Abstract Factory Design Pattern in C#

Back to: Design Patterns in C# With Real-Time Examples

Abstract Factory Design Pattern in C# with Real-Time Example

In this article, I am going to discuss the **Abstract Factory Design Pattern in C#** with an example. Please read our previous article where we discussed the **Factory Method Design Pattern in C#** with an example. The Abstract Factory Design Pattern belongs to the creational design pattern category and is one of the most used design patterns in real-world applications. As part of this article, we are going to discuss the following things.

What is Abstract Factory Design Pattern?

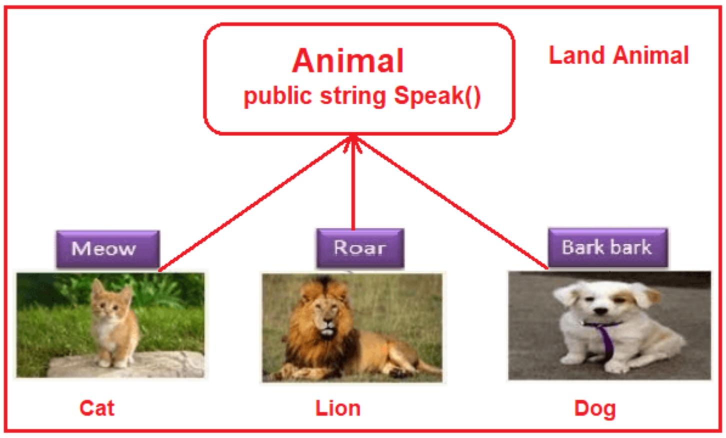
According to Gang of Four Definition: "**The Abstract Factory Design Pattern provides a way to encapsulate a group of individual factories that have a common theme without specifying their concrete classes**".

In simple words we can say, the Abstract Factory is a super factory that creates other factories. This Abstract Factory is also called the Factory of Factories.

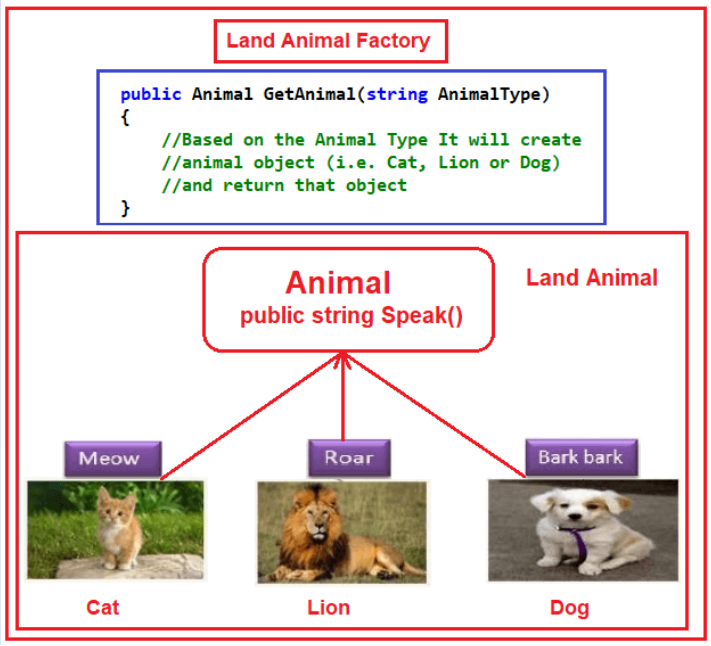
Understanding Abstract Factory Design Pattern:

Let us understand the abstract factory design pattern with one simple example. Suppose we want to create the objects of a group of land animals such as Cat, Lion, and Dog.

Please have a look at the following diagram. Here, as you can see we have three classes i.e. Cat, Lion, and Dog. And these three classes are the subclasses of Animal superclass or super interface. The Animal superclass or super interface has one method i.e. Speak() method. The Cat class will implement that Speak method and return Meow. Similarly, the Lion class will implement the Speak() method and will return Roar and in the say the Dog class will implement the Speak() method and return Bark bark. The Cat, Lion, and Dog are living in the Land, so they belong to the Land Animal group.

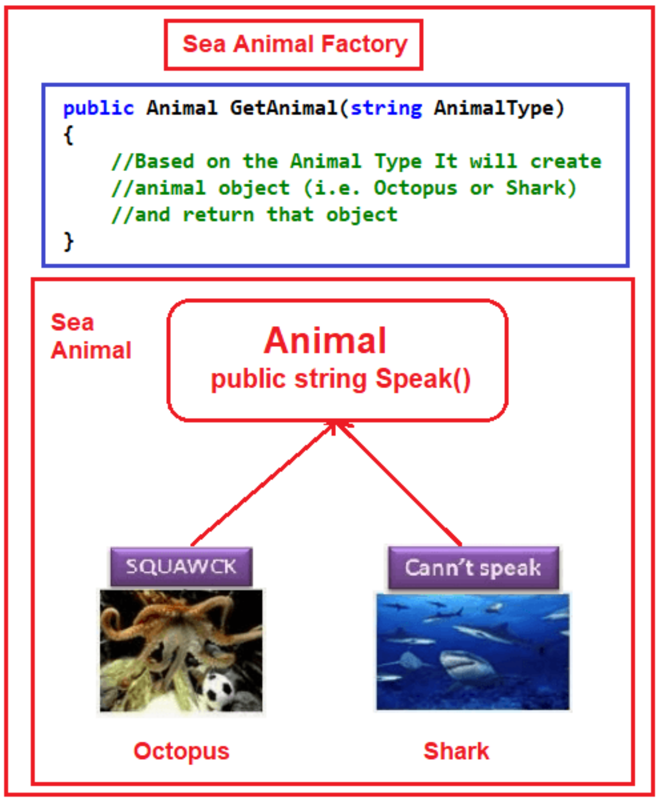


Using Factory Design Patter we can implement the above very easily. Please have a look at the following diagram. As per the factory design pattern, LandAnimalFactory is the factory class and that class has one method i.e. GetAnimal. This method takes one parameter i.e. the animal type and then it will create and return the appropriate object. In this case, the animal object can be a dog, lion, or cat. This method will return the Superclass or super interface i.e. Animal. For example, if you pass the Animal Type as a cat, then it will create the Cat class object and assign that object to the Superclass reference variable i.e. Animal and return that Superclass reference variable to the caller.



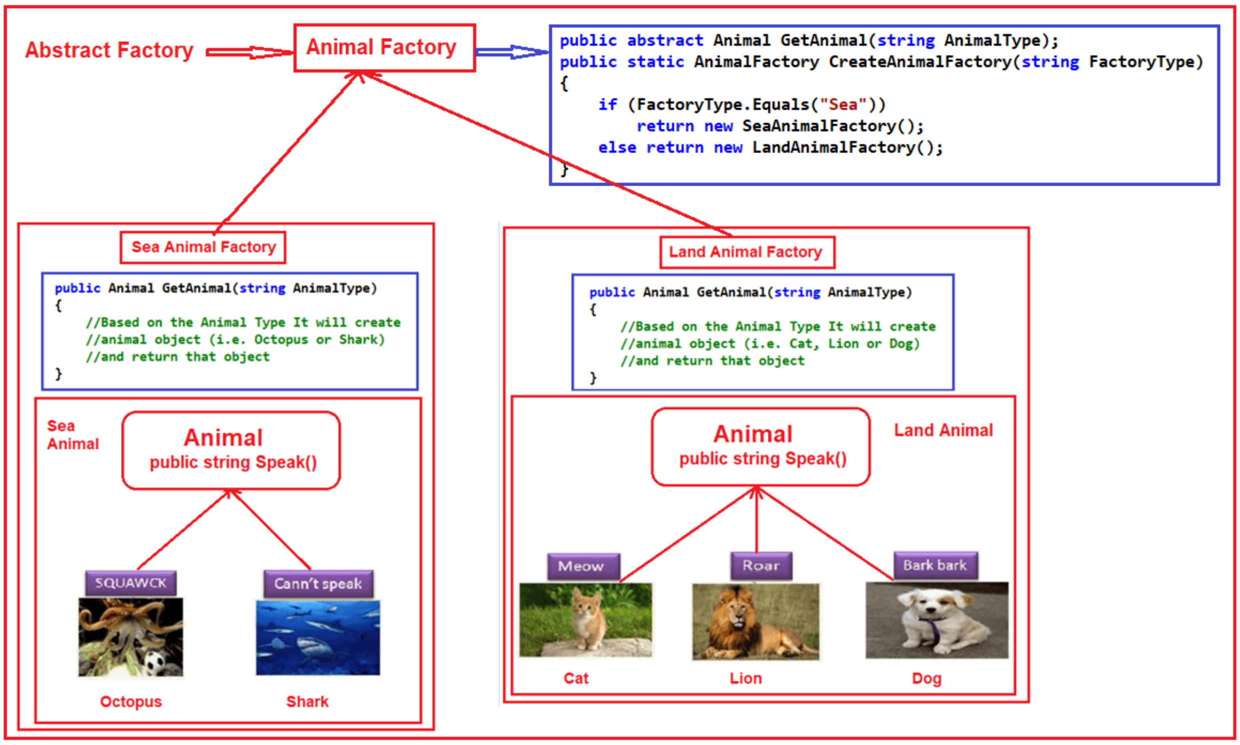
```
public Animal GetAnimal(string AnimalType)
{
    //Based on the Animal Type It will create
    //animal object (i.e. Cat, Lion or Dog)
    //and return that object
}
```

Let say, we have another group of sea animals such as Octopus and Shark. The way we implement the Land animals, in the same way, we need to implement the Sea animals. Please have a look at the following diagram for a better understanding.



```
public Animal GetAnimal(string AnimalType)
{
    //Based on the Animal Type It will create
    //animal object (i.e. Octopus or Shark)
    //and return that object
}
```

So, here we have two factories i.e. Land Animal Factory and Sea Animal Factory. Please have a look at the following diagram. The Land Animal Factory and Sea Animal Factory are the subclasses of Animal Factory. Here, the Animal Factory is nothing but your Abstract Factory.


Abstract Factory
Animal Factory
public abstract Animal GetAnimal(string AnimalType);
public static AnimalFactory CreateAnimalFactory(string FactoryType)
{
if (FactoryType.Equals("Sea"))
return new SeaAnimalFactory();
else return new LandAnimalFactory();
}
Sea Animal Factory
public Animal GetAnimal(string AnimalType)
{
//Based on the Animal Type It will create
//animal object (i.e. Octopus or Shark)
//and return that object
}
Land Animal Factory
public Animal GetAnimal(string AnimalType)
{
//Based on the Animal Type It will create
//animal object (i.e. Cat, Lion or Dog)
//and return that object
}
Sea Animal
Animal
public string Speak()
SQUAWCK
Cann't speak
Octopus
Shark
Land Animal
Animal
public string Speak()
Meow
Roar
Bark bark
Cat
Lion
Dog

The Animal Factory is responsible for creating the Sea Animal Factory and Land Animal Factory. Here, the animal factory has two methods. The abstract GetAnimal method is implemented by the Sea Animal Factory and Land Animal Factory. The second method is CreateAnimalFactory. What basically this method does is, takes the factory type as an input parameter, and based on the factory type it will create either the Sea Animal factory object or the Land Animal factory object and return that object to the caller. So, the Animal Factory acts as the Super Factory for Sea and Land animal factories.

Implementation of Abstract Factory Design Pattern in C#:

Let us implement the Abstract Factory Design Pattern in C# step by step.

Step1: Creating Abstract Product

Create an interface with the name Animal and then copy and paste the following code into it. This interface declares the method of the Product. In our example, it is the Speak method of the animal object.

```
namespace AbstractFactoryDesignPattern
{
    public interface Animal
    {
        string speak();
    }
}
```

Step2: Creating Concrete Product

This is a class that implements the Abstract Factory interface (in our case Animal) to create concrete products. In our example, the Product classes are Cat, Lion, Dog, Octopus, and Sharp.

Cat.cs: Create a class file with the name **Cats.cs** and then copy and paste the following in it. Here, it implements the Speak method of the Animal interface.

```
namespace AbstractFactoryDesignPattern
{
    public class Cat : Animal
    {
        public string speak()
        {
            return "Meow Meow Meow";
        }
    }
}
```

Note: We are going to do the same thing for all other product classes.

Lion.cs: Create a class file with the name **Lion.cs** and then copy and paste the following in it.

```
namespace AbstractFactoryDesignPattern
{
    public class Lion : Animal
    {
        public string speak()
        {
            return "Roar";
        }
    }
}
```

Dog.cs: Create a class file with the name **Dog.cs** and then copy and paste the following in it.

```
namespace AbstractFactoryDesignPattern
{
    public class Dog : Animal
    {
        public string speak()
        {
            return "Bark bark";
        }
    }
}
```

Octopus.cs: Create a class file with the name **Octopus.cs** and then copy and paste the following in it.

```
namespace AbstractFactoryDesignPattern
{
    public class Octopus : Animal
    {
        public string speak()
        {
            return "SQUAWCK";
        }
    }
}
```

Shark.cs: Create a class file with the name **Shark.cs** and then copy and paste the following in it.

```
namespace AbstractFactoryDesignPattern
{
    public class Shark : Animal
    {
        public string speak()
        {
            return "Cannot Speak";
        }
    }
}
```

Step3: Creating the Abstract Factory

AbstractFactory declares an interface for operations that will create AbstractProduct objects. In our example, it is going to be AnimalFactory. So, create a class file with the name AnimalFactory.cs and then copy and paste the following in it. As you can see this class contains two methods. The GetAnimal method is an abstract method that is going to be implemented by the child factory classes. The CreateAnimalFactory method takes an input parameter i.e. factory type and then creates and returns the appropriate factory object to the caller.

```
namespace AbstractFactoryDesignPattern
{
    public abstract class AnimalFactory
    {
        public abstract Animal GetAnimal(string AnimalType);

        public static AnimalFactory CreateAnimalFactory(string FactoryType)
        {
            if (FactoryType.Equals("Sea"))
                return new SeaAnimalFactory();
            else
                return new LandAnimalFactory();
        }
    }
}
```

Step4: Creating Concrete Factory

This is a class that implements the Abstract Factory (Animal Factory) class. In our example, this class is going to implement the GetAnimal method of the Animal factory class. There are two types of concrete abstract factories in our example i.e. LandAnimalfactory and SeaAnimalfactory. So, create a class file with the name **LandAnimalFactory.cs** and then copy and paste the following code into it.

```
namespace AbstractFactoryDesignPattern
{
    public class LandAnimalFactory : AnimalFactory
    {
        public override Animal GetAnimal(string AnimalType)
        {
            if (AnimalType.Equals("Dog"))
            {
                return new Dog();
            }
            else if (AnimalType.Equals("Cat"))
            {
                return new Cat();
            }
            else if (AnimalType.Equals("Lion"))
            {
                return new Lion();
            }
            else
                return null;
        }
    }
}
```

As you can see in the above code, the GetAnimal method creates and return the appropriate Land Animal object (i.e. Dog, cat, and Lion) based on the input parameter i.e. AnimalType it received. Similarly, create another class file with the name **SeaAnimalFactory.cs** and then copy and paste the following code into it.

```
namespace AbstractFactoryDesignPattern
{
    public class SeaAnimalFactory : AnimalFactory
    {
        public override Animal GetAnimal(string AnimalType)
        {
            if (AnimalType.Equals("Shark"))
            {
                return new Shark();
            }
            else if (AnimalType.Equals("Octopus"))
            {
                return new Octopus();
            }
            else
                return null;
        }
    }
}
```

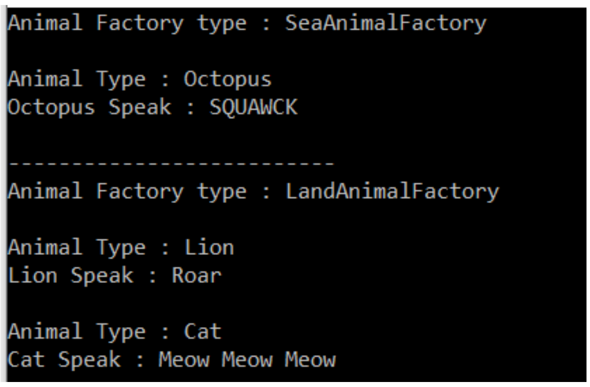
As you can see in the above code, based on the AnimalType parameter value the GetAnimal method creates and returns the object of either Shark or Octopus class.

Step5: Client

The Client uses the Abstract Factory and Abstract Product interfaces to create a family of related objects. So, modify the Main method as shown below.

```
using System;
namespace AbstractFactoryDesignPattern
{
    class Program
    {
        static void Main(string[] args)
        {
            Animal animal = null;
            AnimalFactory animalFactory = null;
            string speakSound = null;

            // Create the Sea Factory object by passing the factory type as Sea
            animalFactory = AnimalFactory.CreateAnimalFactory("Sea");
            Console.WriteLine("Animal Factory type : " + animalFactory.GetType().Name);
            Console.WriteLine();

            // Get Octopus Animal object by passing the animal type as Octopus
            animal = animalFactory.GetAnimal("Octopus");
            Console.WriteLine("Animal Type : " + animal.GetType().Name);
            speakSound = animal.speak();
            Console.WriteLine(animal.GetType().Name + " Speak : " + speakSound);
            Console.WriteLine();

            Console.WriteLine("--------------------------");
            // Create Land Factory object by passing the factory type as Land
            animalFactory = AnimalFactory.CreateAnimalFactory("Land");
            Console.WriteLine("Animal Factory type : " + animalFactory.GetType().Name);
            Console.WriteLine();

            // Get Lion Animal object by passing the animal type as Lion
            animal = animalFactory.GetAnimal("Lion");
            Console.WriteLine("Animal Type : " + animal.GetType().Name);
            speakSound = animal.speak();
            Console.WriteLine(animal.GetType().Name + " Speak : " + speakSound);
            Console.WriteLine();

            // Get Cat Animal object by passing the animal type as Cat
            animal = animalFactory.GetAnimal("Cat");
            Console.WriteLine("Animal Type : " + animal.GetType().Name);
            speakSound = animal.speak();
            Console.WriteLine(animal.GetType().Name + " Speak : " + speakSound);

            Console.Read();
        }
    }
}
```

Output:



```
Animal Factory type : SeaAnimalFactory

Animal Type : Octopus
Octopus Speak : SQUAWCK

-------------------------
Animal Factory type : LandAnimalFactory

Animal Type : Lion
Lion Speak : Roar

Animal Type : Cat
Cat Speak : Meow Meow Meow
```

Pointe to Remember:

1. Abstract Factory Pattern provides an interface for creating families of related dependent objects without specifying their concrete classes.
2. The Abstract Factory Pattern provides a way to encapsulate a group of individual factories that have a common theme without specifying their concrete classes.
3. The abstract factory design pattern is merely an extension to the factory method pattern or factory pattern, which allows you to create objects without being concerned about the actual class of the object being created.
4. Abstract means hiding some information and factory means which produces the products and pattern means a design. So, the Abstract Factory Pattern is a software design pattern that provides a way to encapsulate a group of individual factories that have a common theme.

When to use it Abstract Factory Design Pattern?

When we need to use the Abstract Factory Design Pattern in the following cases

1. When you want to create a set of related objects or dependent objects which must be used together.
2. When the system should configure to work with multiple families of products.
3. When the Concrete classes should be decoupled from the clients.

Differences between Abstract Factory and Factory Method Design Pattern:

1. Abstract Factory Design Pattern adds a layer of abstraction to the Factory Method Design Pattern
2. The Abstract Factory design pattern implementation can have multiple factory methods
3. Similar products of a factory implementation are grouped in the Abstract factory
4. The Abstract Factory Pattern uses object composition to decouple applications from specific implementations
5. The Factory Method Pattern uses inheritance to decouple applications from specific implementations

In the next article, I am going to discuss the **Builder Design Pattern in C#** with examples. Here, in this article, I try to explain the **Abstract Factory Design Pattern** in C# step by step with an example. I hope this article will help you with your needs. I would like to have your feedback. Please post your feedback, question, or comments about this article.