Factory Method and Prototype

The main goals of creational design patterns are making more reusable code and defining separate object creation methods

What is a factory method?

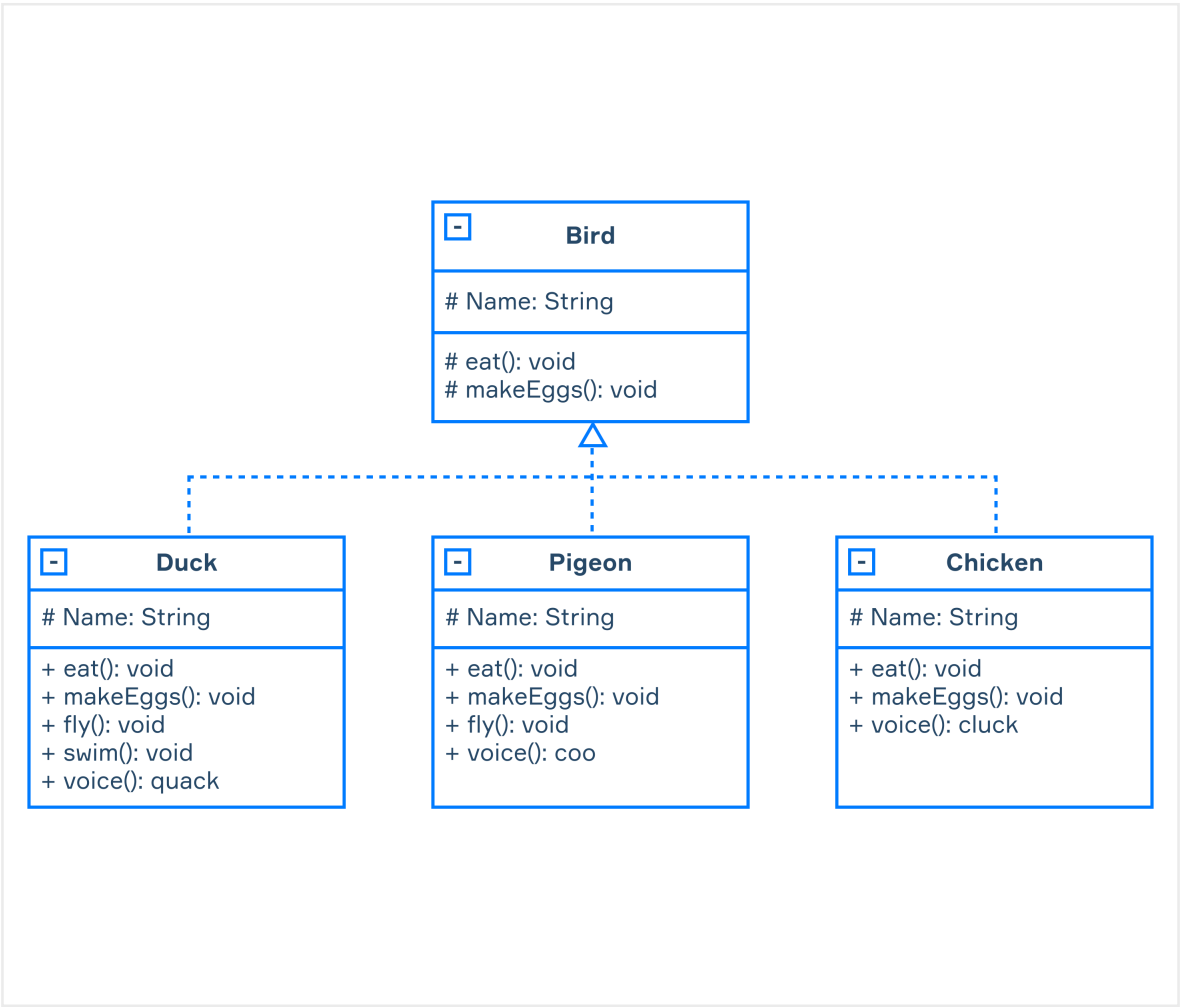
The factory method is a creational design pattern that describes how you can, through interface or abstract class, make a method to create objects which could be overridden in subclasses.

It's one of the basic patterns that describe how to solve recurring design problems to write more flexible and reusable code.

It's relatively easy to describe what the factory method is supposed to do. It solves the following two problems:

- Defining a separate object creation method in the form of a factory
- Allowing subclasses to call factory for object creation

Let's create a factory method that generates various types of birds from a shared template with unique functions.



We have an interface called *bird* which can be used to describe the common functions of all birds.

Through this interface, you can 'make' different types of birds. This picture describes duck, pigeon, and chicken. They all share the same functions they inherited from a *bird* interface, but they also added new methods.

When can we apply the factory method?

There are a few situations where you can use this pattern:

1. It's useful when you don't know the exact types of objects your code should work with. If your application has to expand the types of objects it works with, it will be easier with the factory method. You will only need to create a new factory subclass that will override the factory method in it.
2. A lot of frameworks and libraries are made with the factory method in mind, which allows you to modify them in your code. So if you are writing a framework that in your plans will be used by other people, you could apply the factory method making the extensions of your code possible.

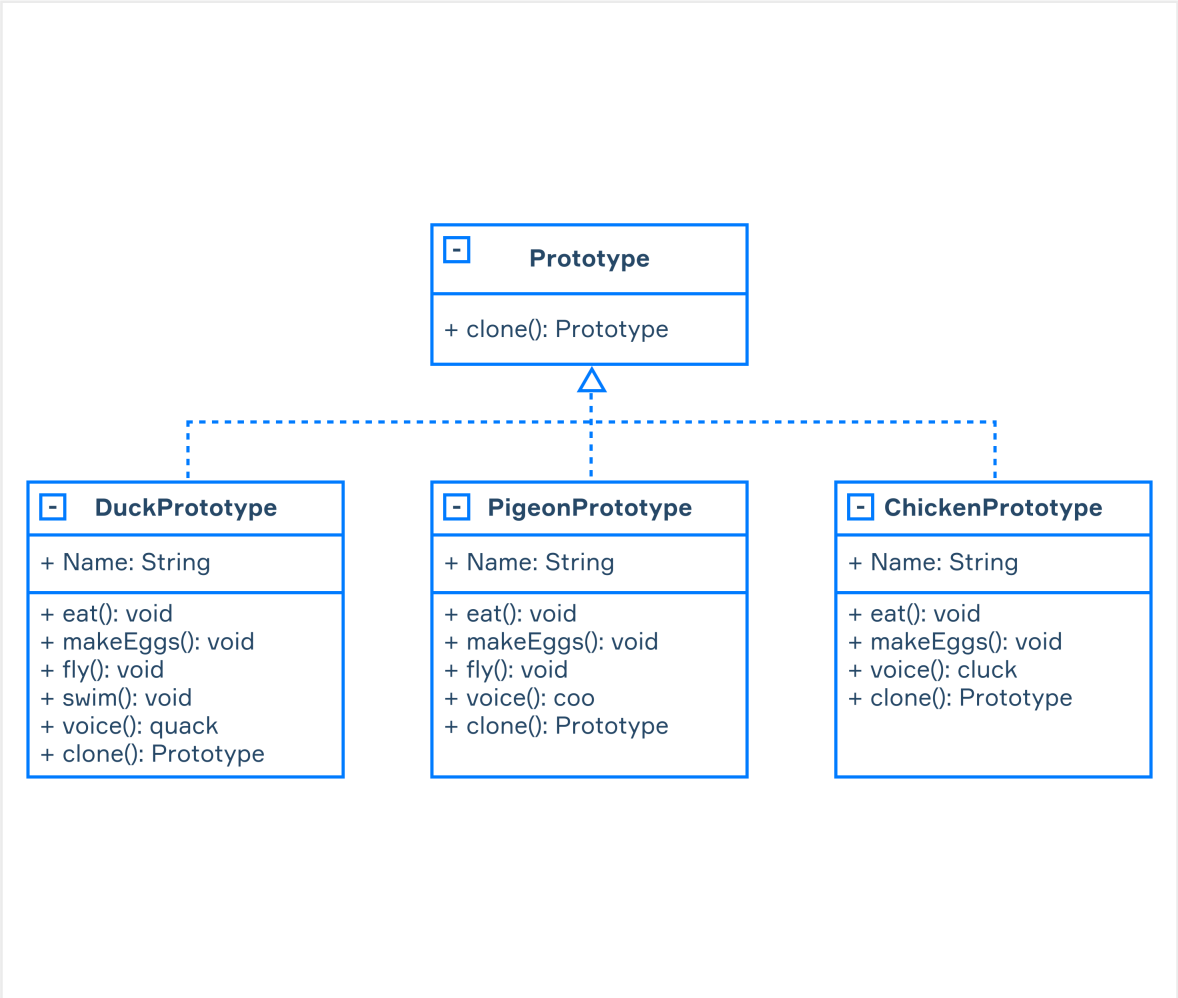
3. This pattern could also be used when you want to save system resources by reusing existing objects instead of rebuilding new ones each time.

Its main disadvantage is more complicated code that consists of a lot of subclasses needed to implement it.

Prototype pattern

A prototype is a creational design pattern that is based on the concept of copying an existing object to create a new one.

This pattern delegates the copying process to the object that is being copied. It commonly uses an interface that contains the clone() method.



Here you have a prototype interface that contains the clone() method. This method is implemented in our prototype classes. They create objects and then copy them through our clone() method.

To explain the difference between the factory method and prototype, let's use pseudocode:

```
// factory method
pigeon = Pigeon()
bird = pigeon as Bird
bird.makeEggs()

// prototype when we have an existing pigeon instance
prototype = pigeon as Prototype
newPigeon = prototype.clone() as Pigeon
newPigeon.fly()
```

As you can see, in the factory method, you call our Bird interface using different classes to create objects. Then you call object methods. In the prototype, you create an object by copying one that already exists. You can use both of these patterns together.

When can you apply the prototype?

You can use this pattern in two situations:

1. When your code shouldn't depend on a class that creates prototype objects. This can occur when you are working with objects which were passed to you from a 3rd-party code via some interface. In this case, you simply can't rely on them, because these classes are unknown. Prototype pattern allows working around these classes by providing an interface that works with any object that supports the copying process.
2. When you want to reduce the number of subclasses that only differ by how they initialize objects. The pattern lets you use a set of pre-built objects, configured in various ways, as prototypes.

It's a more flexible pattern than the factory method, but it can be more complicated to implement. Especially with more complex objects with circular references.