Design Patterns in Software Development

Design patterns are **reusable solutions** to common software design problems. They are **best practices** that help developers **write clean, maintainable, and scalable** code.

Why Do We Need Design Patterns?

1. **Code Reusability** – Avoids reinventing the wheel by using proven solutions.
2. **Maintainability** – Code is easier to understand and modify.
3. **Scalability** – Helps in designing flexible and extensible applications.
4. **Standardization** – Promotes consistency in code across a project or team.
5. **Improved Communication** – Developers can easily discuss and understand code using standard patterns.

Types of Design Patterns

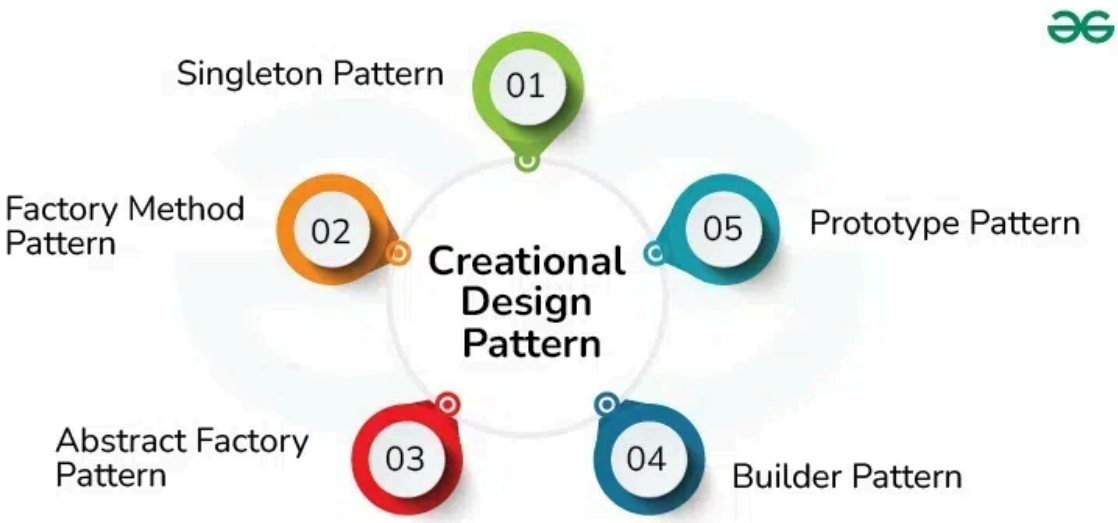
Design patterns are mainly classified into three categories:

1. **Creational Patterns** – Focus on object creation mechanisms.
2. **Structural Patterns** – Define the structure of classes and objects.
3. **Behavioral Patterns** – Deal with object interactions and responsibilities.

1. Creational Design Patterns

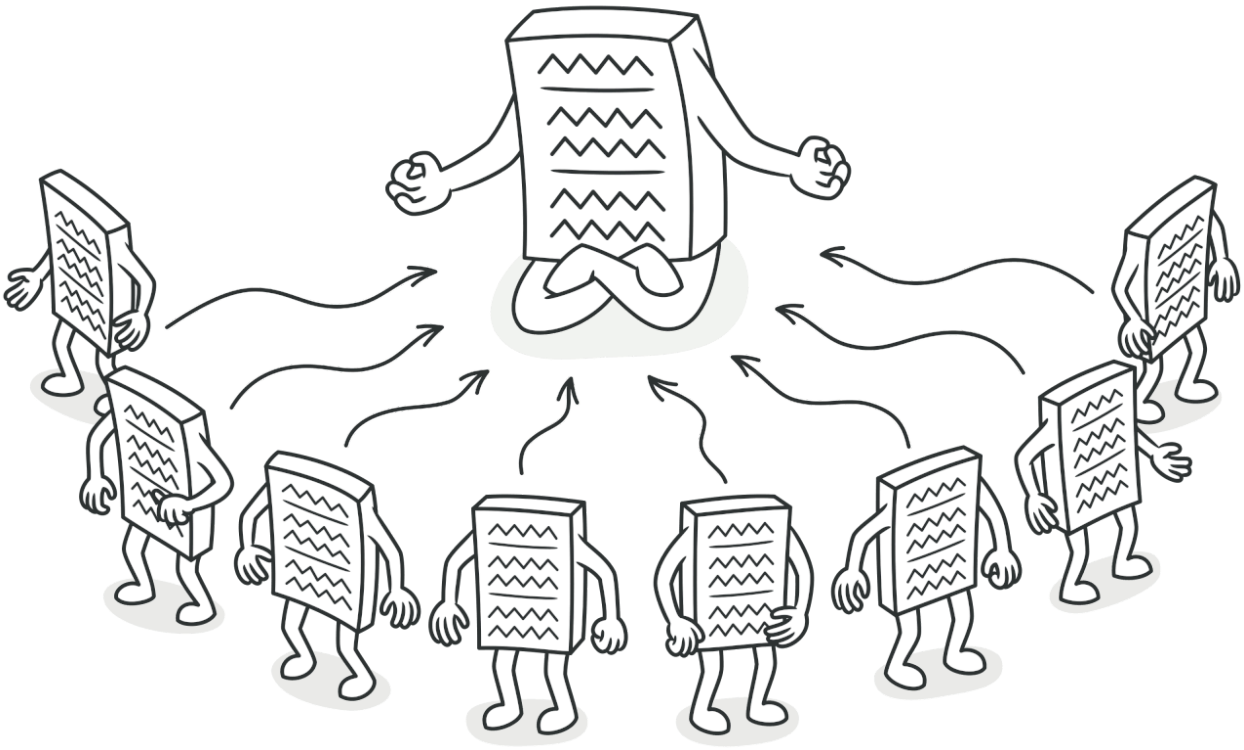
Creational design patterns are a category of design patterns in software development that focus on the process of creating objects.

They aim to enhance flexibility and efficiency in object creation, allowing systems to remain independent of how their objects are constructed, composed, and represented.



1) Singleton Pattern

Ensures that a class has only one instance and provides a global point of access to that instance.



Here's how it works: imagine that you created an object, but after a while decided to create a new one. Instead of receiving a fresh object, you'll get the one you already created.

Note that this behavior is impossible to implement with a regular constructor since a constructor call must always return a new object by design.

Key Components of Singleton Pattern

- **Private Constructor** → Prevents multiple object instantiations.

```
class Singleton {
  // Making the constructor as Private
  private Singleton()
  {
    // Initialization code here
  }
```

```
}
```

- **Static Instance Variable** → Stores the single instance and this static member ensures that memory is allocated only once.

  Ex:
```
// Static member to hold the single instance
private static Singleton instance;
```

- **Public Static Method** → Provides access to the instance.

  This method acts as a gateway, providing a global point of access to the Singleton object. When someone requests an instance, this method either creates a new instance (if none exists) or returns the existing instance to the caller.

```
// Static factory method for global access
public static Singleton getInstance()
{
    // Check if an instance exists
    if (instance == null) {
        // If no instance exists, create one
        instance = new Singleton();
    }
    // Return the existing instance
    return instance;
}
```

Types of Singleton Initialization

1. **Eager Initialization** → Instance is created at the time of class loading.Its drawback is that Instance is always initialized whether it is being used or not.

2. **Lazy Initialization** → Instance is created only when requested.

3. **Thread-safe Singleton** → Uses synchronization to ensure safe access in multi-threaded environments.

Ex: Lazy initializations

In below both examples ,The instance of Singleton / Logger is **not created** at the time of class loading.

Instead, it is created **only when** getInstance() is called for the first time.

The check if (instance == null) ensures that the instance is only created when needed.

```
	package com.designpatterns;
class Singleton {
	// Static member to hold the single instance
	private static Singleton instance;
	// Private Constructor
	private Singleton() {
	}
	public static Singleton getInstance() {
		if (instance == null) {
			instance = new Singleton();
		}
		return instance;
	}
	public static void doSomething() {
		System.out.println("Somethong is Done.");
	}
}
public class SingleTonExample {
	public static void main(String[] args) {
		Singleton.getInstance().doSomething();
	}
}
/*
* The getInstance method, we check whether the instance is null. If the
* instance is not null, it means the object was created before; otherwisewe
* create it using the new operator.
*/
```

Output
Somethong is Done.

Ex2:

```
package com.designpatterns;
class Logger {
	private static Logger instance;
	private Logger() {
	} // Private Constructor
	public static Logger getInstance() {
		if (instance == null) {
			instance = new Logger(); // Lazy Initialization
		}
		return instance;
	}
	public void log(String message) {
		System.out.println("Log: " + message);
	}
}
// Usage
public class SingleTonExample2 {
	public static void main(String[] args) {
		Logger logger1 = Logger.getInstance();
		Logger logger2 = Logger.getInstance();
		logger1.log("Application started");
		System.out.println(logger1 == logger2); // true (Same instance)
	}
}
```

Log: Application started
True

EX3:

```
package com.designpatterns;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
public class DatabaseManager {
  private static DatabaseManager instance;
```

```
    private Connection connection;
    private DatabaseManager() {
        try {
            String url = "jdbc:mysql://localhost:3306/mydatabase";
            String user = "root";
            String password = "password";
            connection = DriverManager.getConnection(url, user, password);
        } catch (SQLException e) {
            throw new RuntimeException("Error connecting to the database", e);
        }
    }
    public static DatabaseManager getInstance() {
        if (instance == null) {
            instance = new DatabaseManager();
        }
        return instance;
    }
    public Connection getConnection() {
        return connection;
    }
}
```

Eager Initialization

we have created an instance of a singleton in a static initializer. JVM executes a static initializer when the class is loaded and hence this is guaranteed to be thread-safe. Use this method only when your singleton class is light and is used throughout the execution of your program.

```
package com.designpatterns;
class Logger {
    private static final Logger instance = new Logger(); // Eager Initialization
    private Logger() {
    } // Private Constructor
    public static Logger getInstance() {
        return instance; // Returns the already created instance
    }
    public void log(String message) {
        System.out.println("Log: " + message);
    }
}
```

```
// Usage
public class SingleTonExampleEager3 {
  public static void main(String[] args) {
     Logger logger1 = Logger.getInstance();
     Logger logger2 = Logger.getInstance();
     logger1.log("Application started");
     System.out.println(logger1 == logger2); // true (Same instance)
  }
}
```

Log: Application started
true

Thread-safe Singleton

```
// Thread Synchronized Java implementation of
// singleton design pattern
class Singleton {
    private static Singleton obj;
    private Singleton() {}

    // Only one thread can execute this at a time
    public static synchronized Singleton getInstance()
    {
        if (obj == null)
            obj = new Singleton();
        return obj;
    }
}
```

Here using synchronized makes sure that only one thread at a time can execute **getInstance()**. The main disadvantage of this method is that using synchronized every time while creating the singleton object is expensive and may decrease the performance of your program. However, if the performance of **getInstance()** is not critical for your application this method provides a clean and simple solution.

```
package com.designpatterns;
class Logger {
        private static Logger instance;
        private Logger() {
        } // Private Constructor
        public static synchronized Logger getInstance() {
                if (instance == null) {
                        instance = new Logger(); // Lazy Initialization
                }
                return instance;
        }
        public void log(String message) {
                System.out.println("Log: " + message);
```

```
        }
}
// Usage
public class SingleTonExample2 {
        public static void main(String[] args) {
                Logger logger1 = Logger.getInstance();
                Logger logger2 = Logger.getInstance();
                logger1.log("Application started");
                System.out.println(logger1 == logger2); // true (Same instance)
        }
}
```

Log: Application started
True

Thread-Safe Singleton with Double-Checked Locking

Method 4 – Most Efficient || Use "Double Checked Locking" to implement singleton design pattern

If you notice carefully once an object is created synchronization is no longer useful because now obj will not be null and any sequence of operations will lead to consistent results. So we will only acquire the lock on the getInstance() once when the obj is null. This way we only synchronize the first way through, just what we want.

```
// Double Checked Locking based Java implementation of
// singleton design pattern
class Singleton {
    private static volatile Singleton obj = null;
    private Singleton() {}

    public static Singleton getInstance()
    {
        if (obj == null) {
            // To make thread safe
            synchronized (Singleton.class)
            {
                // check again as multiple threads
                // can reach above step
                if (obj == null)
                    obj = new Singleton();
            }
        }
        return obj;
    }
}
```

We have declared the obj volatile which ensures that multiple threads offer the obj variable correctly when it is being initialized to the Singleton instance. This method drastically reduces the overhead of calling the synchronized method every time.

```
package com.designpatterns;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
```

```
public class DatabaseManager {
  private static volatile DatabaseManager instance;
  private Connection connection;
  private DatabaseManager() {
    try {
      String url = "jdbc:mysql://localhost:3306/mydatabase";
      String user = "root";
      String password = "password";
      connection = DriverManager.getConnection(url, user, password);
    } catch (SQLException e) {
      throw new RuntimeException("Error connecting to the database", e);
    }
  }
  public static DatabaseManager getInstance() {
    if (instance == null) {
      synchronized (DatabaseManager.class) {
        if (instance == null) {
          instance = new DatabaseManager();
        }
      }
    }
    return instance;
  }
  public Connection getConnection() {
    return connection;
  }
}
```

Principles Behind Singleton

- Encapsulation → Restricts object instantiation.
- Global Access Point → Provides controlled access.

Real-time Example

i) Logger service – A single instance handles all logging requests.
ii) Managing database connections or configuration settings.

Where is Singleton Used in Spring Boot?

Spring Beans (Default Scope: Singleton): By default, Spring manages beans as **singletons**, meaning only **one instance** of the bean exists in the Spring container.

Service Layer (@Service)
Repository Layer (@Repository)
Controller Layer (@RestController)

Example: Singleton Service in Spring Boot

java Copy Edit

```java
import org.springframework.stereotype.Service;

@Service // Singleton by default
public class LoggerService {
    public void log(String message) {
        System.out.println("Log: " + message);
    }
}
```

Usage in Controller

java Copy Edit

```java
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/api")
public class LoggerController {

    private final LoggerService loggerService;

    // Constructor Injection (Singleton Bean is injected)
    public LoggerController(LoggerService loggerService) {
        this.loggerService = loggerService;
    }

    @GetMapping("/log")
    public String logMessage() {
        loggerService.log("API Called");
        return "Logged Successfully!";
    }
}
```

Since LoggerService is a Singleton Bean, every request to /api/log will use the same instance.

@Autowired vs constructor injection

```
package com.designpatterns;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
@Service
public class OrderService {
  @Autowired  // Injects the Singleton instance of PaymentService
  private PaymentService paymentService;
  public void processOrder() {
     System.out.println("Processing order...");
     paymentService.processPayment();
  }
}
```

Why Use a Private Instance?

- **Encapsulation**: Restricts direct access to the field.
- **Spring Dependency Injection**: Spring injects the PaymentService automatically.
- **Singleton by Default**: The same PaymentService instance is used throughout the application.

Alternative: Constructor Injection (Recommended)

Instead of using @Autowired on a private field, it's better to use **Constructor Injection** (best practice):

```
package com.designpatterns;
import org.springframework.stereotype.Service;
@Service
public class OrderService {
  private final PaymentService paymentService;
  // Constructor Injection (Preferred)
  public OrderService(PaymentService paymentService) {
     this.paymentService = paymentService;
  }
  public void processOrder() {
```

```
        System.out.println("Processing order...");
        paymentService.processPayment();
    }
}
```

✅ Advantages of Singleton

1. **Ensures Single Instance** – Prevents multiple object creation.
2. **Saves Memory & Resources** – Avoids redundant objects.
3. **Thread-Safe (if implemented correctly)** – Prevents race conditions.
4. **Global Access Point** – Easy access throughout the application.
5. **Useful for Shared Resources** – Ideal for **logging, caching, database connections**.

❌ Disadvantages of Singleton

1. **Hard to Unit Test** – Introduces **global state**, making tests dependent.
2. **Breaks SOLID Principles** – Violates **Single Responsibility Principle (SRP)**.
3. **Memory Issues (Eager Initialization)** – Wastes memory if unused.
4. **Thread-Safety Issues (Lazy Initialization)** – Can cause race conditions if not synchronized.
5. **Hinders Scalability** – Becomes a **bottleneck** in high-load applications.

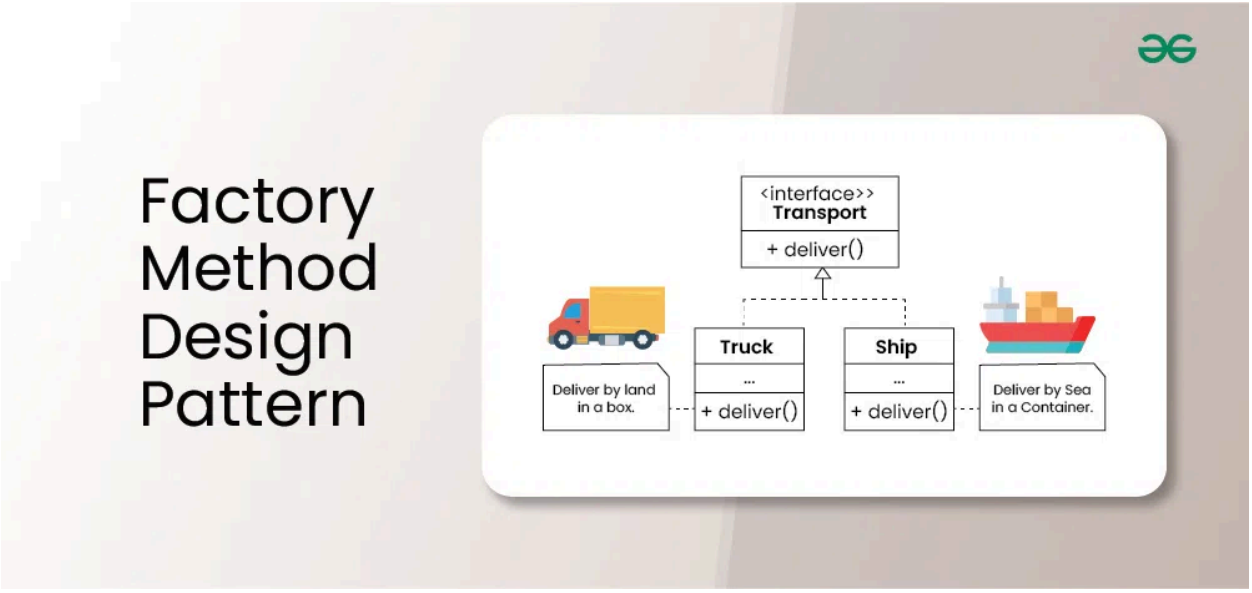
When to Use?

Logging, Database Connections, Configuration Managers, Caching

2) Factory Method

Factory Method is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

- In the Factory Method, we do not expose the creation logic to the client. Instead, we refer to the generated object using a standard interface.



Key Components of Factory Pattern

- **Factory Method** → Centralized object creation logic.
- **Product Interface** → Defines the structure for objects.
- **Concrete Products** → Different object implementations.

When to Use Factory Pattern?

- When the **exact type of object** is unknown until runtime.
- When object **creation logic** is complex and needs to be centralized.
- When object creation is **complex or frequently changes**.
- When working with **multiple related objects**.
- For **frameworks** or **libraries** where clients should not worry about object instantiation.

Principles Behind Factory Pattern:

Encapsulation → Hides the object creation process.

Open-Closed Principle → Allows adding new products without modifying existing code.

Real-time Example: Payment Gateway

A **payment system** that selects the appropriate payment method (CreditCard, PayPal, UPI) based on user input.

Step 1: Define the Transport product Interface

```
interface Transport {
  void deliver();
}
```

Step 2: Create Concrete Transport Classes

```
class Truck implements Transport {
  public void deliver() {
    System.out.println("Deliver by land in a box.");
  }
}
class Ship implements Transport {
  public void deliver() {
    System.out.println("Deliver by sea in a container.");
  }
}
```

Step 3: Define the Factory Method

```
abstract class TransportFactory {
  abstract Transport createTransport();
}
```

Step 4: Implement Specific Factories

```
class TruckFactory extends TransportFactory {
```

```
    public Transport createTransport() {
        return new Truck();
    }
}
class ShipFactory extends TransportFactory {
    public Transport createTransport() {
        return new Ship();
    }
}
```

Step 5: Test the Factory Method

```
public class FactoryMethodExample {
    public static void main(String[] args) {
        TransportFactory truckFactory = new TruckFactory();
        Transport truck = truckFactory.createTransport();
        truck.deliver();  // Output: Deliver by land in a box.
        TransportFactory shipFactory = new ShipFactory();
        Transport ship = shipFactory.createTransport();
        ship.deliver();  // Output: Deliver by sea in a container.
    }
}
```

TransportFactory truckFactory = new TruckFactory();

- **Creates an instance of TruckFactory**.
- Since TruckFactory is a subclass of TransportFactory, it **inherits the abstract method** createTransport(), which returns a Truck object.

Transport truck = truckFactory.createTransport();

- Calls the createTransport() method from TruckFactory, which **creates and returns a new Truck object**.
- Now, truck holds a reference to an instance of Truck.

truck.deliver(); // Output: Deliver by land in a box.

- Call the deliver() method on the Truck instance.

Since Truck implements Transport, it **overrides deliver()** to print:

Deliver by land in a box.

Advantages

- Promotes **loose coupling** by using interfaces/abstract classes.
- Simplifies **code maintenance** and **scalability**.
- Supports **dynamic object creation** based on input.

Disadvantages

- Increases **complexity** due to additional classes/methods.
- Can be **overkill** for simple object creation scenarios.

3) Abstract Factory

Provides an interface for creating families of related or dependent objects without specifying their concrete classes and implementation, in simpler terms the Abstract Factory Pattern is a way of organizing how you create groups of things that are related to each other.

An extension of the Factory Pattern, it provides an **interface for creating families of related objects**.

Key Components of Abstract Factory Pattern

- **Abstract Factory** → Declares methods for object creation.
- **Concrete Factories** → Implement object creation logic.
- **Abstract Product Interface** → Defines structure for objects.
- **Concrete Products** → Implement different product variants.

When to Use Abstract Factory Pattern?

- When multiple related objects need to be created together.
- When object creation must be decoupled from usage.

- When object creation is **complex or frequently changes**.

Principles Behind Abstract Factory

- **Encapsulation** → Hides product creation logic.
- **Dependency Injection** → Promotes loose coupling.

A car factory producing **different parts (Engine, Wheels, Interiors)** based on the car brand.

Vehicle Factory

We will create **two types of vehicles (Car, Bike)** and two categories of factories **(Electric Factory, Petrol Factory)** to produce **Electric and Petrol versions** of each vehicle.

Step 1: Define Abstract Product Interfaces

These interfaces define the general behavior for **Car** and **Bike**.

```
interface Car {
  void drive();
}
interface Bike {
  void ride();
}
```

Step 2: Create Concrete Product Classes

Each type of vehicle (Car, Bike) will have **Electric and Petrol variants**.

```
//Concrete Car Implementations
class ElectricCar implements Car {
public void drive() {
   System.out.println("Driving an electric car.");
}
}
class PetrolCar implements Car {
public void drive() {
   System.out.println("Driving a petrol car.");
}
}
```

```
//Concrete Bike Implementations
class ElectricBike implements Bike {
public void ride() {
    System.out.println("Riding an electric bike.");
}
```

Step 3: Define Abstract Factory

This abstract class defines methods for **creating cars and bikes**

```
interface VehicleFactory {
   Car createCar();
   Bike createBike();
}
```

Step 4: Implement Concrete Factories

Each factory will produce a specific type of vehicle (**Electric or Petrol**).

```
class ElectricVehicleFactory implements VehicleFactory {
   public Car createCar() {
      return new ElectricCar();
   }

   public Bike createBike() {
      return new ElectricBike();
   }
}
class PetrolVehicleFactory implements VehicleFactory {
   public Car createCar() {
      return new PetrolCar();
   }

   public Bike createBike() {
      return new PetrolBike();
   }
}
```

Step 5: Use the Abstract Factory in Main Class

```
public class AbstractFactoryExample {
   public static void main(String[] args) {
```

```
        // Create an Electric Vehicle Factory
        VehicleFactory electricFactory = new ElectricVehicleFactory();
        Car electricCar = electricFactory.createCar();
        Bike electricBike = electricFactory.createBike();
        electricCar.drive();  // Output: Driving an electric car.
        electricBike.ride();  // Output: Riding an electric bike.
        // Create a Petrol Vehicle Factory
        VehicleFactory petrolFactory = new PetrolVehicleFactory();
        Car petrolCar = petrolFactory.createCar();
        Bike petrolBike = petrolFactory.createBike();
        petrolCar.drive();  // Output: Driving a petrol car.
        petrolBike.ride();  // Output: Riding a petrol bike.
    }
}
```

Creating an Electric Vehicle Factory

VehicleFactory electricFactory = new ElectricVehicleFactory();

- We **declare** a variable electricFactory of type VehicleFactory (abstract factory interface).
- We **instantiate** ElectricVehicleFactory, which is a **concrete factory** that produces **Electric Cars & Electric Bikes**.
- This ensures that we can **create only electric vehicles** using this factory.

Creating an Electric Car

Car electricCar = electricFactory.createCar();

- Calls createCar() from the ElectricVehicleFactory class.
- Returns an instance of ElectricCar, which implements the Car interface.
- Now, electricCar holds an **electric car object**.
  Creating an Electric Bike

Bike electricBike = electricFactory.createBike();

- Calls createBike() from the ElectricVehicleFactory class.

- Returns an instance of ElectricBike, which implements the Bike interface.
- Now, electricBike holds an **electric bike object**.

Calling Methods on Electric Vehicles

electricCar.drive(); // Output: Driving an electric car.

- Call the drive() method of ElectricCar.
- Outputs: **"Driving an electric car."**

electricBike.ride(); // Output: Riding an electric bike.

- Call the ride() method of ElectricBike.
- Outputs: **"Riding an electric bike."**

Advantages

- Ensures consistent object families are used together.
- Improves scalability when adding new object types.
- Encapsulates complex creation logic into one place.

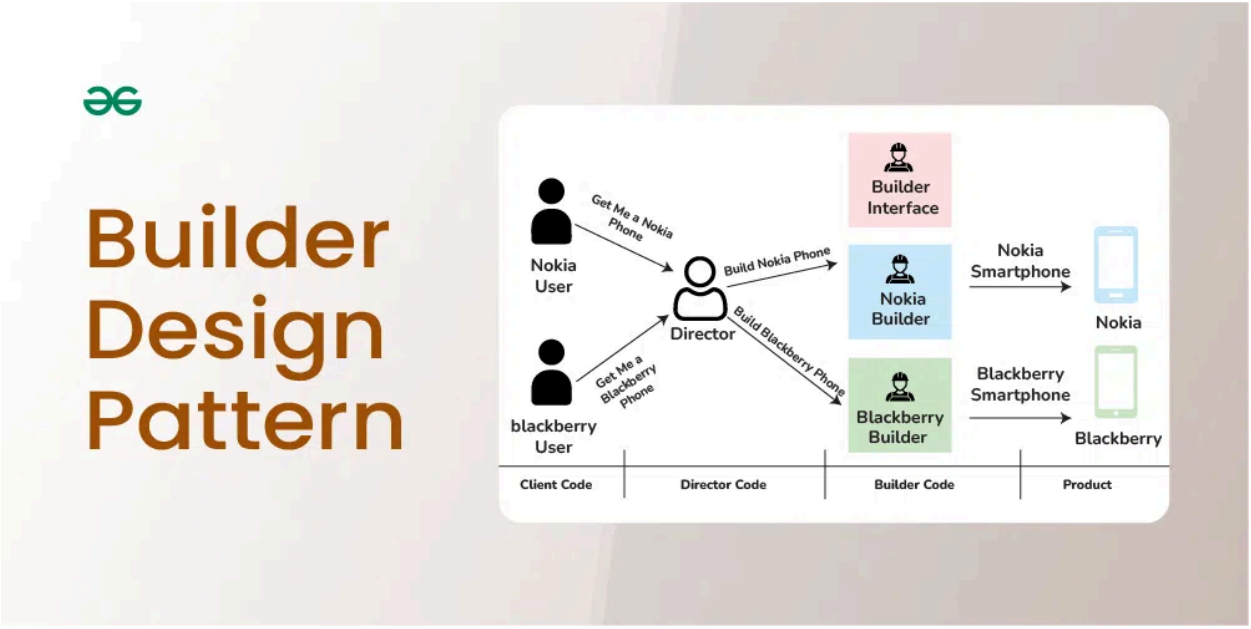
Disadvantages

- Increases complexity due to multiple factory classes.
- Difficult to modify once implemented for a product family.

4) Builder

Used in software design to construct a complex object step by step. It allows the construction of a product in a step-by-step manner, where the construction process can change based on the type of product being built.

The pattern allows us to produce different types and representations of an object using the same construction code.



1. **Client Code (Users)**
    - Nokia User: Requests a **Nokia Smartphone**.
    - Blackberry User: Requests a **Blackberry Smartphone**.
2. **Director (Construction Manager)**
    - The **Director** handles the construction process.
    - It **delegates the building steps** to the appropriate builder (Nokia or Blackberry).
3. **Builder Interface**
    - Defines the structure for building smartphones.
    - Ensures that all concrete builders (Nokia & Blackberry) follow the same set of building steps.
4. **Concrete Builders**
    - **Nokia Builder**: Implements the builder interface to create a **Nokia Smartphone**.

    - **Blackberry Builder**: Implements the builder interface to create a **Blackberry Smartphone**.

5. **Final Product**
    - The built **Nokia Smartphone**.
    - The built **Blackberry Smartphone**.

Key Components of Builder Pattern

- **Builder Interface** → Defines methods to set object properties.
- **Concrete Builder** → Implements the builder logic.
- **Director** → Guides the object construction process.
- **Product** → The final complex object.

When to Use Builder Pattern?

- When constructing **complex objects with multiple attributes**.
- When **step-by-step** object creation is needed.
- When **constructing objects step by step** improves clarity.
- When **objects have multiple optional parameters**.

Principles Behind Builder Pattern

- **Encapsulation** → Hides object details from the client.
- **Fluent Interface** → Provides a chainable API.

Real-time Example

i)Constructing a complex report or meal order in a restaurant.
ii) Resume Builder: A user fills in different sections (Personal Info, Education, Experience) step-by-step.

Step 1: Define the Phone Product

```
class Phone {
```

```
  private String brand;
  private String OS;

  public void setBrand(String brand) { this.brand = brand; }
  public void setOS(String OS) { this.OS = OS; }

  public void showPhone() {
     System.out.println("Phone Brand: " + brand + ", OS: " + OS);
  }
}
```

Step 2: Create the PhoneBuilder Interface

```
interface PhoneBuilder {
  void buildBrand();
  void buildOS();
  Phone getPhone();
}
```

Step 3: Implement Concrete Builders

Nokia Builder

```
class NokiaBuilder implements PhoneBuilder {
  private Phone phone = new Phone();
  public void buildBrand() { phone.setBrand("Nokia"); }
  public void buildOS() { phone.setOS("Android"); }

  public Phone getPhone() { return phone; }
}
```

Blackberry Builder

```
class BlackberryBuilder implements PhoneBuilder {
  private Phone phone = new Phone();
  public void buildBrand() { phone.setBrand("Blackberry"); }
  public void buildOS() { phone.setOS("Blackberry OS"); }

  public Phone getPhone() { return phone; }
}
```

Step 4: Create the Director

```
class PhoneDirector {
  private PhoneBuilder builder;
  public PhoneDirector(PhoneBuilder builder) {
    this.builder = builder;
  }
  public Phone constructPhone() {
    builder.buildBrand();
    builder.buildOS();
    return builder.getPhone();
  }
}
```

Step 5: Client Code to Build Phones

```
public class BuilderPatternExample {
  public static void main(String[] args) {
    // Building a Nokia Phone
    PhoneBuilder nokiaBuilder = new NokiaBuilder();
    PhoneDirector nokiaDirector = new PhoneDirector(nokiaBuilder);
    Phone nokiaPhone = nokiaDirector.constructPhone();
    nokiaPhone.showPhone(); // Output: Phone Brand: Nokia, OS: Android
    // Building a Blackberry Phone
    PhoneBuilder blackberryBuilder = new BlackberryBuilder();
    PhoneDirector blackberryDirector = new PhoneDirector(blackberryBuilder);
    Phone blackberryPhone = blackberryDirector.constructPhone();
    blackberryPhone.showPhone(); // Output: Phone Brand: Blackberry, OS: Blackberry OS
  }
}
```

Output

```
Phone Brand: Nokia, OS: Android
Phone Brand: Blackberry, OS: Blackberry OS
```

```
// Building a Nokia Phone

PhoneBuilder nokiaBuilder = new NokiaBuilder();
```

- **Creates an instance of NokiaBuilder**, which is a concrete implementation of the PhoneBuilder interface.

- This builder is responsible for constructing a **Nokia phone** by setting its brand and OS.

PhoneDirector nokiaDirector = new PhoneDirector(nokiaBuilder);

- **Creates a PhoneDirector object** and passes nokiaBuilder as a parameter.

- The **Director** is responsible for controlling the construction steps (i.e., calling methods like buildBrand() and buildOS() internally).

Phone nokiaPhone = nokiaDirector.constructPhone();

- Calls the **constructPhone()** method in PhoneDirector, which:

    1. Calls buildBrand() → sets the brand to **"Nokia"**.

    2. Calls buildOS() → sets the OS to **"Android"**.

    3. Returns the fully constructed **Nokia Phone** object.

nokiaPhone.showPhone(); // Output: Phone Brand: Nokia, OS: Android

- Calls the **showPhone()** method to **print the details** of the built phone.

- **Output:**
  Phone Brand: Nokia, OS: Android

Key points

Separation of Concerns – Construction logic is separate from the product.

Step-by-Step Object Creation – The Director ensures consistent object building.
Flexibility – Can add new builders (e.g., AppleBuilder, SamsungBuilder) without modifying existing code.
Immutable Objects – Helps in creating objects with different configurations.

Advantages

- **Improves readability** when creating complex objects.

- Ensures **immutability** by creating final objects.
- **Flexible** and supports different configurations.

Disadvantages

- **Increases code size** due to multiple builder classes.
- Can be **overkill** for simple object creation.

5) Prototype

the creation of new objects by copying an existing object

Prototype is a creational design pattern that lets you copy existing objects without making your code dependent on their classes.

Key Components of Prototype Pattern

1. **Prototype Interface** → Declares the cloning method (clone()).
2. **Concrete Prototype** → Implements the clone() method to copy itself.
3. **Client** → Requests cloning of objects instead of direct instantiation.

Types of Initialization in Prototype Pattern

1. **Shallow Copy** → Creates a new object but only copies references for nested objects (not deep copies).
2. **Deep Copy** → Creates a new object along with new copies of any referenced objects.

When to Use Prototype Pattern?

When **creating objects is expensive** (e.g., database operations, network calls).
When object **initialization is complex** and involves many configurations.
When many **similar objects** are needed with slight modifications.
When **copying an existing object** is more efficient than creating a new one.

Real-time Example

i) Copying a document template instead of creating a new one from scratch.

1) Cloning an Employee Object in Java (Shallow Copy)

A shallow copy means creating a new object but copying references for any nested objects. This means that if the original object contains references to other objects, both the original and cloned objects share the same referenced objects.

Key Characteristics of Shallow Copy

Primitive fields (like int, double, String, etc.) are copied as-is.
Reference fields (objects) are not duplicated; both the original and clone share the same referenced object.
Modifying the referenced object affects both the original and cloned object.

```
class Address {
  String city;
  public Address(String city) {
    this.city = city;
  }
}
class Employee implements Cloneable {
  String name;
  int age;
  Address address; // Reference type
  public Employee(String name, int age, Address address) {
    this.name = name;
    this.age = age;
    this.address = address;
  }
  // Overriding clone() method for Shallow Copy
  @Override
  public Employee clone() throws CloneNotSupportedException {
    return (Employee) super.clone(); // Default clone() creates a shallow copy
  }
  public void display() {
    System.out.println("Employee: " + name + ", Age: " + age + ", City: " + address.city);
```

```
    }
}
// Usage
public class Main {
    public static void main(String[] args) throws CloneNotSupportedException {
        Address address = new Address("New York");
        Employee emp1 = new Employee("John", 30, address);
        Employee emp2 = emp1.clone(); // Cloning emp1
        emp2.address.city = "Los Angeles"; // Changing cloned object's address
        emp1.display();
        emp2.display();
    }
}
```

Output

Employee: John, Age: 30, City: Los Angeles

Employee: John, Age: 30, City: Los Angeles

Explanation

- emp1 and emp2 are separate objects.
- But **both share the same Address object** (reference is copied, not the actual object).
- Changing emp2.address.city also affects emp1 because both point to the **same Address object**.

Example 2: Deep Copy (Handling Nested Objects)

A deep copy means creating a new object along with copies of all nested objects. This ensures that the cloned object does not share references with the original object.

Key Characteristics of Deep Copy

All fields, including referenced objects, are duplicated.
Original and cloned objects are completely independent.
Modifying the nested objects in the clone does not affect the original.

```
class Address {
  String city;
  public Address(String city) {
    this.city = city;
  }
}
class Employee implements Cloneable {
  String name;
  int age;
  Address address;
  public Employee(String name, int age, Address address) {
    this.name = name;
    this.age = age;
    this.address = address;
  }
  // Deep Copy Implementation
  @Override
  public Employee clone() throws CloneNotSupportedException {
    Employee cloned = (Employee) super.clone();
    cloned.address = new Address(this.address.city); // Creating a new Address object
    return cloned;
  }
  public void display() {
    System.out.println("Employee: " + name + ", Age: " + age + ", City: " + address.city);
  }
}
// Usage
public class Main {
  public static void main(String[] args) throws CloneNotSupportedException {
    Address address = new Address("New York");
    Employee emp1 = new Employee("John", 30, address);
    Employee emp2 = emp1.clone(); // Cloning emp1
    emp2.address.city = "Los Angeles"; // Modifying cloned object's address
    emp1.display();
    emp2.display();
  }
}
```

Output:

Employee: John, Age: 30, City: New York
Employee: John, Age: 30, City: Los Angeles

- Deep Copy → The cloned emp2 object has a separate Address object.

Advantages

- Reduces object creation overhead, improving performance.
- Ensures independent object copies (deep copy).
- Useful for duplicating objects dynamically.

Disadvantages

- Cloning can be complex if objects have deep dependencies.
- Shallow copy vs deep copy requires careful handling.

Most Interviewed Question

1. Can you name a few design patterns used in the standard JDK library?

1. Factory Method Design Patterns
2. Abstract Factory Method Design Patterns
3. Singleton Method Design Pattern
4. Prototype Method Design Patterns
5. Builder Method Design Patterns