Design Patterns - Factory Method Pattern

Normally, we create objects using constructors.

```
Rectangle rect = Rectangle();
```

Is this a good thing?

No. In fact this approach has a major architectural disadvantage:

- You have to know the **exact class type** you want to instantiate.

Why is this bad?

Think of this: What if we wanted to create an app that used different types of shapes? Ones that we have not even thought of yet?

Design Patterns - Factory Method Pattern

Let's look at a scenario:

1. Assume that you are creating a space-shooter game.
2. In this game you have **different types of bullets** that users could use.
   a. But, you do not know which bullets the users will want to use.
   b. So how do you create them?
3. Additionally, you might in the future have new bullet types.

```
FastBullet fb = FastBullet();
SlowBullet sb = SlowBullet();
SplashBullet spb = SplashBullet();
? ob = ?();
```

```
fb.explode();
sb.explode();
. . .
```

The main problem with this approach is that your code is unnecessarily aware of the type of objects that it creates and thus it is strongly coupled with the type.

Design Patterns - Factory Method Pattern

Factory Method is a creational design pattern that lets us abstract-out the creation logic of specific class instances, so it provides a mechanism for creation of objects without exposing the instantiation logic to the client.

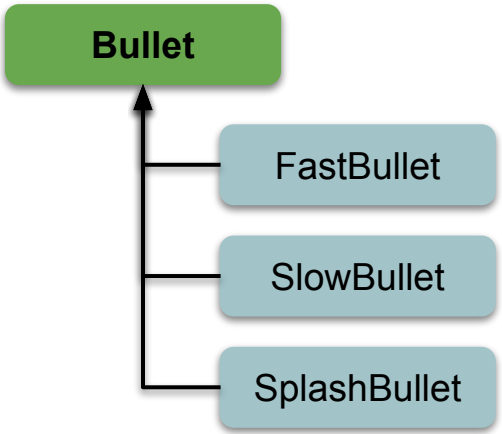
There are two main and crucial points in understanding this pattern:

1. Objects are created by calling a factory method **instead** of calling **a constructor**.
2. Objects are created through an **abstraction not a concretion**.

So, in the **Factory Method** design pattern, we create objects without exposing the creation logic to the caller, and the caller refers to newly created object through a common interface.

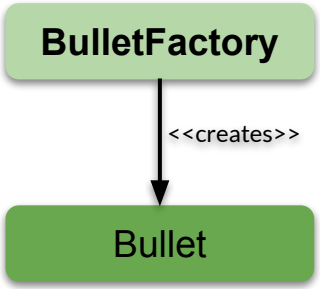
Design Patterns - Factory Method Pattern

If we make the Bullet an **abstract class** we can make it to be the parent class of all Bullets:



So how does this help us?

We can now create Bullet instances based only on the Bullet interface/contract instead of the specific Bullet definition. We then use the Factory Method design pattern as follows:



Design Patterns - Factory Method Pattern

The idea is pretty straightforward. In code, the caller will ask for a Bullet instance based on some abstract name. For example:

```
Bullet myBullet001 = BulletFactory.create(Bullet.FAST_BULLET);
Bullet myBullet002 = BulletFactory.create(Bullet.SPLASH_BULLET);
```

The main advantage is that the caller does not need to know how the factory creates the bullet. The caller only needs to know **HOW to call** the factory and **with what initialization data**.

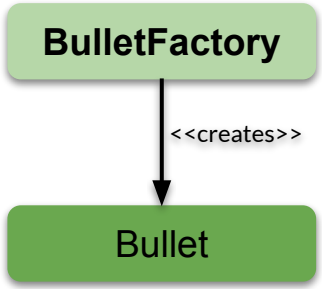
We could even make this more generic, by using some type of context data to initialize the instance:

```
BulletBuildContext context = // some initialization data
Bullet myBullet002 = BulletFactory.create(context);
```

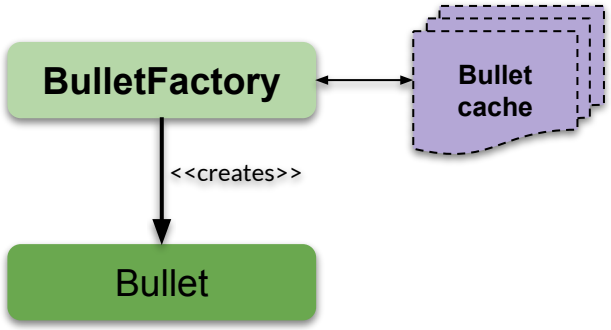
Design Patterns - Factory Method Pattern

One example advantage of using a **Factory Method pattern** is that we could cache objects in the factory for added efficiency:

Here is the factory used without a cache:



And here it is with a cache that is hidden behind the factory call so the caller does not know about this implementation:



Design Patterns - Factory Method Pattern

This is another famous GoF pattern and its motivation is stated as follows:

> *Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.*

This pattern is often used with the following GoF patterns:

1. **Strategy** and **Iterator** design patterns are often used with **Factory Method** pattern to let collection subclasses return different types of iterators that are compatible with the collections.
2. **Factory Method** pattern can also be used in the **Object Pool** pattern when creating cached objects of different subtypes.

Design Patterns - Factory Method Pattern

This is another famous GoF pattern and its motivation is stated as follows:

> *Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.*

This pattern is often used with the following GoF patterns:

1. **Strategy** and **Iterator** design patterns are often used with **Factory Method** pattern to let collection subclasses return different types of iterators that are compatible with the collections.
2. **Factory Method** pattern can also be used in the **Object Pool** pattern when creating cached objects of different subtypes.

Design Patterns - Factory Method Pattern

When to use:

1. When a caller can't anticipate the types of objects it must create.
   a. Let's say you are coding a mobile game. The user can choose any weapon or a spaceship but you do not know which one they will use.
   b. New types of ships or weapons can be added to the game in the future, without any changes to the client code.
2. When you have many objects of a common type (like for example a *Shape* with subclasses such as *Rectangle*, *Circle*, *Triangle*, etc…)

When not to use:

1. There are no specific restrictions as to when NOT to use it.

Design Patterns - Factory Method Pattern

Pros:

1. It allows sub-classes to choose the type of objects to create.
2. Simple to implement (simpler than **Builder** pattern)
3. Promotes loose-coupling by eliminating the need to bind application-specific classes into the code. That means the code interacts only with the resultant interface or abstract class.
4. **Single Responsibility Principle.** You can move the creation code into one place in the program, making the code easier to support.
5. **Open/Closed Principle.** You can introduce new subtypes into the program without breaking existing client code. This results in Clean code.

Design Patterns - Factory Method Pattern

Cons:

1. One disadvantage of the Factory Method pattern is that it can expand the total number of classes in a system. Every concrete class also requires a concrete Creator class. **Note**: the **Parameterized Factory Method** avoids this downside.

Design Patterns - Factory Method Pattern

Design Considerations:

1. When you have many objects of a common type (like for example a **Shape**)
   a. Make all such object types (like for example **IShape**) follow the same interface or extend the same abstract class. This interface should declare methods that make sense in every Shape but in a generalized way.
2. Create a Factory with a static creation method which always returns the same common interface reference (for example **IShape**)
3. Let the Factory know which instance you want, through some sort of a constant id for the specific object type.
   a. This can be accomplished through an **enum** or a **list of constants** which could be integers or strings.

Design Patterns - Factory Method Pattern

There two main variants of the pattern:

1. The very popular **Simple Factory Method** variant (also called Parameterized Factory Method pattern)
   a. We are going to mostly cover the simplified variant here as it is more useful for the majority of projects, but we will briefly mention the classic variant as well.
2. The classic (or original) **GoF Factory Method** variant.