Theory: Factory method

23 minutes 0 / 5 problems solved

The **Factory Method** pattern is a good place to start, especially if you wish to understand the concept of other factory patterns. It is probably the simplest one and you can implement it for sure.

Imagine a situation that you are the boss of a factory producing something — anything you really want. You are lucky to have a qualified engineer in your team who can create any type of product at your factory provided the specification: *TYPE_A* or *TYPE_B.* This is what the Factory Method design pattern is about.

This pattern defines an interface for creating an object but leaves it to the subclasses to decide which class to instantiate. So basically, Factory Method allows the class to delegate instantiation to subclasses. The goal of any factory is to protect customers from the details of creating copies of classes or class hierarchy. Factory Method is a special case of the Template Method pattern, the variable step of which is responsible for creating the desired type of object.

Current topic:

Factory method

Topic depends on:

✕ Abstract class vs interface

✕ Encapsulating object creation

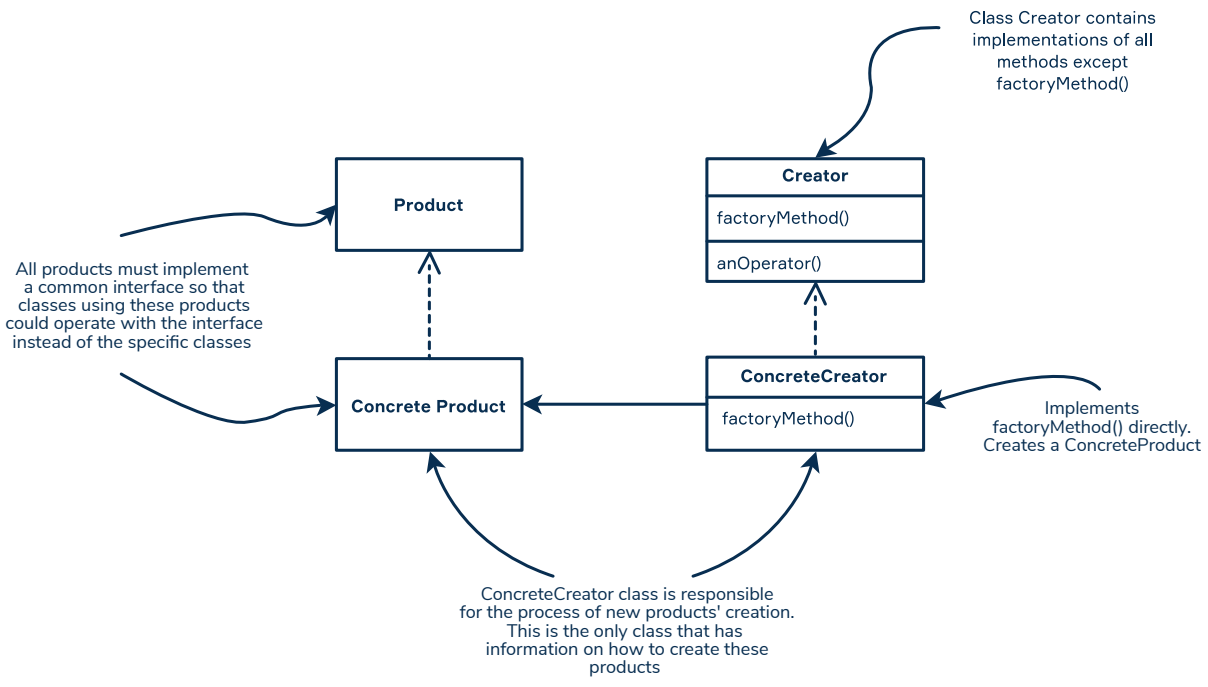
Topic is required for:

Abstract factory

§1. Structure

The Factory Method pattern has the following components:

- Creator;
- Concrete Creator;
- Product;
- Concrete Product.



These 4 components carry out different functions:

1. **Creator** declares an abstract or virtual method of creating a product. It uses the factory method in its implementation. Samples: Hero Factory, Music Factory, Furniture Factory, DB Factory.
2. **ConcreteCreator** implements a factory method that returns ConcreteProduct. Samples: Rock Music Factory, Door Furniture Factory, MongoDB Factory.
3. **Product** defines the interface of products created by the factory method. Samples: Robot, Detail, Transport, Hero, File, Furniture.
4. **ConcreteProduct** determines the specific type of products. Samples: RobotCleaner, ElfHero, MP3File, Detail13.

The pattern in JDK is available in `java.util`, `java.io` and `javax.persistence`.

§2. Practice example

Let's make our abstract example from the beginning a little more vivid and detailed. As you remember, you are the boss of a factory. Suppose the factory makes tables: they are truly indispensable in the house. You work with a qualified employee, an engineer, who, as you might have guessed, is your factory method.

First, let's define the abstract class *Table:*

```
abstract class Table {
    private String name;

    Table(String name) {
        this.name = name;
    }

    String getName() {
        return name;

    }



    void attachLegs() {

        System.out.println("Attaching Legs");

    }



    void attachTableTop() {

        System.out.println("Attaching tabletop");

    }

}
```

Second, we should define two specific tables: *TableOffice* and *TableKitchen* classes. Note that the abstract class has a constructor, which is sometimes tricky for Java developers with little experience.

```
class TableOffice extends Table {
    TableOffice(String name) {
        super(name);
    }
}

class TableKitchen extends Table {
    TableKitchen(String name) {
        super(name);

    }

}
```

Third, let's create your factory. I called it *TableStore,* the implementation of abstract *TableFactory:*

```
abstract class TableFactory {

    abstract Table createTable(String type);

    Table orderTable(String type) {
        Table table = createTable(type);
        if (table == null) {

        System.out.println("Sorry, we are not able to create this kind of tabl
e\n");
            return null;

        }

        System.out.println("Making " + table.getName());

        table.attachLegs();

        table.attachTableTop();

        System.out.println("Created " + table.getName() + "\n");

        return table;

    }

}


class TableStore extends TableFactory {

    @Override

    Table createTable(String type) {

        if (type.equals("office")) {

            return new TableOffice("Office Table");

        } else if (type.equals("kitchen")) {

            return new TableKitchen("Kitchen Table");

        } else return null;

    }

}
```

Finally, our *TestDrive* code and the output:

```
class TestDrive {
    public static void main(String[] args) {
        TableStore tableStore = new TableStore();
        Table strangeTable = tableStore.orderTable("Mysterious table");
        Table officeTable = tableStore.orderTable("office");
        Table kitchenTable = tableStore.orderTable("kitchen");
    }
}
```

```
Sorry, we are not able to create this kind of table

Making Office Table
Attaching Legs
Attaching tabletop
Created Office Table

Making Kitchen Table
Attaching Legs

Attaching tabletop

Created Kitchen Table
```

§3. Conclusion

Factory Method comes in handy in situations when you need to:

- have a complicated process for constructing the objects;
- reduce the time to add another product;
- replace one product with another.

Report a typo

196 users liked this theory. **37** didn't like it. **What about you?**

Start practicing

Comments (18) Hints (0) Useful links (3) Show discussion