The Factory Method Design pattern will be implemented in two ways: Animal Factory and Color Factory. After that, we'll use an Abstract Factory to govern access to them:

We'll start by creating an Animal class family, which we'll subsequently employ in our Abstract Factory.

The Animal interface is as follows:

**public interface Animal {**

**String getAnimal();**

**String makeSound();**

**}**

and a concrete implementation Duck:

**public** **class** **Duck** **implements** **Animal** {

@Override

**public** String **getAnimal**() {

**return** "Duck";

}

@Override

**public** String **makeSound**() {

**return** "Squeks";

}

}

Furthermore, we may develop more concrete Animal interface implementations.

The Abstract Factory works with dependent object families. With that in mind, we'll introduce another interface family, Color, with a few implementations.

We can construct an AbstractFactory interface for them now that we have numerous families ready:

**public** **interface** **AbstractFactory**<T> {

T **create**(String animalType) ;

}

Then, following the Factory Method design pattern that we described in the last part, we'll create an AnimalFactory:

**public** **class** **AnimalFactory** **implements** **AbstractFactory**<Animal> {

@Override

**public** Animal **create**(String animalType) {

**if** ("Dog".equalsIgnoreCase(animalType)) {

**return** **new** **Dog**();

} **else** **if** ("Duck".equalsIgnoreCase(animalType)) {

**return** **new** **Duck**();

}

**return** null;

}

}

Using the same design approach, we can create a factory for the Color interface.

When all of this is in place, we'll construct a FactoryProvider class that will offer us an AnimalFactory or ColorFactory implementation, depending on the parameter we pass to the getFactory() method:

**public** **class** **FactoryProvider** {

**public** **static** AbstractFactory **getFactory**(String choice){

**if**("Animal".equalsIgnoreCase(choice)){

**return** **new** **AnimalFactory**();

}

**else** **if**("Color".equalsIgnoreCase(choice)){

**return** **new** **ColorFactory**();

}

**return** null;

}

}

To handle the new kind of object, the AbstractFactory class and all of its subclasses will need to be updated.