



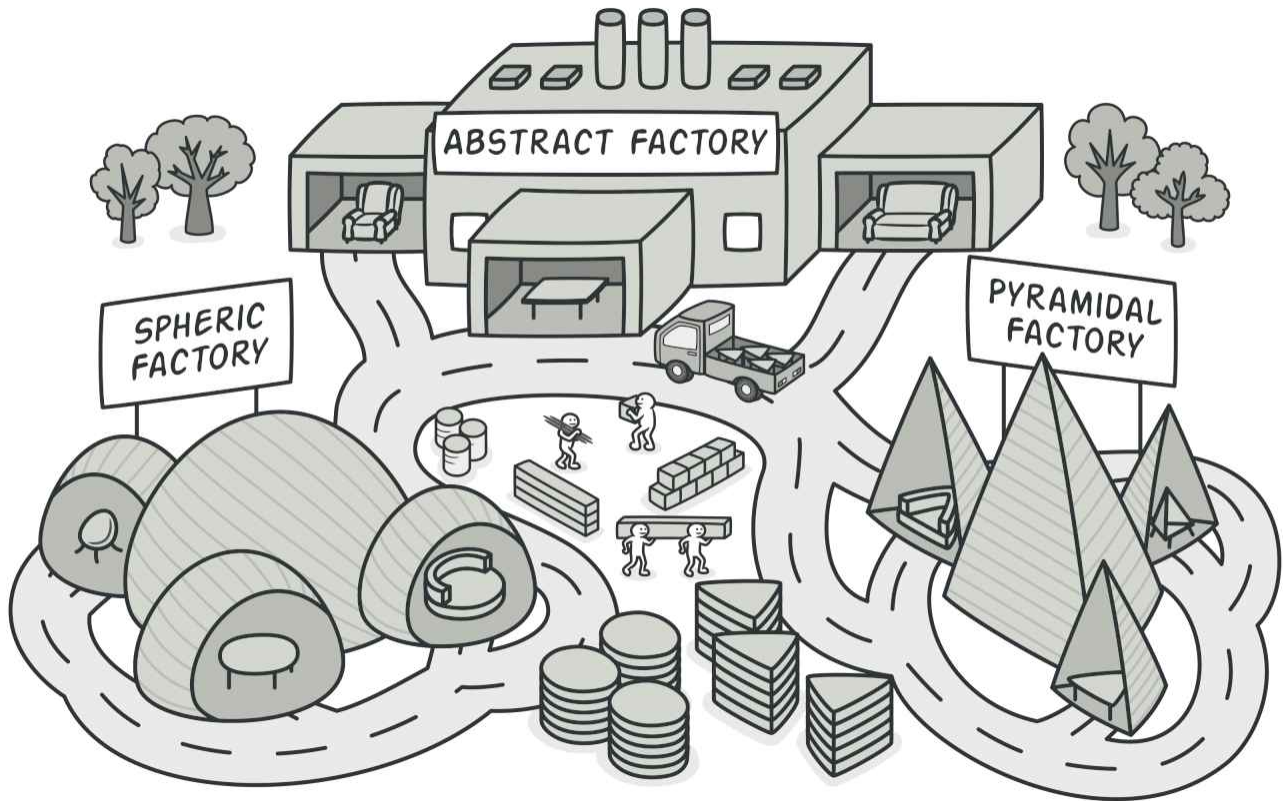
29-Factory Pattern

Author: [Vincent Lau](#)

Note: This material is intended for educational purposes only. All rights reserved. Any unauthorized sharing or copying of this material, in any form, to any individual or party, for any use without prior permission, is strictly prohibited.

Learning Objectives

- Understand how builder pattern benefits programming.
- Understand how to code a builder pattern for a normal class.



Introduction



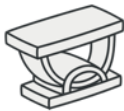






The **Factory Pattern** is a creational design pattern that **provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created**. It helps in creating instances of classes without exposing instantiation logic to the client.

The Factory Pattern promotes **loose coupling between client code and the classes being instantiated**.

Problem

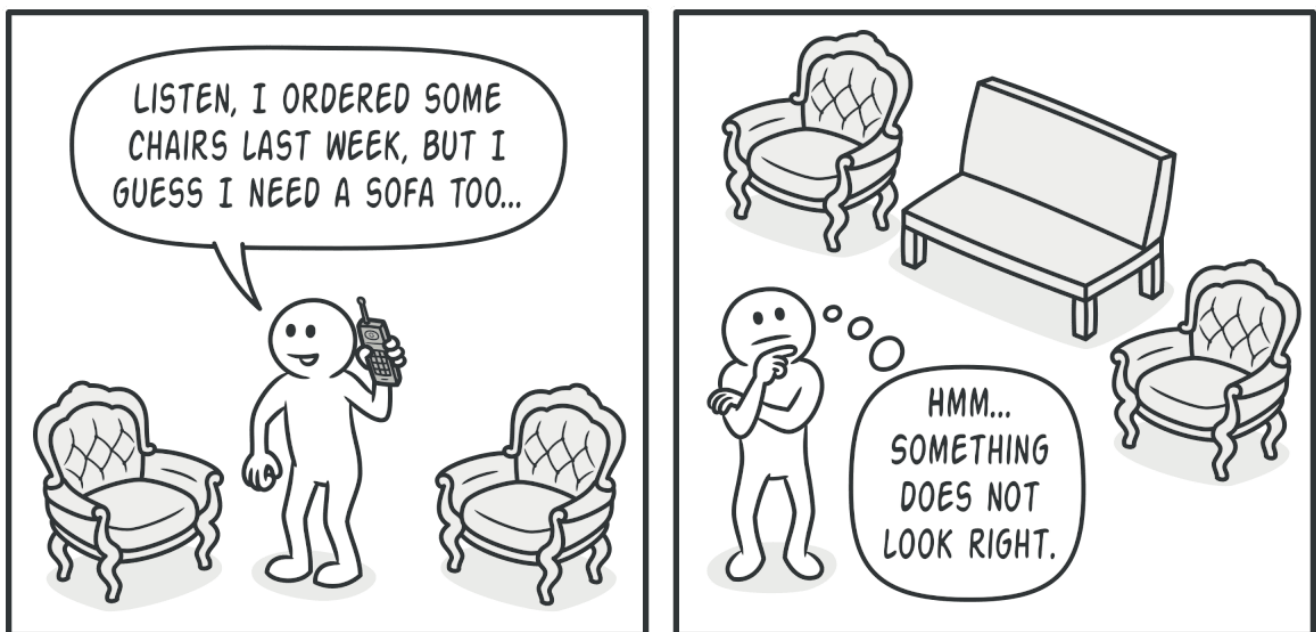
Imagine that you're creating a furniture shop simulator. Your code consists of classes that represent:

- A family of related products, say: `Chair` + `Sofa` + `CoffeeTable` .
- Several variants of this family. For example, products `Chair` + `Sofa` + `CoffeeTable` are available in these variants: `Modern` , `Victorian` , `ArtDeco` .

	Chair	Sofa	Coffee Table
Art Deco			
Victorian			
Modern			

Product families and their variants

You need a way to create individual furniture objects so that they match other objects of the same family. Customers get quite mad when they receive non-matching furniture.

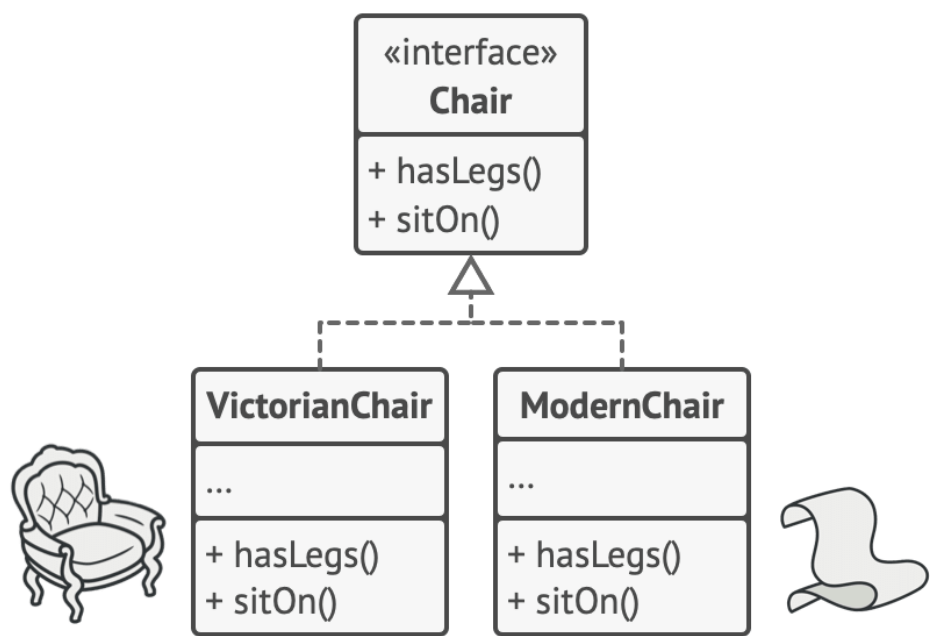


A Modern-style sofa doesn't match Victorian-style chairs

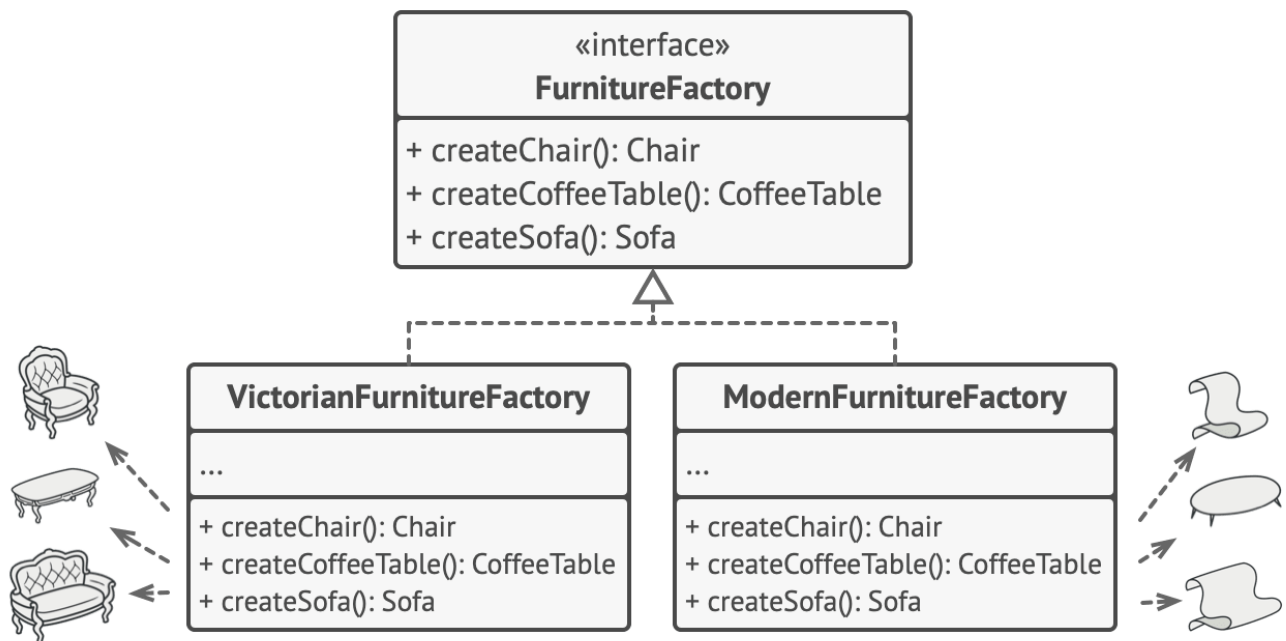
It's crucial to avoid altering the existing code while introducing new products or product categories into the program. Given the frequent updates in furniture catalogs by vendors, modifying the core code each time would be impractical and undesirable.

Solution - Factory Pattern

The initial step proposed by the Abstract Factory pattern involves defining distinct interfaces for each product within the product family, such as chairs, sofas, or coffee tables. Subsequently, all variations of products can adhere to these interfaces. To illustrate, all variations of chairs can implement the Chair interface, and all variations of coffee tables can implement the CoffeeTable interface, and so forth.



The subsequent step involves defining the Abstract Factory—an interface encompassing a collection of creation methods for all products within the product family. These methods, such as `createChair`, `createSofa`, and `createCoffeeTable`, are designed to return abstract product types that correspond to the previously identified interfaces: **Chair**, **Sofa**, **CoffeeTable**, and others.



Now, let's consider the product variations. To accommodate each variant within a product family, we establish distinct factory classes rooted in the AbstractFactory interface. These factories serve to produce products of specific categories. For instance, the ModernFurnitureFactory specializes in crafting ModernChair, ModernSofa, and ModernCoffeeTable objects.

The client code must interact with both factories and products through their corresponding abstract interfaces. This approach empowers you to modify the factory type provided to the client code, as well as the product variant received by the client code, without causing disruptions to the core functionality of the client code.

Sample Classes

```
1 public interface Chair {
2
3     boolean hasLeg();
4     // ...
5
6     public static Chair create(FurnitureFactory factory) {
7         return factory.createChair();
8     }
9 }
```

```
1 public class ModernChair implements Chair {
2
3     @Override
4     public boolean hasLeg() {
5         return true;
6     }
7 }
```

```
1 public class VictorianChair implements Chair {
2
3     @Override
4     public boolean hasLeg() {
5         return false;
6     }
7 }
```

```
1 public interface Sofa {
2
3     boolean isSingleSeat();
4     // ...
5 }
```

```
1 public interface Sofa {
2
3     boolean isSingleSeat();
4     // ...
5
6     public static Sofa create(FurnitureFactory factory) {
7         return factory.createSofa();
8     }
9 }
```

```
1 public class VictorianSofa implements Sofa {
2
3     @Override
4     public boolean isSingleSeat() {
5         return false;
6     }
7 }
```

```
1 public interface FurnitureFactory {
2
3     Chair createChair();
4
5     Sofa createSofa();
6
7     public static FurnitureFactory create(Style style) throws Exception {
8         switch (style) {
9             case MODERN:
10                 return new ModernFactory();
11             case VICTORIAN:
12                 return new VictorianFactory();
13             }
14         throw new Exception("Unknown Style of Furniture");
15     }
16 }
```

```

1 public class ModernFactory implements FurnitureFactory {
2
3     @Override
4     public Chair createChair() {
5         System.out.println("I have my own way to create Modern chair");
6         return new ModernChair();
7     }
8
9     @Override
10    public Sofa createSofa() {
11        System.out.println("I have my own way to create Modern sofa");
12        return new ModernSofa();
13    }
14 }

```

```

1 public class VictorianFactory implements FurnitureFactory {
2
3     @Override
4     public Chair createChair() {
5         System.out.println("I have my own way to create Victorian chair");
6         return new VictorianChair();
7     }
8
9     @Override
10    public Sofa createSofa() {
11        System.out.println("I have my own way to create Victorian sofa");
12        return new VictorianSofa();
13    }
14 }

```

The following codes achieve loose coupling. Clients don't have to care about how the Chair and Sofa are being created by the corresponding factory.

In the future, this code pattern enables better maintainability and easier review process. We can confirm the client codes should not be revised if we just need to add a new Factory or new Furniture.

```

1 public class Customer {
2     public static void main(String[] args) throws Exception {
3         // Get Factory
4         FurnitureFactory factory = FurnitureFactory.create(Style.MODERN);
5         // Get chair

```



```
6     Chair chair = factory.createChair();
7     System.out.println(chair.hasLeg()); // true
8     // Get Sofa
9     Sofa sofa = factory.createSofa();
10    System.out.println(sofa.isSingleSeat()); // true
11 }
12 }
```

Advantages of Factory Pattern

1. Encapsulation: The pattern encapsulates the object creation process in a separate class, promoting better **encapsulation** and **abstraction**.
2. Flexibility: The client code can work with interfaces or abstract classes, making it easy to **switch between different implementations without affecting the client code**.
3. Centralized Object Creation: Factory classes centralize object creation logic, which can be helpful for maintaining consistency and **avoiding code duplication**.
4. **Loose Coupling**: The client code is not tightly coupled with concrete class implementations, enhancing code **maintainability and scalability**.

Disadvantages of Factory Pattern

1. Complexity: Adding new concrete product classes might require creating corresponding concrete factory classes, which can lead to more code.
2. Code Overhead: The pattern introduces extra classes and interfaces, which can increase the overall codebase and development effort.
3. Increased Indirection: Using factory classes adds an additional layer of indirection, which might be unnecessary for simple cases.

Summary

Using the Factory Pattern in Java is beneficial when you need to create objects with complex instantiation logic, when you want to centralize object creation, or when you want to abstract away the concrete class implementations from the client code. It helps in achieving loose coupling and maintainable code but might introduce some additional complexity due to the presence of factory classes.