading to provide multiple functionalities for a single method name.
* Implement method overriding to achieve runtime polymorphism and specific subclass behaviors.
* Perform safe upcasting and downcasting using the `instanceof` operator.
* Understand how dynamic method dispatch enables runtime polymorphism.

#### Week 6: Abstraction: Hiding Complexity, Revealing Essentials

* \*\*Title:\*Abstraction: Hiding Complexity, Revealing Essentials
* \*\*Topics:\*\*
* \*\*Introduction to Abstraction:\*The concept of hiding implementation details and showing only essential features.
* \*\*Abstract Classes:\*Defining abstract methods and abstract classes using the `abstract` keyword; rules and usage.
* \*\*Interfaces:\*Defining contracts using the `interface` keyword; implementing interfaces (`implements` keyword).
* \*\*Multiple Inheritance in Java:\*Achieved through interfaces.
* \*\*Default and Static Methods in Interfaces (Java 8+):\*Providing concrete implementations in interfaces.
* \*\*Functional Interfaces and Lambda Expressions (Basic):\*A brief introduction to simplify code for single-method interfaces.
* \*\*Comparing Abstract Classes vs. Interfaces:\*When to use which.
* \*\*Learning Outcomes:\*\*
* Understand and apply the principle of abstraction in Java programming.
* Design and implement abstract classes to provide a common base for related classes.
* Define and implement interfaces to enforce contracts and achieve type-based multiple inheritance.
* Differentiate between abstract classes and interfaces and select the appropriate construct for different design scenarios.
* Gain a basic understanding of functional interfaces and lambda expressions.

---

### Module 4: Advanced OOP Concepts & Best Practices

#### Week 7: Robustness and Data Structures: Exception Handling and Collections

* \*\*Title:\*Robustness and Data Structures: Exception Handling and Collections
* \*\*Topics:\*\*
* \*\*Exception Handling Fundamentals:\*Understanding errors versus exceptions, the exception hierarchy (`Throwable`, `Exception`, `RuntimeException`).
* \*\*Checked vs. Unchecked Exceptions:\*Differentiating and handling them.
* \*\*`try-catch-finally` Blocks:\*Catching and managing exceptions gracefully.
* \*\*`throw` and `throws` Keywords:\*Throwing exceptions and declaring checked exceptions.
* \*\*Custom Exceptions:\*Creating user-defined exception classes.
* \*\*Introduction to Java Collections Framework:\*Overview of the framework's purpose.
* \*\*Core Interfaces:\*`List`, `Set`, `Map`.
* \*\*Basic Implementations:\*`ArrayList`, `HashSet`, `HashMap` – basic usage and operations.
* \*\*Learning Outcomes:\*\*
* Implement robust code using `try-catch-finally` blocks to handle exceptions effectively.
* Differentiate between checked and unchecked exceptions and apply appropriate handling strategies.
* Design and implement custom exception classes.
* Understand the fundamental purpose and structure of the Java Collections Framework.
* Utilize basic implementations of `List`, `Set`, and `Map` to store and manage collections of objects.

#### Week 8: Applying OOP: Design Principles and Practical Project

* \*\*Title:\*Applying OOP: Design Principles and Practical Project
* \*\*Topics:\*\*
* \*\*Review of All OOP Pillars:\*Consolidating understanding of Abstraction, Encapsulation, Inheritance, and Polymorphism.
* \*\*Introduction to SOLID Principles:\*\*
* \*\*S\*\*ingle Responsibility Principle (SRP)
* \*\*O\*\*pen/Closed Principle (OCP)
* \*\*L\*\*iskov Substitution Principle (LSP)
* \*\*I\*\*nterface Segregation Principle (ISP)
* \*\*D\*\*ependency Inversion Principle (DIP)
* \*\*Common Design Patterns (Brief Overview):\*Singleton, Factory Method (examples and use cases).
* \*\*Refactoring for OOP Principles:\*Identifying and improving code that violates OOP or SOLID principles.
* \*\*Mini-Project/Case Study:\*Applying all learned OOP concepts to design and implement a small, practical Java application.
* \*\*Learning Outcomes:\*\*
* Articulate and apply all four core OOP principles in software design.
* Explain and apply the SOLID principles to develop more maintainable, flexible, and scalable Java applications.
* Recognize and briefly describe the purpose of common design patterns.
* Refactor existing code to adhere better to OOP and design principles.
* Develop a small-scale Java application demonstrating a comprehensive understanding and application of OOP concepts.

```

## Learning Material

Here is a detailed list of learning materials for the 'OOPs in Java' curriculum, structured week-by-week:

```markdown

# Learning Materials: OOPs in Java Curriculum

---

### Module 1: Foundations of OOP & Java Basics

#### Week 1: Introduction to Object-Oriented Programming and Java Setup

* \*\*Lecture Notes:\*\*
* \*\*"From Procedural to OOP: A Paradigm Shift"\*\*: Explaining the motivations behind OOP, limitations of procedural programming.
* \*\*"The Four Pillars of OOP - An Overview"\*\*: High-level introduction to Abstraction, Encapsulation, Inheritance, and Polymorphism with simple analogies.
* \*\*"Java Ecosystem: JVM, JRE, JDK Explained"\*\*: Detailed notes on Java's platform independence, the role of each component.
* \*\*"Setting Up Your Java Development Environment"\*\*: Step-by-step guide for JDK installation (Windows/macOS/Linux), configuring `PATH` variable, and basic setup for IntelliJ IDEA/Eclipse.
* \*\*"Java Fundamentals: Data Types, Variables, Operators"\*\*: Comprehensive notes on primitive data types, variable declaration, naming conventions, and common operators.
* \*\*"Control Flow in Java: Decisions and Loops"\*\*: Detailed explanations and syntax for `if-else`, `switch`, `for`, `while`, `do-while` statements.
* \*\*Code Examples:\*\*
* `HelloWorld.java`: The classic first program.
* `BasicDataTypes.java`: Demonstrating all primitive data types and their ranges.
* `OperatorsDemo.java`: Examples of arithmetic, relational, logical, and assignment operators.
* `ConditionalStatements.java`: Examples of `if-else` and `switch` statements (e.g., grading system, day of the week).
* `LoopExamples.java`: Demonstrations of `for`, `while`, `do-while` loops (e.g., printing numbers, calculating sum, factorial).
* `UserInputDemo.java`: Using `Scanner` for basic console input.
* \*\*Case Studies/Exercises:\*\*
* \*\*Guided Lab\*\*: "Your First Java Program" - Walkthrough of setting up JDK, IDE, writing, compiling, and running "Hello World".
* \*\*Problem Set 1\*\*:
* Write a Java program to calculate the area of a rectangle given its length and width.
* Create a program that checks if a given number is positive, negative, or zero.
* Develop a program to print the first `N` natural numbers and their sum using a `for` loop.

#### Week 2: Building Blocks: Classes, Objects, and Basic Encapsulation

* \*\*Lecture Notes:\*\*
* \*\*"Classes as Blueprints, Objects as Instances"\*\*: In-depth explanation of class definition, attributes (fields), and behaviors (methods).
* \*\*"Object Creation and Manipulation"\*\*: Understanding the `new` keyword, dot operator for accessing members.
* \*\*"Instance vs. Static Members: When to Use Which"\*\*: Clear differentiation with examples, covering `static` variables and methods.
* \*\*"The `this` Keyword: Self-Reference in Java"\*\*: Explaining `this` for disambiguation and method chaining.
* \*\*"Introduction to Encapsulation: Data Hiding with Access Modifiers"\*\*: Focus on `public` and `private`, the concept of information hiding.
* \*\*"Getters and Setters: Controlled Data Access"\*\*: Best practices for implementing accessor and mutator methods.
* \*\*"Packages: Organizing Your Java Code"\*\*: Syntax and conventions for package declaration, `import` statements.
* \*\*Code Examples:\*\*
* `Student.java`: A class with private fields (`name`, `age`, `studentId`) and public methods (`study()`, `getDetails()`).
* `School.java`: Demonstrating creation of multiple `Student` objects and calling their methods.
* `Circle.java`: Class with `radius` (instance) and `PI` (static final constant), `calculateArea()` (instance method) and `getNumberOfCircles()` (static method).
* `ThisKeywordDemo.java`: Examples showing `this` to refer to current instance variables and for method chaining.
* `BankAccount.java`: Illustrating basic encapsulation with private `balance` and public `deposit()`, `withdraw()`, `getBalance()` methods.
* `com.example.app.model.Person` and `com.example.app.main.AppRunner`: Demonstrating package structure and import.
* \*\*Case Studies/Exercises:\*\*
* \*\*Design Exercise\*\*: "Car Class Design" - Design a `Car` class. Identify its attributes (make, model, year, color, currentSpeed) and behaviors (startEngine, accelerate, brake, displayInfo). Implement basic encapsulation with private fields and public getters/setters.
* \*\*Problem Set 2\*\*:
* Create a `Book` class with `title`, `author`, `isbn` as private attributes. Add a constructor and `displayBookInfo()` method.
* Modify the `Student` class to include a `static` variable `totalStudents` that increments every time a new `Student` object is created.
* Implement a `Rectangle` class with `length` and `width`. Include a `calculateArea()` method and ensure `length` and `width` can only be positive using setters.

---

### Module 2: Core OOP Principles I - Encapsulation & Inheritance

#### Week 3: Mastering Encapsulation and Object Initialization

* \*\*Lecture Notes:\*\*
* \*\*"Constructors: The Object Initializers"\*\*: Default, parameterized, and overloaded constructors, their purpose and syntax.
* \*\*"Constructor Chaining with `this()`"\*\*: How to reuse constructors within the same class to avoid code duplication.
* \*\*"The `final` Keyword: Constants, Immutable Methods, and Classes"\*\*: Comprehensive guide on `final` for variables, methods (preventing overriding), and classes (preventing inheritance).
* \*\*"Java's Automatic Memory Management: Garbage Collection"\*\*: Overview of the JVM's Garbage Collector, how it works, and its benefits.
* \*\*"Essential `Object` Class Methods: `toString()`, `equals()`, `hashCode()`"\*\*: Focus on overriding `toString()` for meaningful object representation. Basic introduction to `equals()` and `hashCode()` contract.
* \*\*Code Examples:\*\*
* `Product.java`: Class with multiple overloaded constructors (e.g., `Product()`, `Product(name)`, `Product(name, price)`).
* `ConstructorChainingDemo.java`: Example showing `this()` to call other constructors.
* `FinalKeywordDemo.java`: Demonstrating `final` with variables (`PI`), methods (`calculateArea` in `Shape`), and classes (`String`, custom `ImmutableClass`).
* `GarbageCollectionDemo.java`: Simple example demonstrating object eligibility for GC (though GC run is not guaranteed).
* `Person.java`: Overriding `toString()` to return a descriptive string (e.g., "Person[name=..., age=...]").
* `Point.java`: Basic implementation of `equals()` and `hashCode()` methods.
* \*\*Case Studies/Exercises:\*\*
* \*\*Refactoring Exercise\*\*: "Enhanced BankAccount" - Refactor the `BankAccount` class from Week 2 to include:
* A default constructor that initializes balance to 0.
* A parameterized constructor that sets account number and initial balance.
* A constructor that takes account holder name, account number, and initial balance, using constructor chaining.
* \*\*Problem Set 3\*\*:
* Create an `InvoiceItem` class. Ensure its `itemId` and `price` are set only once using `final` fields and a constructor.
* Design an `Address` class (`street`, `city`, `zipCode`). Override its `toString()` method to provide a formatted address string.
* Implement `equals()` and `hashCode()` for a `Course` class (comparing by `courseId` and `courseName`).

#### Week 4: Inheritance: Reusability and Specialization

* \*\*Lecture Notes:\*\*
* \*\*"Inheritance Fundamentals: The 'is-a' Relationship"\*\*: Explaining `extends` keyword, superclass and subclass concepts.
* \*\*"Types of Inheritance in Java"\*\*: Detailed discussion and diagrams for single, multilevel, and hierarchical inheritance through classes.
* \*\*"The `super` Keyword: Interacting with the Parent"\*\*: Using `super` to invoke superclass constructors and access superclass members.
* \*\*"Method Overriding: Redefining Parent Behaviors"\*\*: Rules for overriding, the `@Override` annotation, and its benefits.
* \*\*"Preventing Inheritance with `final` Class"\*\*: Revisiting `final` applied to classes.
* \*\*Code Examples:\*\*
* `Animal.java` (superclass) and `Dog.java` (subclass): Demonstrating basic inheritance, `Dog` inheriting `eat()` and adding `bark()`.
* `Vehicle.java` -> `Car.java` -> `ElectricCar.java`: Example of multilevel inheritance.
* `Shape.java` (superclass) -> `Circle.java`, `Rectangle.java`, `Triangle.java` (subclasses): Example of hierarchical inheritance.
* `SuperKeywordDemo.java`: Using `super()` in constructors and `super.methodName()` for overridden methods.
* `MethodOverridingDemo.java`: Illustrating method overriding with `@Override` annotation.
* `FinalClassExample.java`: A `final` class that cannot be extended.
* \*\*Case Studies/Exercises:\*\*
* \*\*Design Exercise\*\*: "University Hierarchy" - Design a class hierarchy for a university system. Start with a `Person` class. Derive `Student` and `Professor` classes from `Person`. `Student` might have `major` and `gpa`, `Professor` might have `department` and `salary`. Add appropriate methods and demonstrate inheritance.
* \*\*Problem Set 4\*\*:
* Create a `Shape` class with a method `getArea()`. Create subclasses `Circle` and `Rectangle` that override `getArea()` to calculate their specific areas.
* Implement a `Manager` class that `extends` `Employee`. The `Manager` class should have an additional `bonus` attribute and override a `calculateSalary()` method from `Employee` to include the bonus.

---

### Module 3: Core OOP Principles II - Polymorphism & Abstraction

#### Week 5: Polymorphism: Many Forms, One Interface

* \*\*Lecture Notes:\*\*
* \*\*"Polymorphism: The Power of Many Forms"\*\*: Introduction to the concept, its role in flexible and extensible code.
* \*\*"Compile-time Polymorphism: Method Overloading"\*\*: Detailed explanation of overloading rules (different parameter lists).
* \*\*"Run-time Polymorphism: Method Overriding and Dynamic Method Dispatch"\*\*: How the JVM determines which overridden method to call at runtime.
* \*\*"Upcasting and Downcasting: Safe Object Conversions"\*\*: Explaining implicit upcasting and explicit downcasting with the `instanceof` operator for type safety.
* \*\*"Covariant Return Types (Java 5+)"\*\*: Allowing overridden methods to return a more specific subclass type.
* \*\*Code Examples:\*\*
* `Calculator.java`: Demonstrating method overloading with `add(int, int)`, `add(double, double)`, `add(int, int, int)`.
* `AnimalSound.java`: `Animal` class with `makeSound()`. `Dog` and `Cat` override `makeSound()`. Main method demonstrates `Animal a = new Dog(); a.makeSound();`.
* `PolymorphismDemo.java`: Array of `Animal` objects, iterating and calling `makeSound()` polymorphically.
* `CastingDemo.java`: Examples of upcasting (`Dog` to `Animal`) and safe downcasting (`Animal` back to `Dog` using `instanceof`).
* `CovariantReturnExample.java`: `Shape` class with `getShape()` method returning `Shape`. `Circle` subclass overrides `getShape()` to return `Circle`.
* \*\*Case Studies/Exercises:\*\*
* \*\*Scenario\*\*: "Payment Processing System" - Design a `PaymentProcessor` class. Implement a `processPayment` method that is overloaded to handle different payment types (e.g., `processPayment(CreditCard card)`, `processPayment(PayPalAccount account)`, `processPayment(CashPayment cash)`).
* \*\*Problem Set 5\*\*:
* Create a `Vehicle` class with a `start()` method. Create `Car` and `Motorcycle` subclasses that override `start()`. In your `main` method, create an `ArrayList` of `Vehicle` objects containing `Car` and `Motorcycle` instances, then iterate and call `start()` on each.
* Write a program that takes an `Object` reference as input. If it's an instance of `String`, print its length; if it's an instance of `Integer`, print its value squared. Use `instanceof` and casting.

#### Week 6: Abstraction: Hiding Complexity, Revealing Essentials

* \*\*Lecture Notes:\*\*
* \*\*"Abstraction: Focus on 'What', Not 'How'"\*\*: Explaining the principle of abstraction in OOP, hiding implementation details.
* \*\*"Abstract Classes: Partial Implementations and Contracts"\*\*: Definition, rules for abstract methods and classes, when to use.
* \*\*"Interfaces: Pure Contracts and Multiple Inheritance"\*\*: Definition, `implements` keyword, achieving multiple inheritance through interfaces.
* \*\*"Java 8+ Enhancements: Default and Static Methods in Interfaces"\*\*: How to provide concrete implementations in interfaces, use cases.
* \*\*"Functional Interfaces and Lambda Expressions (Basic)"\*\*: A brief introduction to simplify code for single-method interfaces.
* \*\*"Abstract Class vs. Interface: A Comparative Analysis"\*\*: Detailed comparison table and guidelines for choosing between them.
* \*\*Code Examples:\*\*
* `AbstractShape.java` (abstract class with abstract `calculateArea()` and concrete `displayColor()`). `ConcreteCircle.java` and `ConcreteRectangle.java` extending `AbstractShape`.
* `Flyable.java` (interface with `fly()` method). `Bird.java` and `Airplane.java` implementing `Flyable`.
* `AmphibiousVehicle.java`: Class implementing `LandVehicle` and `WaterVehicle` interfaces.
* `MyInterface.java`: Interface with `default` and `static` methods.
* `FunctionalInterfaceDemo.java`: Simple example of a functional interface and its implementation using a lambda expression (e.g., `Runnable`, custom `Greeting`).
* \*\*Case Studies/Exercises:\*\*
* \*\*Design Exercise\*\*: "Musical Instruments" - Create an abstract class `Instrument` with an abstract method `play()`. Derive concrete classes like `Guitar` and `Piano`, each implementing `play()` differently. Create a `Musician` class that can `playInstrument(Instrument i)`.
* \*\*Problem Set 6\*\*:
* Define an interface `Printable` with a method `print()`. Create classes `Document` and `Image` that implement `Printable`.
* Design an `OnlineOrder` system. Create an interface `Processable` with a `process()` method. Create classes `PhysicalProductOrder` and `DigitalProductOrder` that implement `Processable` and have different processing logic.
* Discuss a scenario where an abstract class would be more suitable than an interface, and vice-versa, with justifications.

---

### Module 4: Advanced OOP Concepts & Best Practices

#### Week 7: Robustness and Data Structures: Exception Handling and Collections

* \*\*Lecture Notes:\*\*
* \*\*"Exception Handling Fundamentals: Errors vs. Exceptions"\*\*: Understanding the `Throwable` hierarchy (`Exception`, `RuntimeException`).
* \*\*"Checked vs. Unchecked Exceptions: When to Handle"\*\*: Differentiating between compile-time enforced vs. runtime exceptions.
* \*\*"`try-catch-finally`: Managing Exceptions Gracefully"\*\*: Detailed explanation of blocks, flow of control, and `try-with-resources`.
* \*\*"`throw` vs. `throws`: Throwing and Declaring Exceptions"\*\*: Syntax and usage for explicitly throwing exceptions and declaring checked exceptions.
* \*\*"Custom Exceptions: Tailoring Error Handling"\*\*: Steps and best practices for creating user-defined exception classes.
* \*\*"Java Collections Framework: An Overview"\*\*: Purpose, core interfaces (`List`, `Set`, `Map`), and their benefits.
* \*\*"Basic Collections Implementations: `ArrayList`, `HashSet`, `HashMap`"\*\*: Fundamental usage, common operations (add, remove, get, iterate).
* \*\*Code Examples:\*\*
* `ExceptionHandlingDemo.java`: Examples of `ArithmeticException`, `NullPointerException`, `ArrayIndexOutOfBoundsException` with `try-catch`.
* `CheckedExceptionDemo.java`: Demonstrating `IOException` with `throws` and `try-catch`.
* `FinallyBlockDemo.java`: Example showing `finally` block execution.
* `CustomException.java` and `InsufficientFundsException.java`: Creating and using a custom exception in a `BankAccount` class.
* `ArrayListExample.java`: Adding, removing, iterating `String` and custom objects.
* `HashSetExample.java`: Demonstrating storing unique elements.
* `HashMapExample.java`: Storing key-value pairs (e.g., `StudentId` to `StudentObject`).
* \*\*Case Studies/Exercises:\*\*
* \*\*Scenario\*\*: "User Authentication System" - Implement a `UserAuthenticator` class with a `login(String username, String password)` method. This method should throw a custom `InvalidCredentialsException` if the username/password combination is incorrect.
* \*\*Problem Set 7\*\*:
* Write a program that attempts to read an integer from the console. Use `try-catch` to handle `InputMismatchException` if the user enters non-integer input.
* Create a simple `Inventory` class that uses an `ArrayList` to store `Product` objects. Implement methods to add a product, remove a product by ID, and display all products.
* Modify the `Inventory` class to use a `HashMap` where product IDs are keys and `Product` objects are values. Add a method to search for a product by ID.

#### Week 8: Applying OOP: Design Principles and Practical Project

* \*\*Lecture Notes:\*\*
* \*\*"Consolidating OOP Pillars"\*\*: Comprehensive review and synthesis of Abstraction, Encapsulation, Inheritance, and Polymorphism with advanced examples.
* \*\*"SOLID Principles: Building Maintainable and Scalable Systems"\*\*: Detailed explanation of each principle (SRP, OCP, LSP, ISP, DIP) with real-world analogies and code patterns.
* \*\*"Introduction to Common Design Patterns: Singleton and Factory Method"\*\*: Brief overview of their purpose, structure, and simple use cases.
* \*\*"Refactoring for OOP Principles"\*\*: Techniques and examples for improving existing code quality by adhering to OOP and SOLID principles.
* \*\*Code Examples:\*\*
* `SRPExample.java`: Before and after refactoring a class violating Single Responsibility Principle (e.g., a `ReportGenerator` class also handling data fetching and printing).
* `OCPExample.java`: Demonstrating Open/Closed Principle with a `Shape` interface and `AreaCalculator` class that can be extended without modification.
* `LSPExample.java`: Illustrating Liskov Substitution Principle with `Rectangle` and `Square` classes.
* `ISPExample.java`: How Interface Segregation Principle leads to smaller, more specific interfaces.
* `DIPExample.java`: Dependency Inversion Principle using interfaces for dependencies (e.g., `Logger` interface and `ConsoleLogger` implementation).
* `SingletonPattern.java`: Basic implementation of the Singleton design pattern.
* `FactoryMethodPattern.java`: Simple factory for creating different types of `Vehicle` objects based on input.
* \*\*Case Studies/Exercises:\*\*
* \*\*Mini-Project\*\*: "Simple E-commerce Order Processing System"
* \*\*Requirements\*\*:
* Classes: `Product`, `Customer`, `Order`, `ShoppingCart`.
* `Product`: (id, name, price, stock) - Encapsulated.
* `Customer`: (id, name, email) - Encapsulated.
* `Order`: Contains `Customer` and a `List` of `Product`s. Calculate total, apply discounts.
* `ShoppingCart`: Add/remove products, view cart, checkout.
* \*\*OOP Principles to Apply\*\*:
* \*\*Encapsulation\*\*: All class fields private with getters/setters.
* \*\*Inheritance\*\*: Optional: Different types of products (e.g., `ElectronicsProduct`, `BookProduct`) inheriting from `Product`.
* \*\*Polymorphism\*\*: If using product types, use polymorphism when calculating total order price.
* \*\*Abstraction\*\*: An `OrderProcessor` interface or abstract class.
* \*\*Exception Handling\*\*: Handle `InsufficientStockException`, `ProductNotFoundException`.
* \*\*Collections\*\*: Use `ArrayList` for `ShoppingCart` items, `HashMap` for product catalog.
* \*\*SOLID\*\*: Design `OrderProcessor` to be open for extension but closed for modification (OCP). Ensure `Product` class follows SRP.
* \*\*Deliverables\*\*: UML class diagram, well-commented Java code, a brief report explaining how OOP and SOLID principles were applied.
* \*\*Refactoring Challenge\*\*: Given a provided "bad code" snippet (e.g., a monolithic class with many responsibilities), identify violations of SRP and OCP, and refactor it into a cleaner, more modular design.

```

## Assessments

```markdown

# OOPs in Java - Detailed Assessment Plan

### Course Title: Object-Oriented Programming in Java

### Course Duration: 8 Weeks

This assessment plan is designed to comprehensively evaluate students' understanding and application of Object-Oriented Programming (OOP) concepts in Java, as outlined in the course curriculum. A variety of assessment methods will be utilized to cater to different learning styles and to provide multiple opportunities for students to demonstrate their proficiency.

---

### 1. Grade Weighting

The final grade for the course will be calculated based on the following components and their respective weights:

* \*\*Quizzes:\*20%
* \*\*Assignments:\*40%
* \*\*Mid-Term Exam:\*20%
* \*\*Final Project:\*20%

**Total: 100%**

---

### 2. Assessment Components Details

#### 2.1. Quizzes (Total 20%)

* \*\*Purpose:\*To regularly assess understanding of fundamental concepts, syntax, and terminology covered in recent lectures and readings. They serve as quick checks of comprehension and retention.
* \*\*Format:\*Typically short, timed online quizzes consisting of multiple-choice questions, true/false, short answer, and basic code snippet analysis.
* \*\*Frequency:\*4 quizzes, approximately bi-weekly, covering the material from the preceding two weeks.
* \*\*Grading:\*Each quiz is worth 5% of the total course grade.

| Quiz | Week Covered | Topics Assessed (Examples) | Weight |

| :--- | :----------- | :------------------------- | :----- |

| **Quiz 1** | End of Week 2 | OOP vs. Procedural, Java Basics, Data Types, Control Flow, Classes, Objects, `this`, Access Modifiers, Getters/Setters. | 5% |

| **Quiz 2** | End of Week 4 | Constructors, `final` keyword, Garbage Collection basics, Inheritance (`extends`, `super`), Method Overriding. | 5% |

| **Quiz 3** | End of Week 6 | Polymorphism (Overloading/Overriding), Upcasting/Downcasting, Abstract Classes, Interfaces, Default/Static Interface Methods. | 5% |

| **Quiz 4** | End of Week 7 | Exception Hierarchy, `try-catch-finally`, `throw`/`throws`, Custom Exceptions, `List`/`Set`/`Map` basic usage. | 5% |

#### 2.2. Assignments (Total 40%)

* \*\*Purpose:\*To provide hands-on programming experience, apply theoretical knowledge to practical problems, and develop problem-solving and coding skills. Assignments require students to write and submit Java code.
* \*\*Format:\*Programming exercises that involve designing classes, implementing OOP principles, and solving specific problems. Submissions will typically include Java source code files.
* \*\*Frequency:\*4 assignments, approximately bi-weekly, increasing in complexity.
* \*\*Grading:\*Each assignment is worth 10% of the total course grade.

| Assignment | Due Week | Topics Assessed (Examples) | Weight |

| :--------- | :------- | :------------------------- | :----- |

| **Assignment 1** | End of Week 2 | Design and implement a simple class (e.g., `Book`, `Student`) with appropriate attributes, constructors, and basic encapsulation (getters/setters). Demonstrate object creation and manipulation. | 10% |

| **Assignment 2** | End of Week 4 | Create a class hierarchy using inheritance (e.g., `Shape` -> `Circle`, `Rectangle`). Implement method overriding and demonstrate `super` keyword usage. Apply constructor overloading. | 10% |

| **Assignment 3** | End of Week 6 | Design an abstract class and an interface for a given scenario (e.g., `Vehicle` abstract class, `Drivable` interface). Implement concrete classes demonstrating polymorphism through method overriding and interface implementation. | 10% |

| **Assignment 4** | End of Week 7 | Develop a small application that uses exception handling (`try-catch-finally`, custom exceptions) and integrates at least one Java Collection (e.g., `ArrayList` to manage a list of objects). | 10% |

#### 2.3. Mid-Term Exam (Total 20%)

* \*\*Purpose:\*To comprehensively assess students' understanding of the core OOP concepts and Java fundamentals covered in the first half of the course (Modules 1 and 2).
* \*\*Timing:\*Administered at the end of Week 4.
* \*\*Format:\*A timed exam, potentially including a mix of:
* Conceptual questions (definitions, comparisons, explanations).
* Code analysis (identifying errors, predicting output).
* Short coding problems (writing small classes or methods to demonstrate a concept).
* \*\*Content:\*Covers all topics from Week 1 to Week 4, including Java basics, classes, objects, encapsulation (deep dive), constructors, `final` keyword, garbage collection basics, and inheritance.

#### 2.4. Final Project (Total 20%)

* \*\*Purpose:\*To allow students to integrate and apply all learned OOP principles and best practices in the development of a more substantial, practical Java application. It demonstrates the ability to design, implement, and test a small software system.
* \*\*Timing:\*Project requirements introduced in Week 7, due at the end of Week 8.
* \*\*Format:\*Students will design and implement a small-scale Java application (e.g., a simple library management system, a basic game, an inventory system). The project will require:
* Demonstration of \*\*Abstraction\*\*, \*\*Encapsulation\*\*, \*\*Inheritance\*\*, and \*\*Polymorphism\*\*.
* Application of basic \*\*SOLID principles\*where applicable.
* Effective use of \*\*Exception Handling\*\*.
* Utilization of \*\*Java Collections Framework\*for data management.
* Clean, well-structured, and commented code.
* \*\*Deliverables:\*Source code files, a brief design document or README explaining the project structure and how OOP principles were applied, and potentially a short demonstration video or presentation.
* \*\*Evaluation Criteria:\*Correctness, adherence to OOP principles, code quality (readability, modularity, comments), robustness (exception handling), and fulfillment of project requirements.

---

This comprehensive assessment plan aims to provide a fair and thorough evaluation of students' mastery of Object-Oriented Programming in Java, progressing from foundational knowledge to practical application and robust software design.

```

## Resources

# Learning Materials: OOPs in Java Curriculum

---

### Module 1: Foundations of OOP & Java Basics

#### Week 1: Introduction to Object-Oriented Programming and Java Setup

* \*\*Lecture Notes:\*\*
* \*\*"From Procedural to OOP: A Paradigm Shift"\*\*: Explaining the motivations behind OOP, limitations of procedural programming.
* \*\*"The Four Pillars of OOP - An Overview"\*\*: High-level introduction to Abstraction, Encapsulation, Inheritance, and Polymorphism with simple analogies.
* \*\*"Java Ecosystem: JVM, JRE, JDK Explained"\*\*: Detailed notes on Java's platform independence, the role of each component.
* \*\*"Setting Up Your Java Development Environment"\*\*: Step-by-step guide for JDK installation (Windows/macOS/Linux), configuring `PATH` variable, and basic setup for IntelliJ IDEA/Eclipse.
* \*\*"Java Fundamentals: Data Types, Variables, Operators"\*\*: Comprehensive notes on primitive data types, variable declaration, naming conventions, and common operators.
* \*\*"Control Flow in Java: Decisions and Loops"\*\*: Detailed explanations and syntax for `if-else`, `switch`, `for`, `while`, `do-while` statements.
* \*\*Code Examples:\*\*
* `HelloWorld.java`: The classic first program.
* `BasicDataTypes.java`: Demonstrating all primitive data types and their ranges.
* `OperatorsDemo.java`: Examples of arithmetic, relational, logical, and assignment operators.
* `ConditionalStatements.java`: Examples of `if-else` and `switch` statements (e.g., grading system, day of the week).
* `LoopExamples.java`: Demonstrations of `for`, `while`, `do-while` loops (e.g., printing numbers, calculating sum, factorial).
* `UserInputDemo.java`: Using `Scanner` for basic console input.
* \*\*Case Studies/Exercises:\*\*
* \*\*Guided Lab\*\*: "Your First Java Program" - Walkthrough of setting up JDK, IDE, writing, compiling, and running "Hello World".
* \*\*Problem Set 1\*\*:
* Write a Java program to calculate the area of a rectangle given its length and width.
* Create a program that checks if a given number is positive, negative, or zero.
* Develop a program to print the first `N` natural numbers and their sum using a `for` loop.

#### Week 2: Building Blocks: Classes, Objects, and Basic Encapsulation

* \*\*Lecture Notes:\*\*
* \*\*"Classes as Blueprints, Objects as Instances"\*\*: In-depth explanation of class definition, attributes (fields), and behaviors (methods).
* \*\*"Object Creation and Manipulation"\*\*: Understanding the `new` keyword, dot operator for accessing members.
* \*\*"Instance vs. Static Members: When to Use Which"\*\*: Clear differentiation with examples, covering `static` variables and methods.
* \*\*"The `this` Keyword: Self-Reference in Java"\*\*: Explaining `this` for disambiguation and method chaining.
* \*\*"Introduction to Encapsulation: Data Hiding with Access Modifiers"\*\*: Focus on `public` and `private`, the concept of information hiding.
* \*\*"Getters and Setters: Controlled Data Access"\*\*: Best practices for implementing accessor and mutator methods.
* \*\*"Packages: Organizing Your Java Code"\*\*: Syntax and conventions for package declaration, `import` statements.
* \*\*Code Examples:\*\*
* `Student.java`: A class with private fields (`name`, `age`, `studentId`) and public methods (`study()`, `getDetails()`).
* `School.java`: Demonstrating creation of multiple `Student` objects and calling their methods.
* `Circle.java`: Class with `radius` (instance) and `PI` (static final constant), `calculateArea()` (instance method) and `getNumberOfCircles()` (static method).
* `ThisKeywordDemo.java`: Examples showing `this` to refer to current instance variables and for method chaining.
* `BankAccount.java`: Illustrating basic encapsulation with private `balance` and public `deposit()`, `withdraw()`, `getBalance()` methods.
* `com.example.app.model.Person` and `com.example.app.main.AppRunner`: Demonstrating package structure and import.
* \*\*Case Studies/Exercises:\*\*
* \*\*Design Exercise\*\*: "Car Class Design" - Design a `Car` class. Identify its attributes (make, model, year, color, currentSpeed) and behaviors (startEngine, accelerate, brake, displayInfo). Implement basic encapsulation with private fields and public getters/setters.
* \*\*Problem Set 2\*\*:
* Create a `Book` class with `title`, `author`, `isbn` as private attributes. Add a constructor and `displayBookInfo()` method.
* Modify the `Student` class to include a `static` variable `totalStudents` that increments every time a new `Student` object is created.
* Implement a `Rectangle` class with `length` and `width`. Include a `calculateArea()` method and ensure `length` and `width` can only be positive using setters.

---

### Module 2: Core OOP Principles I - Encapsulation & Inheritance

#### Week 3: Mastering Encapsulation and Object Initialization

* \*\*Lecture Notes:\*\*
* \*\*"Constructors: The Object Initializers"\*\*: Default, parameterized, and overloaded constructors, their purpose and syntax.
* \*\*"Constructor Chaining with `this()`"\*\*: How to reuse constructors within the same class to avoid code duplication.
* \*\*"The `final` Keyword: Constants, Immutable Methods, and Classes"\*\*: Comprehensive guide on `final` for variables, methods (preventing overriding), and classes (preventing inheritance).
* \*\*"Java's Automatic Memory Management: Garbage Collection"\*\*: Overview of the JVM's Garbage Collector, how it works, and its benefits.
* \*\*"Essential `Object` Class Methods: `toString()`, `equals()`, `hashCode()`"\*\*: Focus on overriding `toString()` for meaningful object representation. Basic introduction to `equals()` and `hashCode()` contract.
* \*\*Code Examples:\*\*
* `Product.java`: Class with multiple overloaded constructors (e.g., `Product()`, `Product(name)`, `Product(name, price)`).
* `ConstructorChainingDemo.java`: Example showing `this()` to call other constructors.
* `FinalKeywordDemo.java`: Demonstrating `final` with variables (`PI`), methods (`calculateArea` in `Shape`), and classes (`String`, custom `ImmutableClass`).
* `GarbageCollectionDemo.java`: Simple example demonstrating object eligibility for GC (though GC run is not guaranteed).
* `Person.java`: Overriding `toString()` to return a descriptive string (e.g., "Person[name=..., age=...]").
* `Point.java`: Basic implementation of `equals()` and `hashCode()` methods.
* \*\*Case Studies/Exercises:\*\*
* \*\*Refactoring Exercise\*\*: "Enhanced BankAccount" - Refactor the `BankAccount` class from Week 2 to include:
* A default constructor that initializes balance to 0.
* A parameterized constructor that sets account number and initial balance.
* A constructor that takes account holder name, account number, and initial balance, using constructor chaining.
* \*\*Problem Set 3\*\*:
* Create an `InvoiceItem` class. Ensure its `itemId` and `price` are set only once using `final` fields and a constructor.
* Design an `Address` class (`street`, `city`, `zipCode`). Override its `toString()` method to provide a formatted address string.
* Implement `equals()` and `hashCode()` for a `Course` class (comparing by `courseId` and `courseName`).

#### Week 4: Inheritance: Reusability and Specialization

* \*\*Lecture Notes:\*\*
* \*\*"Inheritance Fundamentals: The 'is-a' Relationship"\*\*: Explaining `extends` keyword, superclass and subclass concepts.
* \*\*"Types of Inheritance in Java"\*\*: Detailed discussion and diagrams for single, multilevel, and hierarchical inheritance through classes.
* \*\*"The `super` Keyword: Interacting with the Parent"\*\*: Using `super` to invoke superclass constructors and access superclass members.
* \*\*"Method Overriding: Redefining Parent Behaviors"\*\*: Rules for overriding, the `@Override` annotation, and its benefits.
* \*\*"Preventing Inheritance with `final` Class"\*\*: Revisiting `final` applied to classes.
* \*\*Code Examples:\*\*
* `Animal.java` (superclass) and `Dog.java` (subclass): Demonstrating basic inheritance, `Dog` inheriting `eat()` and adding `bark()`.
* `Vehicle.java` -> `Car.java` -> `ElectricCar.java`: Example of multilevel inheritance.
* `Shape.java` (superclass) -> `Circle.java`, `Rectangle.java`, `Triangle.java` (subclasses): Example of hierarchical inheritance.
* `SuperKeywordDemo.java`: Using `super()` in constructors and `super.methodName()` for overridden methods.
* `MethodOverridingDemo.java`: Illustrating method overriding with `@Override` annotation.
* `FinalClassExample.java`: A `final` class that cannot be extended.
* \*\*Case Studies/Exercises:\*\*
* \*\*Design Exercise\*\*: "University Hierarchy" - Design a class hierarchy for a university system. Start with a `Person` class. Derive `Student` and `Professor` classes from `Person`. `Student` might have `major` and `gpa`, `Professor` might have `department` and `salary`. Add appropriate methods and demonstrate inheritance.
* \*\*Problem Set 4\*\*:
* Create a `Shape` class with a method `getArea()`. Create subclasses `Circle` and `Rectangle` that override `getArea()` to calculate their specific areas.
* Implement a `Manager` class that `extends` `Employee`. The `Manager` class should have an additional `bonus` attribute and override a `calculateSalary()` method from `Employee` to include the bonus.

---

### Module 3: Core OOP Principles II - Polymorphism & Abstraction

#### Week 5: Polymorphism: Many Forms, One Interface

* \*\*Lecture Notes:\*\*
* \*\*"Polymorphism: The Power of Many Forms"\*\*: Introduction to the concept, its role in flexible and extensible code.
* \*\*"Compile-time Polymorphism: Method Overloading"\*\*: Detailed explanation of overloading rules (different parameter lists).
* \*\*"Run-time Polymorphism: Method Overriding and Dynamic Method Dispatch"\*\*: How the JVM determines which overridden method to call at runtime.
* \*\*"Upcasting and Downcasting: Safe Object Conversions"\*\*: Explaining implicit upcasting and explicit downcasting with the `instanceof` operator for type safety.
* \*\*"Covariant Return Types (Java 5+)"\*\*: Allowing overridden methods to return a more specific subclass type.
* \*\*Code Examples:\*\*
* `Calculator.java`: Demonstrating method overloading with `add(int, int)`, `add(double, double)`, `add(int, int, int)`.
* `AnimalSound.java`: `Animal` class with `makeSound()`. `Dog` and `Cat` override `makeSound()`. Main method demonstrates `Animal a = new Dog(); a.makeSound();`.
* `PolymorphismDemo.java`: Array of `Animal` objects, iterating and calling `makeSound()` polymorphically.
* `CastingDemo.java`: Examples of upcasting (`Dog` to `Animal`) and safe downcasting (`Animal` back to `Dog` using `instanceof`).
* `CovariantReturnExample.java`: `Shape` class with `getShape()` method returning `Shape`. `Circle` subclass overrides `getShape()` to return `Circle`.
* \*\*Case Studies/Exercises:\*\*
* \*\*Scenario\*\*: "Payment Processing System" - Design a `PaymentProcessor` class. Implement a `processPayment` method that is overloaded to handle different payment types (e.g., `processPayment(CreditCard card)`, `processPayment(PayPalAccount account)`, `processPayment(CashPayment cash)`).
* \*\*Problem Set 5\*\*:
* Create a `Vehicle` class with a `start()` method. Create `Car` and `Motorcycle` subclasses that override `start()`. In your `main` method, create an `ArrayList` of `Vehicle` objects containing `Car` and `Motorcycle` instances, then iterate and call `start()` on each.
* Write a program that takes an `Object` reference as input. If it's an instance of `String`, print its length; if it's an instance of `Integer`, print its value squared. Use `instanceof` and casting.

#### Week 6: Abstraction: Hiding Complexity, Revealing Essentials

* \*\*Lecture Notes:\*\*
* \*\*"Abstraction: Focus on 'What', Not 'How'"\*\*: Explaining the principle of abstraction in OOP, hiding implementation details.
* \*\*"Abstract Classes: Partial Implementations and Contracts"\*\*: Definition, rules for abstract methods and classes, when to use.
* \*\*"Interfaces: Pure Contracts and Multiple Inheritance"\*\*: Definition, `implements` keyword, achieving multiple inheritance through interfaces.
* \*\*"Java 8+ Enhancements: Default and Static Methods in Interfaces"\*\*: How to provide concrete implementations in interfaces, use cases.
* \*\*"Functional Interfaces and Lambda Expressions (Basic)"\*\*: A brief introduction to simplify code for single-method interfaces.
* \*\*"Abstract Class vs. Interface: A Comparative Analysis"\*\*: Detailed comparison table and guidelines for choosing between them.
* \*\*Code Examples:\*\*
* `AbstractShape.java` (abstract class with abstract `calculateArea()` and concrete `displayColor()`). `ConcreteCircle.java` and `ConcreteRectangle.java` extending `AbstractShape`.
* `Flyable.java` (interface with `fly()` method). `Bird.java` and `Airplane.java` implementing `Flyable`.
* `AmphibiousVehicle.java`: Class implementing `LandVehicle` and `WaterVehicle` interfaces.
* `MyInterface.java`: Interface with `default` and `static` methods.
* `FunctionalInterfaceDemo.java`: Simple example of a functional interface and its implementation using a lambda expression (e.g., `Runnable`, custom `Greeting`).
* \*\*Case Studies/Exercises:\*\*
* \*\*Design Exercise\*\*: "Musical Instruments" - Create an abstract class `Instrument` with an abstract method `play()`. Derive concrete classes like `Guitar` and `Piano`, each implementing `play()` differently. Create a `Musician` class that can `playInstrument(Instrument i)`.
* \*\*Problem Set 6\*\*:
* Define an interface `Printable` with a method `print()`. Create classes `Document` and `Image` that implement `Printable`.
* Design an `OnlineOrder` system. Create an interface `Processable` with a `process()` method. Create classes `PhysicalProductOrder` and `DigitalProductOrder` that implement `Processable` and have different processing logic.
* Discuss a scenario where an abstract class would be more suitable than an interface, and vice-versa, with justifications.

---

### Module 4: Advanced OOP Concepts & Best Practices

#### Week 7: Robustness and Data Structures: Exception Handling and Collections

* \*\*Lecture Notes:\*\*
* \*\*"Exception Handling Fundamentals: Errors vs. Exceptions"\*\*: Understanding the `Throwable` hierarchy (`Exception`, `RuntimeException`).
* \*\*"Checked vs. Unchecked Exceptions: When to Handle"\*\*: Differentiating between compile-time enforced vs. runtime exceptions.
* \*\*"`try-catch-finally`: Managing Exceptions Gracefully"\*\*: Detailed explanation of blocks, flow of control, and `try-with-resources`.
* \*\*"`throw` vs. `throws`: Throwing and Declaring Exceptions"\*\*: Syntax and usage for explicitly throwing exceptions and declaring checked exceptions.
* \*\*"Custom Exceptions: Tailoring Error Handling"\*\*: Steps and best practices for creating user-defined exception classes.
* \*\*"Java Collections Framework: An Overview"\*\*: Purpose, core interfaces (`List`, `Set`, `Map`), and their benefits.
* \*\*"Basic Collections Implementations: `ArrayList`, `HashSet`, `HashMap`"\*\*: Fundamental usage, common operations (add, remove, get, iterate).
* \*\*Code Examples:\*\*
* `ExceptionHandlingDemo.java`: Examples of `ArithmeticException`, `NullPointerException`, `ArrayIndexOutOfBoundsException` with `try-catch`.
* `CheckedExceptionDemo.java`: Demonstrating `IOException` with `throws` and `try-catch`.
* `FinallyBlockDemo.java`: Example showing `finally` block execution.
* `CustomException.java` and `InsufficientFundsException.java`: Creating and using a custom exception in a `BankAccount` class.
* `ArrayListExample.java`: Adding, removing, iterating `String` and custom objects.
* `HashSetExample.java`: Demonstrating storing unique elements.
* `HashMapExample.java`: Storing key-value pairs (e.g., `StudentId` to `StudentObject`).
* \*\*Case Studies/Exercises:\*\*
* \*\*Scenario\*\*: "User Authentication System" - Implement a `UserAuthenticator` class with a `login(String username, String password)` method. This method should throw a custom `InvalidCredentialsException` if the username/password combination is incorrect.
* \*\*Problem Set 7\*\*:
* Write a program that attempts to read an integer from the console. Use `try-catch` to handle `InputMismatchException` if the user enters non-integer input.
* Create a simple `Inventory` class that uses an `ArrayList` to store `Product` objects. Implement methods to add a product, remove a product by ID, and display all products.
* Modify the `Inventory` class to use a `HashMap` where product IDs are keys and `Product` objects are values. Add a method to search for a product by ID.

#### Week 8: Applying OOP: Design Principles and Practical Project

* \*\*Lecture Notes:\*\*
* \*\*"Consolidating OOP Pillars"\*\*: Comprehensive review and synthesis of Abstraction, Encapsulation, Inheritance, and Polymorphism with advanced examples.
* \*\*"SOLID Principles: Building Maintainable and Scalable Systems"\*\*: Detailed explanation of each principle (SRP, OCP, LSP, ISP, DIP) with real-world analogies and code patterns.
* \*\*"Introduction to Common Design Patterns: Singleton and Factory Method"\*\*: Brief overview of their purpose, structure, and simple use cases.
* \*\*"Refactoring for OOP Principles"\*\*: Techniques and examples for improving existing code quality by adhering to OOP and SOLID principles.
* \*\*Code Examples:\*\*
* `SRPExample.java`: Before and after refactoring a class violating Single Responsibility Principle (e.g., a `ReportGenerator` class also handling data fetching and printing).
* `OCPExample.java`: Demonstrating Open/Closed Principle with a `Shape` interface and `AreaCalculator` class that can be extended without modification.
* `LSPExample.java`: Illustrating Liskov Substitution Principle with `Rectangle` and `Square` classes.
* `ISPExample.java`: How Interface Segregation Principle leads to smaller, more specific interfaces.
* `DIPExample.java`: Dependency Inversion Principle using interfaces for dependencies (e.g., `Logger` interface and `ConsoleLogger` implementation).
* `SingletonPattern.java`: Basic implementation of the Singleton design pattern.
* `FactoryMethodPattern.java`: Simple factory for creating different types of `Vehicle` objects based on input.
* \*\*Case Studies/Exercises:\*\*
* \*\*Mini-Project\*\*: "Simple E-commerce Order Processing System"
* \*\*Requirements\*\*:
* Classes: `Product`, `Customer`, `Order`, `ShoppingCart`.
* `Product`: (id, name, price, stock) - Encapsulated.
* `Customer`: (id, name, email) - Encapsulated.
* `Order`: Contains `Customer` and a `List` of `Product`s. Calculate total, apply discounts.
* `ShoppingCart`: Add/remove products, view cart, checkout.
* \*\*OOP Principles to Apply\*\*:
* \*\*Encapsulation\*\*: All class fields private with getters/setters.
* \*\*Inheritance\*\*: Optional: Different types of products (e.g., `ElectronicsProduct`, `BookProduct`) inheriting from `Product`.
* \*\*Polymorphism\*\*: If using product types, use polymorphism when calculating total order price.
* \*\*Abstraction\*\*: An `OrderProcessor` interface or abstract class.
* \*\*Exception Handling\*\*: Handle `InsufficientStockException`, `ProductNotFoundException`.
* \*\*Collections\*\*: Use `ArrayList` for `ShoppingCart` items, `HashMap` for product catalog.
* \*\*SOLID\*\*: Design `OrderProcessor` to be open for extension but closed for modification (OCP). Ensure `Product` class follows SRP.
* \*\*Deliverables\*\*: UML class diagram, well-commented Java code, a brief report explaining how OOP and SOLID principles were applied.
* \*\*Refactoring Challenge\*\*: Given a provided "bad code" snippet (e.g., a monolithic class with many responsibilities), identify violations of SRP and OCP, and refactor it into a cleaner, more modular design.

---

### External Resources for 'OOPs in Java'

#### Books:

* \*\*A Beginner's Guide to oops in java\*\*: This book is recommended for undergraduate students as it covers fundamental OOPs concepts in Java, aligning with the curriculum's target audience and initial modules.

#### Videos:

* \*\*Introduction to oops in java on Khan Academy\*\*: Khan Academy videos are known for their clear explanations and visual aids, making them suitable supplementary material for undergraduate students learning introductory OOPs concepts.

#### Websites:

* \*\*Official documentation website for oops in java\*\*: The official documentation is an invaluable resource for any Java programmer, providing authoritative and comprehensive information on all aspects of the language, including OOPs. It's essential for undergraduate students to learn to consult this resource for in-depth understanding and reference.