Encapsulating object creation

Sometimes we have a hierarchy of classes with one base class (or interface) and several subclasses, and we need to create a new subclass object depending on its type.

Instead of writing a new operator in client code where we will use the objects, it is convenient to encapsulate the code for creating objects in a separate place and call it from the client code.

These places are known as **factories** that produce instances of classes related to the same hierarchy.

They allow us to simplify the client code and protect it from changes when new classes are added to the hierarchy.

So, a **factory** is a way of creating objects when one part of a program (a class or a method) creates objects and another one processes them.

There are several kinds of factories: **static factory**, **simple factory**, **factory method** and **abstract factory**.

In this topic, we will study only the first two; they are often referred to as *idioms* rather than design patterns. But the latter two are real design patterns that rely on similar principles as those idioms.

Static factory idiom

As an example, we will consider the following hierarchy of computers:

```
class Computer {

    private long ram;
    private long cpu;

    // getters and setters
}

class PC extends Computer {

    // additional members
}

class Laptop extends Computer {

    // additional members
}
```

The static factory is the simplest factory that we can write. It has one static method which creates objects of the hierarchy. The method takes a required type as a string or enum argument and returns a corresponding subclass instance through the base class.

To create new computers, we invoke the static factory passing a required type:

```
class ComputerStaticFactory {

    public static Computer newInstance(String type) {
        if (type.equals("Computer")) {
            return new Computer();
        } else if (type.equals("PC")) {
            return new PC();
        } else if (type.equals("Laptop")) {
            return new Laptop();
        }
        return null; // if not a suitable type
    }
}
```

Depending on the passed type, a suitable branch will work. We also can write the same using switch.
The following client code creates two computers: a laptop and a PC:

```
public class FactoryClient {

    public static void main(String args[]) {

        Computer pc = ComputerStaticFactory.newInstance("PC");
        System.out.println(pc instanceof PC); // prints "true"

        Computer laptop = ComputerStaticFactory.newInstance("Laptop");
        System.out.println(laptop instanceof Laptop); // prints "true"
    }
}
```

It is not difficult to declare a new class that extends Computer and modify the factory to create instances of a new class.
There are several possible implementation features:

- the method newInstance of ComputerStaticFactory takes an enum type to restrict possible values;
- the method newInstance throws an exception if an unsuitable type is passed instead of returning null;

- move the method newInstance directly to the base of the hierarchy: Computer class;
- a factory can have multiple methods that produce instances of different classes or an instance of a default class.

Simple factory idiom
The simple factory idioms differ from the static factory because the method for creating objects is non-static.

```
class ComputerFactory {

    // it may contain some fields

    public Computer newInstance(String type) {
        if (type.equals("Computer")) {
            return new Computer();
        } else if (type.equals("PC")) {
            return new PC();
        } else if (type.equals("Laptop")) {
            return new Laptop();
        }
        return null;
    }
}
```

In the client code, we should create an instance of the factory and then invoke *newInstance* as shown below:

```
ComputerFactory factory = new ComputerFactory();
Computer pc = factory.newInstance("PC");
```

The simple factory idiom has the same features as the static factory. But, unlike the static factory, it is possible to create multiple differently parameterized factories to control instantiating. You can also subclass the factory and override its non-static method that is used to create other factories.

This flexibility comes from the ability to create multiple differently parameterized factories, each responsible for creating objects with specific configurations or behaviors. In contrast, with the static factory pattern, the creation logic is typically static and fixed within the class itself.

Another thing to remember is that a factory may have multiple clients that need to create objects. They do not need to duplicate the same code that calls the new operator to make an instance.