Abstraction

Review of OOP, Inheritance, Encapsulation, and Polymorphism

1. **Encapsulation:**
    - **Definition**: Encapsulation is the practice of bundling the data (variables) and the methods that act on the data into a single unit, known as a class. It also involves restricting direct access to some of an object's components.
    - **Purpose**: Encapsulation helps control access to the class's data and protects it from outside interference and misuse.
    - **How it's Achieved**:
        - Making fields (attributes) private.
        - Providing public getter and setter methods to access and modify the private fields.
2. **Inheritance:**
    - **Definition**: Inheritance allows a new class (subclass) to acquire the properties and methods of an existing class (superclass). It also allows the subclass to extend or override the superclass's functionality.
    - **Purpose**: Inheritance promotes code reuse and establishes a hierarchical relationship between classes.
    - **How it's Achieved**:
        - Using the `extends` keyword in languages like Java.
3. **Polymorphism**
    - **Definition**: Polymorphism allows an object to change its behavior based on the context. In OOP, it often refers to the ability of different classes to be treated as instances of the same class through a common interface. It involves method overriding and overloading.
    - **Purpose**: Polymorphism tailors code to a specific context, allowing methods to behave differently depending on the object they are acting upon.

- **How it's Achieved**:
  - Interclass design (using parent classes and interfaces) to reduce redundancy.
  - Intraclass design (overriding and overloading methods).

Abstraction

- **Definition**: Abstraction is often considered the "fourth pillar" of OOP. It involves creating simpler, high-level classes to represent complex systems by hiding unnecessary details and exposing only the necessary parts.
- **Purpose**: Abstraction simplifies the design by creating smaller objects that work together and reduces complexity by hiding the internal implementation details.

Simple and Complex Classes

Classes in Object-Oriented Programming (OOP) can be broadly categorized into **simple classes** and **complex classes** based on their design and the nature of the data they encapsulate.

- **Simple Classes:**
  - **Definition**: Simple classes are designed to be straightforward containers for data, with fields that are primarily primitive data types (like integers, booleans, strings, etc.).
  - **Characteristics**:
    - **Fields**: Contain primitive data types or simple data types like `int`, `double`, `boolean`, `char`, or `String`.
    - **Purpose**: Serve as a basic "container" for holding data. These classes often represent simple, singular concepts or entities.
    - **Methods**: Typically include getter and setter methods to access or modify fields, as well as some basic operations that can be performed on those fields.
  - **Example of a Simple Class:**

```
public class Point {
    private int x;
    private int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public void setX(int x) {
        this.x = x;
    }

    public int getY() {
        return y;
    }

    public void setY(int y) {
        this.y = y;
    }

    @Override
    public String toString() {
        return "(" + x + ", " + y + ")";
    }
}
```

- Here, `Point` is a simple class that represents a coordinate in a 2D space. It contains two fields (`x` and `y`) that are primitive types (`int`) and provides basic getter, setter, and `toString` methods.

- **Complex Classes:**
  - **Definition**: Complex classes encapsulate more detailed structures and behaviors. These classes contain fields that are themselves objects (instances of other classes), rather than primitive data types.
  - **Characteristics**:
    - **Encapsulated Fields**: Fields within a complex class are often instances of other classes, making the overall structure more detailed and layered.
    - **Purpose**: Designed to represent more sophisticated entities or systems where multiple simple objects work together or depend on each other.
    - **Methods**: Include more advanced logic to interact with and manage these encapsulated objects. They may also provide additional layers of abstraction to simplify interactions with the underlying objects.
  - **Example of a Complex Class:**

```
public class UniversityCourse {
    private final Room room;
    private final Professor professor;
    private final List<Student> roster;
    private final String name;
    private final int credits;

    public UniversityCourse(Room room, Professor profes
sor, String name, int credits) {
        this.room = room;
        this.professor = professor;
        this.roster = new ArrayList<Student>();
        this.name = name;
        this.credits = credits;
    }

    public void enrollStudent(Student s) {
```

```
            if (!roster.contains(s)) {
                roster.add(s);
                s.takeCredits(credits);
            }
        }

        public void dropStudent(Student s) {
            if (roster.contains(s)) {
                roster.remove(s);
                s.takeCredits(-credits);
            }
        }

        @Override
        public String toString() {
            String str = name + "\\n";
            str += "Taught by " + professor.getName() +
"\\n";
            str += "Location: " + room + "\\n";
            str += "----------------------" + "\\n";
            for (Student student : roster) {
                str += student.getName() + "\\n";
            }
            return str;
        }
}
```

- In this example, `UniversityCourse` is a complex class because it encapsulates multiple objects (`Room`, `Professor`, `Student`). The fields `room` and `professor` are instances of other classes, and the `roster` is a list of `Student` objects, which are themselves instances of a `Student` class.

- **Usage in Software Design:**
  - **Simple classes** are used for basic entities, such as a `Point` in a 2D space or a `Person` with a name and age.

- **Complex classes** are used for modeling more intricate systems or entities, such as a `UniversityCourse` that involves multiple other objects (`Room`, `Professor`, and `Student`).

Composition and Aggregation

Both composition and aggregation are ways to form relationships between objects, where **one object (the container) encapsulates instances of other objects (the components).**

- **Composition:**
  - **Definition**: A "strong" form of association where the contained objects (components) are created and managed by the container object. The contained objects do not make sense outside of the containing object.
  - **Characteristics**:
    - The inner objects are usually created internally within the constructor.
    - No setters, and often no getters are provided.
    - Encapsulated objects are not shared with other abstractions.
    - The functionality or state of the encapsulated objects is only accessible through the composition.
- **Aggregation:**
  - **Definition**: A "weaker" form of association where the internal objects (components) could exist independently of the containing object.
  - **Characteristics**:
    - Encapsulated objects are provided externally (e.g., as constructor parameters).
    - Getters and setters might be provided, and the objects can be removed or modified.
    - Encapsulated objects can be independently referenced outside the aggregation.
  - Sometimes, it's hard to say whether a relationship is **definitively** a composition or an aggregation - These are just vocabulary terms to help

developers communicate about code.

- **Example: Modeling a University Course:**
  - **Classes**:
    1. **Room Class**:
       - Encapsulates details about a room, like the building name and room number.
       - Overrides the global `toString()` method to provide a readable representation.
    2. **Course Class**:
       - Encapsulates details about a course, including a room, professor, and a list of students.
       - Uses aggregation to include references to `Room`, `Professor`, and `Student` objects, which could exist independently outside of the `Course` class.
       - Provides methods to enroll or drop students.

    ⇒ The `Course` class is an **aggregation** because it represents a relationship where the encapsulated objects (`Room`, `Professor`, `Student`) are external and independent.

    ⇒ These objects can exist without the `Course`, and they can be shared or referenced elsewhere in the program. In aggregation, the lifecycle of the encapsulated objects is not strictly bound to the container object (`Course`), meaning they can exist even if the `Course` object is destroyed.

    ⇒ We say a `Course` is an aggregation of some `Students`, a `Room`, and a `Professor`.

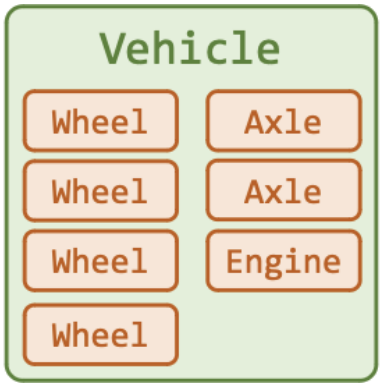
```
public class Room {
    private final String buildingName;
    private final int roomNumber;

    public Room(String buildingName, int roomNumber) {
        this.buildingName = buildingName;
```

```
        this.roomNumber = roomNumber;
    }

    @Override
    public String toString() {
        return buildingName + " " + roomNumber;
    }
}
```

```
public class Course {
    private final Room room;
    private final Professor professor;
    private final List<Student> roster;
    private final String name;
    private final int credits;

    public Course(Room room, Professor professor, String n
ame, int credits) {
        this.room = room;
        this.professor = professor;
        this.roster = new ArrayList<Student>();
        this.name = name;
        this.credits = credits;
    }

    public void enrollStudent(Student s) {
        if (!roster.contains(s)) {
            roster.add(s);
            s.takeCredits(credits);
        }
    }

    public void dropStudent(Student s) {
        if (roster.contains(s)) {
            roster.remove(s);
```

```
            s.takeCredits(-credits);
        }
    }

    @Override
    public String toString() {
        String str = name + "\n";
        str += "Taught by " + professor.getName() + "\n";
        str += "Location: " + room + "\n";
        str += "----------------------" + "\n";
        for (Student student : roster) {
            str += student.getName() + "\n";
        }
        return str;
    }
}
```

- **Example Questions**:
  1. **Vehicle Class Example:**
     - **Prompt:** A `Vehicle` class that encapsulates four `Wheel` objects, two `Axle` objects, and an `Engine` object.
     - **Answer: a. Composition**
     - **Reasoning:**
       - In this example, the `Vehicle` class encapsulates objects (`Wheel`, `Axle`, `Engine`) that do not make sense independently outside of the `Vehicle` class. For instance, a `Wheel` or `Engine` is a part of a vehicle and usually does not exist independently of it.
       - If the `Vehicle` is destroyed, its wheels, axles, and engine would also be destroyed. This tight coupling and dependent lifecycle is characteristic of **composition**.



2. **Backpack Class Example**
    - **Prompt:** A `Backpack` class that encapsulates a list of `Item` objects.
    - **fAnswer: b. Aggregation**
    - **Reasoning:**
        - In this case, the `Backpack` class holds a list of `Item` objects. However, the `Item` objects can exist independently outside the `Backpack`. For example, items can be removed from the backpack, used elsewhere, or transferred to another backpack. The existence of items does not depend on the existence of the `Backpack`.
        - This loose coupling, where the contained objects (`Item`) can exist independently of the containing object (`Backpack`), is characteristic of **aggregation**.

Composition Over Inheritance

- "**Composition over inheritance**" Principle:
    - A principle in object-oriented design that suggests that, when designing a system, you should prefer **composition** over **inheritance** to achieve code reuse and flexibility.
- **Understanding Composition and Inheritance:**

- **Inheritance** is a mechanism in OOP that allows one class (the child or subclass) to inherit fields and methods from another class (the parent or superclass). It represents an "**is-a**" relationship.
    - **Example**: A `Dog` class could inherit from an `Animal` class, making `Dog` an `Animal`.
- **Drawbacks of Inheritance:**
    1. **Tight Coupling**: The child class is tightly coupled to the parent class. Changes in the parent class can lead to unexpected behavior in the child class.
    2. **Inflexibility**: Inheritance forms a rigid hierarchy that can make it difficult to adapt or change functionality without modifying the class hierarchy.
    3. **Limited Reuse**: Inheritance only allows for a single form of reuse — you inherit everything from the parent, even if you only need a part of it.
- **Composition** is a design principle where a class is composed of one or more objects of other classes, instead of inheriting from them. It represents a "**has-a**" relationship.
    - **Example**: A `Car` class can have an `Engine` object, a `Transmission` object, and a `Wheel` object.
- **Advantages of Composition:**
    1. **Loose Coupling**: The composed objects are independent, and changes to one do not directly affect the other.
    2. **Flexibility**: Allows changing behavior by replacing the composed objects. It provides more flexibility in building complex behaviors without rigid class hierarchies.
    3. **Reusability**: Smaller, independent objects can be reused in different contexts.

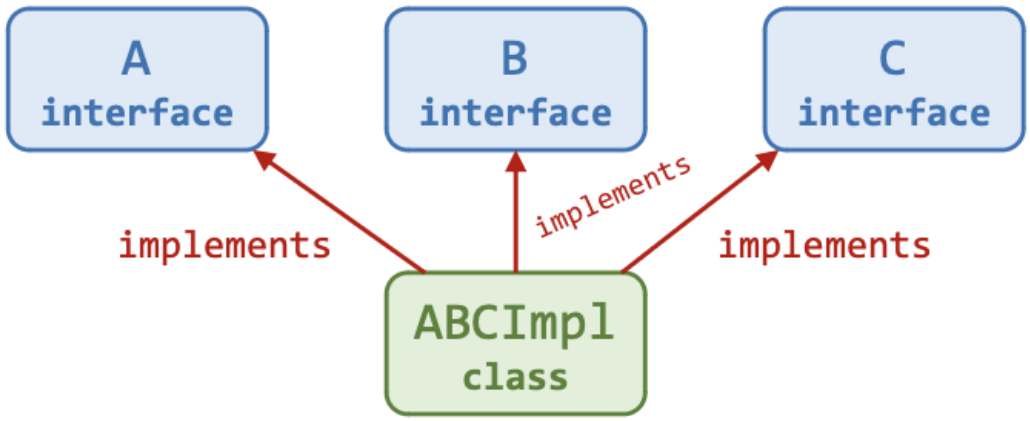
- **Why Favor Composition Over Inheritance?**

- **Flexibility and Modularity**: Composition allows you to change behavior at runtime by swapping out components. It enables more modular and maintainable code.
- **Avoids Tight Coupling**: Inheritance creates tight coupling between the parent and child classes, making changes difficult and error-prone.
- **Single Responsibility**: Composition allows you to delegate responsibilities to specific components, adhering to the Single Responsibility Principle.

- **Example: Write an `ABCImpl` Class That "Is-An" A, B, and C:**
  - Different Approaches to Achieve This Goal:
    1. No Hierarchy Approach
    2. Inheritance Approach
    3. Composition Approach

- **Approach 1: No Hierarchy:** The simplest way to make `ABCImpl` "is-an" `A`, `B`, and `C` is to directly implement all three interfaces in a single class:

```
public class ABCImpl implements A, B, C {
    // Code for implementing methods from A, B, and C
}
```

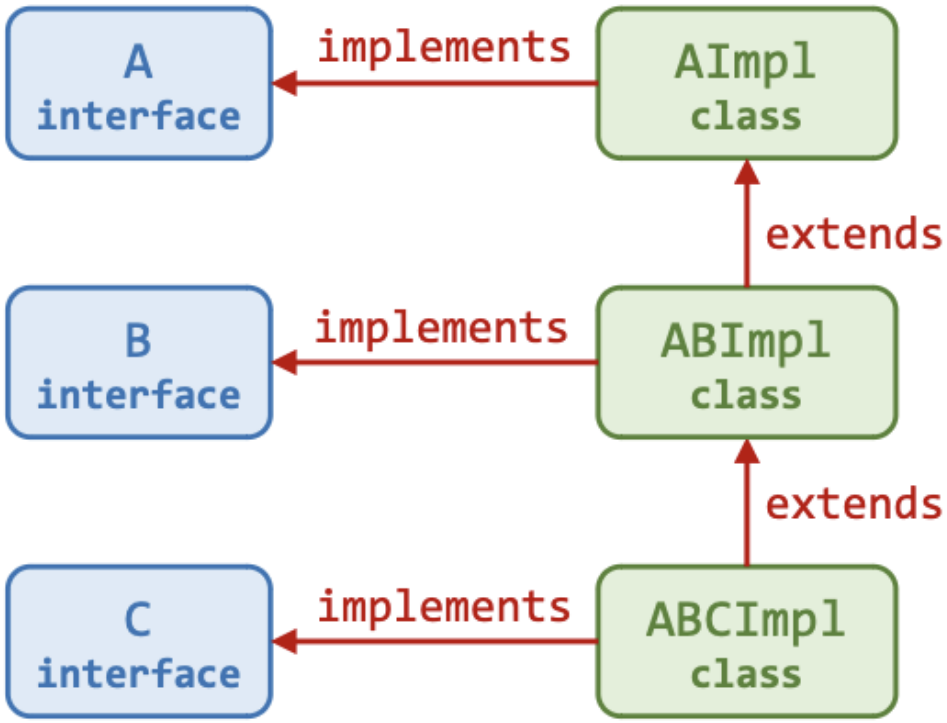


  - **Pros**: Easy to understand; directly satisfies the requirement.

- **Cons**: If `A`, `B`, and `C` have many methods or complex behaviors, `ABCImpl` can become large and difficult to maintain. Additionally, if you need to change the behavior, you may have to modify this large class, violating the **Single Responsibility Principle**.

- **Approach 2: Inheritance:** You could use inheritance to build up functionality in a hierarchical manner:

```
public class AImpl implements A {
    // Code for implementing methods from A
}

public class ABImpl extends AImpl implements B {
    // Code for implementing methods from B
    // Inherits A's methods from AImpl
}

public class ABCImpl extends ABImpl implements C {
    // Code for implementing methods from C
    // Inherits A's methods from AImpl and B's methods fro
m ABImpl
}
```



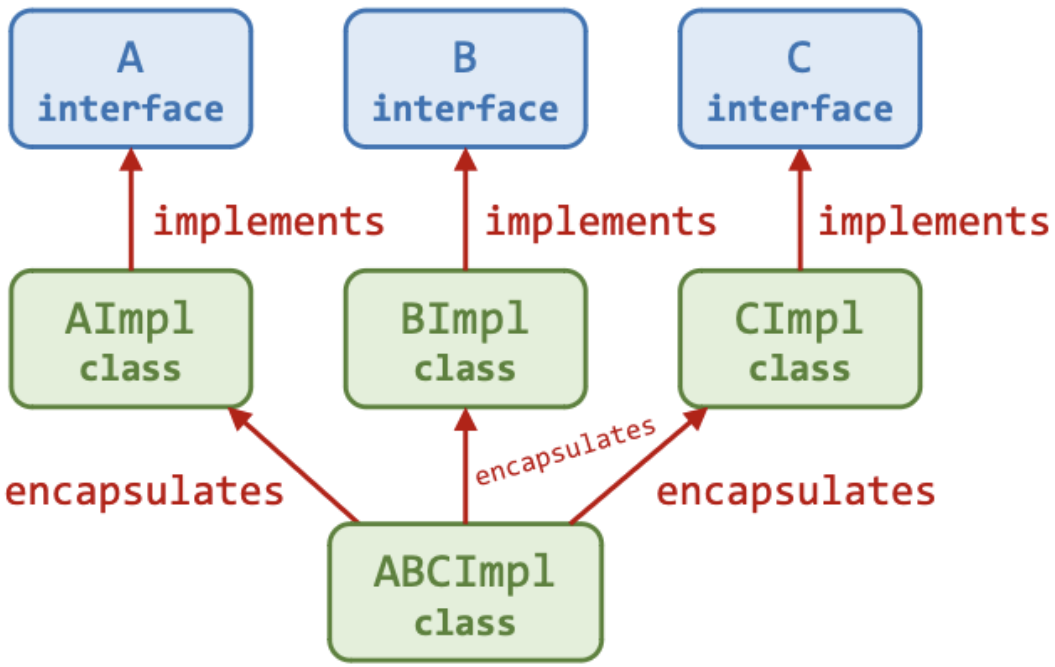
- **Pros**: Allows gradual extension of functionality. Each class adds only the methods it directly needs.
- **Cons**: Introduces tight coupling between the classes. The hierarchy is rigid and hard to change or extend. All classes must follow a specific inheritance structure, which limits flexibility and adaptability.

⇒ Works best if there is a hierarchical structure for A, B, and C.

- **Approach 3: Composition:** The composition approach allows `ABCImpl` to use smaller, independent classes (`AImpl`, `BImpl`, and `CImpl`) to fulfill its contract.

```
public class AImpl implements A {
    // Code for implementing methods from A
}

public class BImpl implements B {
    // Code for implementing methods from B
}
```

```
public class CImpl implements C {
    // Code for implementing methods from C
}

public class ABCImpl implements A, B, C {
    private A a;
    private B b;
    private C c;

    public ABCImpl() {
        this.a = new AImpl();
        this.b = new BImpl();
        this.c = new CImpl();
    }

    // Methods that delegate to A, B, and C objects
    // Code omitted
}
```



- **Pros**:
  - **Loose Coupling**: `ABCImpl` uses instances of `AImpl`, `BImpl`, and `CImpl` to provide the functionality, making the code more modular and easier to maintain.
    - **Flexibility**: You can easily change the behavior by swapping out the objects `a`, `b`, or `c` for different implementations of `A`, `B`, or `C`.
    - **Reusability**: `AImpl`, `BImpl`, and `CImpl` are reusable in other contexts without modification.
- **Cons**: Slightly more boilerplate code due to delegation methods, but this is outweighed by the flexibility and maintainability benefits.

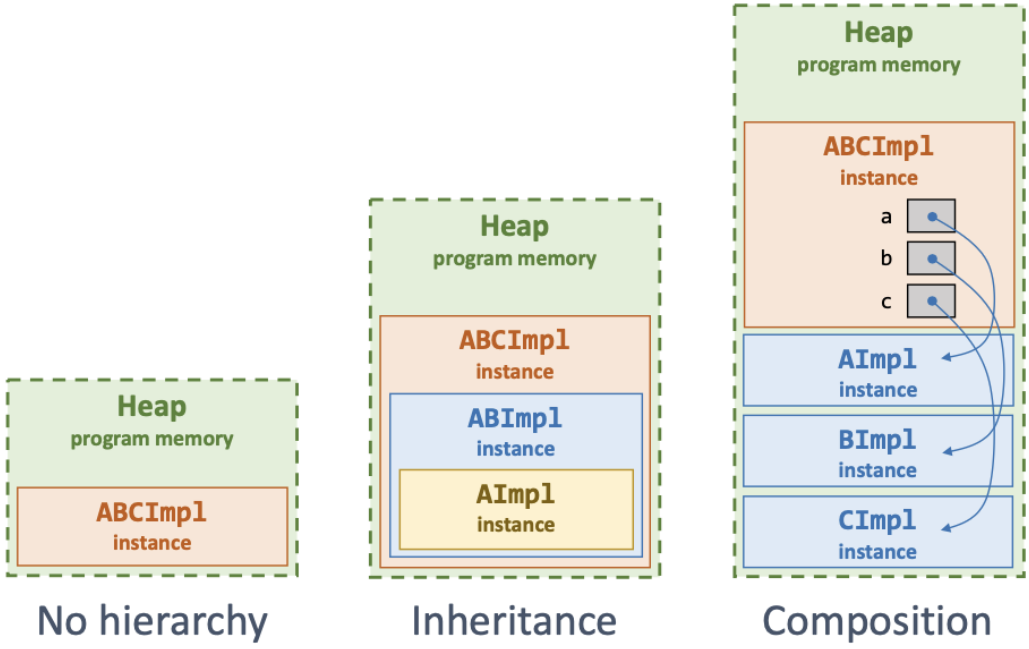
⇒ Works best if A, B, and C functionalities are separable.

- **Differences in Memory Usage:**
  - **Inheritance Memory Structure**:
    - If `ABCImpl` extends `ABImpl`, which in turn extends `AImpl`, the memory for `ABCImpl` includes all the fields and methods from `ABImpl` and `AImpl`. Each instance is part of a single object, but memory is allocated for all parent classes in a hierarchical stack.

```
Heap memory:
    ABCImpl instance
    ABImpl instance
    AImpl instance
```

  - **Composition Memory Structure**:
    - The `ABCImpl` class contains references to objects of `AImpl`, `BImpl`, and `CImpl`. These objects are created separately in memory and are referenced within `ABCImpl`. The memory is allocated for each component object separately, which can be managed or replaced individually.

```
Heap memory:
    ABCImpl instance
      -> AImpl instance
      -> BImpl instance
      -> CImpl instance
```



Dependency Injection (DI) and Coupling

- **Understanding Coupling:**
  - **Coupling** refers to the degree of direct knowledge that one class has of another.
    - When classes are **tightly coupled**, they are highly dependent on each other, making them hard to separate, reuse, and maintain.
    - Conversely, **loosely coupled** classes can function independently of each other, improving the flexibility, maintainability, and readability of the code.
  - **Tightly Coupled Code**: When a class directly references another class by name, it creates a strong dependency between the two. This makes it

difficult to modify one class without affecting the other.

- **Loosely Coupled Code**: Code is separated into well-defined, independent modules that can function and evolve separately.
- **Goal**: Design systems that are **loosely coupled** to make them easier to maintain, modify, and test.

- **Dependency Injection (DI):**
  - **Dependency Injection:** A technique in which an object receives its dependencies from an external source, rather than creating them itself. This is done by **injecting** the required dependencies into the object using constructors, setters, or interfaces.
  - **Dependency**: An object that another object needs to perform its tasks. For example, a `Vehicle` class may need an `Engine` or `Wheel` to operate.
  - **Injection**: The process of providing the required dependencies to a class.
- **Advantages of Dependency Injection:**
  1. **Loose Coupling**: The code becomes loosely coupled because the dependencies are supplied externally rather than being hard-coded within the class.
  2. **Improved Modularity and Reusability**: Classes become more modular and can be reused in different contexts.
  3. **Easier to Test**: Dependency injection makes unit testing easier because dependencies can be mocked or replaced with test-specific implementations.
- **Example: Class Dependencies Without Dependency Injection:**
  - Consider the following example of a `VehicleImpl` class that is tightly coupled with specific implementations of `EngineImpl` and `WheelImpl` classes:

```
public class VehicleImpl {
    private Engine engine;
    private Wheel frontLeft;
    private Wheel frontRight;
    private Wheel rearLeft;
```

```
    private Wheel rearRight;

    public VehicleImpl() {
        engine = new EngineImpl();
        frontLeft = new WheelImpl();
        frontRight = new WheelImpl();
        rearLeft = new WheelImpl();
        rearRight = new WheelImpl();
    }

    // ...
}
```

- **Problems with This Approach:**
  - **Tight Coupling**: `VehicleImpl` is tightly coupled to `EngineImpl` and `WheelImpl`. This makes it difficult to test the `VehicleImpl` class independently or swap out different implementations of `Engine` or `Wheel`.
  - **Lack of Flexibility**: If you want to change the `Engine` to a different type or use a subclass of `Wheel`, you would need to modify the `VehicleImpl` class.
  - **Difficult to Test**: You can't test the `VehicleImpl` class in isolation because it always creates specific instances of `EngineImpl` and `WheelImpl`.

- **Refactoring with Dependency Injection:**
  - **Option 1: Constructor Injection:**
    - Constructor injection is a way to provide dependencies to a class by passing them through its constructor.

```
public class VehicleImpl {
    private Engine engine;
    private Wheel frontLeft;
    private Wheel frontRight;
```

```
    private Wheel rearLeft;
    private Wheel rearRight;

    // Constructor Injection
    public VehicleImpl(Engine engine, Wheel frontLeft, Whe
el frontRight, Wheel rearLeft, Wheel rearRight) {
        this.engine = engine;
        this.frontLeft = frontLeft;
        this.frontRight = frontRight;
        this.rearLeft = rearLeft;
        this.rearRight = rearRight;
    }

    // Additional methods...
}
```

- **Benefits**:
  - **Loose Coupling**: The class is no longer directly dependent on specific implementations (`EngineImpl`, `WheelImpl`). Instead, it depends on the `Engine` and `Wheel` interfaces, allowing any implementation of these interfaces to be used.
  - **Flexibility**: You can easily change the engine or wheels by providing different implementations at runtime.
  - **Testability**: Easier to test because you can inject mock or stub dependencies in the test cases.
- **Option 2: Setter Injection:**
  - Setter injection provides dependencies through setter methods.

```
public class VehicleImpl {
    private Engine engine;
    private Wheel frontLeft;
    private Wheel frontRight;
    private Wheel rearLeft;
```

```
    private Wheel rearRight;

    // Constructor
    public VehicleImpl() {
        // Default constructor
    }

    // Setter methods for Dependency Injection
    public void setEngine(Engine engine) {
        this.engine = engine;
    }

    public void setFrontLeftWheel(Wheel wheel) {
        this.frontLeft = wheel;
    }

    public void setFrontRightWheel(Wheel wheel) {
        this.frontRight = wheel;
    }

    public void setRearLeftWheel(Wheel wheel) {
        this.rearLeft = wheel;
    }

    public void setRearRightWheel(Wheel wheel) {
        this.rearRight = wheel;
    }

    // Additional methods...
}
```

- **Benefits**:
  - **More Configurable**: Allows objects to be configured after they are created, offering more flexibility in certain situations.

    - **Loose Coupling**: Similar benefits as constructor injection; decouples the class from specific implementations.

- **Using Dependency Injection:**
    - **Without Dependency Injection:**

```
Vehicle car = new VehicleImpl();
```

      ⇒ **Problem**: The `VehicleImpl` constructor creates its own dependencies, making it tightly coupled to `EngineImpl` and `WheelImpl`.

    - **With Dependency Injection:**

      1. **Constructor Injection:**

```
Vehicle car = new VehicleImpl(
    new EngineImpl(),
    new WheelImpl(),
    new WheelImpl(),
    new WheelImpl(),
    new WheelImpl()
);
```

      ⇒ The dependencies (`EngineImpl` and `WheelImpl`) are created outside and passed into the `VehicleImpl` class. This allows for more flexibility and easier testing.

      2. **Setter Injection:**

```
Vehicle car = new VehicleImpl();
car.setEngine(new EngineImpl());
car.setFrontLeftWheel(new WheelImpl());
car.setFrontRightWheel(new WheelImpl());
car.setRearLeftWheel(new WheelImpl());
car.setRearRightWheel(new WheelImpl());
```

⇒ With setter injection, dependencies are injected after the object is created, providing flexibility in configuration.

- **Bottom Line: Advantages and Disadvantages of Dependency Injection:**
  - **Advantages of Dependency Injection:**
    1. **Loose Coupling**: Classes are less dependent on specific implementations and more dependent on abstractions (interfaces), making code easier to modify and extend.
    2. **Testability**: Makes it easier to write unit tests by allowing mock or stub implementations to be injected.
    3. **Configurable and Flexible**: Objects are more configurable and can be used in a variety of situations with different dependencies.
    4. **Improved Maintainability**: Because dependencies are injected from the outside, changes to one part of the system have minimal impact on others.
  - **Disadvantages of Dependency Injection:**
    1. **More Code Required**: Requires more code to set up and manage dependencies, which can increase complexity.
    2. **More Development Effort**: Requires understanding of DI patterns and can involve a steeper learning curve.
    3. **Goes Against "Convention over Configuration"**: In some cases, especially in simpler applications, DI can feel cumbersome compared to more conventional, direct instantiation.

Inversion of Control

- **Traditional Control Flow:** In a traditional, procedural program:
  - Execution starts in the `main()` method (or a test method if you're running tests).
  - You, the developer, define and control the flow of execution.
  - For example, consider this basic structure:

```
public class Main {
    public static void main(String[] args) {
        UserService userService = new UserService();
        userService.notifyUser("Hello, User!");
    }
}
```

- Here, you control the flow of execution: you decide when to create `UserService` and when to call its `notifyUser()` method.
- This is **traditional control flow**, where the program's execution is under your full control.

- **Inversion of Control (IoC):** In **Inversion of Control**, **someone else** (like a framework or a higher-level structure) controls the flow. This means:
  - You don't have control over when your methods are called.
  - Your methods are called by some external entity (like a framework, library, or another part of the system).
  - Your responsibility is to respond to method calls, not control when they happen.
  - Frameworks such as Spring, Angular, or event-driven systems (like event handlers in GUI applications) are good examples of IoC.
  - Example of IoC:

```
public class EmailService {
    public void sendEmail(String message) {
        System.out.println("Sending email: " + messag
e);
    }
}
```

  - If this service was part of a larger framework, like Spring, you wouldn't manually instantiate it or control when it sends an email. Instead, the framework would handle that, and you'd just define the behavior.

- **Tightly Coupled Example:**
  - In the original example, the `UserService` directly creates and controls the `EmailService`. This creates **tight coupling**:

```
public class UserService {
    private EmailService emailService = new EmailService
();

    public void notifyUser(String message) {
        emailService.sendEmail(message);
    }
}
```

  - Here, `UserService` has full control over `EmailService`, which is tightly coupled because `UserService` directly creates an instance of `EmailService`. This makes testing, scaling, or reusing `UserService` more difficult, as it is dependent on the specific instantiation of `EmailService`.
- **Inversion of Control Using Dependency Injection (DI):**
  - One way to achieve IoC is by using **Dependency Injection (DI)**. Instead of creating the dependencies (e.g., `EmailService`) inside the class, they are **injected** from outside. This way, the class is not responsible for managing its own dependencies.
  - In DI, control over dependencies is inverted:
    - The framework or another part of the system creates the dependencies and injects them into the class.
    - Your class just uses these dependencies without controlling their lifecycle.
  - Example of Dependency Injection (DI):

```
public class UserService {
    private EmailService emailService;

    // Constructor Injection
```

```
    public UserService(EmailService emailService) {
        this.emailService = emailService;
    }

    public void notifyUser(String message) {
        emailService.sendEmail(message);
    }
}
```

⇒ Now, `UserService` no longer creates `EmailService` on its own. Instead, it expects `EmailService` to be provided from the outside (this is **Dependency Injection**). The main method might look like this:

```
public class Main {
    public static void main(String[] args) {
        EmailService emailService = new EmailService();
        UserService userService = new UserService(emailS
ervice);
        userService.notifyUser("Hello, User!");
    }
}
```

- Or, if using a framework like Spring, the framework would handle the creation and injection of `EmailService` automatically:

```
@Service
public class UserService {
    private final EmailService emailService;

    // Spring would inject this dependency automaticall
y
    @Autowired
    public UserService(EmailService emailService) {
        this.emailService = emailService;
```

```
    }
}
```

- In this case, Spring controls the flow (IoC), creating and injecting dependencies as needed.

- **How IoC and DI Are Related:**
  - **Inversion of Control** is a broader concept where you don't control the execution flow — some other entity does.
  - **Dependency Injection** is a specific way to implement IoC, where dependencies are injected into a class rather than the class creating them internally.

  ⇒ DI is one of the ways to decouple your code and make it more modular, testable, and easier to manage. It is also a practical example of how IoC works in frameworks and complex systems.

Summarizing the Four Pillars of OOP

Pillar	Description	Purpose	Key Benefits	Key Drawbacks
Encapsulation	Bundles data and methods in a single unit and restricts access to some of the object's components.	To protect the internal state of an object and provide a controlled interface.	Reduces complexity, increases maintainability, protects internal state.	Can lead to boilerplate code (getters/setters).
Inheritance	Allows a new class to inherit properties and methods from an existing class.	To promote code reuse and establish a natural hierarchical relationship between classes.	Reduces redundancy, facilitates easier maintenance and extension.	Can create tight coupling, making code less flexible and harder to modify.
Polymorphism	Allows a single interface to be used for	To enable flexibility and reuse by	Increases flexibility,	Overuse can make code harder to

	different underlying data types or objects.	allowing methods to operate differently on different objects.	reduces code redundancy.	understand and maintain; potential runtime errors.
Abstraction	Hides complex implementation details and exposes only the necessary parts of an object.	To reduce complexity and show only essential features, enhancing user experience and security.	Simplifies complex systems, enhances security, improves maintainability.	Requires more design and planning upfront; can lead to more code complexity.

Comprehensive Example: Library Management System:

- Suppose you are tasked with designing a **Library Management System** that needs to handle various types of media, such as books, magazines, and DVDs. Each media type has its own set of characteristics (like title, author, and publication date) and specific behaviors (like checking availability, borrowing, and returning).
- **Applying the Four Pillars of OOP:**
  1. **Encapsulation:**
     - Encapsulation involves bundling the data (attributes) and methods (behaviors) related to an object into a single unit (a class) and restricting access to some of its components.
     - **Encapsulated Classes**:
       - We can create a base class `Media` that has common properties and methods for all media types.
       - We will make the attributes `title`, `author`, `publicationDate`, and `isAvailable` private to protect them from direct access. Instead, we provide public methods (getters and setters) to interact with these attributes.

- **Benefits**: Encapsulation protects the internal state of each `Media` object by restricting direct access to its attributes, ensuring that all interactions with the object's state are controlled through its methods.

```
public class Media {
    private String title;
    private String author;
    private String publicationDate;
    private boolean isAvailable;

    // Constructor
    public Media(String title, String author, String publi
cationDate) {
        this.title = title;
        this.author = author;
        this.publicationDate = publicationDate;
        this.isAvailable = true; // default to available
    }

    // Encapsulated methods to access private attributes
    public String getTitle() {
        return title;
    }

    public String getAuthor() {
        return author;
    }

    public String getPublicationDate() {
        return publicationDate;
    }

    public boolean isAvailable() {
        return isAvailable;
    }
```

```
    public void setAvailable(boolean isAvailable) {
        this.isAvailable = isAvailable;
    }

    public void borrow() {
        if (isAvailable) {
            this.isAvailable = false;
            System.out.println("The media item '" + title
+ "' has been borrowed.");
        } else {
            System.out.println("The media item '" + title
+ "' is not available.");
        }
    }

    public void returnItem() {
        this.isAvailable = true;
        System.out.println("The media item '" + title + "'
has been returned.");
    }
}
```

2. **Inheritance:**
    - Inheritance allows us to create subclasses that inherit properties and behaviors from a parent class, promoting code reuse.
    - Let's create subclasses `Book`, `Magazine`, and `DVD` that inherit from the base class `Media`. Each subclass can have its own additional properties or behaviors while reusing common properties and methods from `Media`.
    - **Benefits**: Inheritance allows `Book`, `Magazine`, and `DVD` to reuse common properties and methods from the `Media` class, reducing code duplication and promoting a hierarchical relationship.

```
public class Book extends Media {
    private String genre;

    public Book(String title, String author, String public
ationDate, String genre) {
        super(title, author, publicationDate); // Inherit
common properties from Media
        this.genre = genre;
    }

    public String getGenre() {
        return genre;
    }
}

public class Magazine extends Media {
    private int issueNumber;

    public Magazine(String title, String author, String pu
blicationDate, int issueNumber) {
        super(title, author, publicationDate);
        this.issueNumber = issueNumber;
    }

    public int getIssueNumber() {
        return issueNumber;
    }
}

public class DVD extends Media {
    private int duration; // Duration in minutes

    public DVD(String title, String author, String publica
tionDate, int duration) {
        super(title, author, publicationDate);
```

```
        this.duration = duration;
    }

    public int getDuration() {
        return duration;
    }
}
```

3. **Polymorphism**

   - Polymorphism allows a single interface to represent different types of objects, providing flexibility in handling different data types and behaviors uniformly.
   - **Polymorphism in Action**:
     - We can use polymorphism to handle different media types (`Book`, `Magazine`, `DVD`) in a uniform way.
     - For example, we can create a list of `Media` objects and perform operations like borrowing and returning without worrying about the specific type of media.
   - **Benefits**: Polymorphism allows the `Library` class to manage all types of media (`Book`, `Magazine`, `DVD`) in a generic way, treating them as instances of `Media`. This reduces code redundancy and increases flexibility.

```
import java.util.ArrayList;
import java.util.List;

public class Library {
    private List<Media> catalog = new ArrayList<>();

    public void addMedia(Media media) {
        catalog.add(media);
    }
```

```
    public void borrowMedia(String title) {
        for (Media media : catalog) {
            if (media.getTitle().equals(title)) {
                media.borrow();
                return;
            }
        }
        System.out.println("Media with title '" + title +
"' not found.");
    }

    public void returnMedia(String title) {
        for (Media media : catalog) {
            if (media.getTitle().equals(title)) {
                media.returnItem();
                return;
            }
        }
        System.out.println("Media with title '" + title +
"' not found.");
    }
}
```

4. **Abstraction:**
    - Abstraction involves hiding complex implementation details and exposing only the necessary parts of an object, focusing on what an object does rather than how it does it.
    - **Abstraction in Action**: The `Media` class provides a general contract for all media types with methods like `borrow()` and `returnItem()`. The user does not need to know the internal details of how these methods work; they only interact with the provided interface.
    - **Abstract Class Example**: If we need to ensure that every type of media must have a `borrow()` and `returnItem()` method, we can make

`Media` an abstract class or an interface and force every subclass to implement these methods.

- **Benefits**: Abstraction hides the complex logic within methods like `borrow()` and `returnItem()`, allowing users to interact with a simple interface without needing to understand the underlying details.

```
public abstract class Media {
    private String title;
    private String author;
    private String publicationDate;
    private boolean isAvailable;

    public Media(String title, String author, String pu
blicationDate) {
        this.title = title;
        this.author = author;
        this.publicationDate = publicationDate;
        this.isAvailable = true;
    }

    public abstract void borrow();

    public abstract void returnItem();

    // Common getters...
}
```

- **Summary:**
  - **Encapsulation** protects the internal state of `Media` objects by providing controlled access through methods.
  - **Inheritance** allows `Book`, `Magazine`, and `DVD` to reuse common properties and methods from `Media`, promoting code reuse.
  - **Polymorphism** enables the `Library` class to handle different types of media in a generic way, reducing code redundancy and increasing

flexibility.

- **Abstraction** provides a simplified interface for interacting with media objects, hiding the complexity of the underlying implementation.