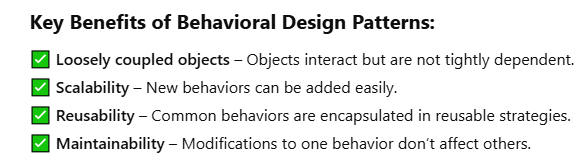
**Behavioral Design Patterns**

**Behavioral Design Patterns** focus on how objects interact with each other. These patterns define communication patterns between objects, making the system **flexible and easy to extend** without modifying the objects themselves.

Behavioral patterns help **delegate responsibilities**, improve **code maintainability**, and **reduce dependencies** between objects.

1. **Chain of Responsibility**
2. **command**
3. Iterator
4. Mediator
5. Memento
6. Observer
7. State
8. Strategy
9. Template



**1:- chain of responsibility**

**Chain of Responsibility** is a behavioral design pattern that lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.

**Chain of Responsibility Pattern in Java & Spring Boot**

**Definition:**

The **Chain of Responsibility** is a **behavioral design pattern** that allows a request to be passed along a chain of handlers. Each handler **processes** the request **or passes it to the next handler** in the chain.

This pattern is useful when multiple objects may handle a request, but the handler is determined dynamically at runtime.

**Use Case in Java & Spring Boot: Request Validation Middleware**

In a Spring Boot application, the **Chain of Responsibility** pattern is useful for implementing **request validation, authentication, or logging middleware**.

**Example Scenario: API Request Validation**

Consider an API request that must go through multiple validation steps before processing:

1. **Check if the request is authenticated**
2. **Check if the request has valid input data**
3. **Check if the request has the required permissions**

**2:- command design pattern**

A command design pattern is a behavioural design pattern from that gang of four patterns. It is used to encapsulate a request as an object and pass it to an invoker the invoker does not know how to service the request from the client. It will take the command and pass it to a receiver who knows how to perform the action typically.

There are five actors in the command design pattern. They are the **command** itself. The **client** the invoker the concrete command that implements the command and a **receiver** who knows how to perform all the actions.

Let's take a look at an **example** to see all these five actors in action. Let's consider a person who is using a television or is watching a television and he uses a remote control typically to perform several operations. But let's simply take the on and off operations and let's see how the command pattern fits in here.Here the person is the client who wants to execute the on and off command on it television, the remote control is the invoker so he uses remote control to invoke a particular command by pressing a button and the commands themselves are the on command and off command that implement a interface called Command which has a execute method.

So the person will wrap this on command passes it to the remote control the remote control will send that command to the television and the television knows how to perform that action based on the command that comes in when it is on.

it will execute the command which is passed in which is the on command execute method. And when it is off it will perform the Execute method of the off command to switch off the television.

Here person is the client remote control is the Invoker command is the command interface on command and off command are the concrete command classes. And finally television is the receiver who knows how to perform the action. The huge advantage of the command pattern is that the invoker which is the client and the remote control are completely decoupled from the receiver.

The person need not touch the television or he need not know how to perform the on and off command he simply uses the remote control and presses the button the remote control also doesn't care how the actions are performed. It simply passes the command to the television. That way they are completely decoupled the invoker does not know the details of the action that needs to be performed.

The receiver here the television can change the implementation of how the on and off should be performed without impacting the remote control and the person.

**Spring Boot Use Case: Background Task Execution**

In a **Spring Boot application**, we can use the **Command Pattern** to decouple and execute **background tasks asynchronously**.

**Example: Processing Orders Asynchronously**

💡 **Use Case:**  
We have an **Order Processing System**, where an OrderService submits orders.  
Instead of processing them synchronously, we will **queue them as commands** and execute in the background.

3:- Iterator design Pattern

## ****Iterator Design Pattern in Java & Spring Boot****

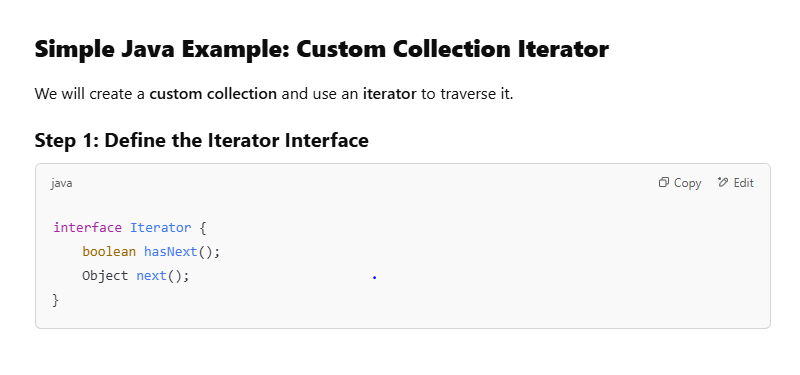
### ****What is the Iterator Design Pattern?****

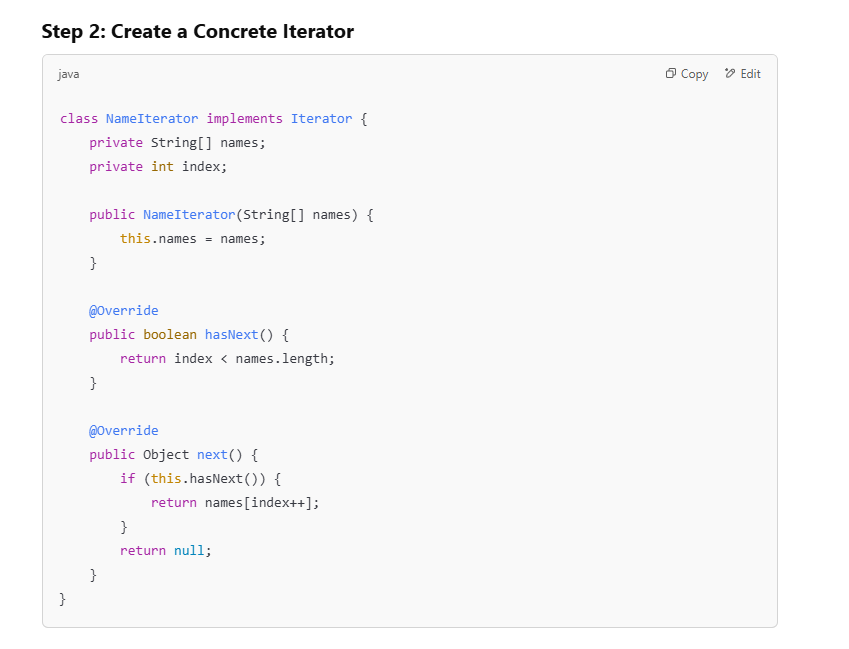
The **Iterator Pattern** is a **behavioral design pattern** that provides a way to **access elements of a collection sequentially without exposing its underlying structure**.

🔹 It allows traversal of a collection **without needing to understand how the collection is implemented**.  
🔹 It provides a **uniform way to iterate through different types of collections** (e.g., lists, sets, trees).  
🔹 It is useful when we want to **iterate over complex data structures** like trees or graphs.

## ****Use Case in Spring Boot****

The **Iterator Pattern** is commonly used in Spring Boot for:  
✅ **Processing Large Data Sets** – Iterating through large database records efficiently.  
✅ **Lazy Loading / Streaming Data** – Fetching records from a database one-by-one instead of loading everything at once.  
✅ **Iterating Over Beans in Application Context** – Accessing beans dynamically from the ApplicationContext.  
✅ **Custom Collections** – Creating custom iterable objects like hierarchical menus, logs, or workflows.









**Explanation:**

* We defined an **iterator (NameIterator)** to traverse a custom collection (NameCollection).
* The hasNext() method ensures safe iteration.
* The next() method returns the next element sequentially.

4:- Mediator design pattern :-

## ****Mediator Design Pattern in Java & Spring Boot****

### ****What is the Mediator Pattern?****

The **Mediator Pattern** is a **behavioral design pattern** that promotes **loose coupling** by centralizing communication between multiple objects. Instead of objects communicating directly with each other, they interact through a **mediator**, which reduces dependencies between components.

🔹 **Key Concept:** Objects don’t communicate directly but instead use a **mediator** to facilitate interactions.  
🔹 **Use Case:** Helps in reducing **tight coupling** when multiple objects need to interact dynamically.

**Real-World Example 🏢**

Think of an **air traffic control (ATC) system** at an airport:

* Different airplanes don’t communicate directly.
* Instead, they **send requests to the ATC**, which coordinates landings and takeoffs.

Here, the **ATC acts as the mediator** that centralizes communication.

**Use Cases in Java & Spring Boot**

✅ **Messaging Systems (e.g., Kafka, RabbitMQ, JMS)** – Mediator acts as a message broker.  
✅ **Microservices Communication** – API Gateway or Event Bus as a mediator.  
✅ **Spring’s Application Event System (ApplicationEvent & ApplicationListener)** – Mediator for decoupling components.  
✅ **Chat Applications** – Users send messages through a mediator instead of directly communicating.  
✅ **Workflow Engines** – Processes interact via a centralized mediator.

**5:- Memento**

**What is the Memento Pattern?**

The **Memento Pattern** is a **behavioral design pattern** that allows an object to save and restore its previous state **without exposing its internal structure**. It helps implement **undo/redo functionality** in applications.

🔹 **Key Concept:**

* Stores an object's state as a **memento** (snapshot).
* Allows restoring an object's previous state.
* Keeps the **internal details encapsulated**.

**Real-World Example 📌**

**Text Editor with Undo/Redo**

Imagine a **text editor** where you type something and later **undo** changes. The editor saves **snapshots (mementos)** of the text so it can restore previous states.

**Use Cases in Java & Spring Boot**

✅ **Undo/Redo functionality in text editors or UI applications**.  
✅ **State management in workflow engines** (e.g., reverting changes).  
✅ **Database Transactions (Saving checkpoints before executing operations)**.  
✅ **Game Development (Saving & restoring player progress)**.

6:- state design pattern

**State Design Pattern in Java & Spring Boot**

**Definition**

The **State pattern** allows an object to change its behavior when its **internal state** changes. It helps manage state transitions cleanly without using large conditional statements.

**Use Case in Spring Boot**

The State pattern is useful in applications that have objects with different states and behaviors, such as:

* **Order Processing System** (e.g., NEW → PROCESSING → SHIPPED → DELIVERED)
* **Workflow Engine** (e.g., user onboarding with different steps)
* **Authentication Flow** (e.g., logged-in vs logged-out user)
* **Finite State Machines** (e.g., a ticketing system)

7:- strategy design pattern :-

**Strategy Design Pattern in Java & Spring Boot**

**Definition**

The **Strategy Pattern** is a behavioral design pattern that defines a family of algorithms, encapsulates each one, and makes them interchangeable. The strategy lets the algorithm vary independently from the clients that use it. This pattern is used when you have multiple ways of performing a specific task, and you want to allow the algorithm to be chosen at runtime.

**Use Case in Spring Boot**

The Strategy pattern is useful when there are different ways to perform a task, and you want to decouple the code that uses the algorithm from the specific implementation. Examples include:

* **Payment Processing Systems**: Support for multiple payment methods like credit card, PayPal, etc.
* **Sorting Algorithms**: Different sorting methods based on user input or data.
* **Discount Calculation**: Apply different discounts based on user type or other conditions.

8: template design pattern

1. Template Method

Template Method is a behavioral design pattern that allows you to defines a skeleton of an algorithm in a base class and let subclasses override the steps without changing the overall algorithm's structure.

The intent of the template method is to define the overall structure of the operation, while allowing subclasses to refine, or redefine, certain steps.

**Template Method:**

The template method pattern is a**behavioural pattern**. And as the name itself says it provides a base template method. When we are working with inheritance in our applications we provide a base template method that should be used by the child classes. The child classes can override certain methods but they should use the base template method as is.

**For example** we have a data renderer class which can read the data, process the data and then render or display that data to the end user.But in our application we want to render the data in the same way no matter in which format the data is coming in that is if it is xml data or if it is CSV data. We want to render it using the render method in the base class reading the data and processing that data is up to the child classes. The child classes can override the readData and the processData.But we want to provide a base template method with all the implementation in it in that data renderer superclass. This pattern is called template method ,as we are providing a template for a particular method from the parent class that should be used by the child classes.