**What is OO?**

A Paradigm or way of thinking / viewing / structuring for software

**Define OOP?**

* A paradigm is very centralized around object concept where class collect data then enforcement
* Figure out object + function + interactions

**Why OOP?**

* **Real-world analogy**: Makes it easier to model real-world systems.
* **Complexity management**: Helps handle large, complex systems by organizing them into smaller, manageable parts (classes).
* **Maintainability**: Promotes easier maintenance, debugging, and testing.
* **Code reuse**: Encourages code reuse and reduces redundancy.
* **Extensibility**: Easily extendable and adaptable to new requirements.
* **Avoiding software crisis**: Helps avoid over-budgeting, overtime, and bugs in large-scale systems.

**What is Procedural Programming?**

* Paradigm focus is on creating a sequence of **steps**
* Control flow is based on **sequential steps** (like loops, conditionals) that modify the state as the program runs.
* Data is often **mutable** and passed to functions where it can be modified.
* The program is usually a collection of tasks
* Functions are used to encapsulate **procedures** or steps that change the state of the system
* Suitable for applications with well-defined, sequential operations

**What is Functional Programming?**

* Paradigm focus is on **pure functions** and **immutable data**.
* Control flow is based on **function calls** and recursion (iteration is usually done via recursion, not loops).
* Data is **immutable**: once created, data cannot be changed. Instead, new data is returned from functions.
* Functions are first-class citizens (assign to variable – pass to function parameter- return function from function)
* Functions given the same inputs, they always return the same outputs and do not modify any external state.
* Suitable for more complex programs where modularity, reusability, and correctness are priorities

**Flaws of Functional Programming?**

* Difficult to handle systems that require frequent state changes
* Recursive functions can cause performance hits or stack overflow if not optimized

**Flaws of Procedural Programming?**

* No owner for data so it can be found everywhere
* Data integrity issue and Duplication
* Missing Data ex: customer address is missing in order (on validation)
* Inconsistent Data (string CutomerName= "Jhon" in place and "Mike"in another part of the program)
* Hard debugging if data corrupted or to update it

**What is the class?**

* **Blueprint** or template or that describes the state and behavior of objects
* Pattern or container or of a given type of copies objects

**What`s class element?**

Declaration: define class name

Body: curly brackets container

Fields: data members defined

Methods: member functions perform operations

Properties: characteristics assigned by access modifiers

**How it works?**

Group all relevant attributes in ONE unit

Provides internal operations on these data using methods

Bind variables **attributes** & operations **methods**

**What is object?**

Tangible entity that represents a specific instance of a given class **attributes**

**Attributes** State: define or describe object in a specific moment as data member

**Methods** Behavior: the specific action that can be done by object which make changes

**What is Encapsulation?**

Group all the data members & member functions within one single unit where can access them using

Setter & Getter

**Why we use encapsulation?**

**Data Hiding**: restricts direct access to some of an object's components only allows changes through well-defined (getters and setters)

**Code Organization**: grouping related data and methods together, reducing complexity and making software manageable

**? Define Accessors & Metuters**

Accessors: is a function used to read / access data member

Mutators: is a function used to change/mutate data member

**“**Don`t ask for the information you need to do the work, ask the object for the information that has to do the work” **provide more functions than use set & get**

**Different Bet~ Data Hiding & Encapsulation?**

Hiding data/functions from outside code using Access modifiers

1. **Public**:
   * The member is accessible from anywhere, both inside and outside the class.
   * No restrictions on access.
2. **Private**:
   * The member is accessible only within the class where it is declared.
   * Cannot be accessed directly from outside the class.
3. **Protected**:
   * The member is accessible within the class and by subclasses (derived classes) but not from outside the class or package.
   * Allows inheritance to access the members.
4. **Default (Package-Private)**:
   * In languages like Java, if no access modifier is specified, the member is accessible within the same package but not outside it.
   * This is also called package-private in some languages.
5. **Internal (C# specific)**:
   * The member is accessible within the same assembly or project but not outside it.
   * Used to restrict access within a specific project.
6. **Protected Internal (C# specific)**:
   * Combines protected and internal. The member is accessible within the same assembly or by derived classes.

**Why we use Data Hiding?**

Reduce outsider’s dependency (coupling) to whole data/functions

Hide complexity & object state **changes ex: Google search engine**

**Different between public variable and private variable uses setter & getter?**

**Accessibility**

**Public** is directly accessed from anywhere (no control or validation) its value can affect behavior of code throughout a program

**Private** Indirect access, allowing you to control how the data is accessed and modified using validations (more secure and flexible)

**Use Public Variables** when:

* The value is constant or immutable.
* You want the variable to be directly accessible and you don’t need validation or protection.
* The class is simple, and you're okay with exposing all its fields.

**Ex:** DTOs (Data Transfer Objects), simple containers of data

**Use Private Variables with Setters/Getters** when:

* You need to protect the data or ensure it remains valid.
* You need flexibility to change the internal implementation without breaking external code.
* You want to encapsulate logic for validation, security, or other behaviors.

**Ex**: Suppose you have a Person class with an age variable. You might want to ensure that the age is always positive and never negative. If the age was public, someone could directly set it to a negative number, which might not make sense in your program.

**Abstraction**

Is about hide **more** unwanted details & showing most essential in a context

**How** hide as possible you can implementation and data

show (What) user will use & context identify what`s needed and what`s not

**Context**

Specific environment or circumstances under which a piece of code is executed

Influenced by factor (variables - state - scope - calling) ex: employee (Google – Toyota)

**Why we need Abstraction?**

**Coupling:** class directly uses another class's methods or data without any abstraction (tight)

**Risk of Errors:** Exposing internal details without abstraction leads to side effects & require changes in the dependent class

**Flexibility and Scalability:** Adding new features or extending functionality becomes harder because changes might ripple through the entire codebase.

**Abstract class & Concrete class & Interface?**

|  |  |  |  |
| --- | --- | --- | --- |
| **Interface** | **Abstract Class** | **Concrete class** | **Features** |
| Can`t be instantiated | Can`t be instantiated | Can be instantiated directly | **Instantiation** |
| Can be implemented by multiple classes (multiple inheritance) | Supports inheritance (single inheritance) | Can be inherited (single inheritance) | **Inheritance** |
| Contains only abstract method declarations (or default methods in some languages) | Can contain both abstract and concrete methods | Contains only concrete methods | **Methods** |
| Complete abstraction (no method implementations) | Partial abstraction (abstract methods and concrete methods) | No abstraction | **Abstraction** |
| Used to define a contract that implementing classes must follow | Used as a base class for other classes | Direct use in code | **Use case** |
| Can`t have a constructor | Can have a constructor | Can have a constructor | **Constructor** |

**Virtual & Pure Virtual Method & Friend?**

**Virtual:** A method in a base class with a default implementation that can be overridden or use its implementation in derived classes.

**Pure Virtual:** A method in a base class with a no implementation that must be overridden in derived classes to ensures that the derived classes provide their own behavior

**Friend Function:** Non-member functions useful for cases where you need access to private members but don’t want to make the function a member of the class itself

**Polymorphism**

Allows objects/methods of different types to be treated as if they were of the same type **have many forms same type**

Allow us to write generic code without knowing exact **children**

|  |
| --- |
| **Ex**: void process(Shape shape) -> can be circle - Rectangle - Triangle |

**Overloading**

Two or more methods in the same class have the **same name** but **different parameters (Type + number)**

Allows a class to have multiple methods that perform similar tasks but with different input types or different numbers of arguments

Overloading is resolved during **compile-time** based on the method signature (name + parameter list)

Overloaded methods can have the **same** or **different** return types, but the return type alone cannot be used to distinguish overloaded method

**Overriding**

Occurs when a **subclass** provides its own specific implementation of a method that is already defined in its **superclass**

Change or extend the behavior of the inherited method

Can use the @Override annotation to explicitly indicate that a method is being overridden

Resolved during **runtime**. The method to be executed is determined based on the actual object type, not the reference type

**Virtual Table**

A mechanism used by **OOP** to implement **runtime polymorphism** and **method overriding**

Virtual table allows dynamic dispatch or late binding

Method that should be executed is determined at **runtime**, based on the actual object type, rather than the reference type.

**How Virtual Table Works:**

* An array of pointers to methods that marked as virtual
* Each class with virtual methods (methods that can be overridden in derived classes) has its own vtable.
* Every object of a class that has virtual functions contains a pointer to the vtable (often called a vptr).
* The vptr is set up when an object is created, and it points to the vtable corresponding to the class of the object.
* When a method is called on an object, the program uses the vptr to look up the correct function in the vtable. This allows the program to call the overridden method of the actual class of the object, not the class of the reference

**Inheritance**

Represents an "is-a" relationship between classes (sub class - superclass)

Allows a class to **copy** properties, methods, and other characteristics from another class

**Base class**: A class that is inherited by another class and it is starting point for other classes. It can be called as general class.

**Super class**: A class that is inherited by another class can be called parent class

**Derived class**: A class that inherits from a base or super class

**Inheritance Abuse**

**Overgeneralization**: this occurs when a **superclass** is created too broadly, meaning it tries to capture too many behaviors and properties from a variety of subclasses, which leads to a **lack of specificity**

**Ex**: Superclass **Vehicle** that includes all possible vehicle-related behavior, such as drive (), sail (), and fly (), even though not all subclasses (e.g., Car and Boat) would use all these behaviors.

**Under generalization superclass** fails to capture the **commonalities** between subclasses that could be shared that leads to subclasses duplicate similar behavior

**EX:** Car and Truck share common properties like engine () and wheels (), but those properties are not extracted into a Vehicle superclass.

**Diamond Problem**: a class inherits from **two or more classes** that share a **common ancestor**, leading to ambiguity in resolving inherited methods or attributes.

**Types of Inheritance**

**Single**: A subclass inherits from only one superclass

**Multiple**: A subclass inherits from multiple super classes **not supported in some languages**

**Multilevel**: A subclass inherits from another class, which itself inherits from a superclass

**Hierarchical**: Multiple subclasses inherit from a common superclass

**Hybrid**: A combination of multiple and hierarchical inheritance

Constructor

Deconstructor

Keywords (static – let – var – val – final – const –sealed – virtual - factory )

Binding -> late – early

Copy by value – reference

Static & dynamic memory

Compile & run Time

**What are the SOLID principles?**

The SOLID principles consist of five design principles for object-oriented programming, introduced by Robert C. Martin. Ensures quality (correct functionality & maintenance) of software and make it easier to apply design pattern they are:

**S: Single Responsibility Principle (SRP)**

A module should have only one reason to change, meaning it should have only one responsibility.

A class has one job and does it well

It helps us in maintenance that we know the class behavior and mission so you can edit it easily

**What if a class has intertwined responsibilities?**

Identify core responsibilities and separate they were possible by create new services to isolate them

**What if splitting a class for SRP causes too many classes?**

Grouping of related tasks, not splitting everything use patterns

**How to apply SRP in MVC or MVVM architectures?**

**Model** handles business logic, **View** handles UI, and **Controller** delegate’s tasks.

**How to handle tightly coupled responsibilities?**

Use **dependency injection** to separate concerns. For example, inject services into a class instead of letting it handle everything

**How to refactor legacy code for SRP without breaking things?**

Separate responsibilities one by one. Focus on refactoring small methods or classes first, and use interfaces to decouple dependencies. Class user – class email service - class user service depend on email & user

public class **UserService** {

private UserRepository userRepository;

private EmailService emailService;

// Constructor injection for dependency injection

public **UserService**(UserRepository userRepository, EmailService emailService) {

this.userRepository = userRepository;

this.emailService = emailService;

}

public void **registerUser**(String username, String password, String email) {

userRepository.**saveUserToDatabase**(username, password); // Delegate to UserRepository

emailService.**sendEmailConfirmation**(email); // Delegate to EmailService

}

}

**What if SRP conflicts with DRY or KISS?**

It’s okay to have a class that does more than one thing if those responsibilities are simple and logically related

Don’t split classes unnecessarily just to follow SRP if it leads to duplication that could be avoided.

**O: Open/Closed Principle (OCP)**

Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.

**Can the Open/Closed Principle be applied to all types of software systems?**

A high level of abstraction and might be easier to maintain by modifying existing code directly

**How does the Open/Closed Principle work in the context of frameworks?**

Add new features or change behavior without modifying the framework itself.

New feature -> in scope of repeated features (state pattern)

New feature -> has its own functionality (strategy pattern)

New feature -> added transport ally to an existing feature to do specific task (decorator)

**What are some potential pitfalls when trying to follow the Open/Closed Principle?**

**Over-engineering:** Trying to create abstract classes and interfaces for every possible extension can result in unnecessary complexity

**Increased testing complexity:** Extending code while ensuring that existing code works perfectly can increase the complexity of testing

**L: Liskov Substitution Principle (LSP)**

Objects of a superclass should be replaceable with objects of a subclass without affecting the functionality of the program.

If B inherits from class A. class B can replace class A without affecting the functionality

If the replacement/ change in a step within a fixed steps -> template method

**How to check if you implement LSP correctly?**

All functionality in parent class implemented in child class without adding public disposed function

If you want to add specific function to child make sure it’s private

**I: Interface Segregation Principle (ISP)**

Split large interfaces into smaller, more specific interfaces. Clients should not be forced to depend on interfaces they do not use. Split depends on business logic - number of clients they would use the interface

**What are the potential drawbacks or challenges of strictly following ISP?**

**Over-abstraction**: Breaking down interfaces too much can lead to **over-engineering**

**Refactoring Overhead**: If an existing system violates ISP and needs to be refactored, it might require significant changes

**How would you refactor a class that violates the Interface Segregation Principle?**

**Can you explain how ISP helps avoid "fat" interfaces?**

**Identify the large interface** that the class is implementing.

**Analyze which methods** are relevant to the specific class.

**Split the large interface into smaller, more focused interfaces**. Each interface should group methods that are logically related to each other

**How does ISP relate to the Single Responsibility Principle (SRP)?**

* **SRP** states that a class should have only one reason to change, meaning it should only be responsible for one task or functionality.
* **ISP** states that clients should not be forced to depend on methods they don’t use.

**D: Dependency Inversion Principle (DIP)**

High-level modules should not depend on low-level modules. Both should depend on abstractions. Furthermore, abstractions should not depend on details. Details should depend on abstractions.

**In DIP in solid when we call the class its high level module and when we call it low level module?**

**High-Level Module**: This module is responsible for the overall logic of the application. It typically defines the business rules or use cases

**Low-Level Module**: These are the modules responsible for more specific details or implementation, such as database access, network communication, or file I/O.

**This is helps us to solve tight coupling between classes**

**Class** image\_picker(crop\_helper.crop) - **interface** crop\_helper(crop) - **class** crop1 – **class** crop2

**Dependency Injection (DI)** is a pattern where the dependencies (services or objects) a class needs are provided (injected) from the outside rather than the class creating them itself.

In simpler terms, **Dependency Injection** allows a class to receive its dependencies from an external source, instead of creating them internally, making the class more flexible, easier to test, and decoupled from its dependencies.

**Inversion of Control (IoC) (Service Locator)** the control over the flow of a program and the creation of objects is shifted from the application to an external framework or container & takes control and provides the necessary dependencies

**Types of DI**

**Constructor Injection**: required at object creation.

**Setter Injection**: when you want to inject dependencies after the object is constructed.

**Interface Injection**: the class controls how the dependency is injected.

**Design patterns**

Best practice to solve common software problems

Solutions in the form of template that may apply to real-world problems

**When we use Design pattern?**

If you face a problem that design patterns solves if not don’t use it

### 1. ****Creational Design Patterns****

Creational design patterns deal with the process of object creation, making it easier to instantiate objects in a way that promotes flexibility and reusability.

#### ****Key Purpose:****

To abstract the instantiation process, making it more flexible and reducing dependency on the specific class being instantiated.

#### ****Examples:****

* **Singleton Pattern**: Ensures a class has only one instance and provides a global point of access to that instance.
* **Factory Method Pattern**: Defines an interface for creating an object, but let’s subclasses alter the type of objects that will be created.
* **Abstract Factory Pattern**: Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
* **Builder Pattern**: Allows for the creation of complex objects by separating the construction process from the actual representation.
* **Prototype Pattern**: Creates new objects by cloning an existing object, allowing the creation of objects with similar properties.

### 2. ****Structural Design Patterns****

Structural design patterns focus on how objects are composed to form larger structures. These patterns help in simplifying the structure of code and improving its scalability and maintainability.

#### ****Key Purpose:****

To ease the design and composition of objects and classes in a way that reduces complexity while maintaining flexibility and scalability.

#### ****Examples:****

* **Adapter Pattern**: Allows incompatible interfaces to work together by providing a wrapper that makes an existing class compatible with another interface.
* **Facade Pattern**: Provides a simplified interface to a complex subsystem, making it easier for clients to interact with the system.
* **Composite Pattern**: Allows you to compose objects into tree-like structures to represent part-whole hierarchies, treating individual objects and compositions of objects uniformly.
* **Decorator Pattern**: Adds new functionality to an object dynamically by wrapping it in another object.
* **Proxy Pattern**: Provides a surrogate or placeholder for another object to control access to it, often used for lazy initialization or access control.

### 3. ****Behavioral Design Patterns****

Behavioral design patterns focus on communication between objects, ensuring that they interact in a flexible and well-defined manner. These patterns are primarily concerned with algorithms, object collaborations, and communication.

#### ****Key Purpose:****

To manage algorithms and the flow of control between objects, improving flexibility and reducing the coupling between objects.

#### ****Examples:****

* **Observer Pattern**: Defines a one-to-many dependency between objects, so that when one object changes state, all its dependents are notified and updated automatically.
* **Strategy Pattern**: Defines a family of algorithms and allows them to be interchangeable at runtime without changing the clients that use them.
* **Command Pattern**: Encapsulates a request as an object, allowing for parameterization of clients with different requests and decoupling the sender from the receiver.
* **State Pattern**: Allows an object to alter its behavior when its internal state changes, making the object appear to change its class.
* **Chain of Responsibility Pattern**: Allows a request to be passed along a chain of handlers, where each handler can process the request or pass it to the next handler in the chain.
* **Template Method Pattern**: Defines the skeleton of an algorithm in a method, allowing subclasses to implement specific steps of the algorithm without changing its structure.

### Factory Design Pattern (Brief Overview)

#### ****Definition:****

The Factory Design Pattern is a creational pattern that provides an interface for creating objects while allowing subclasses to alter the type of objects that will be created. It helps encapsulate the object creation process, making the system more flexible and maintainable.

#### ****Usage:****

* Use when the type of object to be created is unknown beforehand.
* Use when object creation is complex or requires various configurations.
* Decouples client code from specific object types.
* Allows flexibility by providing a simple interface for object creation.

#### ****Structure:****

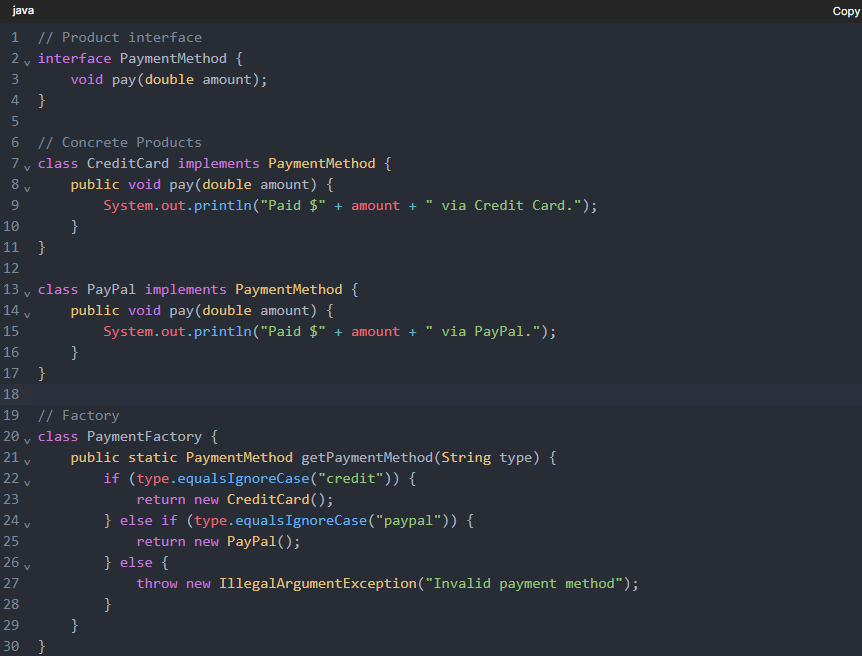
1. **Product:** Interface or abstract class that defines the object to be created.
2. **ConcreteProduct:** Concrete class that implements the Product interface.
3. **Creator (Factory):** Abstract class or interface with a factory method.
4. **ConcreteCreator:** Subclass of Creator that implements the factory method.

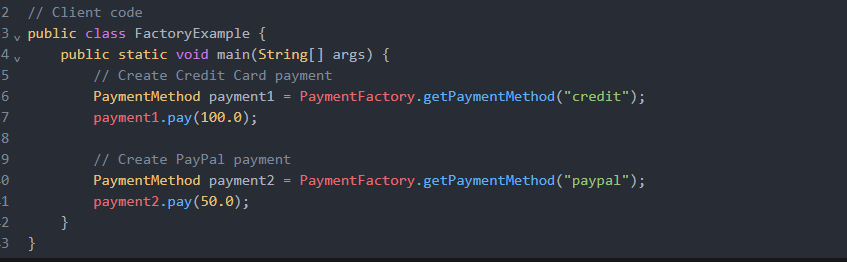
#### ****Advantages:****

* **Decoupling:** Client code is not tightly coupled with the concrete classes.
* **Centralized object creation logic.**
* **Easier code maintenance:** Adding new products doesn’t affect the client code, only the factory.
* **Reusability:** The object creation logic can be reused across different clients.
* **Simplifies complex creation logic.**

#### ****Disadvantages:****

* **Increased complexity:** More classes in the system, which can add overhead in small projects.
* **Scaling issues:** As the number of products grows, the number of factories increases.
* **Limited sub classing flexibility:** Requires specific factory subclasses for each product.
* **Over-engineering:** May be unnecessary for simple creation logic, making the design overly complex.





### Abstract Factory Pattern (Brief Overview)

#### ****Definition:****

The **Abstract Factory Pattern** is a creational design pattern that provides an interface for creating families of related or dependent objects without specifying their concrete classes. It allows the client to create objects from different product families, ensuring that the created objects are compatible.

#### ****Usage:****

* Use when the system needs to create multiple types of related objects.
* When a client needs to create objects from different families but doesn’t want to deal with the specifics of object creation.
* To provide a way to ensure product compatibility across different families of objects.

#### ****Structure:****

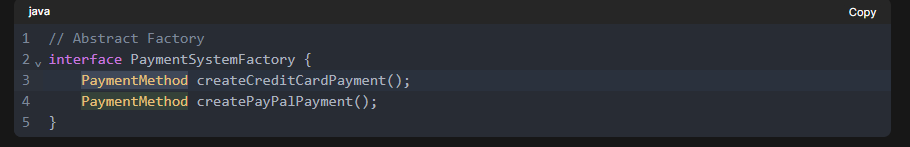
1. **AbstractProduct:** Defines the interface for a type of product.
2. **ConcreteProduct:** Implements the AbstractProduct interface for a specific product.
3. **AbstractFactory:** Defines methods for creating abstract product families.
4. **ConcreteFactory:** Implements the AbstractFactory methods to create specific families of products.

#### ****Advantages:****

* **Product consistency:** Ensures that products from the same family are compatible.
* **Flexibility:** Can easily introduce new product families without modifying client code.
* **Decoupling:** The client code is decoupled from the concrete product classes.

#### ****Disadvantages:****

* **Increased complexity:** Can add more classes and increase complexity, especially when there are many families of products.
* **Harder to modify:** Adding new products to a family requires updating both the factory and the client code.
* **Overhead:** Can be overkill for simple object creation needs.



### 

### 

### ****Builder Design Pattern****

#### ****Definition:****

The **Builder Design Pattern** is a creational design pattern that provides a way to construct a complex object step by step. This pattern allows you to separate the construction of an object from its representation, meaning you can create different types of objects using the same construction process.

#### ****Usage:****

* The Builder pattern is useful when you need to create an object with many parts or configurations, especially when the object can have multiple representations (e.g., a product with different configurations).
* It allows you to construct the object incrementally, making it easier to create a complex object without needing a constructor with a large number of parameters.

#### ****Structure of the Builder Pattern:****

1. **Product**: The object being constructed.
2. **Builder**: Abstract interface that defines the methods for creating parts of the product.
3. **Concrete Builder**: Implements the builder interface and constructs the parts of the product.
4. **Director**: class uses the builder to construct the product in a specific order. It orchestrates the construction process.
5. **Client**: The class that uses the director to construct the object.

#### ****Use Case:****

* **Complex Object Creation**: When an object requires multiple steps to construct and involves multiple components or sub-objects.
* **Example**: Consider a Car object. It may have different parts like wheels, engine, doors, and windows, with variations. A builder can incrementally assemble these parts into a final Car object, allowing flexibility in the configurations without cluttering the main Car class with multiple constructors.

#### ****Example Scenario:****

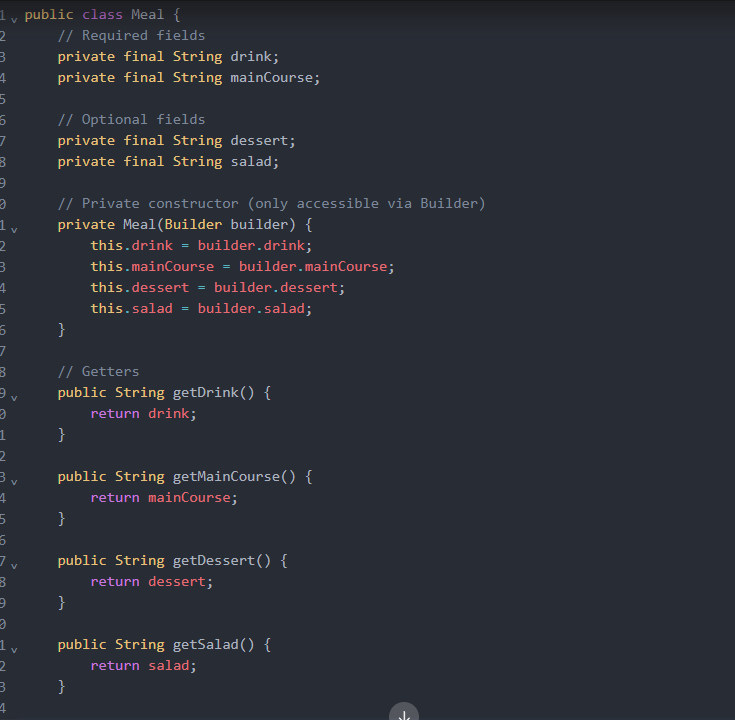
Imagine you're building a **Meal** object in a restaurant that can have different combinations of **Starter**, **Main Course**, and **Dessert**. The builder would allow customers to construct different meal configurations without having to create multiple constructors.

#### ****Advantages:****

1. **Separation of Concerns**: Builder pattern separates the construction logic of an object from its representation, making code more modular and easier to maintain.
2. **Flexibility**: It allows the creation of complex objects with different representations using the same building process.
3. **Readable Code**: Reduces the need for complex constructors, which can be difficult to manage, especially with a large number of parameters.
4. **Immutable Objects**: The pattern often produces immutable objects, improving safety and consistency in multi-threaded environments.

#### ****Disadvantages:****

1. **Increased Complexity**: It introduces additional classes (e.g., Director, Builder, Concrete Builder), which can make the design more complex than needed for simple object creation.
2. **More Code**: Since the construction logic is spread across multiple classes, it can result in more code compared to direct instantiation.
3. **Not Always Necessary**: For simple objects with few fields, the Builder pattern may be overkill, as traditional constructors or factory methods might be sufficient.

****

### 

### 

### ****Prototype Design Pattern****

#### ****Definition:****

The **Prototype Design Pattern** is a **creational design pattern** that allows you to create new objects by copying an existing object, known as the prototype. Instead of creating a new instance from scratch, the prototype pattern enables you to clone an object and make necessary modifications, which can be more efficient than creating new objects from scratch, especially for complex objects.

#### ****Usage:****

* When you need to create several identical or similar objects and cloning an existing object is more efficient than constructing a new one.
* When the cost of creating an object is too high (e.g., because of complex setup or resource-intensive processes).
* When the object’s internal state might change dynamically and you want to create a clone with modified properties.

#### ****Structure of the Prototype Pattern:****

1. **Prototype Interface**: This is an abstract interface that defines the cloning operation. Typically, it has a method like clone () which returns a copy of the object.
2. **Concrete Prototype**: This class implements the prototype interface and provides the specific cloning implementation.
3. **Client**: The client class is responsible for creating copies of objects using the prototype.

#### ****Advantages:****

1. **Efficiency**: Cloning an object can be more efficient than constructing it from scratch, especially when the object requires expensive setup or complex initialization.
2. **Customization**: You can create new objects based on existing prototypes and customize only the necessary fields or properties.
3. **Reduced Object Creation Cost**: It reduces the need to repeatedly recreate identical or similar objects, saving resources.

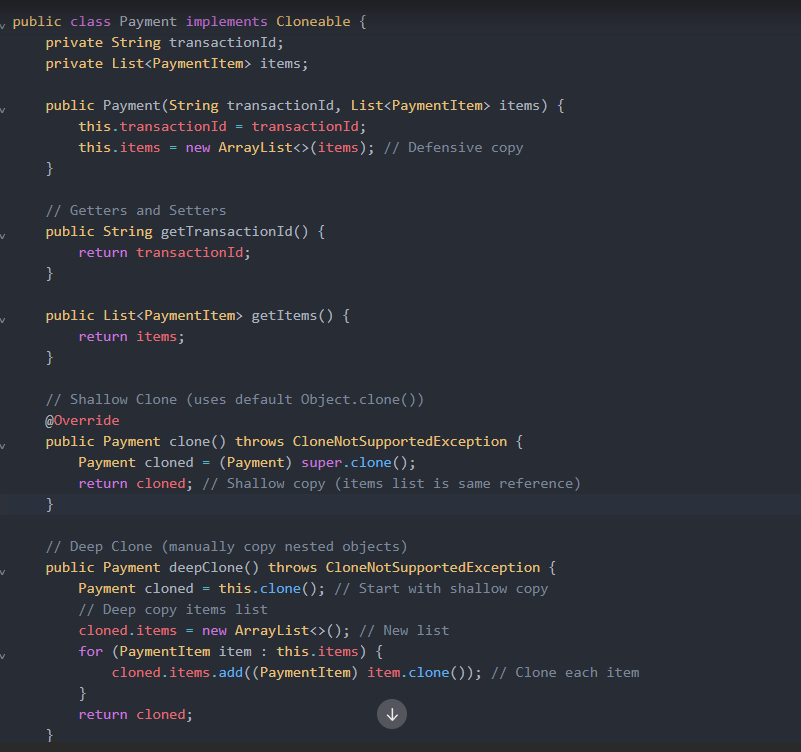
#### ****Disadvantages:****

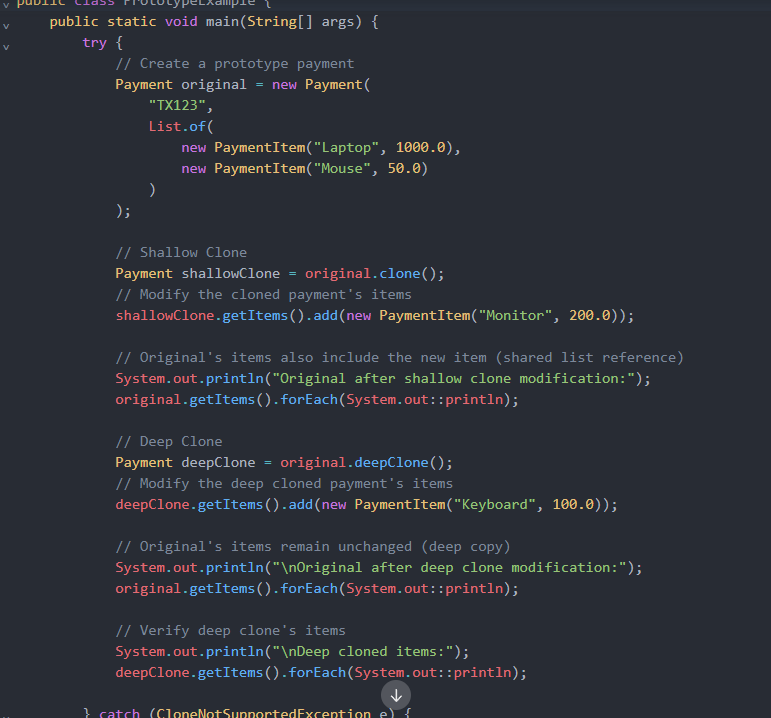
1. **Complexity**: The prototype pattern might introduce unnecessary complexity when the objects to be cloned are simple and don’t require frequent cloning.
2. **Deep Copy vs. Shallow Copy**: Depending on the design, the clone method can either perform a shallow copy (copying references to objects) or a deep copy (copying all referenced objects as well). Managing deep copies can be more complex.

A **deep copy** creates new instances ensuring that the original object and the clone do not share references to any internal states

A **shallow copy** duplicates only the references of the inner objects, not the actual objects themselves. This means that if you modify an internal object in the cloned prototype, it will also affect the original prototype.

****

****

****

### Singleton Design Pattern (Brief Overview)

#### ****Definition:****

The **Singleton Design Pattern** is a creational pattern that ensures a class has only one instance and provides a global point of access to that instance. It restricts the instantiation of a class to just one object, ensuring consistent access to that object throughout the application.

#### ****Usage:****

* Use when you need to control access to a shared resource, such as a configuration manager or a database connection.
* When only one instance of a class is required to coordinate actions across the system (e.g., logging, caching).
* To prevent the creation of multiple instances of a class, which might be inefficient or cause issues in the system.

#### ****Structure:****

1. **Singleton:** The class itself that is responsible for creating and managing its own unique instance.
2. **Instance:** A static member variable holds the single instance of the class.
3. **Access method:** A static method (often called getInstance()) to access the unique instance.

#### ****Advantages:****

* **Global access point:** Ensures that there is a single point of access to the instance.
* **Controlled access:** You can control how the instance is created and accessed, preventing unnecessary instantiations.
* **Lazy initialization:** The instance can be created only when needed, saving resources if the instance is never used.

#### ****Disadvantages:****

* **Global state:** Can lead to hidden dependencies and make unit testing harder since it introduces a global state.
* **Concurrency issues:** In multithreaded applications, special care needs to be taken to ensure that only one instance is created (e.g., thread safety).
* **Overuse:** Over-relying on the Singleton pattern can lead to design issues, as it may hinder modularity and make code harder to maintain.

### 

### Facade Design Pattern (Brief Overview)

#### ****Definition:****

The **Facade Design Pattern** is a structural design pattern that provides a simplified interface to a complex subsystem. It hides the complexities of the system and exposes a higher-level interface that makes the subsystem easier to use. The facade acts as a front-facing interface that delegates client requests to appropriate objects within the subsystem.

#### ****Usage:****

* Use when you want to provide a simplified interface to a complex set of classes or subsystems.
* When you want to reduce the dependencies between clients and the subsystems they use.
* To make a complex system easier to use and understand by abstracting the interactions with the subsystems.

#### ****Structure:****

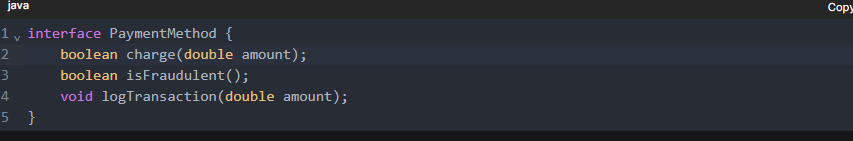
1. **Facade:** The class that provides a simplified interface to the complex subsystem.
2. **Subsystem Classes:** The individual classes or components of the complex subsystem that the facade interacts with.
3. **Client:** The code that interacts with the facade, using the simplified interface

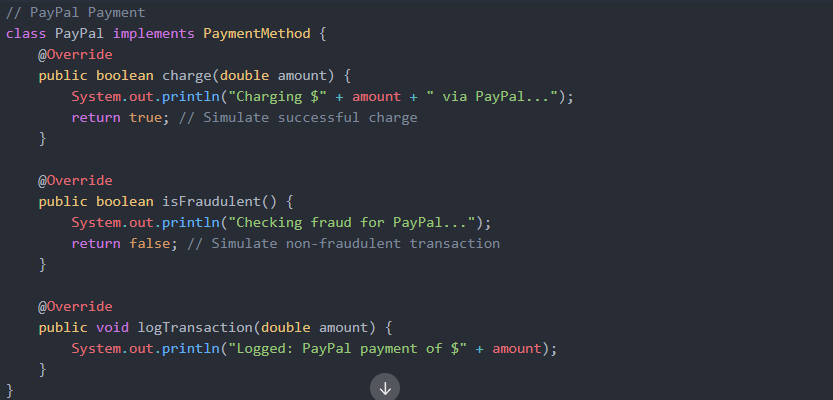
#### ****Advantages:****

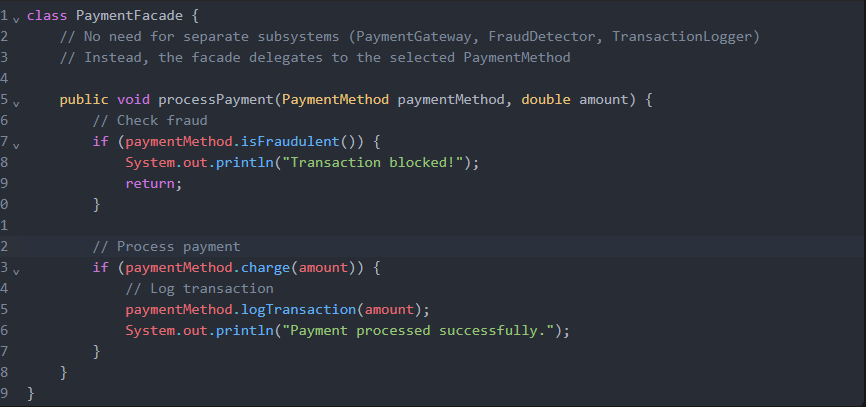
* **Simplified Interface:** Reduces the complexity of interacting with a complex system by providing a unified interface.
* **Loose Coupling:** The client code interacts with the facade, not the subsystem components directly, making it easier to change or replace parts of the subsystem without affecting the client.
* **Improved Readability:** By abstracting the details, the code becomes more readable and understandable.

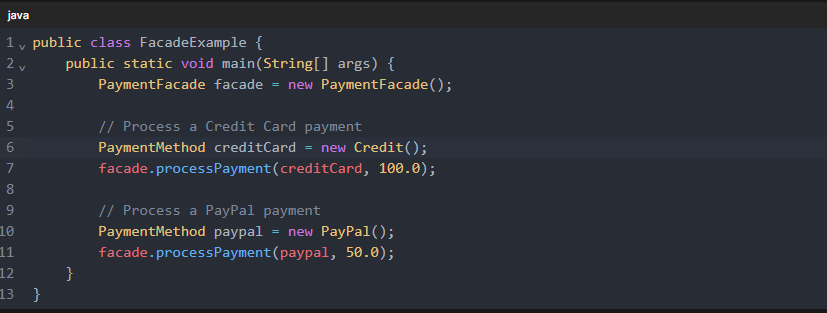
#### ****Disadvantages:****

* **Over-simplification:** If the facade is too generic, it may limit access to specific subsystem functionality that clients might need.

****

****

****

****

**Decorator pattern**

**Purpose**

Dynamically adds responsibilities or behaviors (features) to an object without affecting other objects of the same class at runtime.

Provides a flexible alternative to sub classing for extending functionality.

**Structure**

Component : The interface or abstract class defining the core functionality.

Concrete Component : The base class implementing the component interface.

Decorator: A wrapper class that extends the component interfaces and adds new behavior.

Concrete Decorators: Specific decorators that add additional functionality.

**Use Case**

When you want to add responsibilities to individual objects dynamically and transparently.

When you need to extend the functionality of objects at runtime without using inheritance.

**Advantages**

Adheres to the open/closed principle: You can add new decorators without modifying existing code.

More flexible than inheritance because you can combine multiple decorators.

Avoids feature-laden classes with lots of optional features.

**Disadvantages**

Can lead to a large number of small classes if there are many decorators

Code can become harder to read and debug due to multiple layers of wrapping

// Component Interface

interface Coffee {

String getDescription();

double getCost();

}

// Concrete Component

class SimpleCoffee implements Coffee {

public String getDescription() { return "Simple Coffee"; }

public double getCost() { return 1.0; }

}

// Decorator Abstract Class

abstract class CoffeeDecorator implements Coffee {

protected **Coffee** decoratedCoffee;

public CoffeeDecorator(Coffee c) {

this.decoratedCoffee = c;

}

public String getDescription() {

return decoratedCoffee.getDescription();

}

public double getCost() {

return decoratedCoffee.getCost();

}

}

// Concrete Decorators

class MilkDecorator extends CoffeeDecorator {

public MilkDecorator(Coffee c) {

super(c); // Pass the Coffee object to the parent class constructor

}

public String getDescription() {

return decoratedCoffee.getDescription() + ", Milk";

}

public double getCost() {

return decoratedCoffee.getCost() + 0.5;

}

}

class SugarDecorator extends CoffeeDecorator {

public SugarDecorator(Coffee c) {

super(c); // Pass the Coffee object to the parent class constructor

}

public String getDescription() {

return decoratedCoffee.getDescription() + ", Sugar";

}

public double getCost() {

return decoratedCoffee.getCost() + 0.2;

}

}

**State Design Pattern**

**Purpose**

Allows an object to alter its behavior when it’s internal state changes.

The object appears to change its class (feature) based on its state.

**Structure**

Context: The class whose behavior depends on its state.

State Interface: Defines the common interface for all concrete states.

Concrete States: Implementations of the state interface, each representing a specific state and its associated behavior.

**Use Case**

When an object's behavior depends on its state, and you want to avoid complex conditional logic.

When you need to model finite state machines.

**Advantages**

Encapsulates state-specific behavior, making the code more modular and easier to maintain.

Eliminates long if-else or switch-case statements by delegating behavior to state objects.

Follows the single responsibility principle by isolating state-specific logic.

**Disadvantages**

Can lead to an increase in the number of classes if there are many states.

May require additional effort to manage transitions between states.

// State interface defines the common behavior for all states

interface BookState {

void nextState(); // Transition to the next state

void printStatus(); // Print the current status of the book

}

// State A

class StateA implements BookState {

private String name = "A";

@Override

public void nextState() {

System.out.println("Transitioning from " + name + " to B");

// Transition logic: Move to State B

}

@Override

public void printStatus() {

System.out.println("Current state: " + name);

}

}

class LibraryBook {

private BookState currentState;

public LibraryBook(BookState initialState) {

this.currentState = initialState;

}

public void printCurrentStatus() {

currentState.printStatus();

}

public void goNextSate() {

currentState. nextState ();

}

}

**Strategy Design Pattern**

**Purpose**

Defines a family of algorithms or behaviors, encapsulates each one, and makes them interchangeable.

Allows the client to dynamically select an algorithm at runtime without modifying the code.

**Structure**

Context : The class that uses the strategy (algorithm).

Strategy Interface : Defines the common interface for all concrete strategies.

Concrete Strategies : Implementations of the strategy interface, each representing a specific algorithm or behavior.

**Use Case**

When you need to define multiple interchangeable algorithms or behaviors.

When you want to avoid conditional statements (if-else) by encapsulating behaviors in separate classes.

**Advantages**

Promotes open/closed principle: You can add new strategies without modifying existing code.

Reduces code duplication by separating algorithms from the context.

Makes the system more flexible and easier to maintain.

**Disadvantages**

Can lead to an increase in the number of classes if there are many strategies.

Clients must understand the differences between various strategies to choose the right one.

**Template pattern**

**Purpose**

The **Template Method** pattern is a behavioral design pattern that defines the structure of an algorithm in a **method**, **deferring** some steps to subclasses. This allows subclasses to redefine certain steps of the algorithm without changing its overall structure.

**Structure**

 Abstract **Class (or Base Class)**:

* Contains the template **method** that defines the sequence of steps in the algorithm.
* Defines some steps **as abstract methods** that must be implemented by concrete subclasses.

 Concrete **Subclasses**:

* Implement the abstract steps defined in the base class.
* These subclasses override the specific steps of the algorithm while following the structure defined by the base class.

**Usage**

Ensure that **certain steps of the algorithm remain unchanged**, but still allow customization for parts of the process.

**Advantage**

**Control over Algorithm**: The base class controls the structure of the algorithm, ensuring that the process follows a well-defined order.

**Disadvantage**

**Tight Coupling**: The base class and its subclasses are tightly coupled because the subclasses rely on the template method. Changes in the base class may require changes in the subclasses

using System;

// Abstract class defining the template method

public abstract class Pizza

{

// Template method defining the skeleton of making a pizza

public void MakePizza()

{

PrepareDough();

AddSauce();

AddToppings();

BakePizza();

CutPizza();

BoxPizza();

}

// Concrete method that can be shared by all pizzas

private void PrepareDough()

{

Console.WriteLine("Preparing the dough.");

}

private void AddSauce()

{

Console.WriteLine("Adding sauce.");

}

private void BakePizza()

{

Console.WriteLine("Baking pizza.");

}

private void CutPizza()

{

Console.WriteLine("Cutting the pizza.");

}

private void BoxPizza()

{

Console.WriteLine("Boxing the pizza.");

}

*// Abstract method for adding toppings, to be implemented by subclasses*

protected abstract void AddToppings();

}

// Concrete subclass for Margherita pizza

public class MargheritaPizza : Pizza

{

protected override void AddToppings()

{

Console.WriteLine("Adding mozzarella cheese and basil.");

}

}

// Concrete subclass for Pepperoni pizza

public class PepperoniPizza : Pizza

{

protected override void AddToppings()

{

Console.WriteLine("Adding pepperoni slices and cheese.");

}

}

// Concrete subclass for Veggie pizza

public class VeggiePizza : Pizza

{

protected override void AddToppings()

{

Console.WriteLine("Adding bell peppers, onions, and olives.");

}

}

public class Program

{

public static void Main(string[] args)

{

Console.WriteLine("Making a Margherita pizza:");

Pizza margheritaPizza = new MargheritaPizza();

margheritaPizza.MakePizza();

Console.WriteLine("\nMaking a Pepperoni pizza:");

Pizza pepperoniPizza = new PepperoniPizza();

pepperoniPizza.MakePizza();

Console.WriteLine("\nMaking a Veggie pizza:");

Pizza veggiePizza = new VeggiePizza();

veggiePizza.MakePizza();

}

}

Different between Strategy & Façade & Factory

**Strategy**: Focuses on **swapping algorithms** (payment methods) at runtime.

**Facade**: Focuses on **simplifying complexity** of multiple subsystems.

**Factory**: Focuses on **centralizing object creation** while hiding implementation details