Factory method

This pattern defines an interface for creating an object but leaves it to the subclasses to decide which class to instantiate.
Simply put, the Factory Method allows the class to delegate instantiation to subclasses.
The goal of any factory is to relieve a client from the task of creating class instances or class hierarchy.

Factory Method is a special case of the **Template Method pattern**, the variable step responsible for creating the desired type of object.

The structure of Factory Method

The Factory Method pattern has the following components:

- Creator;
- ConcreteCreator;
- Product;
- ConcreteProduct.

https://ucarecdn.com/7ddd67b8-4b2d-455e-b27c-42aab7213234/

These 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. Some of the examples are HeroFactory, MusicFactory, FurnitureFactory, DBFactory.
2. **ConcreteCreator** implements a factory method that returns ConcreteProduct. For example, RockMusicFactory, DoorFurnitureFactory, MongoDBFactory.
3. **Product** defines the interface of products created by the factory method. For example, Robot, Detail, Transport, Hero, File, Furniture.
4. **ConcreteProduct** determines a specific type of products, such as RobotCleaner, ElfHero, MP3File, Detail13.

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

Practice example

You remember, you are the CEO 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:
the 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 our factory. We called it TableStore, the implementation of the 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 table\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, here is 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 unable 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
```

Conclusion

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

- deal with a complicated process of constructing objects;
- reduce the time for adding another product;
- replace one product with another.