**Design Patterns**

[Design patterns](https://www.geeksforgeeks.org/software-design-patterns/) in Java refer to structured approaches involving objects and classes that aim to solve recurring design issues within specific contexts. These patterns offer reusable, general solutions to common problems encountered in software development, representing established best practices. By utilizing design patterns, developers can communicate more effectively about their approaches to problem-solving, fostering collaboration and consistency in their code.

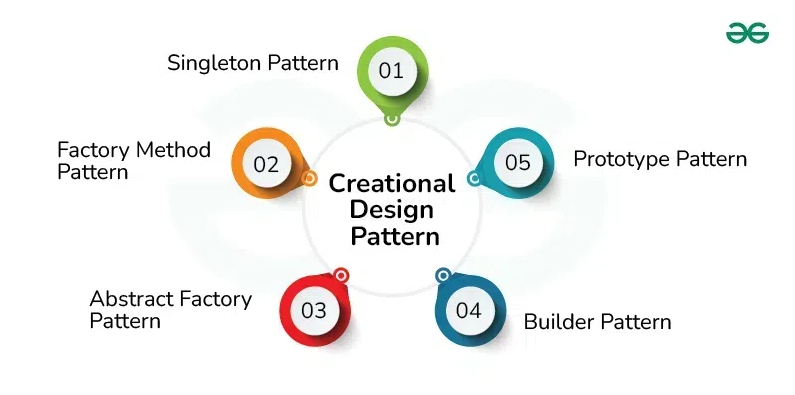
**What are Design Patterns?**

A design pattern is a reusable solution for common problems in software design used in engineering. It is not a full design ready for coding but rather a guideline or model for solving issues. Design patterns can be adapted to different situations and contexts, providing flexibility in problem-solving.

**Types of Software Design Patterns in Java**

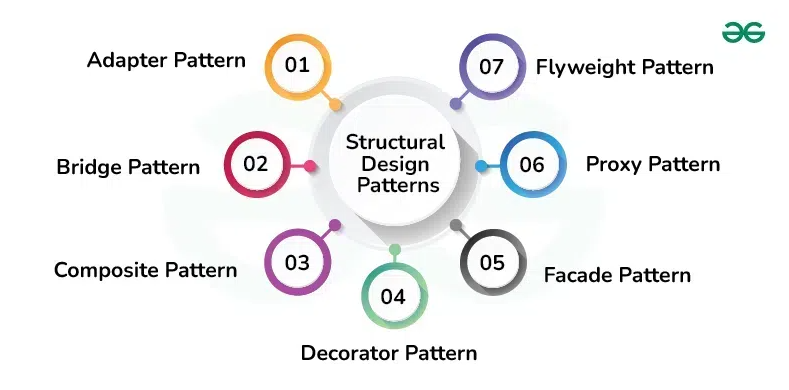
**1. Creational Design Patterns in Java**

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. This approach helps streamline the instantiation process and can lead to more maintainable and adaptable code



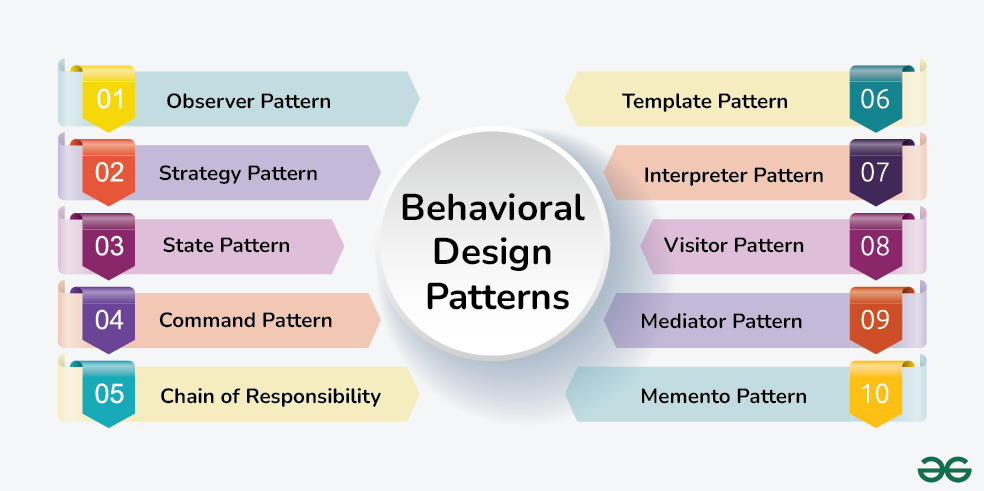
**2. Structural Design Patterns in Java**

Structural design patterns focus on how classes and objects are arranged to create larger, more complex structures in software development. They help organize relationships between objects, making systems more flexible, reusable, and maintainable. By using these patterns, developers can create easier-to-understand and modify systems.



**3. Behavioral Design Patterns in Java**

Behavioral design patterns are a group of design patterns that focus on how objects and classes interact and communicate in software development. They emphasize the collaboration between objects to effectively accomplish tasks and responsibilities, making the system more manageable and adaptable



**Factory method Design Pattern**

The Factory Method Design Pattern is a [creational design pattern](https://www.geeksforgeeks.org/creational-design-pattern/) that provides an interface for creating objects in a superclass, allowing subclasses to alter the type of objects that will be created. This pattern is particularly useful when the exact types of objects to be created may vary or need to be determined at runtime, enabling flexibility and extensibility in object creation.

**What is the Factory Method Design Pattern?**

The Factory Method Design Pattern is a creational design pattern used in software development. It provides an interface for creating objects in a superclass while allowing subclasses to specify the types of objects they create.

* This pattern simplifies the object creation process by placing it in a dedicated method, promoting loose coupling between the object creator and the objects themselves.
* This approach enhances flexibility, extensibility, and maintainability, enabling subclasses to implement their own factory methods for creating specific object types.

**When to Use the Factory Method Design Pattern**

Below is when to use factory method design pattern:

* If your object creation process is complex or varies under different conditions, using a factory method can make your client code simpler and promote reusability.
* The Factory Method Pattern allows you to create objects through an interface or abstract class, hiding the details of concrete implementations. This reduces dependencies and makes it easier to modify or expand the system without affecting existing code.
* If your application needs to create different versions of a product or may introduce new types in the future, the Factory Method Pattern provides a flexible way to handle these variations by defining specific factory methods for each product type.
* Factories can also encapsulate configuration logic, allowing clients to customize the object creation process by providing parameters or options to the factory method.

**Components of Factory Method Design Pattern**

Below are the main components of Factory Design Pattern:

* **Creator**: This is an abstract class or an interface that declares the factory method. The creator typically contains a method that serves as a factory for creating objects. It may also contain other methods that work with the created objects.
* **Concrete Creator**: Concrete Creator classes are subclasses of the Creator that implement the factory method to create specific types of objects. Each Concrete Creator is responsible for creating a particular product.
* **Product**: This is the interface or abstract class for the objects that the factory method creates. The Product defines the common interface for all objects that the factory method can create.
* **Concrete Product**: Concrete Product classes are the actual objects that the factory method creates. Each Concrete Product class implements the Product interface or extends the Product abstract class.

**Use Cases of the Factory Method**

Below are the main use cases of factory method design pattern:

* Used in JDBC for creating connections and in frameworks like Spring for managing beans.
* Libraries like Swing and JavaFX use factories to create flexible UI components.
* Tools like Log4j rely on factories to create configurable loggers.
* Factories help create objects from serialized data, supporting various formats.

**Advantages of the Factory Method**

Below are the main advantages of factory method design pattern:

* Separates creation logic from client code, improving flexibility.
* New product types can be added easily.
* Simplifies unit testing by allowing mock product creation.
* Centralizes object creation logic across the application.
* Hides specific product classes from clients, reducing dependency.

**Disadvantages of the Factory Method**

Below are the main advantages of factory method design pattern:

* Adds more classes and interfaces, which can complicate maintenance.
* Slight performance impacts due to polymorphism.
* Concrete creators are linked to their products.
* Clients need knowledge of specific subclasses.
* May lead to unnecessary complexity if applied too broadly.
* Factory logic can be harder to test.

**Abstract Factory Pattern**

The Abstract Factory Pattern is one of the creational design patterns that 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.

**What is the Abstract Factory Pattern?**

The Abstract Factory Pattern is a way of organizing how you create groups of things that are related to each other. It provides a set of rules or instructions that let you create different types of things without knowing exactly what those things are. This helps you keep everything organized and lets you switch between different types easily.

* Abstract Factory pattern is almost same as [Factory Pattern](https://www.geeksforgeeks.org/design-patterns-set-2-factory-method/) and is considered as another layer of abstraction over factory pattern.
* Abstract Factory patterns work around a super-factory which creates other factories.
* At runtime, the abstract factory is coupled with any desired concrete factory which can create objects of the desired type.

**Components of Abstract Factory Pattern**

To understand abstract factory pattern, we have to understand the components of it and relationships between them.

* **Abstract Factory**:
  + Abstract Factory provides a high-level blueprint that defines rules for creating families of related object without specifying their concrete classes.
  + It provides a way such that concrete factories follow a common interface, providing consistent way to produce related set of objects.
* **Concrete Factories**:
  + Concrete Factories implement the rules specified by the abstract factory. It contain the logic for creating specific instances of objects within a family.
  + Also multiple concrete factories can exist, each produce a distinct family of related objects.
* **Abstract Products**:
  + Abstract Products represents a family of related objects by defining a set of common methods or properties.
  + It acts as an abstract or interface type that all concrete products within a family must follow to and provides a unified way for concrete products to be used interchangeably.
* **Concrete Products**:
  + They are the actual instances of objects created by concrete factories.
  + They implement the methods declared in the abstract products, ensuring consistency within a family and belong to a specific category or family of related objects.
* **Client**:
  + Client utilizes the abstract factory to create families of objects without specifying their concrete types and interacts with objects through abstract interfaces provided by abstract products.

**Example of Abstract Factory Design Pattern**

Let's understand Abstract Factory Design Pattern using an example:

***Imagine you're managing a global car manufacturing company***

* *You want to design a system to create cars with specific configurations for different regions, such as North America and Europe.*
* *Each region may have unique requirements and regulations, and you want to ensure that cars produced for each region meet those standards.*

**What can be the challenges while implementing this system?**

* Different regions have different cars with different features, so designing this can be challenging.
* The other main challenge is to ensure consistency in the production of cars and their specifications within each region.
* There can be updation in having new cars in different regions so adapting the system to changes in regulations or introducing new features for a specific region becomes challenging.
* So, Modifications would need to be made in multiple places, increasing the chances of introducing bugs and making the system more prone to errors.

**How Abstracy Factory Pattern help to solve above challenges?**

Below is how abstract factory pattern help to solve the above challenges. After using this pattern:

* Different regions has their own factory to create cars for local needs.
* This helps to keeps the design and features the same for vehicles in each region.
* You can change one region without affecting others (e.g., updating North America doesn’t impact Europe).
* To add a new region, just create a new factory, no need to change existing code.
* The pattern keeps car creation separate from how they are used.

**Benefits of using Abstract Factory Pattern**

Below are the main benefits of abstract factory pattern:

* The Abstract Factory pattern separates the creation of objects, so clients don’t need to know specific classes.
* Clients interact with objects through abstract interfaces, keeping class names hidden from client code.
* Changing the factory allows for different product configurations, as all related products change together.
* The pattern ensures that an application uses objects from only one family at a time for better compatibility.

**Challenges of using Abstract Factory Pattern**

Below are the main challenges of using abstract factory pattern:

* The Abstract Factory pattern can add unnecessary complexity to simpler projects with multiple factories and interfaces.
* Adding new product types may require changes to both concrete factories and the abstract factory interface, impacting existing code.
* Introducing more factories and product families can quickly increase the number of classes, making code management difficult in smaller projects.
* It may violate the Dependency Inversion Principle if client code depends directly on concrete factories rather than abstract interfaces.

**When to use Abstract Factory Pattern**

Choose using abstract factory pattern when:

* When your system requires multiple families of related products and you want to ensure compatibility between them.
* When you need flexibility and extensibility, allowing for new product variants to be added without changing existing client code.
* When you want to encapsulate the creation logic, making it easier to modify or extend the object creation process without affecting the client.
* When you aim to maintain consistency across different product families, ensuring a uniform interface for the products.

**When not to use Abstract Factory Pattern**

Aviod using abstract factory pattern when:

* The product families are unlikely to change, as it may add unnecessary complexity.
* When your application only requires single, independent objects and isn't concerned with families of related products.
* When overhead of maintaining multiple factories outweighs the benefits, particularly in smaller applications.
* When simpler solutions, like the Factory Method or Builder pattern, if they meet your needs without adding the complexity of the Abstract Factory pattern.

**Singleton Method Design Pattern**

The Singleton Method Design Pattern ensures a class has only one instance and provides a global access point to it. It’s ideal for scenarios requiring centralized control, like managing database connections or configuration settings. This article explores its principles, benefits, drawbacks, and best use cases in software development.

**What is Singleton Method Design Pattern?**

*The Singleton method or Singleton Design pattern is one of the simplest design patterns. It ensures a class only has one instance, and provides a global point of access to it.*

**Singleton Design Pattern Principles**

Below are the principles of the Singleton Pattern:

* **Single Instance:** Singleton ensures that only one instance of the class exists throughout the application.
* **Global Access:** Provide a global point of access to that instance.
* **Lazy or Eager Initialization:** Support creating the instance either when needed (lazy) or when the class is loaded (eager).
* **Thread Safety:** Implement mechanisms to prevent multiple threads from creating separate instances simultaneously.
* **Private Constructor:** Restrict direct instantiation by making the constructor private, forcing the use of the access point

**When to use Singleton Method Design Pattern?**

Use the Singleton method Design Pattern when:

* Consider using the Singleton pattern when you need to ensure that only one instance of a class exists in your application.
* Use it when you want to provide a straightforward way for clients to access that instance from a specific location in your code.
* If you think you might want to extend the class later, the Singleton pattern is a good choice. It allows for subclassing, so clients can work with the extended version without changing the original Singleton.
* This pattern is often used in situations like logging, managing connections to hardware or databases, caching data, or handling thread pools, where having just one instance makes sense

**Initialization Types of Singleton**

Singleton class can be instantiated by two methods:

* **Early initialization :** In this method, class is initialized whether it is to be used or not. The main advantage of this method is its simplicity. You initiate the class at the time of class loading. Its drawback is that class is always initialized whether it is being used or not.
* **Lazy initialization :** In this method, class in initialized only when it is required. It can save you from instantiating the class when you don't need it. Generally, lazy initialization is used when we create a singleton class.

Different Ways to Implement Singleton Method Design Pattern

Sometimes we need to have only one instance of our class for example a single DB connection shared by multiple objects as creating a separate DB connection for every object may be costly. Similarly, there can be a single configuration manager or error manager in an application that handles all problems instead of creating multiple managers.

Let’s see various design options for implementing such a class. If you have a good handle on static class variables and access modifiers this should not be a difficult task.

**Method 1 - Classic Implementation || Make getInstance() static to implement Singleton Method Design Pattern**