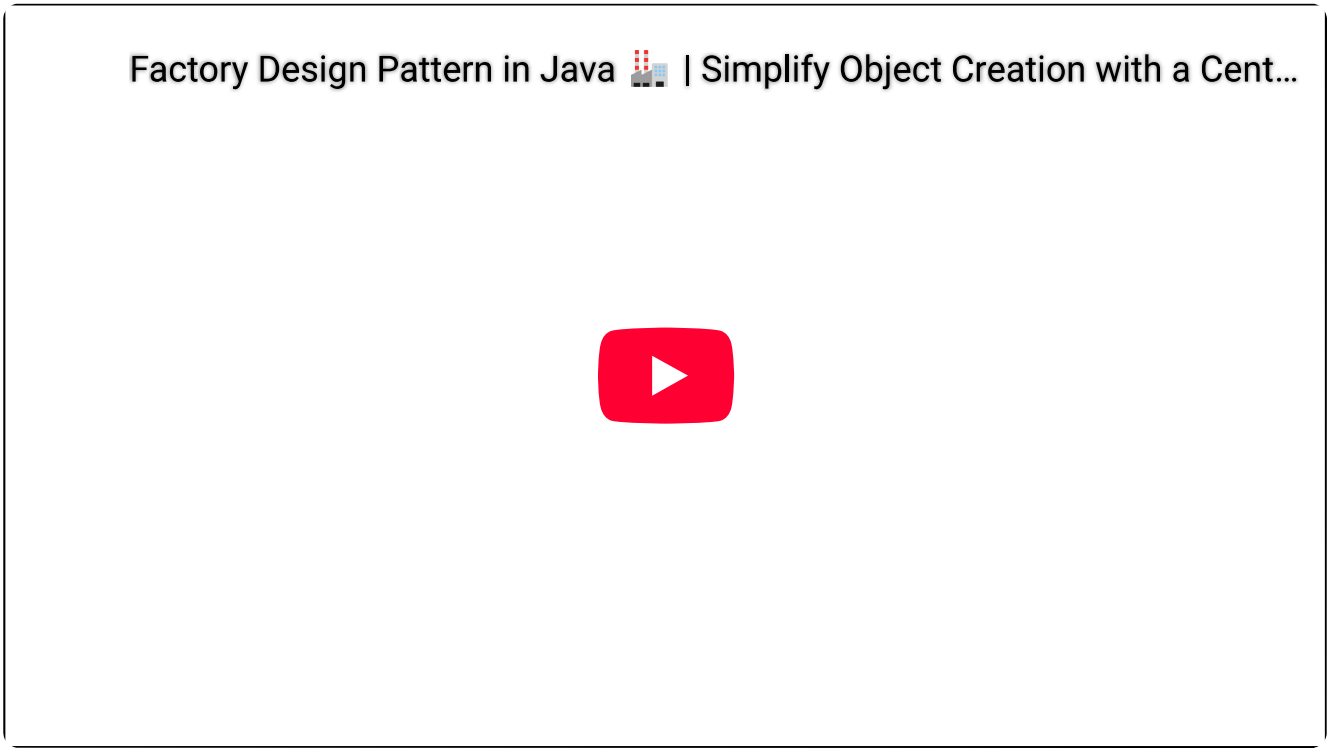
Search articles...

Factory Design Pattern



Topic Tags:

System Design LLD

Problem Statement: Creating Objects Dynamically 🚗🏍️🚚

Imagine you're building a software system to manage vehicles for a transportation company. The system needs to create different types of vehicles such as Car, Truck, and Bike. These vehicles have different characteristics, but they all share a few common behaviors like start() and stop().
Now, each time you need to create a vehicle, you have to decide which class to instantiate: Car, Truck, or Bike. If your application has many places where vehicles are created, maintaining all these object creation codes in different classes becomes messy. 😩

So, the problem is: How can we create vehicles easily and cleanly without hard-coding the class names everywhere in the code? 🤔

🚀 Problem Statement: Creating Objects Dynamically

🏗️ Scenario:

Imagine you're building a software system to manage vehicles for a transportation company. The system needs to create different types of vehicles such as:

✅ Car 🚗
✅ Truck 🚚
✅ Bike 🏍️

Each of these vehicles has different characteristics, but they all share a few common behaviors:

- 🔹 start() ▶️
- 🔹 stop() ⏹️

⚠️ The Problem

Each time you need to create a vehicle, you have to decide manually which class to instantiate:

➡️ Car 🚗
➡️ Truck 🚛
➡️ Bike 🏍️

If your application has many places where vehicles are created, maintaining all these object creation codes in different classes becomes messy 😵💥!

> 🤯 *Hard-coding class names everywhere = bad maintainability!*

❓ The Challenge

How can we create vehicles easily and cleanly without hard-coding the class names everywhere in the code?

🔧 Solving it with Traditional Approach

Let's start by solving the problem in the traditional way, where each class creates its own objects!

Java

```java
// Vehicle.java - Common interface
public interface Vehicle {
  void start();
  void stop();
}

// Car.java - Concrete class for Car
public class Car implements Vehicle {
  public void start() {
```

```
    System.out.println("Car is starting...");
  }
  public void stop() {
    System.out.println("Car is stopping...");
  }
}

// Truck.java - Concrete class for Truck
public class Truck implements Vehicle {
  public void start() {
    System.out.println("Truck is starting...");
  }
  public void stop() {
    System.out.println("Truck is stopping...");
  }
}

// Bike.java - Concrete class for Bike
public class Bike implements Vehicle {
  public void start() {
    System.out.println("Bike is starting...");
  }
  public void stop() {
    System.out.println("Bike is stopping...");
  }
}

// Main.java - Code to create vehicles
public class Main {
  public static void main(String[] args) {
    Vehicle vehicle1 = new Car();
    vehicle1.start();
    vehicle1.stop();
    Vehicle vehicle2 = new Truck();
    vehicle2.start();
    vehicle2.stop();
    Vehicle vehicle3 = new Bike();
    vehicle3.start();
    vehicle3.stop();
```

```
49    }
50  }
```

Here, the Main class creates each vehicle explicitly by calling the constructor of the respective vehicle class. But what if we need to add more vehicle types later, or if we need to change the way vehicles are created? 🤷‍♂️

🤔 Interviewer's Follow-up Questions: Can We Improve the Code?

An interviewer might ask:
• What if we need to add more vehicle types in the future? 🚙
• What if the logic of vehicle creation changes? 🔧

In this case, the code could become harder to maintain as you add more vehicle types or change the vehicle creation logic. For example, if you had to introduce new behavior or properties for vehicle creation, you would need to modify the creation code in many places, which could lead to potential errors. 😱

🛠️ Ugly Code: When We Realize the Code Needs Restructuring

Let's say, instead of creating vehicles directly, the vehicle creation process is now complex. For example, you have to choose the vehicle based on user input, configuration files, or network requests. If you don't address this early on, the object creation code quickly becomes cumbersome and ugly. 🤦‍♂️

It might look something like this:

Java

```java
// Main.java becomes a mess as you add more vehicle creation logic
public class Main {
  public static void main(String[] args) {
    String vehicleType = "Truck"; // Imagine this value is dynamic
    Vehicle vehicle;
    if (vehicleType.equals("Car")) {
      vehicle = new Car();
    } else if (vehicleType.equals("Truck")) {
      vehicle = new Truck();
    } else if (vehicleType.equals("Bike")) {
      vehicle = new Bike();
```

```
    } else {
      throw new IllegalArgumentException("Unknown vehicle type");
    }
    vehicle.start();
    vehicle.stop();
  }
}
```

This code is fragile. If we want to add another vehicle type, we need to modify this code again, which is error-prone and hard to maintain. 😖

🦸 The Savior: Factory Design Pattern

Now, let's introduce the Factory Design Pattern to rescue us. 🦸 The Factory Pattern will allow us to handle the object creation in a centralized manner, so that we don't need to keep repeating the logic of choosing which vehicle to create in multiple places.

The Factory Design Pattern is named after a "factory" because, just like a factory produces different types of products, the pattern provides a central place (the factory) to create objects of different types. 🏭 Instead of directly instantiating objects, the factory method is responsible for producing the correct object, making the system more flexible and organized.

🏭 Solving the Problem with Factory Design Pattern

Here's how we can solve this problem by introducing a Factory that creates the vehicles:

Java

```java
// Vehicle.java - Common interface
public interface Vehicle {
  void start();
  void stop();
}

// Concrete vehicle classes remain the same
public class Car implements Vehicle {
  public void start() {
    System.out.println("Car is starting...");
  }
  public void stop() {
```

```
    System.out.println("Car is stopping...");
  }
}

public class Truck implements Vehicle {
  public void start() {
    System.out.println("Truck is starting...");
  }
  public void stop() {
    System.out.println("Truck is stopping...");
  }
}

public class Bike implements Vehicle {
  public void start() {
    System.out.println("Bike is starting...");
  }
  public void stop() {
    System.out.println("Bike is stopping...");
  }
}

// VehicleFactory.java - Factory to create vehicles
public class VehicleFactory {
  public static Vehicle getVehicle(String vehicleType) {
    if (vehicleType.equals("Car")) {
      return new Car();
    } else if (vehicleType.equals("Truck")) {
      return new Truck();
    } else if (vehicleType.equals("Bike")) {
      return new Bike();
    } else {
      throw new IllegalArgumentException("Unknown vehicle type");
    }
  }
}

// Main.java - Simplified with Factory
public class Main {
  public static void main(String[] args) {
    Vehicle vehicle1 = VehicleFactory.getVehicle("Car");
```

```
54      vehicle1.start();
55      vehicle1.stop();
56      Vehicle vehicle2 = VehicleFactory.getVehicle("Truck");
57      vehicle2.start();
58      vehicle2.stop();
59      Vehicle vehicle3 = VehicleFactory.getVehicle("Bike");
60      vehicle3.start();
61      vehicle3.stop();
62    }
63  }
```

🏆 Advantages of Using the Factory Design Pattern

Let's review how the Factory Pattern improves our solution:

1. Centralized Object Creation:

The VehicleFactory class handles all the logic of creating vehicles. Now, you only need to call the getVehicle() method with the desired vehicle type, and the factory will take care of the rest. This makes the code much cleaner and easier to maintain. ✨

2. Scalability:

If you want to add a new vehicle type, say Bus, you only need to add the Bus class and update the VehicleFactory class. No changes are needed in the rest of the application. 🔄

3. Encapsulation:

The client code (in Main.java) no longer needs to know how to create the vehicles. The logic is abstracted away in the VehicleFactory class, which makes the system easier to manage. 🔑

🌎 Real-life Use Cases and Examples of the Factory Design Pattern

The Factory Design Pattern is widely used in real-world software development. Here are some examples:

• Database Connections:

When creating a connection to different types of databases (e.g., MySQL, PostgreSQL, Oracle), the factory can handle the creation of database connections based on configuration parameters without exposing the details to the client. 💾

• User Interface Elements:

In GUI libraries, different platforms (Windows, Mac, Linux) may require different implementations of buttons, windows, and menus. A factory pattern can be used to create the appropriate UI elements for the specific platform. 💻

• Logging:
Depending on the logging requirements (e.g., logging to a file, console, or database), a factory can create the correct type of logger, allowing different components of the system to use the logger without knowing its exact implementation. 📜

🎯 Conclusion

The Factory Design Pattern simplifies object creation by centralizing it in a factory, making the code cleaner, more maintainable, and easier to extend. It ensures that we can easily add new types or change the instantiation logic without touching the client code. This pattern is highly beneficial when your application needs to create a variety of objects in a flexible and scalable way. 🏗️