Design Patterns: Factory Method Pattern in TypeScript

Master the Factory Method Pattern and Easily Handle the Creation of Objects.w me on Medium or Twitter.

Welcome to the **Design Patterns in TypeScript** series, which introduces some useful design patterns in web development using TypeScript.

Previous articles are as follows:

Strategy Pattern in TypeScript

Chain of Responsibility Pattern in TypeScript

Observer Pattern in TypeScript

Template Method Pattern in TypeScript

Adapter Pattern in TypeScript

Factory Method Pattern in TypeScript

Abstract Factory Pattern in TypeScript

Design patterns are very important for web developers and we can write better code by mastering them. In this article, I will use **TypeScript** to introduce the **Factory Method Pattern.**

The factory method pattern, also known as the factory pattern, is also called the polymorphic factory pattern, which belongs to the creational pattern.

In the factory method pattern, the factory parent class is responsible for defining the public interface for creating product objects, while the factory subclass is responsible for generating specific product objects. **The purpose of this is to delay the instantiation of the product class to the factory subclass.** That is, through the factory subclass to determine which specific product class should be instantiated.

In the above picture, I simulated the user’s car purchase process. Bytefer and Chris1993 ordered SuperX01 and SuperX02 models from the SuperX01 and SuperX02 factories respectively, and then the factory produced the corresponding models and delivered them to the users after the production was completed.

Let’s look at how to use the factory method to describe the process of producing a given model of car in a vehicle factory.

In order to better understand the following code, let’s first look at the corresponding UML diagram:

The factory method pattern includes the following roles:

* **Product(Vehicle)**: abstract product
* **Concrete Product(SuperX01)**: concrete product
* **Factory(VehicleFactory)**: abstract factory
* **ConcreteFactory(SuperX01Factory)**: concrete factory

Next, we define an abstract class Vehicle and its two subclasses SuperX01 and SuperX02 to represent different types of vehicles.

abstract class Vehicle {  
 abstract run(): void;  
}class SuperX01 extends Vehicle {  
 run(): void {  
 console.log("SuperX01 start");  
 }  
}class SuperX02 extends Vehicle {  
 run(): void {  
 console.log("SuperX02 start");  
 }  
}

Then, we define the VehicleFactory class to represent the vehicle factory. The abstract class contains an abstract method produceVehicle, which is the so-called factory method.

abstract class VehicleFactory {  
abstract produceVehicle(): Vehicle;  
}

Based on the VehicleFactory abstract class, we define the SuperX01Factory and SuperX02Factory factory classes for the production of **SuperX01** and **SuperX02** models of vehicles:

class SuperX01Factory extends VehicleFactory {  
 produceVehicle(): Vehicle {  
 return new SuperX01();  
 }  
}class SuperX02Factory extends VehicleFactory {  
 produceVehicle(): Vehicle {  
 return new SuperX02();  
 }  
}

After creating the SuperX01Factory and SuperX02Factory factory classes, we can start producing cars:

const superX01Factory = new SuperX01Factory();  
const superX02Factory = new SuperX02Factory();const superX01Vehicle = superX01Factory.produceVehicle();  
const superX02Vehicle = superX02Factory.produceVehicle();superX01Vehicle.run();  
superX02Vehicle.run();

When you successfully run the above code, the terminal will output the following result:

SuperX01 start  
SuperX02 start

Finally, let’s summarize the usage scenarios of the factory method pattern:

* In the factory method pattern, the abstract factory class only needs to provide an interface for creating products, and its subclasses determine the specific objects to be created, using object-oriented polymorphism and the Liskov substitution principle, when the program is running , subclass objects will override parent class objects, making the system easier to extend.

If you have any questions, please feel free to leave me a message. I will continue to introduce other patterns later, if you are interested, you can follow me on [Medium](https://medium.com/@bytefer) or [Twitter](https://twitter.com/Tbytefer).