Factory Comparison

This article will show the difference between:

1. Factory
2. Creation method
3. Static creation (or factory) method
4. Simple factory
5. **Factory Method** pattern
6. **Abstract Factory** pattern

You can find references to these terms all around the web. Though they may look similar, they all have different meanings. A lot of people don't realize that, which leads to confusion and misunderstanding.

So let's try to figure out the difference and solve this problem once and for all.

1. Factory

Factory is an ambiguous term that stands for a function, method or class that supposed to be producing something. Most commonly, factories produce objects. But they may also produce files, records in databases, etc.

For instance, any of these things may be casually referenced as a "factory":

- a function or method that creates a program's GUI;
- a class that creates users;
- static method that calls a class constructor in a certain way;
- one of the creational design patterns.

Usually, when someone says a word “factory,” the exact meaning is supposed to be clear from a context. But if you’re in doubt, just ask. There’s a chance that the author is ignorant.

2. Creation method

Creation method defined in the **Refactoring To Patterns** book as “a method that creates objects.” This means that every result of a factory method pattern is a “creation method” but not necessarily the reverse. It also means that you can substitute the term “creation method” wherever Martin Fowler uses the term “factory method” in **Refactoring** and wherever Joshua Bloch uses the term “static factory method” in **Effective Java**.

In reality, the creation method is just a wrapper around a constructor call. It may just have a name that better expresses your intentions. On the other hand, it may help to isolate your code from changes to the constructor. It could even contain some particular logic that would return existing objects instead of creating new.

A lot of people would call such methods “factory method” just because it produces new objects: The logic is straightforward: the method creates objects and since all *factories* creates objects, this method should clearly be a *factory method*. Naturally, there’s a lot of confusion when it comes to the real **Factory Method** pattern.

In the following example, `next` is a creation method:

```
class Number {
    private $value;

    public function __construct($value) {
        $this->value = $value;
    }

    public function next() {
        return new Number ($this->value + 1);
    }
}
```

3. Static creation method

Static creation method is a creation method declared as `static`. In other words, it can be called on a class and doesn't require an object to be created.

Don't be confused when someone calls methods like this a "static factory method". That's just a bad habit. The **Factory Method** is a design pattern that relies on inheritance. If you make it `static`, you can no longer extend it in subclasses, which defeats the purpose of the pattern.

When a static creation method returns new objects it becomes an alternative constructor.

It might be useful when:

- You have to have several different constructors that have different purposes but their signatures match. For instance, having both `Random(int max)` and `Random(int min)` is impossible in Java, C++, C#, and many other languages. And the most popular workaround is to create several static methods that call the default constructor and set appropriate values afterward.

- You want to reuse existing objects, instead of instancing new ones (see, the **Singleton** pattern). Constructors in most programming languages have to return new class instances. The static creation method is a workaround to this limitation. Inside a static method, your code can decide whether to create a fresh instance by calling the constructor or return an existing object from some cache.

In the following example, the `load` method is a static creation method. It provides a convenient way to retrieve users from a database.

```
class User {
    private $id, $name, $email, $phone;

    public function __construct($id, $name, $email, $phone) {
        $this->id = $id;
        $this->name = $name;
        $this->email = $email;
        $this->phone = $phone;
    }
```

```
    public static function load($id) {
        list($id, $name, $email, $phone) = DB::load_data('users', 'id', 'name'
        $user = new User($id, $name, $email, $phone);
        return $user;
    }
}
```

4. *Simple factory* pattern

The **Simple factory** pattern ⓘ describes a class that has one creation method with a large conditional that based on method parameters chooses which product class to instantiate and then return.

People usually confuse *simple factories* with a general *factories* or with one of the creational design patterns. In most cases, a simple factory is an intermediate step of introducing **Factory Method** or **Abstract Factory** patterns.

A simple factory is usually represented by a single method in a single class. Over time, this method might become too big, so you may decide to extract parts of the method to subclasses. Once you do it several times, you might discover that the whole thing turned into the classic *factory method* pattern.

By the way, if you declare a simple factory `abstract`, it doesn't magically become the *abstract factory* pattern.

Here's an example of *simple factory*:

```
class UserFactory {
    public static function create($type) {
        switch ($type) {
            case 'user': return new User();
            case 'customer': return new Customer();
            case 'admin': return new Admin();
            default:
                throw new Exception('Wrong user type passed.');
        }
```

```
    }
}
```

5. *Factory Method* pattern

The **Factory Method** ⓘ is a creational design pattern that provides an interface for creating objects but allows subclasses to alter the type of an object that will be created.

If you have a creation method in base class and subclasses that extend it, you might be looking at the factory method.

```php
abstract class Department {
    public abstract function createEmployee($id);

    public function fire($id) {
        $employee = $this->createEmployee($id);
        $employee->paySalary();
        $employee->dismiss();
    }
}

class ITDepartment extends Department {
    public function createEmployee($id) {
        return new Programmer($id);
    }
}

class AccountingDepartment extends Department {
    public function createEmployee($id) {
        return new Accountant($id);
    }
}
```

6. *Abstract Factory* pattern

The **Abstract Factory** ⓘ is a creational design pattern that allows producing families of related or dependent objects without specifying their concrete classes.

What are the "families of objects"? For instance, take this set of classes: `Transport` + `Engine` + `Controls`. There might be several variants of these:

1. `Car + CombustionEngine + SteeringWheel`
2. `Plane + JetEngine + Yoke`

If your program doesn't operate with product families, then you don't need an abstract factory.

And again, a lot of people mix-up the *abstract factory* pattern with a simple factory class declared as `abstract`. Don't do that!

Afterword

Now that you know the difference, take a fresh look at the design patterns:

- **Factory Method**
- **Abstract Factory**

RETURN

← Abstract Factory

READ NEXT

Builder →