## Factory Method Pattern:

## The factory method pattern is a popular software design pattern that allows you to create objects without specifying their concrete classes. It can help you achieve performance and maintainability goals in your software solution architecture, but it also comes with some trade-offs and challenges.

Or define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate.

The Factory Method Pattern is also known as **Virtual Constructor.**

**Advantages:**

1. The factory method pattern can improve performance by reducing the number of object creations and dependencies, and by enabling lazy initialization and caching of objects.
2. It can also enhance maintainability by decoupling the client code from the concrete classes, and by allowing you to add new types of objects without modifying the existing code.
3. The factory method pattern can also support the principle of open-closed design, which states that software entities should be open for extension but closed for modification.

**Disadvantages:**

1. The factory method pattern can also introduce some drawbacks that can affect performance and maintainability. For example, it can increase the complexity and size of the code, as you need to create a separate factory class or method for each type of object.
2. It can also introduce an extra level of abstraction and indirection, which can make the code harder to understand and debug. Moreover, it can create tight coupling between the factory and the concrete classes, which can make the code less flexible and testable.

**When to use:**

When using the factory method pattern, you need to take into account several factors that can impact the performance and maintainability of your software solution architecture. These include the number and variety of objects to create and manage, the frequency and cost of object creation and initialization, the desired level of abstraction and encapsulation, the degree of flexibility and extensibility required, as well as the trade-off between performance and readability.

**Best Practice :**

To balance performance and maintainability when using the factory method pattern, you should adhere to certain best practices.

1. This includes only using the factory method pattern when it makes sense for your problem domain and design goals, avoiding creating unnecessary or redundant factory classes or methods, and using meaningful and consistent naming conventions for your factory classes or methods.
2. Additionally, you should document your factory classes or methods and explain their purpose and logic.
3. Moreover, you can use dependency injection or inversion of control to reduce coupling and improve testability. Furthermore, you can use design principles and patterns that complement the factory method pattern, such as single responsibility, interface segregation, and dependency inversion.

**Example:**

We define some main concepts related to this pattern. **The factory method pattern loosens the coupling code by separating our Product‘s construction code from the code that uses this Product**. This design makes it easy to extract the Product construction independently from the rest of the application. Besides, it allows the introduction of new products without breaking existing code.

1. we define the MotorVehicle interface. This interface only has a method build(). This method is used to build a specific motor vehicle.

**public** **interface** **MotorVehicle** {

**void** **build**();

}

1. Implement the concrete classes that implement the MotorVehicle interface. We create two types: Motorcycle and Car.

**public** **class** **Motorcycle** **implements** **MotorVehicle** {

@Override **public** **void** **build**() {

System.out.println("Build Motorcycle");

}

}

**public** **class** **Car** **implements** **MotorVehicle** {

@Override **public** **void** **build**() {

System.out.println("Build Car");

}

}

1. we create the MotorVehicleFactory class. This class is responsible for creating every new vehicle instance. It’s an abstract class because it makes a specific vehicle for its particular factory.

**public** **abstract** **class** **MotorVehicleFactory** {

**public** MotorVehicle **create**() {

**MotorVehicle** vehicle = createMotorVehicle();

vehicle.build();

**return** vehicle;

}

**protected** **abstract** MotorVehicle **createMotorVehicle**();

}

the method create() calls to the abstract method createMotorVehicle() to create a specific type of motor vehicle. That’s why each particular motor vehicle factory must implement its correct MotorVehicle type. Previously, we implemented two MotorVehicle types, Motorcycle and Car.

1. First, the *MotorcycleFactory*class:

**public** **class** **MotorcycleFactory** **extends** **MotorVehicleFactory** {

@Override

**protected** MotorVehicle **createMotorVehicle**() {

**return** **new** **Motorcycle**();

}

}

Then, the CarFactory class:

**public** **class** **CarFactory** **extends** **MotorVehicleFactory** {

@Override

**protected** MotorVehicle **createMotorVehicle**() {

**return** **new** **Car**();

}

}

**public** **class** FactoryPatternClient {

**public** **static** **void** main(String[] args) {

**MotorVehicleFactory** factory = **new** CarFactory();

factory.create();

}

}

# Abstract Factory Pattern :

# Interface or abstract class for creating families of related (or dependent) objects but without specifying their concrete sub-classes.

After our fatory class, two new vehicle brand companies are interested in our system: NextGen and FutureVehicle. These new companies build not only fuel-only vehicles but also electric vehicles. Each company has its vehicle design.

Our Factory calss is not ready to address these new scenarios. We must support electric vehicles and consider that each company has its design. To resolve these problems, we can use the [Abstract Factory Pattern](https://www.baeldung.com/java-abstract-factory-pattern).

# When to use :

# **This pattern is commonly used when we start using the Factory Method Pattern, and we need to evolve our system to a more complex system**. **It centralizes the product creation code in one place.**

1. **Multiple families of related products:** When your system needs to be configured with multiple families of related products, and you want to ensure that the products from one family are compatible with the products from another family.
2. **Flexibility and extensibility:** If you need to allow for variations or extensions in the products or their families, the Abstract Factory pattern provides a way to introduce new product variants without modifying existing client code.
3. **Encapsulation of creation logic:** The pattern encapsulates the creation of objects, making it easier to change or extend the creation process without affecting the client code.
4. **Consistency across product families:** If you want to enforce consistency among the products created by different factories, the Abstract Factory pattern can help maintain a uniform interface.

**Advantages:**

The Abstract Factory pattern has several benefits that make it useful for software design.

1. It promotes loose coupling between the client code and the concrete products, as the client only interacts with the abstract interfaces and not the implementation details.
2. It supports the principle of open/closed, as you can introduce new variants of products without breaking the existing code.
3. It helps you avoid hard-coded dependencies and create objects that are compatible with each other.
4. **Exchanging Product Families easily:**
   * The class of a concrete factory appears only once in an application, that is where it’s instantiated.
   * This makes it easy to change the concrete factory an application uses.
   * It can use various product configurations simply by changing the concrete factory.
   * Because an abstract factory creates a complete family of products, the whole product family changes at once.
5. **Promoting consistency among products:**
   * When product objects in a family are designed to work together, it’s important that an application use objects from only one family at a time. AbstractFactory makes this easy to enforce

**Disadvantages:**

1. **Complexity:**
   * Abstract Factory can introduce additional complexity to the codebase.
   * Having multiple factories and abstract product interfaces may be overkill for simpler projects.
2. **Rigidity with New Product Types:**
   * Adding new product types (classes) to the system can be challenging.
   * You might need to modify not just the concrete factories but also the abstract factory interface, potentially impacting existing code.
3. **Increased Number of Classes:**
   * As you introduce more abstract factories and product families, the number of classes in your system can grow rapidly.
   * This can make the code harder to manage and understand, particularly for smaller projects.
4. **Dependency Inversion Principle Violation:**
   * In some cases, the Abstract Factory pattern may lead to a violation of the Dependency Inversion Principle, especially if client code directly depends on concrete factory implementations rather than the abstract interfaces.
5. **Limited Extensibility:**
   * Extending the abstract factory hierarchy or introducing new product families might require modifications to multiple parts of the code, potentially leading to cascading changes and making the system less extensible.
6. **Not Ideal for Simple Systems:**
   * The Abstract Factory pattern may be overkill for smaller, less complex systems where the overhead of defining abstract factories and products outweighs the benefits of the pattern.

**Example:**

1. We already have the MotorVehicle interface. Additionally, we must add an interface to represent electric vehicles.

**public** **interface** **ElectricVehicle** {

**void** **build**();

}

1. Next, we create our abstract factory. The new class is abstract because the responsibility of object creation will be for our concrete factory.

**public** **abstract** **class** **Corporation** {

**public** **abstract** MotorVehicle **createMotorVehicle**();

**public** **abstract** ElectricVehicle **createElectricVehicle**();

}

1. Before we create the concrete factory for each company, we must implement some vehicles for our new companies. Let’s make some new classes for FutureVehicle company.

**public** **class** **FutureVehicleMotorcycle** **implements** **MotorVehicle** {

@Override **public** **void** **build**() {

System.out.println("Future Vehicle Motorcycle");

}

}

1. We do the same for the NexGen company:

**public** **class** **NextGenMotorcycle** **implements** **MotorVehicle** {

@Override **public** **void** **build**() {

System.out.println("NextGen Motorcycle");

}

}

1. Additionally, the other electric car concrete implementation:

**public** **class** **NextGenElectricCar** **implements** **ElectricVehicle** {

@Override **public** **void** **build**() {

System.out.println("NextGen Electric Car");

}

}

1. Finally, we are ready to build our concrete factories. First, FutureVehicle factory:

**public** **class** **FutureVehicleCorporation** **extends** **Corporation** {

@Override **public** MotorVehicle **createMotorVehicle**() {

**return** **new** **FutureVehicleMotorcycle**();

}

@Override **public** ElectricVehicle **createElectricVehicle**() {

**return** **new** **FutureVehicleElectricCar**();

}

}

Next, the other one:

**public** **class** **NextGenCorporation** **extends** **Corporation** {

@Override **public** MotorVehicle **createMotorVehicle**() {

**return** **new** **NextGenMotorcycle**();

}

@Override **public** ElectricVehicle **createElectricVehicle**() {

**return** **new** **NextGenElectricCar**();

}

}

**public** **class** AbstractFactoryPatternClient {

**public** **static** **void** main(String[] args) {

Corporation futureCorporation = **new** FutureVehicleCorporation();

futureCorporation.createElectricVehicle().build();

futureCorporation.createMotorVehicle().build();

System.***out***.println("---------------------------------------------");

Corporation nextgenCorporation = **new** NextGenCorporation();

nextgenCorporation.createElectricVehicle().build();

nextgenCorporation.createMotorVehicle().build();

}

}

**Singleton Pattern:**

The Singleton pattern is a design pattern that restricts the instantiation of a class to a single object. It ensures that only one instance of a class exists in the system and provides a global point of access to that instance.

### ****When to use Singleton****

### There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.

1. When the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

## Problem it solves?

There are situations where we need to have only one instance of a class throughout the lifetime of an application. For example, a logging class that records all events that occur in the application should only have one instance. If multiple instances of the logging class are created, it can cause problems such as conflicting log entries, loss of log data, and increased memory usage. The Singleton pattern solves this problem by ensuring that only one instance of the class is created and providing a global point of access to it.

## Advantages:

The Singleton pattern has several advantages, including:

1. Global point of access: The Singleton pattern provides a single point of access to the object, which can be accessed from anywhere in the system.
2. Controlled access: The Singleton pattern allows access to the object to be controlled, as only one instance of the class exists.
3. Reduced memory usage: The Singleton pattern reduces memory usage by ensuring that only one instance of the class exists in the system.
4. Simplified object creation: The Singleton pattern simplifies object creation by ensuring that only one instance of the class is created and providing a global point of access to it.

## Disadvantages

The Singleton pattern also has some disadvantages, including:

1. **Tight coupling**: The Singleton pattern can lead to tight coupling between classes, as the Singleton class is globally accessible.
2. **Thread safety**: The Singleton pattern can be prone to thread safety issues, as multiple threads can attempt to create or access the Singleton instance simultaneously.
3. **Testing difficulties:** The Singleton pattern can make unit testing difficult, as it is not easy to substitute the Singleton with a mock object during testing.

**The non-thread-safe singleton pattern can lead to several problems, including below:**

1. **Multiple instances**: In a multithreaded environment, multiple threads can create instances of the Singleton class simultaneously, resulting in multiple instances of the Singleton object. This defeats the purpose of the pattern, as the Singleton should only have one instance.
2. **Inconsistent state**: When multiple threads access the Singleton instance simultaneously, they may modify its state in unpredictable ways. This can lead to inconsistent behavior and difficult-to-debug errors.
3. **Race conditions**: The non-thread-safe Singleton pattern is susceptible to race conditions, where multiple threads try to access or modify the same data simultaneously. This can cause data corruption, deadlock, or other issues.
4. **Deadlocks**: Deadlocks can occur when two or more threads are waiting for each other to release a lock, causing them to block indefinitely.
5. **Reduced performance**: Synchronization overhead can cause reduced performance in heavily multithreaded applications.

### Understanding early Instantiation of Singleton Pattern

**class** A{

**private** **static** A obj=**new** A();//Early, instance will be created at load time

**private** A(){}

**public** **static** A getA(){

**return** obj;

}

**public** **void** doSomething(){

//write your code

}

}

### Understanding lazy Instantiation of Singleton Pattern

**class** A{

**private** **static** A obj;

**private** A(){}

**public** **static** A getA(){

**if** (obj == **null**){

**synchronized**(Singleton.**class**){

**if** (obj == **null**){

    obj = **new** Singleton();

//instance will be created at request time

}

  }

      }

**return** obj;

}

**public** **void** doSomething(){

//write your code

}

}