عنوان: **آشنایی با الگوی طراحی Fly Weight**
نویسنده: وحید فرهمندیان
تاریخ: ۱۳۹۳/۱۲/۰۵ ۱۴:۲۰
آدرس: www.dotnettips.info
گروه‌ها: C#, OOP, DesignPatterns

سناریوی زیر را در نظر بگیرید:
فرض کنید از شما خواسته شده است تا یک پردازشگر متن را بنویسید. خوب در این پردازشگر با یک‌سری کاراکتر روبرو هستید که هر کاراکتر احتمالا آبجکتی از نوع کلاس خود می‌باشد؛ برای مثال آبجکت XYZ که آبجکتی از نوع کلاس A هست و برای نمایش کاراکتر A استفاده می‌شود. این آبجکت‌ها دارای دو دسته خصیصه هستند: (مطالعه بیشتر)
خصیصه‌های ثابت: یعنی همه کاراکترهای A دارای یک شکل مشخص هستند. در واقع مشخصات ذاتی آبجکت می‌باشند.

خصیصه‌های پویا: یعنی هر کاراکتر دارای فونت، سایز و رنگ خاص خود است. در واقع خصیصه‌هایی که از یک آبجکت به آبجکت دیگر متفاوت هستند .

خوب احتمالا در ساده‌ترین راه حل، به ازای تک تک کاراکترهایی که کاربر وارد می‌کند، یک آبجکت از نوع کلاس متناسب با آن ساخته می‌شود. ولی بحث مهم این است که با این همه آبجکت که هر یک مصرف خود را از حافظه دارند، می‌خواهید چکار کنید؟ احتمالا به مشکل حافظه برخورد خواهید کرد! پس باید یک سناریوی بهتر ایجاد کرد.
سناریوی پیشنهادی این است که برای هر نوع کاراکتر، یک کلاس داشته باشیم، همانند قبل(یک کلاس برای A یک کلاس برای B و غیره) و یک استخر پر از آبجکت داشته باشیم که آبجکت‌های ایجاد شده در آن ذخیره شوند.
سپس کاربر، کاراکتر A را درخواست می‌کند. ابتدا به این استخر نگاه می‌کنیم. اگر کاراکتر A موجود بود، آن را برمی‌گردانیم و اگر موجود نبود، یک آبجکت از نوع A می‌سازیم، سپس این آبجکت را در استخر ذخیره می‌کنیم و آبجکت را بر می‌گردانیم. در این صورت اگر کاربر دوباره درخواست A را کرد، دیگر نیازی به ساخت آبجکت جدید نیست و از آبجکت قبلی می‌توانیم استفاده نماییم. با این شرایط تکلیف خصایص ایستا مشخص است. ولی مشکل مهم با خصایص پویا این است که می‌توانند بین آبجکت‌ها متفاوت باشند که برای این هم یک متد در کلاس‌ها قرار می‌دهیم تا این خصایص را تنظیم نماید.
به کد زیر دقت نمایید:

```
public interface IAlphabet
    {
        void Render(string font);//Define Extrinsic and non-static states for each object
    }

    public class A : IAlphabet
    {
        public void Render(string font) { Console.WriteLine(GetType().Name + " has font of type " +
font); }
    }
    public class B : IAlphabet
    {
        public void Render(string font) { Console.WriteLine(GetType().Name + " has font of type " +
font); }
    }
```

از متد Render برای تنظیم نمودن خصایص پویا استفاده خواهد شد.
سپس در ادامه به یک موتور نیاز داریم که قبل از ساخت آبجکت، استخر را بررسی نماید:

```
public class FlyWeightFactory
    {
        private readonly Dictionary<string, IAlphabet> _dictionary = new Dictionary<string,
IAlphabet>();
        public int Count { get { return _dictionary.Count; } }
        public IAlphabet GetObject(string name)
        {
            if (!_dictionary.ContainsKey(name))
                switch (name)
                {
                    case "A":
                        _dictionary.Add(name, new A());
```

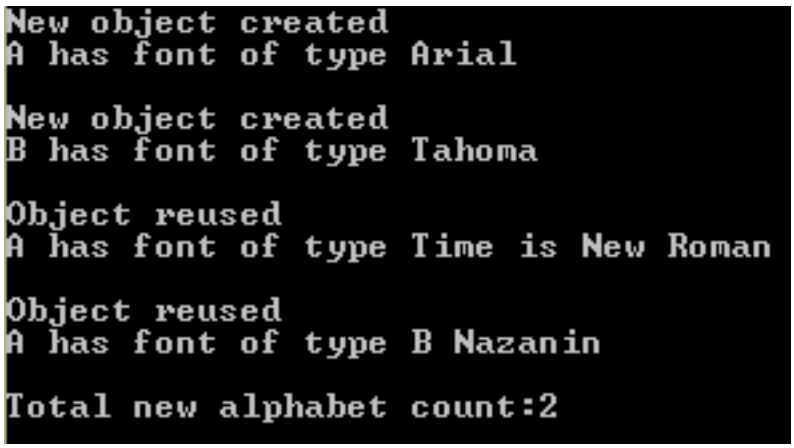
```
                    Console.WriteLine("New object created");
                    break;
                case "B":
                    _dictionary.Add(name, new B());
                    Console.WriteLine("New object created");
                    break;
                default:
                    throw new Exception("Factory can not create given object");
            }
        else
            Console.WriteLine("Object reused");
        return _dictionary[name];
    }
}
```

در اینجا _dictionaries همان استخر ما می‌باشد که قرار است آبجکت‌ها در آن ذخیره شوند. Count برای نمایش تعداد آبجکت‌های موجود در استخر استفاده می‌شود (حداکثر مقدار آن چقدر خواهد بود؟). GetObject نیز همان موتور اصلی کار است که در آن ابتدای استخر بررسی می‌شود. اگر آبجکت در استخر نبود، یک نمونه‌ی جدید از آن ساخته شده، به استخر اضافه گردیده و برگردانده می‌شود.

لذا برای استفاده‌ی از این کد داریم:

```
FlyWeightFactory flyWeightFactory = new FlyWeightFactory();
 IAlphabet alphabet = flyWeightFactory.GetObject(typeof(A).Name);
 alphabet.Render("Arial");
 Console.WriteLine();
 alphabet = flyWeightFactory.GetObject(typeof(B).Name);
 alphabet.Render("Tahoma");
 Console.WriteLine();
 alphabet = flyWeightFactory.GetObject(typeof(A).Name);
 alphabet.Render("Time is New Roman");
 Console.WriteLine();
 alphabet = flyWeightFactory.GetObject(typeof(A).Name);
 alphabet.Render("B Nazanin");
 Console.WriteLine();
 Console.WriteLine("Total new alphabet count:" + flyWeightFactory.Count);
```

با اجرای این کد خروجی زیر را مشاهده خواهید نمود:



نکته‌ی قابل توجه این است که این الگو بصورت داخلی از الگوی Factory Method استفاده می‌کند. با توجه بیشتر به پیاده سازی Flyweight Factory شباهت هایی بین آن و Singleton Pattern می‌بینیم. کلاس‌هایی از این دست را Multiton می نامند. در Multiton نمونه‌ها بصورت زوج کلیدهایی نگهداری می‌شوند و بر اساس Key دریافت شده نمونه‌ی متناظر بازگردانده می‌شود. همچنین در Singleton تضمین می‌شود که از کلاس مربوطه فقط یک نمونه در کل Application وجود دارد. در Multiton Pattern تضمین می‌شود که برای هر Key تنها یک Instance وجود دارد.