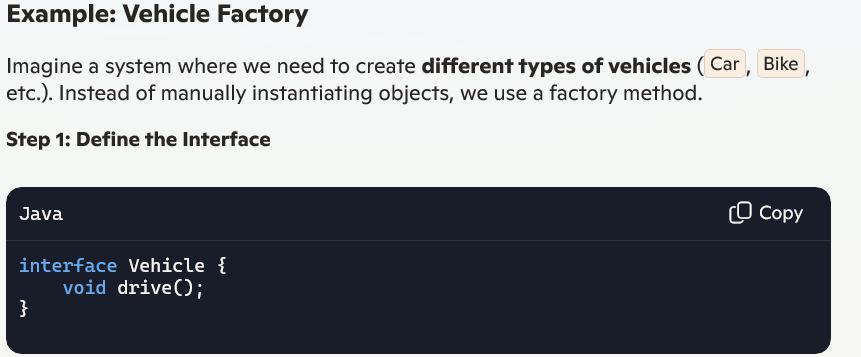
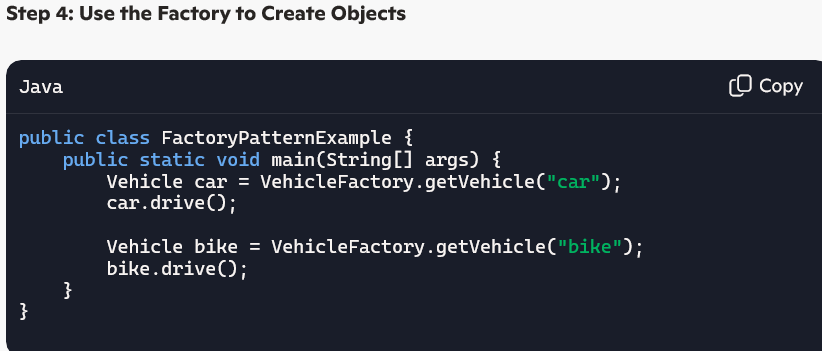


**Factory Design Patter:**

A Factory Pattern or Factory Method Pattern says that just define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate. In other words, subclasses are responsible to create the instance of the class.

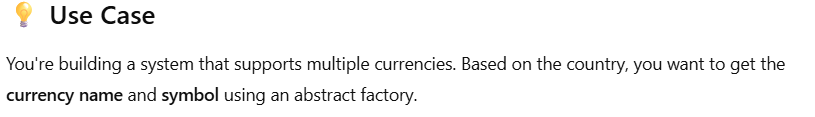






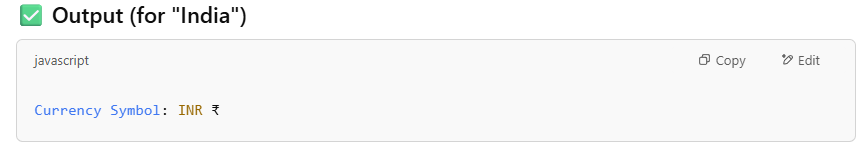
**Abstract Factory Pattern:**

The **Abstract Factory Pattern** is a **creational design pattern** used when you need to create **families of related objects** without specifying their concrete classes.



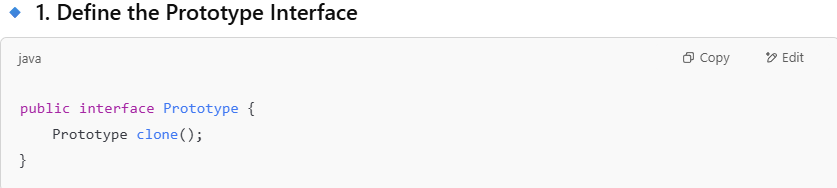


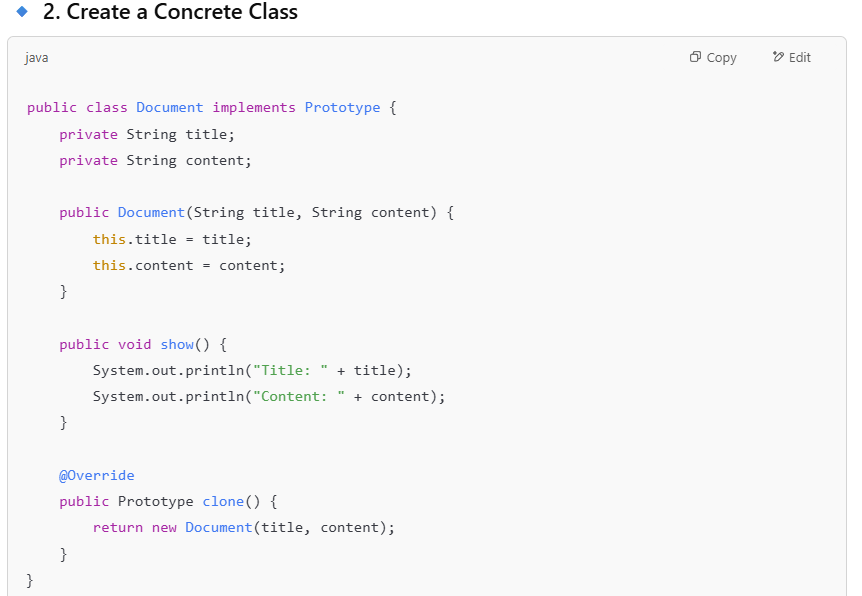


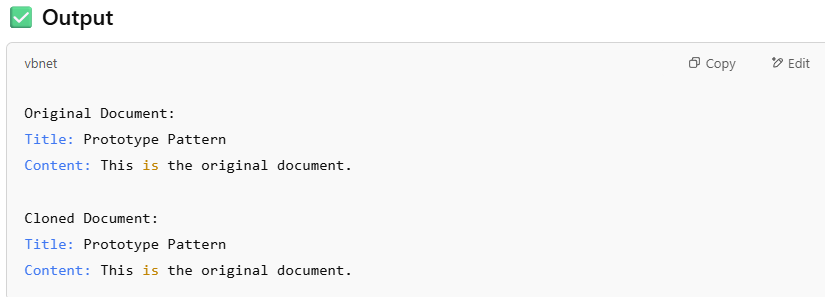
**Prototype Design Pattern:**

The **Prototype Pattern** is a creational design pattern used to **clone objects** instead of creating new instances. This improves performance when object creation is expensive. It uses a clone() method to create a copy of an object



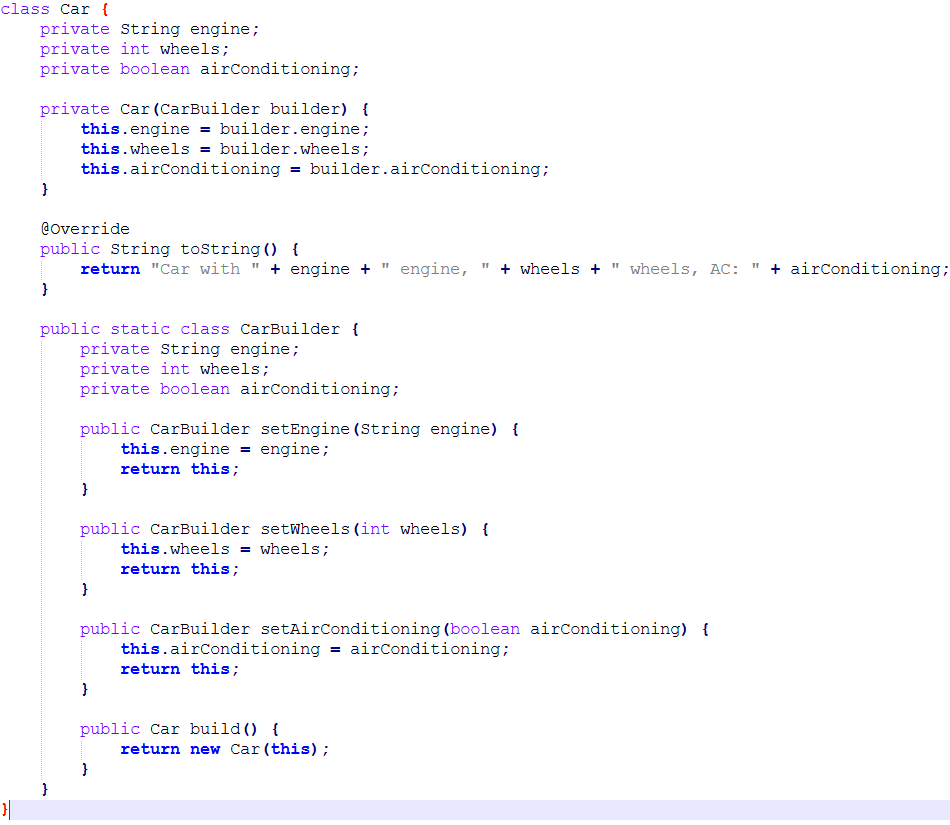


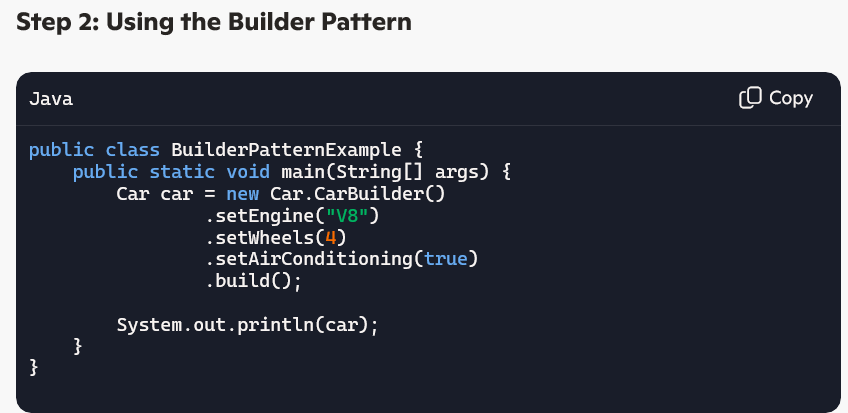




**Builder Design Pattern:**

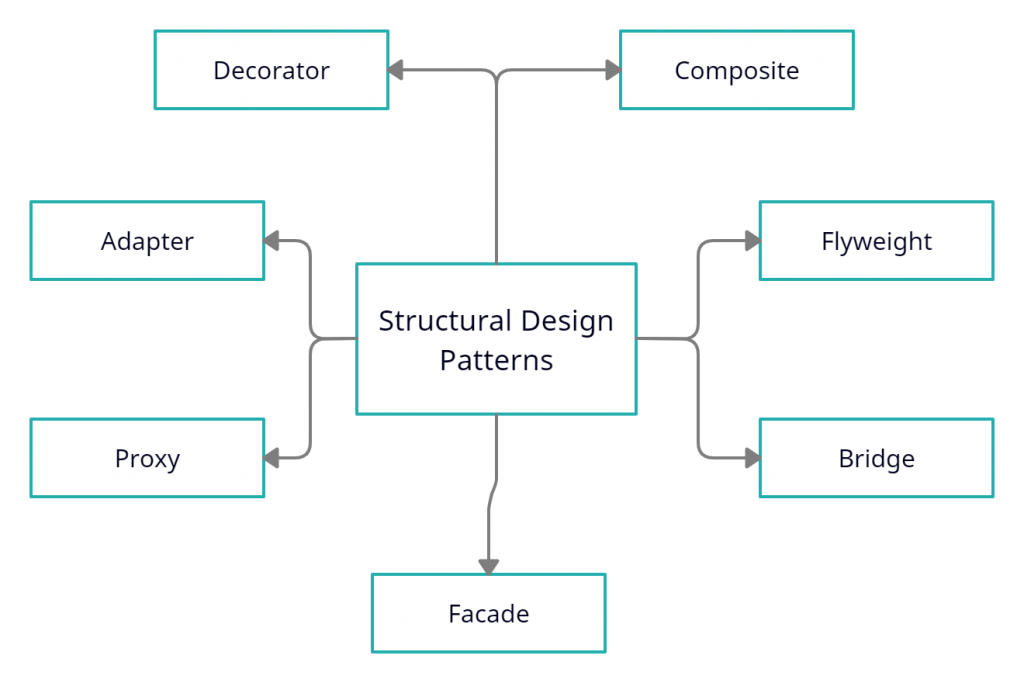
The Builder Pattern is a creational design pattern that lets you construct complex objects step by step. It separates the construction of an object from its representation.





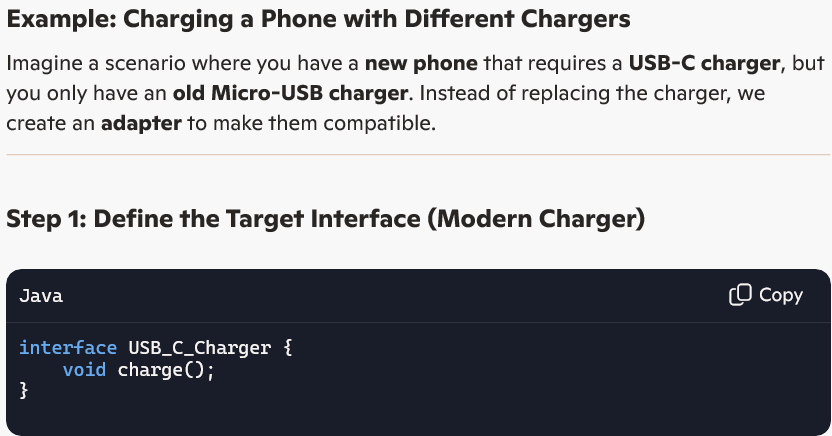
# Structural Design Patterns:

The structural design patterns simplifies the structure by identifying the relationships. Structural design patterns are concerned with how classes and objects can be composed

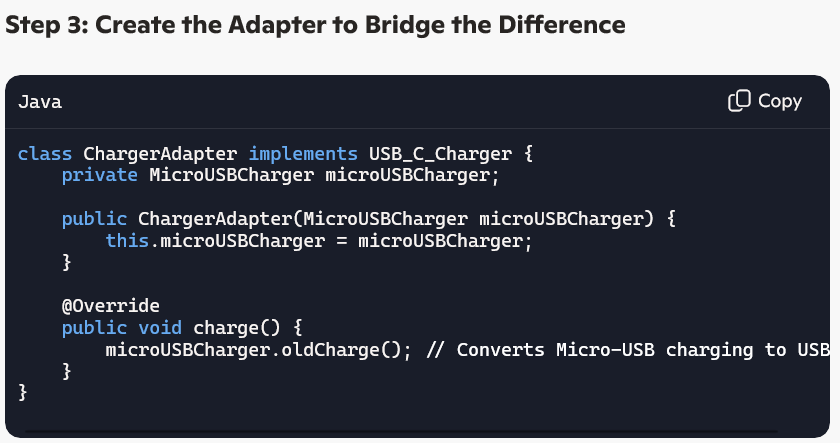


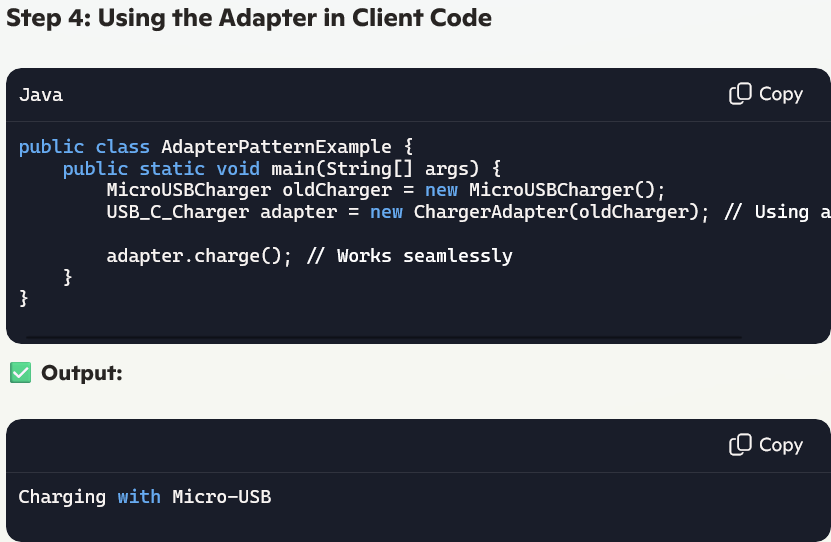
# Adaptor Design Pattern:

The **Adapter Pattern** is a **structural design pattern** that allows two incompatible interfaces to work together **without modifying their source code**.



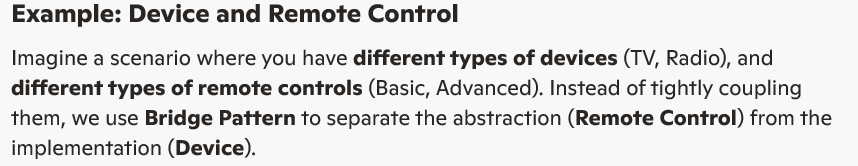


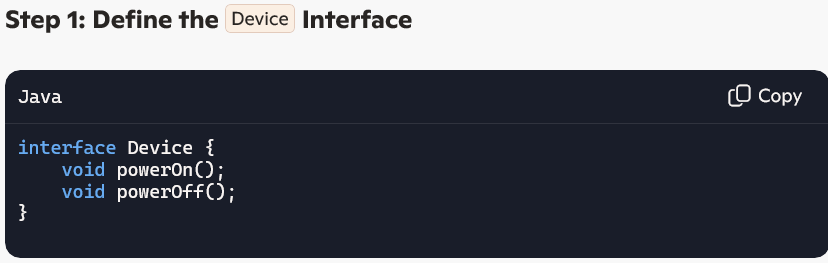




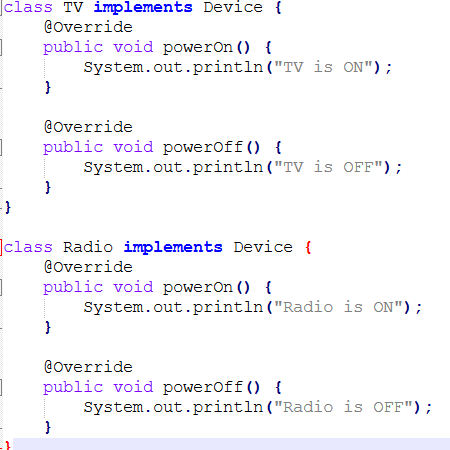
# Bridge Design Pattern:

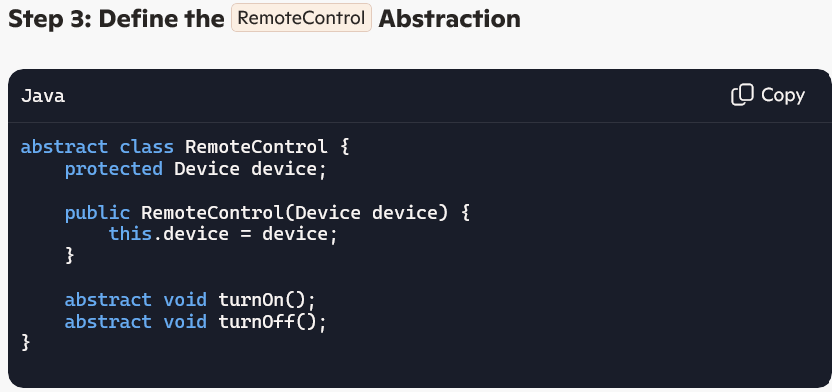
The **Bridge Pattern** is a **structural design pattern** that helps separate **abstraction** from its **implementation**, allowing changes independently.

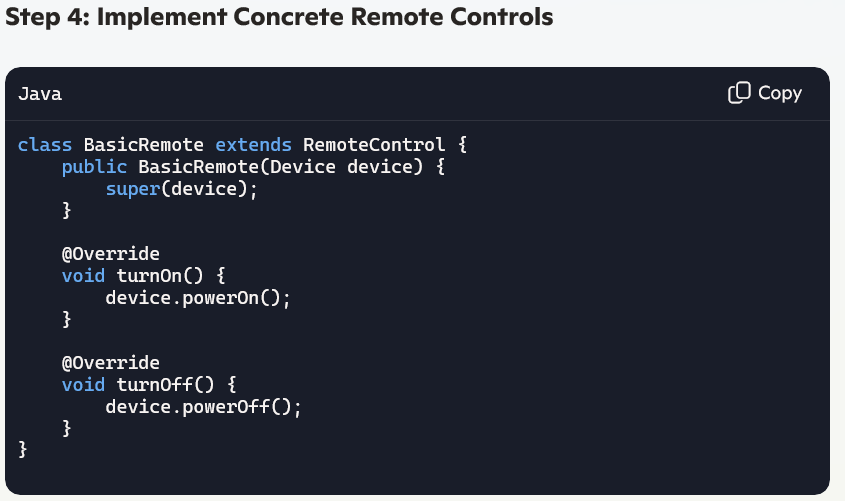


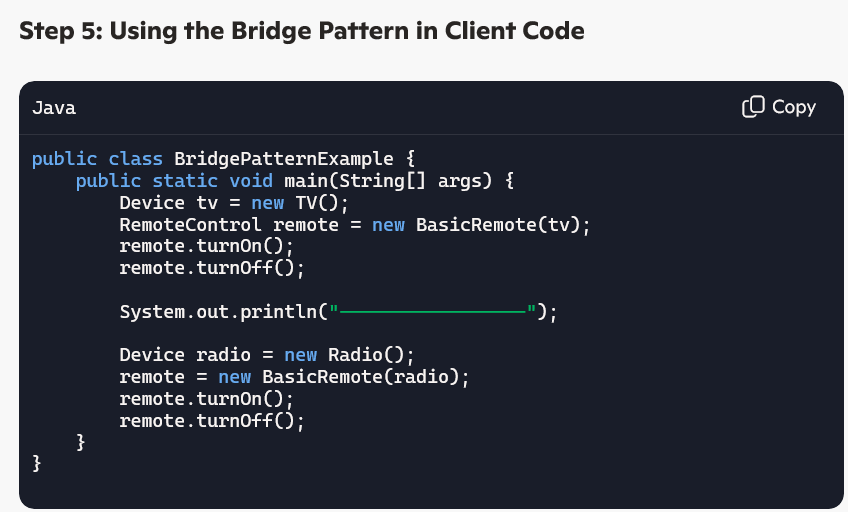














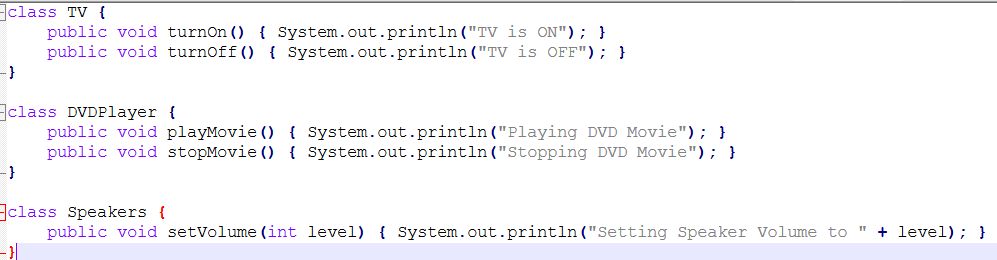
# **Façade Design Pattern:**

The **Facade Pattern** is a **structural design pattern** that provides a **simplified interface** to complex subsystems. It hides the internal complexity and makes a system easier to use.

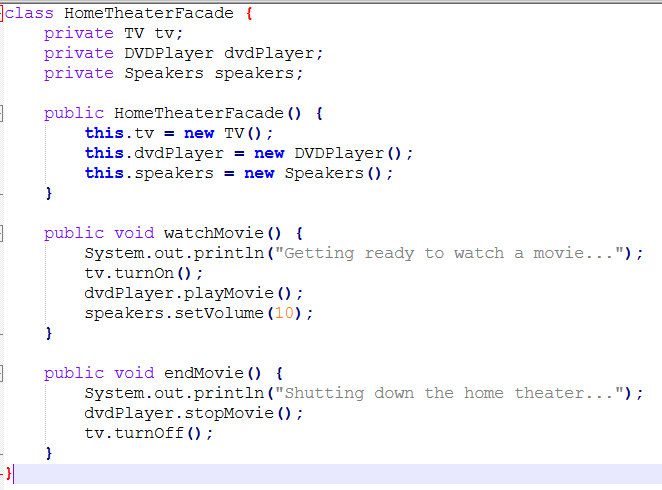
**Example: Home Theater System**

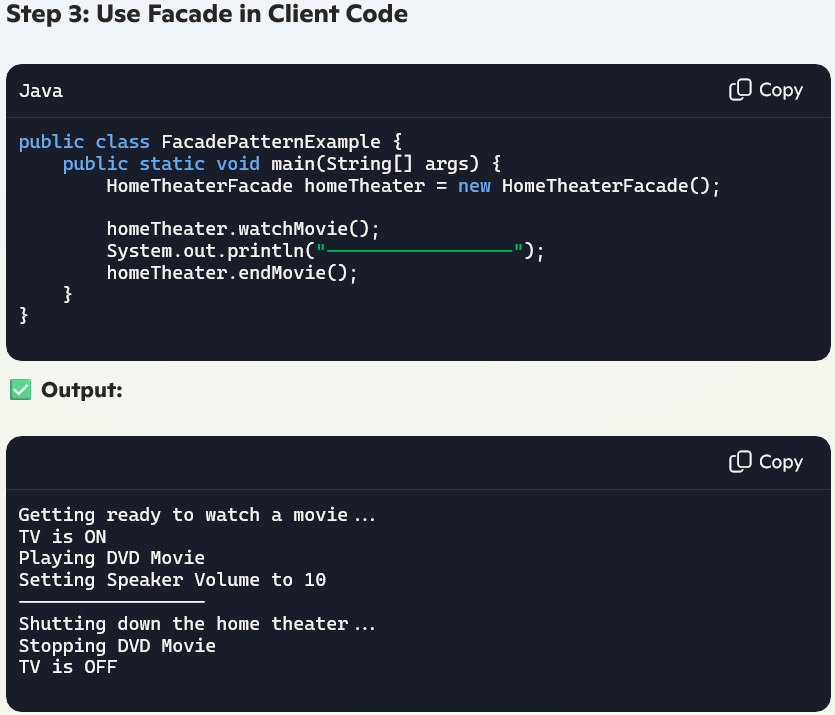
Imagine a **home theater system** with multiple components (**TV, Speakers, DVD Player, etc.**). Instead of calling multiple methods for each device, we create a **Facade** that simplifies the user experience.





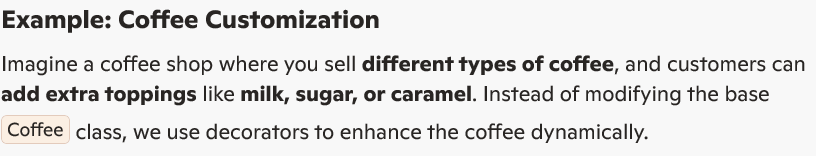






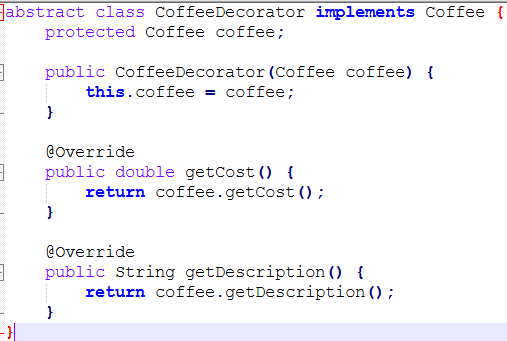
# **Decorator Design Pattern:**

The **Decorator Pattern** is a **structural design pattern** used to dynamically add behavior or responsibilities to objects **without modifying their structure**. It helps **extend object functionality** without altering the existing class.

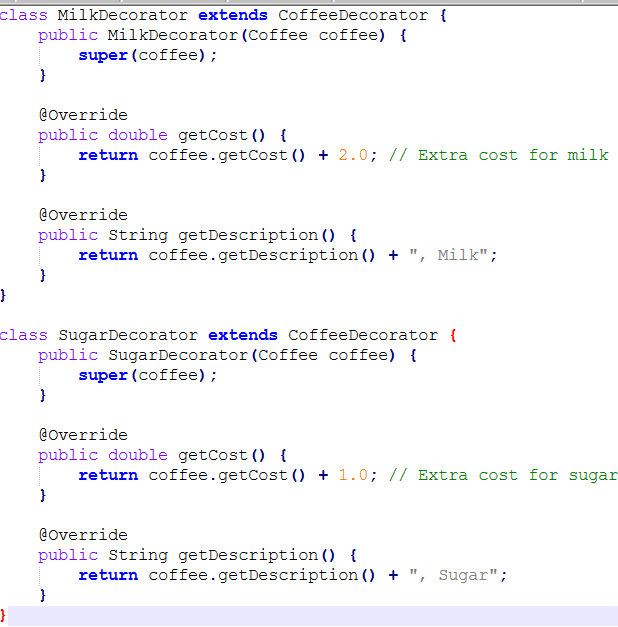




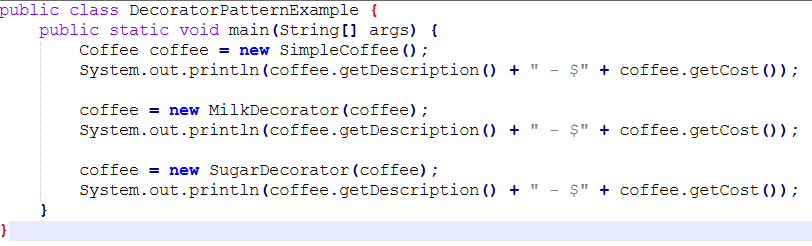


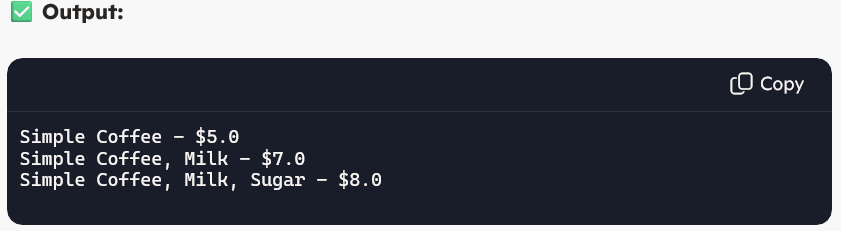






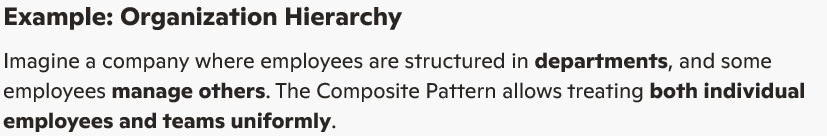


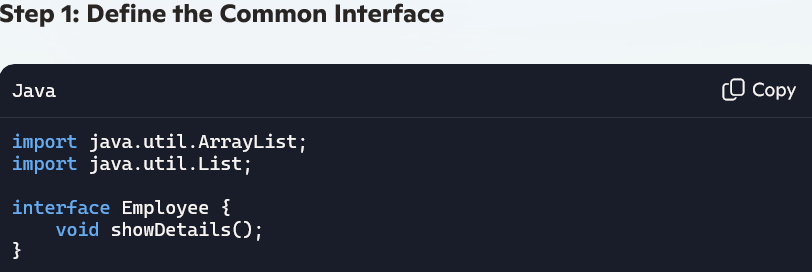


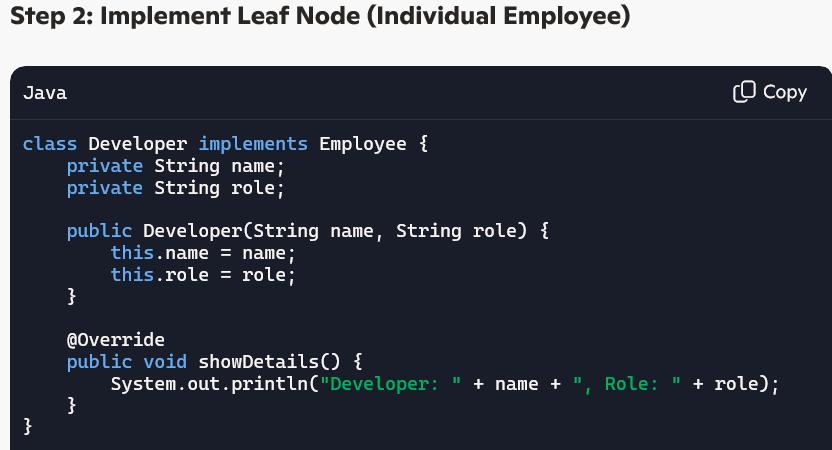


# Composite Design Pattern:

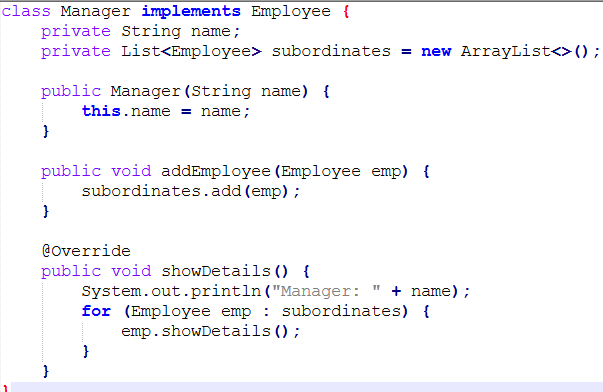
The **Composite Pattern** is a **structural design pattern** that treats **individual objects and groups of objects** uniformly. It helps represent **tree-like hierarchies**, such as a **file system, organizational structure, or UI components**

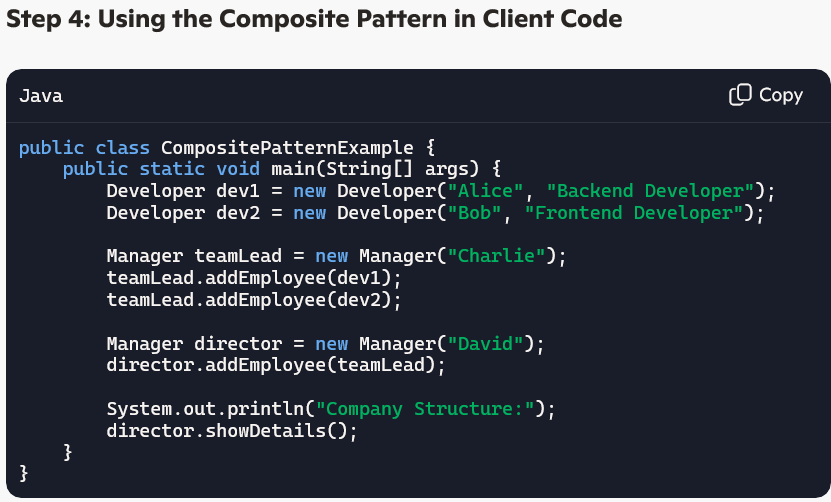


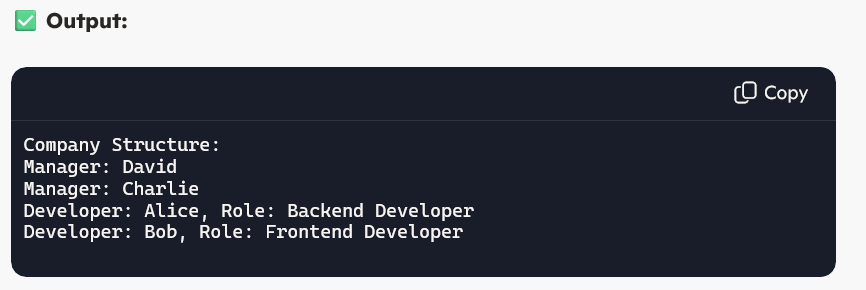






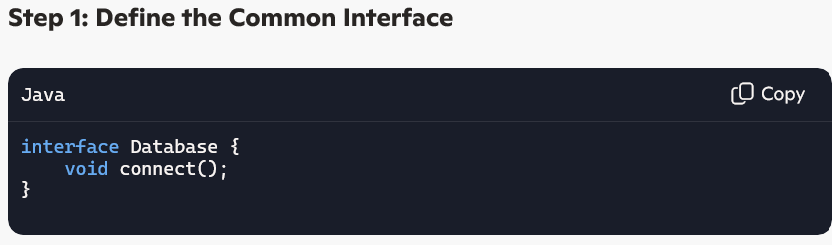


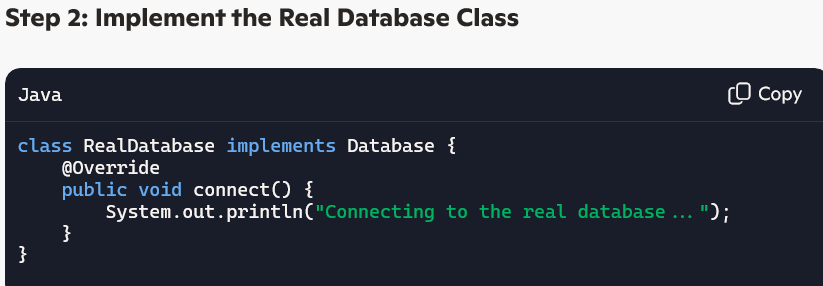




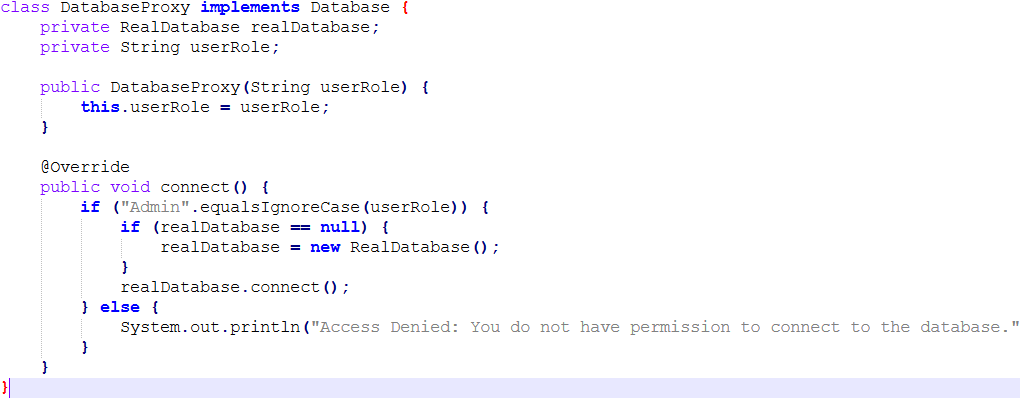
# Proxy Design Pattern:

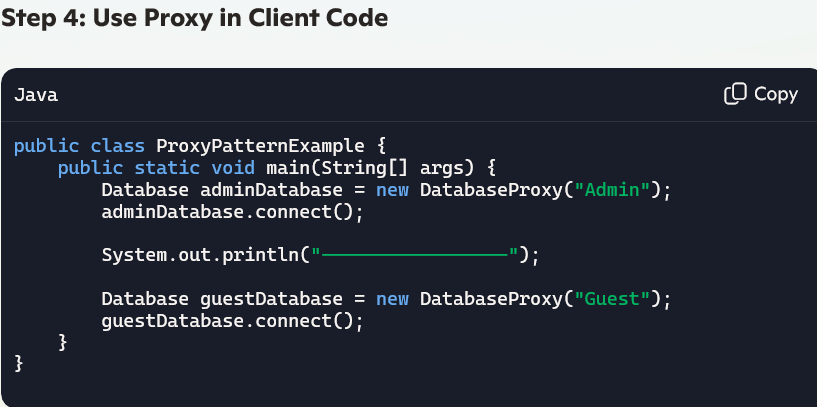
The **Proxy Pattern** is a **structural design pattern** that provides a surrogate or placeholder object to control access to another object. It is useful when you need **lazy initialization, security control, or logging before accessing a resource-heavy object**.

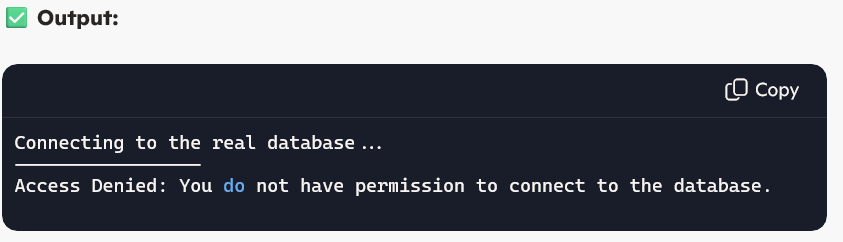






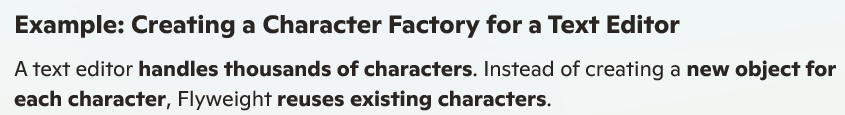




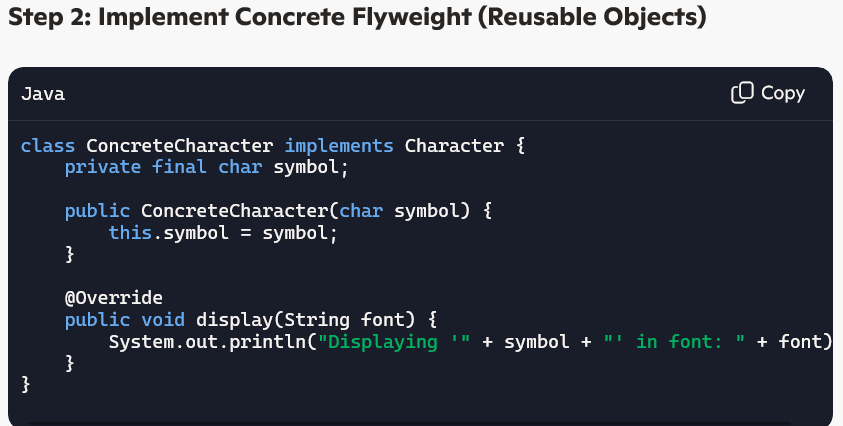


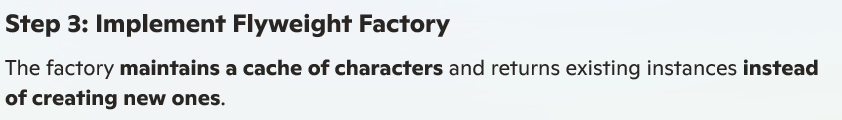
# Fly Weight Design Pattern:

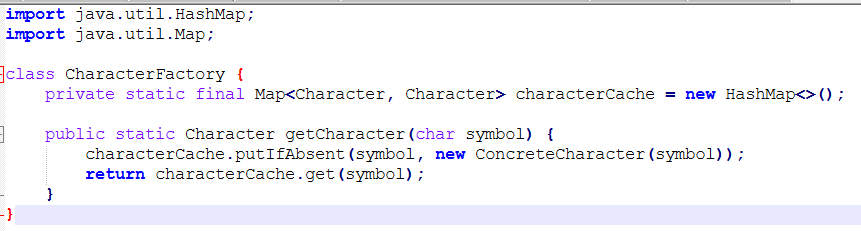
The **Flyweight Pattern** is a **structural design pattern** that minimizes **memory usage** by sharing **common objects** instead of creating **new instances**. It is particularly useful for applications with **large numbers of similar objects**, such as **text editors, gaming sprites, or caching mechanisms**.



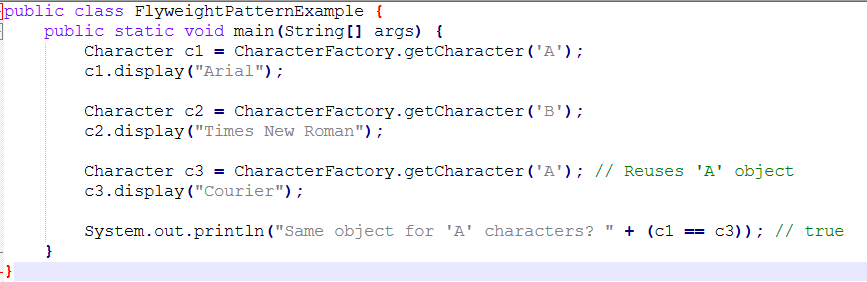


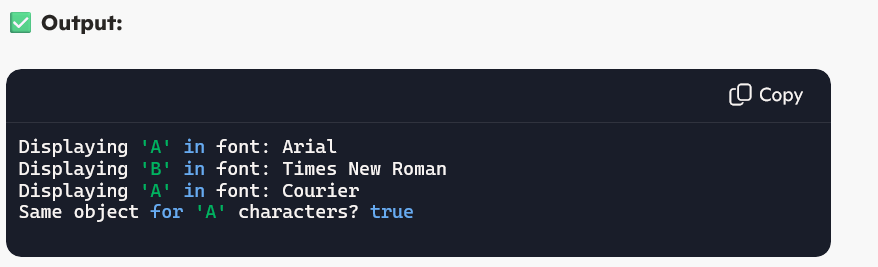






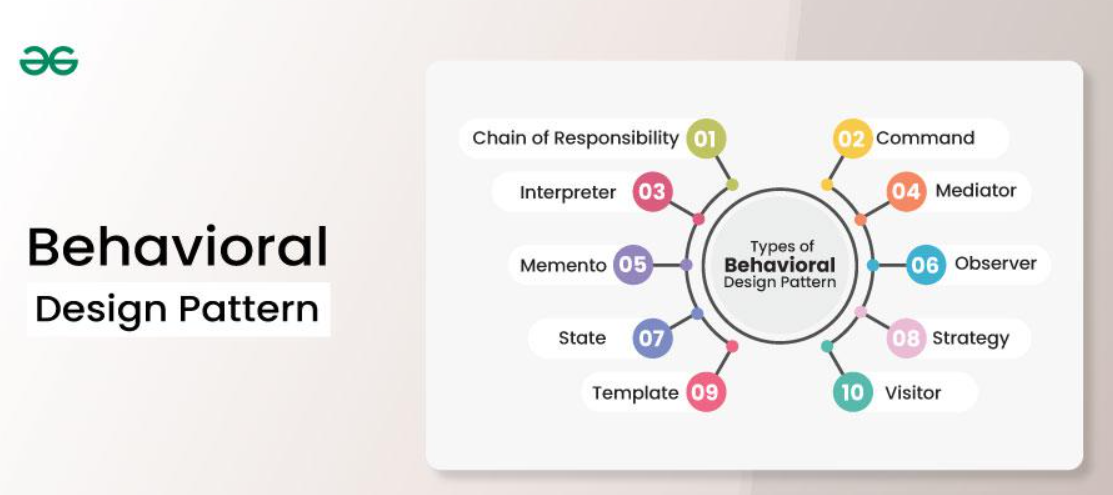






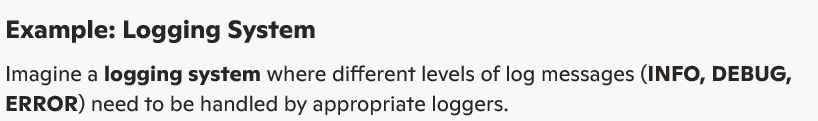
# Behavioural Design Patterns:

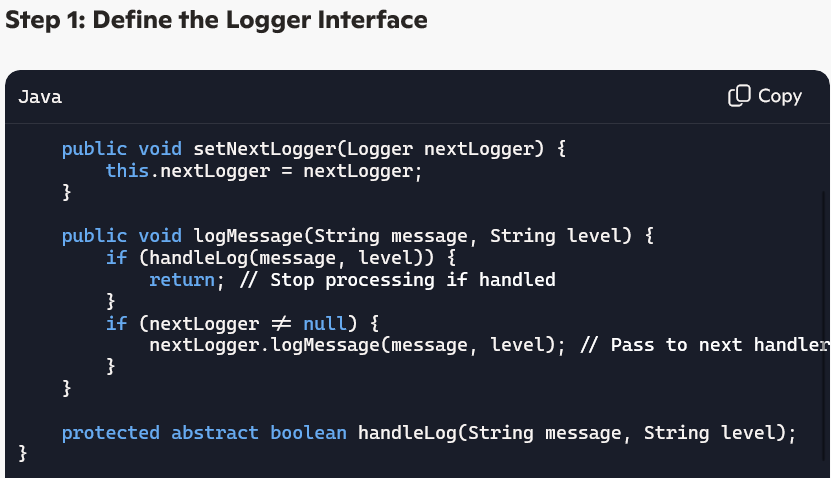
Behavioural design patterns are a category of [design patterns](https://www.geeksforgeeks.org/software-design-patterns/) that focus on the interactions and communication between objects. They help define how objects collaborate and distribute responsibility among them, making it easier to manage complex control flow and communication in a system.



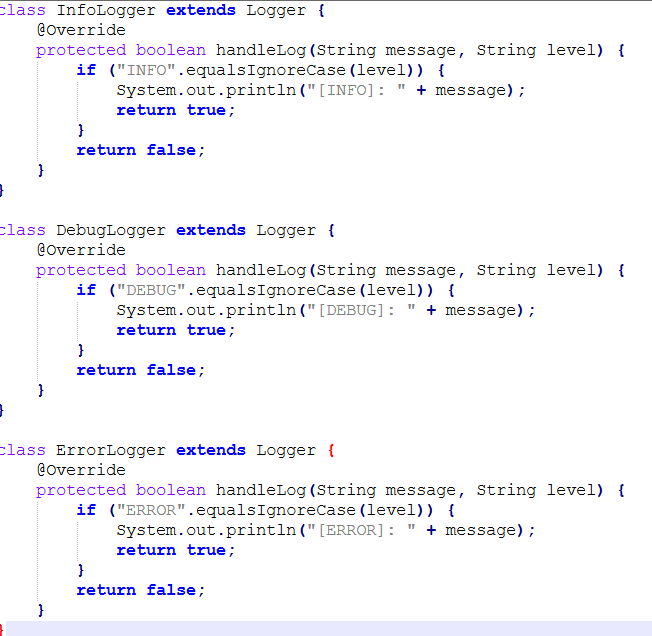
# Chain of Responsibility Pattern:

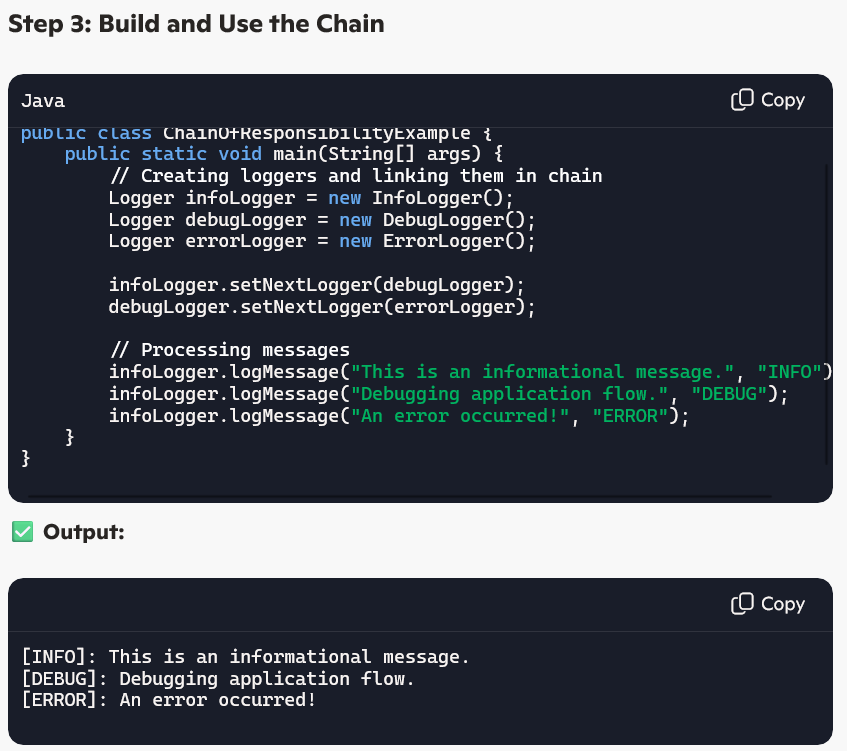
The **Chain of Responsibility Pattern** is a **behavioral design pattern** that allows multiple handlers to process a request in **sequence** until one of them **handles it**. It’s commonly used in **logging frameworks, authentication mechanisms, request validation, and event handling**.



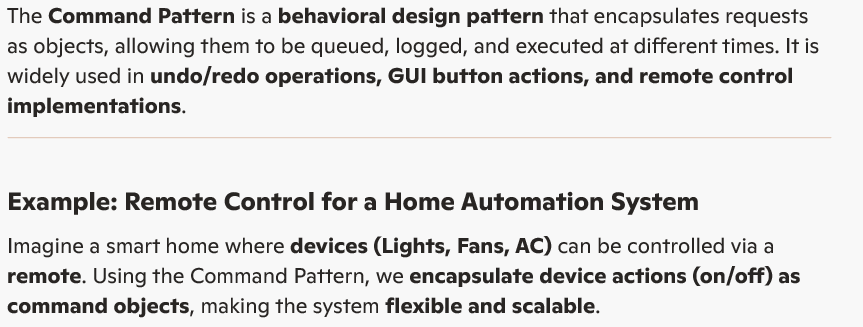




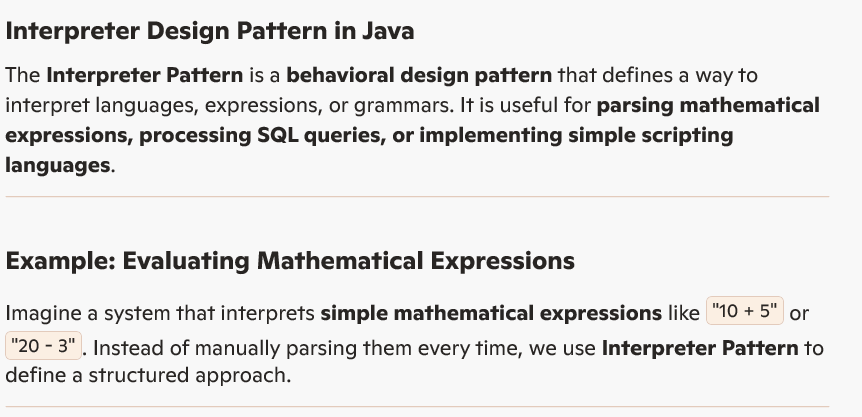




# Command Design Pattern:



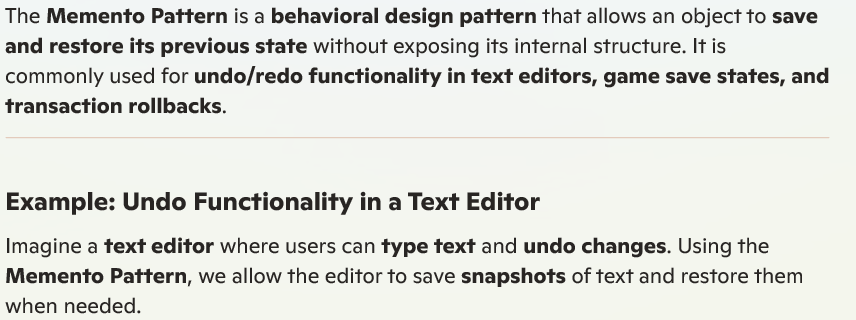
# Interpreter Design Pattern:



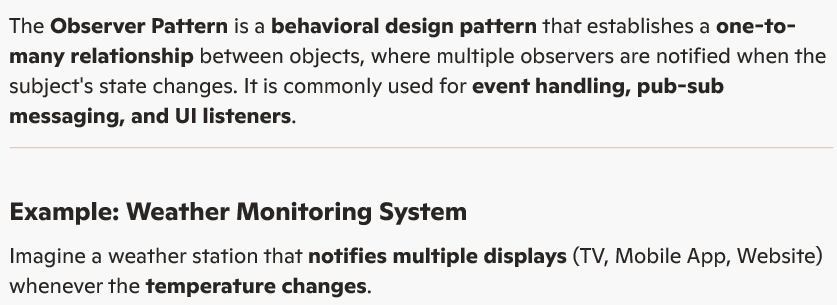
# Mediator Design Pattern:



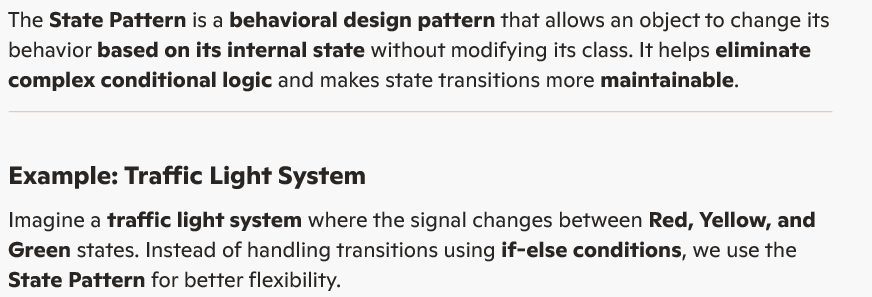
# Memento Design Pattern



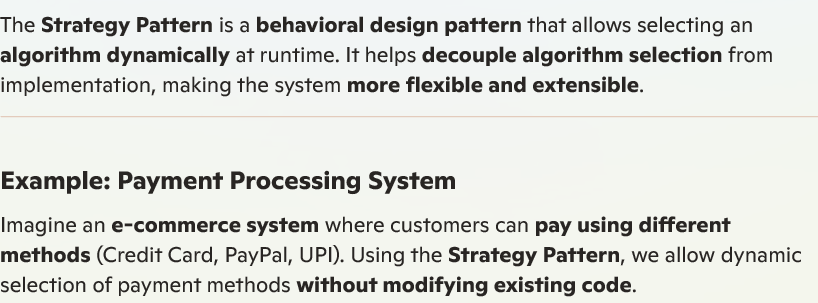
# Observer Method Design Pattern



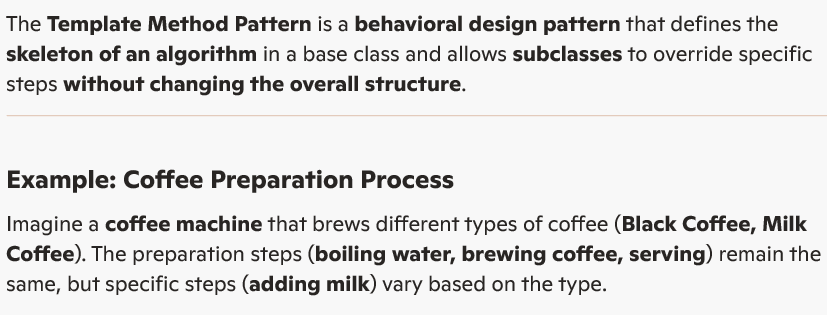
State Method Design Pattern



Strategy Method Design Pattern



Template Method Design Pattern



Visitor Method Design Pattern

