**The Factory method**



The factory method is a creational pattern that delegates the responsibility of object instantiation to its concrete subclasses.

**Motivation**

Imagine a drive-thru where customers order meals by number, such as "Combo 1". Customers don't specify ingredients or preparation details; they trust the kitchen to handle that.

But how do we design this from a software perspective, such that we create a flexible and loosely coupled design?

Let's take a look at two candidate solutions:

**1. Direct Instantiation by the client**: At first glance, one way we can handle this is direct instantiation. Imagine creating burger objects based on multiple conditionals:

Burger burger;

if (TYPES.CHEESE) {

burger = new CheeseBurger();

} else if (TYPES.DELUXE) {

burger = new DeluxeCheeseBurger();

} else if (TYPES.VEGAN) {

burger = new VeganBurger();

}

*// ...*

This *does* work, but what happens when we introduce a new burger type or a limited-time special?

Each addition requires us to modify this logic. This isn't just tedious but also risks introducing errors, particularly if this decision-making code sprawls across the application.

We have discussed "program to an interface and not to an implementation" before, meaning that clients should remain unaware of the specific types of objects they use, as long as the objects adhere to the interface that they expect.

While the code above adheres to this principle, we are still limiting ourselves to the concrete implementation when we perform instantiation. This direct instantiation within the client code is brittle. It makes the system less adaptable to change and increases the chances of mistakes.

We also violate the ***single-responsibility principle*** because we are handling the logic of cooking, preparing the drink and the side, all together in one class. If we make a mistake updating any of the conditional code, we might end up taking down the entire burger class. In a real kitchen, the cashier does not decide how to cook the burger or what ingredients to use. The same should be the case with our code.

Taking into consideration the pitfalls of this approach, let's address them in the next candidate solution.

**2. Encapsulate what varies using a simple factory**: We know that the burger menu may evolve over time, but we would prefer not to directly modify all of the parts of code that instantiate a burger.  
Taking the design principle, "encapsulate what varies" into consideration, we can create a class called SimpleBurgerFactory and isolate the burger creation logic to a method called createBurger.

Enum BurgerType {

CHEESEBURGER,

DELUXE\_CHEESEBURGER,

VEGANBURGER,

}

class SimpleBurgerFactory {

public Burger createBurger(BurgerType type) {

Burger burger = null;

switch (type) {

case CHEESEBURGER:

burger = new CheeseBurger();

break;

case DELUXE\_CHEESEBURGER:

burger = new DeluxeCheeseBurger();

break;

case VEGANBURGER:

burger = new VeganBurger();

break;

}

return burger;

}

}

This is the essence of the **simple** factory pattern - a class dedicated to producing objects based on conditions.

The above solution still violates the open/closed principle but it also violates the dependency inversion principle. The simple factory has direct dependencies on the concrete classes that it is creating. And while the simple factory is abstracting away the creation process, the product type still needs to be provided to the factory.

**The solution**

The Factory Method pattern improves upon the simple factory by promoting loose coupling and adhering to the open/closed principle.

**1. Building on the Simple Factory**

While the simple factory centralized the creation process in one class, the Factory Method pattern defers the instantiation process to subclasses.

If we introduce a new burger type, for instance, we just need to extend our base factory class and provide the creation logic specific to that new type — all without touching existing code. This will make more sense soon.

**2. Promoting the Open/Closed Principle**

One of the core strengths of the Factory Method is its adherence to the open/closed principle. It allows our system to be open for extension (by adding new burger types) but closed for modification (no need to change existing code).

**3. Decoupling and the Power of Abstraction**

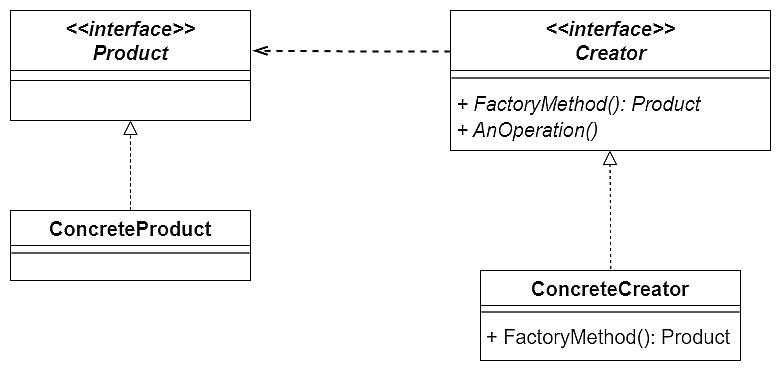
The Factory Method shines in promoting decoupling. Clients order a burger interacting solely with the BurgerStore abstract class. The specific details of which burger are abstracted away.

This means we can provide different implementations of the BurgerStore and the client code will not need to change. This is known as the ***dependency inversion principle***.

You might have come across another pattern named the abstract factory. The Factory Method is all about creating individual products, delegating the 'how' to subclasses. In contrast, the Abstract Factory is concerned with creating families of related products. Think of the Factory Method as a singular method, while the Abstract Factory is more of an object comprising multiple factory methods.

**Design and Implementation**

The factory method UML class diagram can be conceptualized like the following:



Let's expand on the components that are collaborating together in the above diagram.

**1. Creator Interface or Abstract Class**

This class/interface provides the declaration of the factory method. This method might be abstract or concrete, depending on the specific scenario.

If this method is declared as abstract, it just means that it is to be implemented by the concrete classes that extend it. This **does not** mean that the pattern becomes an abstract factory pattern.

Whether we choose to make use of an abstract class or an interface depends on the specific scenario. If there are shared behaviors or attributes among each concrete implementation of the product, an abstract class is a better choice. However, if it is just the methods that need to be implemented, an interface would suffice.

The factory method might have default implementation in some cases. This factory method **must** return an instance of the Product type - we shall see why.

**2. Concrete Creators**

These are the subclasses that extend/implement the creator. They will be overriding the factory method to produce concrete products based on some input parameter.

Even though the factory method has a return type Product, we can still return an instance of a concrete product because it implements (or extends) the Product interface/abstract class.

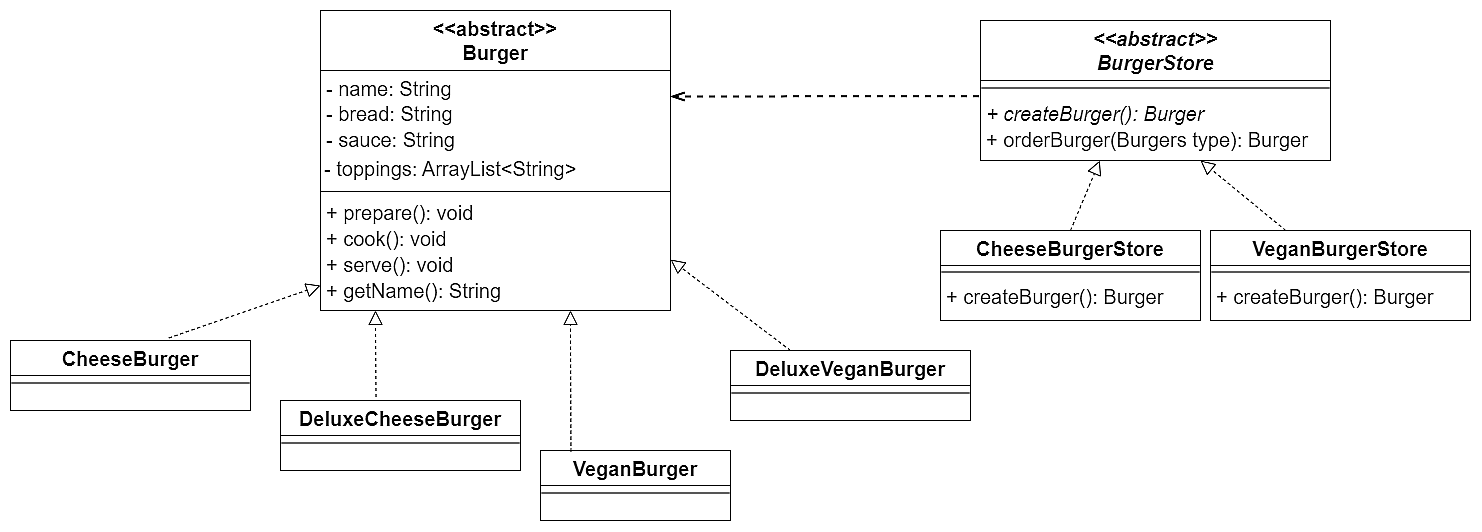
**3. Product Interface or Abstract Class**

This represents the contract for the types of products the factory will produce.

**4. Concrete Products**

These are concrete implementations of the product interface or the abstract class. They define the specifics of each product variant. Adding functionality each time is done through extension, rather than modification. This follows the Open/Closed principle closely.

If we apply all of the above components to our scenario, we end up with the following:



The above class diagram demonstrates the class diagram applied to our scenario.

The complete implementation of the code is available in the `View Code`` section above.

Enum Burgers {

CHEESE,

DELUXECHEESE,

VEGAN,

DELUXEVEGAN,

}

abstract class BurgerStore {

abstract Burger createBurger(Burgers item);

public Burger orderBurger(Burgers type) {

Burger burger = createBurger(type);

System.out.println("--- Making a " + burger.getName() + " ---");

burger.prepare();

burger.cook();

burger.serve();

return burger;

}

}

class CheeseBurgerStore extends BurgerStore {

Burger createBurger(Burgers item) {

if (item.equals(Burgers.CHEESE)) {

return new CheeseBurger();

} else if (item.equals(Burgers.DELUXECHEESE)) {

return new DeluxeCheeseBurger();

} else return null;

}

}

class VeganBurgerStore extends BurgerStore {

*// ...*

}

abstract class Burger {

String name;

String bread;

String sauce;

ArrayList<String> toppings = new ArrayList<String>();

void prepare() {

*// ...*

}

void cook() {

*//...*

}

void serve() {

*// ...*

}

public String getName() {

return name;

}

}

class CheeseBurger extends Burger {

*// ...*

}

class DeluxeCheeseBurger extends Burger {

*// ...*

}

class VeganBurger extends Burger {

*// ...*

}

class DeluxeVeganBurger extends Burger {

*// ...*

}

cheese\_store = CheeseBurgerStore()

vegan\_store = VeganBurgerStore()

burger = cheese\_store.order\_burger(Burgers.CHEESE)

print(f"Ethan ordered a {burger.get\_name()}")

*# ...*

**Key points to note:**

1. We used ABC and abstractmethod for abstract classes and methods respectively.
2. Python doesn't have explicit private variables, but convention is to prefix with an underscore to indicate that they're intended as private.
3. Python supports type hints, so we can specify types for parameters and return values.

**Limitations and Pitfalls**

Just like any other pattern, there are always pitfalls:

**1. Increased Complexity**

Adding the factory method requires the creation of additional classes and interfaces, which increases overhead as well. The code might become very complicated since we need to introduce a lot of new subclasses to implement the pattern.

In the example above, we only had two different kind of burgers per factory, but let's say we were designing a factory that produces jets with different models. This could lead to an explosion of classes.

**2. Might be hard to refactor**

We saw how meticulous we had to be with the set up of the factory method. The factory pattern is deeply embedded into our system and it will require a lot of effort to replace it if in the future we decide to use some other implementation.

*Click here for some additional remarks on the downsides*

**Use Cases**

**1. User Interface (UI) Libraries**

Instead of having the client code littered with conditionals to create OS-specific components, the library provides a generic interface for creating these components.

Each OS-specific library then implements a factory that produces components that conform to that OS's look and feel.

Clients just use the factory methods without needing to be aware of the specifics of each OS's components.

This way, the creation process is abstracted, and adding support for a new OS means adding a new factory without changing the client code.

**2. Database Connection Pooling**

The connection pooling library might use a Factory Method to abstract the creation of database-specific connection objects.

When the pool needs a new connection, it uses the appropriate factory to get a connection for the target database. By leveraging the Factory Method, the connection pooling logic can remain database-agnostic, and support for a new database type simply requires adding a new connection factory.

**Closing Notes**

Given the factory method, we achieve the following design principles:

**1. Encapsulate what varies**

We encapsulated the creation of the concrete objects within a single method. This way, the creator classes can evolve without affecting the rest of the application. The object creation process can also change without affecting the rest of the client code.

**2. Dependency Inversion Principle**

Dependency Inversion Principle promotes loose coupling and says that clients (high-level modules) should interact with abstractions (interfaces), rather than concrete implementations (low-level modules).

Traditionally, the client would be interacting directly with the concrete implementation of the Burger and BurgerStore. But now a client can be provided a variety of BurgerStores at runtime. If we introduce a new kind of BurgerStore we won't need to update the client to accommodate it.

**3. Open/Closed Principle**

The factory method pattern allows for new types of products to be added without modifying any existing code or the factory itself. If a new product needs to be added, we can simply extend the base factory and implement the factory method for the new product type. The existing client code does not need to change.