عنوان: **پیاده سازی پروژه نقاشی (Paint) به صورت شی گرا #1**
نویسنده: صابر فتح الهی
تاریخ: ۱۳۹۱/۱۱/۰۷ ۹:۴۰
آدرس: www.dotnettips.info
برچسب‌ها: OOP, C#.NET, Project

قصد داریم در طی چند پست متوالی، یک پروژه Paint را به صورت شی گرا پیاده سازی کنیم. خوب، پروژه ای که می‌خواهیم پیاده سازی کنیم باید دارای این امکانات باشه که مرحله به مرحله پیش میریم و پروژه کامل در نهایت در قسمت پروژه‌ها ی همین سایت قرار خواهد گرفت.

قابلیت ترسیم اشیا روی بوم گرافیکی دلخواه
قابلیت جابجایی اشیا
قابلیت تغییر رنگ اشیا
ترسیم اشیا توپر و تو خالی
تعیین پهنای خط شی ترسیم شده
تعیین رنگ پس زمینه در صورت تو پر بودن شی
قابلیت پیش نمایش رسم شکل در زمان ترسیم اشیا
توانایی انتخاب اشیا
تعیین عمق شی روی بوم گرافیکی مورد نظر
ترسیم اشیایی مانند خط، دایره، بیضی، مربع، مستطیل، لوزی، مثلث
قابلیت تغییر اندازه اشیا ترسیم شده

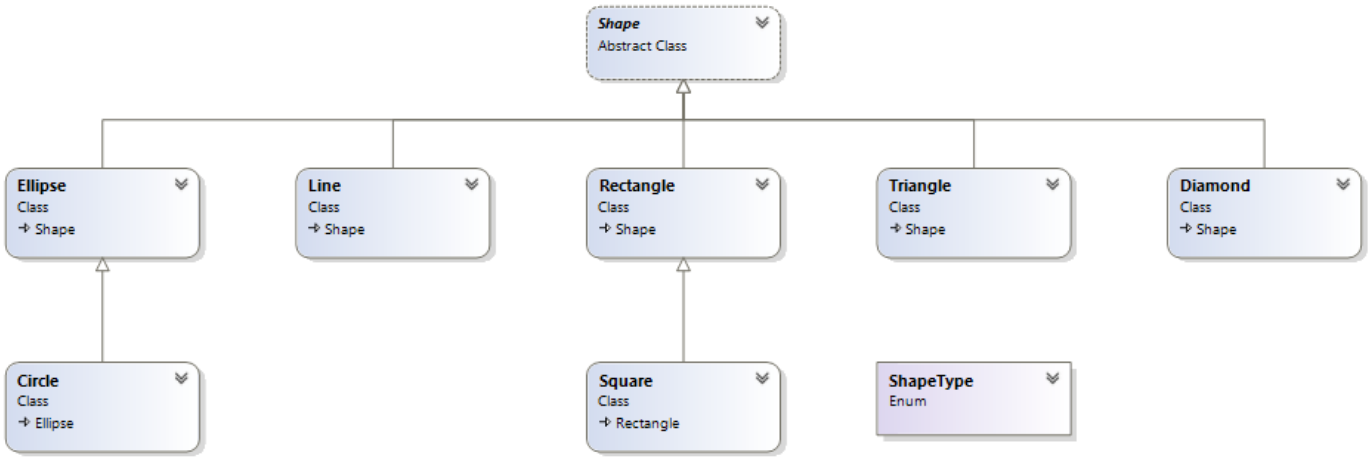
خوب برای شروع ابتدا یک پروژه از نوع Windows Application ایجاد می‌کنیم (البته برای این قسمت می‌توانیم یک پروژه Class Library ایجاد کنیم)

سپس یک پوشه به نام Models به پروژه اضافه نمایید.

خوب در این پروژه یک کلاس پایه به نام Shape در نظر می‌گیریم.

همه اشیا ما دارای نقطه شروع، نقطه پایان، رنگ قلم، حالت انتخاب، رنگ پس زمینه، نوع شی، .... می‌باشند که بعضی از خصوصیات را توسط خصوصیات دیگر محاسبه می‌کنیم. مثلا خاصیت Width و Height و X و Y توسط خصوصیات نقطه شروع و پایان می‌توانند محاسبه شوند.

ساختار کلاس‌های پروژه ما به صورت زیر است که مرحله به مرحله کلاس‌ها پیاده سازی خواهند شد.



با توجه به تصویر بالا (البته این تجزیه تحلیل شخصی من بوده و دوستان به سلیقه خود ممکن است این ساختار را تغییر دهند)

نوع شمارشی ShapeType: در این نوع شمارشی انواع شی‌های موجود در پروژه معرفی شده است

محتوای این نوع به صورت زیر تعریف شده است:

```
namespace PWS.ObjectOrientedPaint.Models
{
    /// <summary>
    /// Shape Type in Paint
    /// </summary>
    public enum ShapeType
    {
        /// <summary>
        /// هیچ
        /// </summary>
        None = 0,
        /// <summary>
        /// خط
        /// </summary>
        Line = 1,
        /// <summary>
        /// مربع
        /// </summary>
        Square = 2,
        /// <summary>
        /// مستطیل
        /// </summary>
        Rectangle = 3,
        /// <summary>
        /// بیضی
        /// </summary>
        Ellipse = 4,
        /// <summary>
        /// دایره
        /// </summary>
        Circle = 5,
        /// <summary>
        /// لوزی
        /// </summary>
        Diamond = 6,
        /// <summary>
        /// مثلث
        /// </summary>
        Triangle = 7,
    }
}
```

انشاالله در پست‌های بعدی ما بقی کلاس‌ها به مرور پیاده سازی خواهند شد.

نظرات خوانندگان

نویسنده: علی صداقت
تاریخ: ۱۳۹۱/۱۱/۰۷ ۱۷:۱۷

مبحث مفید و جالبیست. منتظر ادامه این مبحث هستیم. موفق باشید.

نویسنده: Parham
تاریخ: ۱۳۹۱/۱۱/۰۸ ۱۰:۳۷

نوع شمارشی ShapeType یک فایل کد ساده است، درسته؟

نویسنده: صابر فتح الهی
تاریخ: ۱۳۹۱/۱۱/۱۲ ۳:۲

بله درسته

نویسنده: masoud
تاریخ: ۱۳۹۱/۱۱/۱۴ ۱۴:۶

با تشکر اقای فتح الهی ممنون میشم این بحث رو تا آخر پیش ببرید استفاده میکنیم.

نویسنده: مسعود بهرامی
تاریخ: ۱۳۹۲/۰۸/۳۰ ۳:۲۰

قرار دادن کلاس Square به عنوان sub type زیر کلاس Rectangle اصل LSP رو نقض می‌کنه بهتره که هر دو از Shape به ارث برسن با بهتره بگم Implement کنن

نویسنده: صابر فتح الهی
تاریخ: ۱۳۹۲/۰۸/۳۰ ۱۰:۴۰

بله کاملا درسته از نظر شی گرای یک مربع نمیتونه مستطیل باشه، این پروژه هنوز تمام نشده و در خروجی نهایی اصلاح خواهد شد الان قابلیت ویرایش وجود نداره.
البته پست بعدی پیاده سازی پروژه نقاشی (Paint) به صورت شی گرا #2 نگاه کنین شاید نظراون عوض شد.
موفق و موید باشید

نویسنده: خوزستان
تاریخ: ۱۳۹۲/۰۸/۳۰ ۱۱:۳۰

اصل SPR , LSP ,.... از کجا و چه منابعی برای اینا موجود هست ؟
آیا کتابهای مهندسی نرم افزار باید خوند یا کلن مبحث جداگانه‌ای داره ؟

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۰۸/۳۰ ۱۲:۷

از اینجا شروع کنید. 5 قسمت هست:

اصول طراحی شی گرا SOLID - #بخش اول اصل SRP

نویسنده: خوزستان
تاریخ: ۱۳۹۲/۰۸/۳۰ ۱۳:۱

تشکر .
آیا کتاب فارسی در این باره وجود دارد ؟
اساتید کتابی هست که معرفی کنن ؟

نویسنده: صابر فتح الهی
تاریخ: ۱۳۹۲/۰۸/۳۰ ۱۳:۵۹

توی گوگل SOLID در شی گرا جستجو کنین

یا توی همین سایت این برچسب دنبال کنید

نویسنده: مسعود بهرامی
تاریخ: ۱۳۹۲/۰۹/۰۱ ۳:۵۷

سلام مهندس فتح الهی عزیز من خوندم کامل ولی به نظر من بهتره از هم جدا شدن تا اصل LSP رو نقض نکنن درسته رفتار ترسیمشون مثل همه و دیگه تو کلاس مربع شما تابع رسم رو ننوشتین ولی باعث نقض LSPشدین به نظر من بهتر هردوشون از یه Abstract Class دیگه یه ارث برسن و تابع Draw این دوشکل که مثل هم هستش رو بزارین اونجا با اینکار هم LSP رعایت شده و هم تکرا تابع Draw رو هم ندارین
در ضمن تابع رسم پیش نمایش اشکالات اساسی از نظر OOP داشت که من با اجازتون اونو زیر پست خودش میزارم Re factor که به نظرم بهتر میومد. دست گلتونم درد نکنه

نویسنده: صابر فتح الهی
تاریخ: ۱۳۹۲/۰۹/۰۱ ۱۲:۸

خوشحال میشم کد شمارو ببینم
منتظر روش شما هستم من که خودم چیزی به ذهنم نرسید

عنوان: **پیاده سازی پروژه نقاشی (Paint) به صورت شی گرا #2**
نویسنده: صابر فتح الهی
تاریخ: ۱۳۹۱/۱۱/۱۸ ۰:۳۵
آدرس: www.dotnettips.info
برچسب‌ها: OOP, C#, Project, Paint

در ادامه مطلب پیاده سازی پروژه نقاشی (Paint) به صورت شی گرا #1 به تشریح مابقی کلاس‌های برنامه می‌پردازیم.

با توجه به تجزیه و تحلیل انجام شده تمامی اشیا از کلاس پایه به نام Shape ارث بری دارند حال به توضیح کدهای این کلاس می‌پردازیم. (به دلیل اینکه توضیحات این کلاس در دو پست نوشته خواهد شد برای این کلاس‌ها از partial class استفاده شده است)

```
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Net;

namespace PWS.ObjectOrientedPaint.Models
{
    /// <summary>
    /// Shape (Base Class)
    /// </summary>
    public abstract partial class Shape
    {
        #region Fields (1)

        private Brush _backgroundBrush;

        #endregion Fields

        #region Properties (16)

        /// <summary>
        /// Gets or sets the brush.
        /// </summary>
        /// <value>
        /// The brush.
        /// </value>
        public Brush BackgroundBrush
        {
            get { return _backgroundBrush ?? (_backgroundBrush = new SolidBrush(BackgroundColor)); }
            private set
            {
                _backgroundBrush = value ?? new SolidBrush(BackgroundColor);
            }
        }

        /// <summary>
        /// Gets or sets the color of the background.
        /// </summary>
        /// <value>
        /// The color of the background.
        /// </value>
        public Color BackgroundColor { get; set; }

        /// <summary>
        /// Gets or sets the end point.
        /// </summary>
        /// <value>
        /// The end point.
        /// </value>
        public PointF EndPoint { get; set; }

        /// <summary>
        /// Gets or sets the color of the fore.
        /// </summary>
        /// <value>
        /// The color of the fore.
        /// </value>
        public Color ForeColor { get; set; }

        /// <summary>
        /// Gets or sets the height.
        /// </summary>
        /// <value>
```

```
        /// The height.
        /// </value>
        public float Height
        {
            get
            {
                return Math.Abs(StartPoint.Y - EndPoint.Y);
            }
            set
            {
                if (value > 0)
                    EndPoint = new PointF(EndPoint.X, StartPoint.Y + value);
            }
        }

        /// <summary>
        /// Gets or sets a value indicating whether this instance is fill.
        /// </summary>
        /// <value>
        ///   <c>true</c> if this instance is fill; otherwise, <c>false</c>.
        /// </value>
        public bool IsFill { get; set; }

        /// <summary>
        /// Gets or sets a value indicating whether this instance is selected.
        /// </summary>
        /// <value>
        /// <c>true</c> if this instance is selected; otherwise, <c>false</c>.
        /// </value>
        public bool IsSelected { get; set; }

        /// <summary>
        /// Gets or sets my pen.
        /// </summary>
        /// <value>
        /// My pen.
        /// </value>
        public Pen Pen
        {
            get
            {
                return new Pen(ForeColor, Thickness);
            }
        }

        /// <summary>
        /// Gets or sets the type of the shape.
        /// </summary>
        /// <value>
        /// The type of the shape.
        /// </value>
        public ShapeType ShapeType { get; protected set; }

        /// <summary>
        /// Gets the size.
        /// </summary>
        /// <value>
        /// The size.
        /// </value>
        public SizeF Size
        {
            get
            {
                return new SizeF(Width, Height);
            }
        }

        /// <summary>
        /// Gets or sets the start point.
        /// </summary>
        /// <value>
        /// The start point.
        /// </value>
        public PointF StartPoint { get; set; }

        /// <summary>
        /// Gets or sets the thickness.
        /// </summary>
        /// <value>
        /// The thickness.
        /// </value>
```

```
        public byte Thickness { get; set; }

        /// <summary>
        /// Gets or sets the width.
        /// </summary>
        /// <value>
        /// The width.
        /// </value>
        public float Width
        {
            get
            {
                return Math.Abs(StartPoint.X - EndPoint.X);
            }
            set
            {
                if (value > 0)
                    EndPoint = new PointF(StartPoint.X + value, EndPoint.Y);
            }
        }

        /// <summary>
        /// Gets or sets the X.
        /// </summary>
        /// <value>
        /// The X.
        /// </value>
        public float X
        {
            get
            {
                return StartPoint.X;
            }
            set
            {
                if (value > 0)
                    StartPoint = new PointF(value, StartPoint.Y);
            }
        }

        /// <summary>
        /// Gets or sets the Y.
        /// </summary>
        /// <value>
        /// The Y.
        /// </value>
        public float Y
        {
            get
            {
                return StartPoint.Y;
            }
            set
            {
                if (value > 0)
                    StartPoint = new PointF(StartPoint.X, value);
            }
        }

        /// <summary>
        /// Gets or sets the index of the Z.
        /// </summary>
        /// <value>
        /// The index of the Z.
        /// </value>
        public int Zindex { get; set; }

        #endregion Properties
        }
}
```

ابتدا به تشریح خصوصیات کلاس می‌پردازیم:

خصوصیات:

BackgroundColor : در صورتی که شی مورد نظر به صورت توپررسم شود، این خاصیت رنگ پس زمینه شی را مشخص می‌کند.
BackgroundBrush : خاصیتی است که با توجه به خاصیت BackgroundColor یک الگوی پر کردن زمینه شی می‌سازد.
StartPoint : نقطه شروع شی را در خود نگهداری می‌کند.
EndPoint : نقطه انتهای شی را در خود نگهداری می‌کند. (قبلا گفته شد که هر شی را در صورتی که در یک مستطیل فرض کنیم یک نقطه شروع و یک نقطه پایان دارد)
ForeColor : رنگ قلم ترسیم شی مورد نظر را تعیین می‌کند.
Height : ارتفاع شی مورد نظر را تعیین می‌کند (این خصوصیت اختلاف عمودی StartPoint.Y و EndPoint.Y را محاسبه می‌کند و در زمان مقدار دهی EndPoint جدیدی ایجاد می‌کند).
Width : عرض شی مورد نظر را تعیین می‌کند (این خصوصیت اختلاف افقیStartPoint.X و EndPoint.X را محاسبه می‌کند و در زمان مقدار دهی EndPoint جدیدی ایجاد می‌کند).
IsFill : این خصوصیت تعیین کننده توپر و یا توخالی بودن شی است.
IsSelected : این خاصیت تعیین می‌کند که آیا شی انتخاب شده است یا خیر (در زمان انتخاب شی چهار مربع کوچک روی شی رسم می‌شود).
Pen : قلم خط ترسیم شی را مشخص می‌کند. (قلم با ضخامت دلخواه)
ShapeType : این خصوصیت نوع شی را مشخص می‌کند (این خاصیت بیشتر برای زمان پیش نمایش ترسیم شی در زمان اجراست البته به نظر خودم اضافه هست اما راه بهتری به ذهنم نرسید)
Size : با استفاده از خصوصیات Height و Width ایجاد شده و تعیین کننده Size شی است.
Thickness : ضخامت خط ترسیمی شی را مشخص می‌کند، این خاصیت در خصوصیت Pen استفاده شده است.
X : مقدار افقی نقطه شروع شی را تعیین می‌کند در واقع StartPoint.X را برمی‌گرداند (این خاصیت اضافی بوده و جهت راحتی کار استفاده شده می‌توان آن را ننوشت).
Y : مقدار عمودی نقطه شروع شی را تعیین می‌کند در واقع StartPoint.Y را برمی‌گرداند (این خاصیت اضافی بوده و جهت راحتی کار استفاده شده می‌توان آن را ننوشت).
Zindex : در زمان ترسیم اشیا ممکن است اشیا روی هم ترسیم شوند، در واقع Zindex تعیین کننده عمق شی روی بوم گرافیکی است.

در پست بعدی به توضیح متدهای این کلاس می‌پردازیم.

نظرات خوانندگان

نویسنده: بتیسا
تاریخ: ۱۳۹۱/۱۱/۱۸ ۱۰:۲۶

با سلام

از مطلب مفیدی که تهیه کردید ممنون.
می‌شود از طریق خاصیت Brush که فعلا فقط خواندنی هست، طرح‌های مختلفی برای پس زمینه اشیاع ایجاد کرد. مانند Paint.net و یا MS Paint.

اگر به صورت زیر تعریف کنیم فکر می‌کنم کمی کامل‌تر باشه!

```
private Brush _backgroundBrush;

        /// <summary>
        /// Gets or sets the brush.
        /// </summary>
        /// <value>
        /// The brush.
        /// </value>
        public Brush BackgroundBrush
        {
            get
            {
                return _backgroundBrush;
            }
            private set
            {
                _backgroundBrush = (value != null) ? value : new SolidBrush(BackgroundColor);
            }
        }
        //-------------------------------------[Methode for set brush]---------------------------------
-----
        public virtual void SetBackgroundBrushAsHatch(HatchStyle hatchStyle)
        {
            HatchBrush brush = new HatchBrush(hatchStyle, BackgroundColor);
            BackgroundBrush = brush;
        }

        public virtual void SetBackgroundBrushAsSolid()
        {
            SolidBrush brush = new SolidBrush(BackgroundColor);
            BackgroundBrush = brush;
        }

        public virtual void SetBackgroundBrushAsLinearGradient()
        {
            LinearGradientBrush brush = new LinearGradientBrush(StartPoint, EndPoint, ForeColor,
BackgroundColor);
            BackgroundBrush = brush;
        }
```

که اگر بخواهیم میتونیم باز بیشتر Customize بکنیمشون.

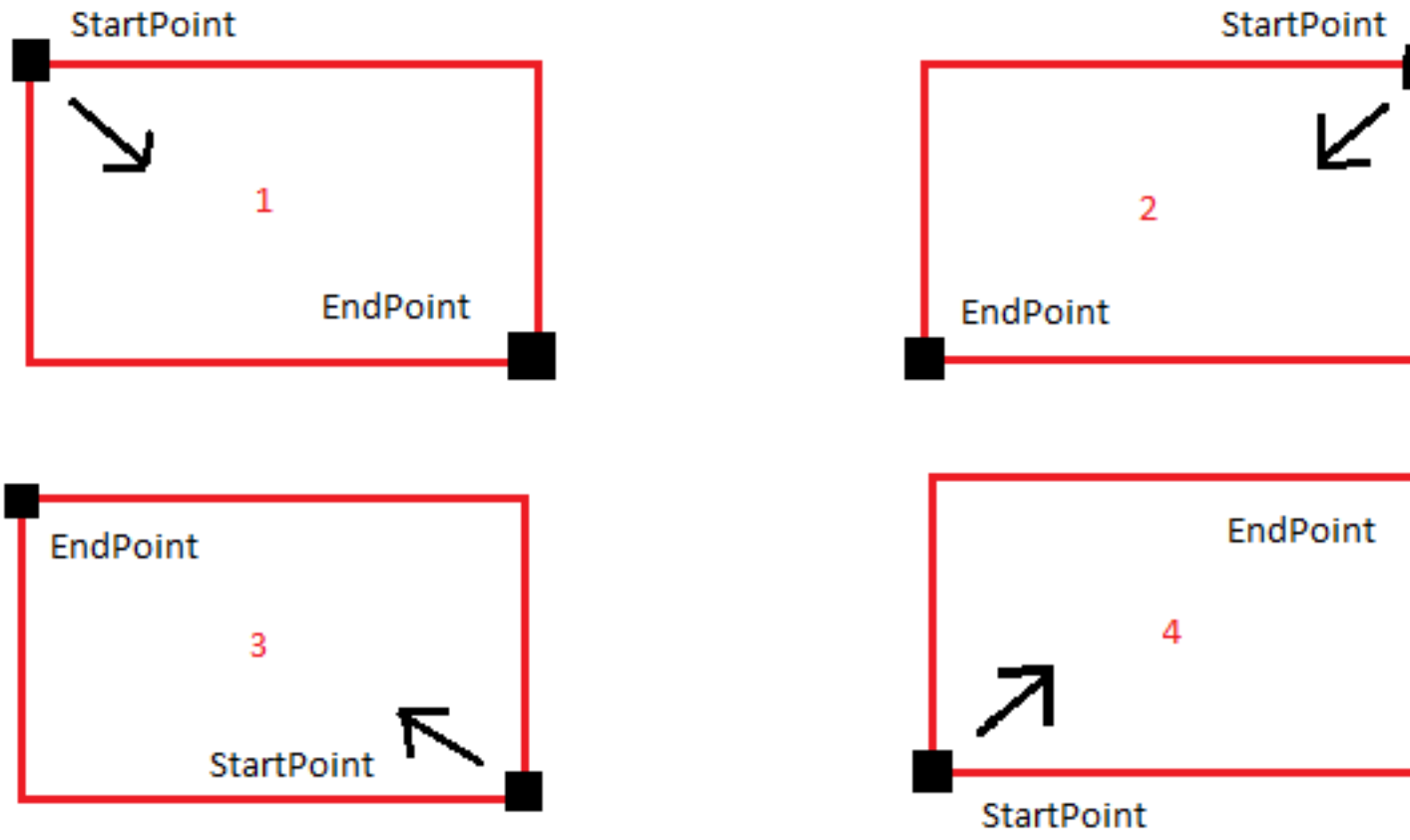
نویسنده: صابر فتح الهی
تاریخ: ۱۳۹۱/۱۱/۱۸ ۱۰:۴۰

بله کاملا حق با شماست خیلی کارها می‌شه روش انجام داد (قصد آموزش یک مبحث به زبان ساده بود)==> نظر شما اعمال شد.
سعی می‌کنم در زمان ارائه پروژه نهایی همه اینها اعمال بشه

عنوان: **پیاده سازی پروژه نقاشی (Paint) به صورت شی گرا #3**
نویسنده: صابر فتح الهی
تاریخ: ۱۳۹۱/۱۱/۱۸ ۲:۱۵
آدرس: www.dotnettips.info
برچسب‌ها: OOP, C#, Project, Paint

در ادامه مطالب قبل
پیاده سازی پروژه نقاشی (Paint) به صورت شی گرا #1
پیاده سازی پروژه نقاشی (Paint) به صورت شی گرا #2

قبل از شروع توضیحات متدهای کلاس Shape در ادامه پست‌های قبل در ^ و ^ ابتدا به تشریح یک تصویر می‌پردازیم.



خوب همانگونه که در تصویر بالا مشاده می‌نمایید، برای رسم یک شی چهار حالت متفاوت ممکن است پیش بیاید. (دقت کنید که ربع اول محور مختصات روی بوم گرافیکی قرار گرفته است، در واقع گوشه بالا و سمت چپ بوم گرافیکی نقطه (0 و 0) محور مختصات است و عرض بوم گرافیکی محور Xها و ارتفاع بوم گرافیکی محور Yها را نشان می‌دهد)

در این حالت StartPoint.X < EndPoint.X و StartPoint.Y < EndPoint.Y خواهد بود. (StartPoint نقطه ای است که ابتدا ماوس شروع به ترسیم می‌کند، و EndPoint زمانی است که ماوس رها شده و پایان ترسیم را مشخص می‌کند.)
در این حالت StartPoint.X > EndPoint.X و StartPoint.Y > EndPoint.Y خواهد بود.
در این حالت StartPoint.X > EndPoint.X و StartPoint.Y > EndPoint.Y خواهد بود.
در این حالت StartPoint.X < EndPoint.X و StartPoint.Y > EndPoint.Y خواهد بود.

ابتدا یک کلاس کمکی به صورت استاتیک تعریف می‌کنیم که متدی جهت پیش نمایش رسم شی در حالت جابجایی ، رسم، و تغییر اندازه دارد.

```
using System;
using System.Drawing;

namespace PWS.ObjectOrientedPaint.Models
{
    /// <summary>
    /// Helpers
    /// </summary>
    public static class Helpers
    {
        /// <summary>
        /// Draws the preview.
        /// </summary>
        /// <param name="g">The g.</param>
        /// <param name="startPoint">The start point.</param>
        /// <param name="endPoint">The end point.</param>
        /// <param name="foreColor">Color of the fore.</param>
        /// <param name="thickness">The thickness.</param>
        /// <param name="isFill">if set to <c>true</c> [is fill].</param>
        /// <param name="backgroundBrush">The background brush.</param>
        /// <param name="shapeType">Type of the shape.</param>
        public static void DrawPreview(Graphics g, PointF startPoint, PointF endPoint, Color foreColor,
byte thickness, bool isFill, Brush backgroundBrush, ShapeType shapeType)
        {
            float x = 0, y = 0;
            float width = Math.Abs(endPoint.X - startPoint.X);
            float height = Math.Abs(endPoint.Y - startPoint.Y);
            if (startPoint.X <= endPoint.X && startPoint.Y <= endPoint.Y)
            {
                x = startPoint.X;
                y = startPoint.Y;
            }
            else if (startPoint.X >= endPoint.X && startPoint.Y >= endPoint.Y)
            {
                x = endPoint.X;
                y = endPoint.Y;
            }
            else if (startPoint.X >= endPoint.X && startPoint.Y <= endPoint.Y)
            {
                x = endPoint.X;
                y = startPoint.Y;
            }
            else if (startPoint.X <= endPoint.X && startPoint.Y >= endPoint.Y)
            {
                x = startPoint.X;
                y = endPoint.Y;
            }

            switch (shapeType)
            {
                case ShapeType.Ellipse:
                    if (isFill)
                        g.FillEllipse(backgroundBrush, x, y, width, height);
                    //else
                    g.DrawEllipse(new Pen(foreColor, thickness), x, y, width, height);
                    break;
                case ShapeType.Rectangle:
                    if (isFill)
                        g.FillRectangle(backgroundBrush, x, y, width, height);
                    //else
                    g.DrawRectangle(new Pen(foreColor, thickness), x, y, width, height);
                    break;
                case ShapeType.Circle:
                    float raduis = Math.Max(width, height);

                    if (isFill)
                        g.FillEllipse(backgroundBrush, x, y, raduis, raduis);
                    //else
                    g.DrawEllipse(new Pen(foreColor, thickness), x, y, raduis, raduis);
                    break;
                case ShapeType.Square:
                    float side = Math.Max(width, height);

                    if (isFill)
                        g.FillRectangle(backgroundBrush, x, y, side, side);
                    //else
                    g.DrawRectangle(new Pen(foreColor, thickness), x, y, side, side);
```

```
                    break;
                case ShapeType.Line:
                    g.DrawLine(new Pen(foreColor, thickness), startPoint, endPoint);
                    break;
                case ShapeType.Diamond:
                    var points = new PointF[4];
                    points[0] = new PointF(x + width / 2, y);
                    points[1] = new PointF(x + width, y + height / 2);
                    points[2] = new PointF(x + width / 2, y + height);
                    points[3] = new PointF(x, y + height / 2);
                    if (isFill)
                        g.FillPolygon(backgroundBrush, points);
                    //else
                    g.DrawPolygon(new Pen(foreColor, thickness), points);
                    break;
                case ShapeType.Triangle:
                    var tPoints = new PointF[3];
                    tPoints[0] = new PointF(x + width / 2, y);
                    tPoints[1] = new PointF(x + width, y + height);
                    tPoints[2] = new PointF(x, y + height);
                    if (isFill)
                        g.FillPolygon(backgroundBrush, tPoints);
                    //else
                    g.DrawPolygon(new Pen(foreColor, thickness), tPoints);
                    break;
            }
            if (shapeType != ShapeType.Line)
            {
                g.DrawString(String.Format("({0},{1})", x, y), new Font(new FontFamily("Tahoma"), 10),
new SolidBrush(foreColor), x - 20, y - 25);
                g.DrawString(String.Format("({0},{1})", x + width, y + height), new Font(new
FontFamily("Tahoma"), 10), new SolidBrush(foreColor), x + width - 20, y + height + 5);
            }
            else
            {
                g.DrawString(String.Format("({0},{1})", startPoint.X, startPoint.Y), new Font(new
FontFamily("Tahoma"), 10), new SolidBrush(foreColor), startPoint.X - 20, startPoint.Y - 25);
                g.DrawString(String.Format("({0},{1})", endPoint.X, endPoint.Y), new Font(new
FontFamily("Tahoma"), 10), new SolidBrush(foreColor), endPoint.X - 20, endPoint.Y + 5);
            }

        }
    }
}
```

متد های این کلاس:

DrawPreview : این متد پیش نمایشی برای شی در زمان ترسیم، جابجایی و تغییر اندازه آماده می‌کند، پارامترهای آن عبارتند از : **بوم گرافیکی** ، **نقطه شروع** ، **نقطه پایان** و **رنگ قلم ترسیم** پیش نمایش شی، **ضخامت خط** ، **آیا شی توپر باشد** ؟، **الگوی پر کردن پس زمینه شی** ، و **نوع شی ترسیمی** می‌باشد.

در ادامه پست‌های قبل ادامه کد کلاس Shape را تشریح می‌کنیم.

```
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Net;

namespace PWS.ObjectOrientedPaint.Models
{
    /// <summary>
    /// Shape (Base Class)
    /// </summary>
    public abstract partial class Shape
    {
#region Constructors (2)

        /// <summary>
        /// Initializes a new instance of the <see cref="Shape" /> class.
        /// </summary>
        /// <param name="startPoint">The start point.</param>
        /// <param name="endPoint">The end point.</param>
        /// <param name="zIndex">Index of the z.</param>
        /// <param name="foreColor">Color of the fore.</param>
        /// <param name="thickness">The thickness.</param>
```

```
        /// <param name="isFill">if set to <c>true</c> [is fill].</param>
        /// <param name="backgroundColor">Color of the background.</param>
        protected Shape(PointF startPoint, PointF endPoint, int zIndex, Color foreColor, byte
thickness, bool isFill, Color backgroundColor)
        {
            CalulateLocationAndSize(startPoint, endPoint);
            Zindex = zIndex;
            ForeColor = foreColor;
            Thickness = thickness;
            IsFill = isFill;
            BackgroundColor = backgroundColor;
        }

        /// <summary>
        /// Initializes a new instance of the <see cref="Shape" /> class.
        /// </summary>
        protected Shape() { }

#endregion Constructors

#region Methods (10)

// Public Methods (9)

        /// <summary>
        /// Draws the specified g.
        /// </summary>
        /// <param name="g">The g.</param>
        public virtual void Draw(Graphics g)
        {
            if (!IsSelected) return;
            float diff = Thickness + 4;
            Color myColor = Color.DarkSeaGreen;
            g.DrawString(String.Format("({0},{1})", StartPoint.X, StartPoint.Y), new Font(new
FontFamily("Tahoma"), 10), new SolidBrush(myColor), StartPoint.X - 20, StartPoint.Y - 25);
            g.DrawString(String.Format("({0},{1})", EndPoint.X, EndPoint.Y), new Font(new
FontFamily("Tahoma"), 10), new SolidBrush(myColor), EndPoint.X - 20, EndPoint.Y + 5);
            if (ShapeType != ShapeType.Line)
            {
                g.DrawRectangle(new Pen(myColor), X, Y, Width, Height);

                //  1 2 3
                //  8   4
                //  7 6 5
                var point1 = new PointF(StartPoint.X - diff / 2, StartPoint.Y - diff / 2);
                var point2 = new PointF((StartPoint.X - diff / 2 + EndPoint.X) / 2, StartPoint.Y - diff
/ 2);
                var point3 = new PointF(EndPoint.X - diff / 2, StartPoint.Y - diff / 2);
                var point4 = new PointF(EndPoint.X - diff / 2, (EndPoint.Y + StartPoint.Y) / 2 - diff /
2);
                var point5 = new PointF(EndPoint.X - diff / 2, EndPoint.Y - diff / 2);
                var point6 = new PointF((StartPoint.X - diff / 2 + EndPoint.X) / 2, EndPoint.Y - diff /
2);
                var point7 = new PointF(StartPoint.X - diff / 2, EndPoint.Y - diff / 2);
                var point8 = new PointF(StartPoint.X - diff / 2, (EndPoint.Y + StartPoint.Y) / 2 - diff
/ 2);


                g.FillRectangle(new SolidBrush(myColor), point1.X, point1.Y, diff, diff);
                g.FillRectangle(new SolidBrush(myColor), point2.X, point2.Y, diff, diff);
                g.FillRectangle(new SolidBrush(myColor), point3.X, point3.Y, diff, diff);
                g.FillRectangle(new SolidBrush(myColor), point4.X, point4.Y, diff, diff);
                g.FillRectangle(new SolidBrush(myColor), point5.X, point5.Y, diff, diff);
                g.FillRectangle(new SolidBrush(myColor), point6.X, point6.Y, diff, diff);
                g.FillRectangle(new SolidBrush(myColor), point7.X, point7.Y, diff, diff);
                g.FillRectangle(new SolidBrush(myColor), point8.X, point8.Y, diff, diff);
            }
            else
            {
                var point1 = new PointF(StartPoint.X - diff / 2, StartPoint.Y - diff / 2);
                var point2 = new PointF(EndPoint.X - diff / 2, EndPoint.Y - diff / 2);
                g.FillRectangle(new SolidBrush(myColor), point1.X, point1.Y, diff, diff);
                g.FillRectangle(new SolidBrush(myColor), point2.X, point2.Y, diff, diff);
            }
        }

        /// <summary>
        /// Points the in sahpe.
        /// </summary>
        /// <param name="point">The point.</param>
        /// <param name="tolerance">The tolerance.</param>
```

```
        /// <returns>
        ///   <c>true</c> if [has point in sahpe] [the specified point]; otherwise, <c>false</c>.
        /// </returns>
        public virtual bool HasPointInSahpe(PointF point, byte tolerance = 5)
        {
            return point.X > (StartPoint.X - tolerance) && point.X < (EndPoint.X + tolerance) &&
point.Y > (StartPoint.Y - tolerance) && point.Y < (EndPoint.Y + tolerance);
        }

        /// <summary>
        /// Moves the specified location.
        /// </summary>
        /// <param name="location">The location.</param>
        /// <returns></returns>
        public virtual PointF Move(Point location)
        {
            StartPoint = new PointF(location.X, location.Y);
            EndPoint = new PointF(location.X + Width, location.Y + Height);
            return StartPoint;
        }

        /// <summary>
        /// Moves the specified dx.
        /// </summary>
        /// <param name="dx">The dx.</param>
        /// <param name="dy">The dy.</param>
        /// <returns></returns>
        public virtual PointF Move(int dx, int dy)
        {
            StartPoint = new PointF(StartPoint.X + dx, StartPoint.Y + dy);
            EndPoint = new PointF(EndPoint.X + dx, EndPoint.Y + dy);
            return StartPoint;
        }

        /// <summary>
        /// Resizes the specified dx.
        /// </summary>
        /// <param name="dx">The dx.</param>
        /// <param name="dy">The dy.</param>
        /// <returns></returns>
        public virtual SizeF Resize(int dx, int dy)
        {
            EndPoint = new PointF(EndPoint.X + dx, EndPoint.Y + dy);
            return new SizeF(Width, Height);
        }

        /// <summary>
        /// Resizes the specified start point.
        /// </summary>
        /// <param name="startPoint">The start point.</param>
        /// <param name="currentPoint">The current point.</param>
        public virtual void Resize(PointF startPoint, PointF currentPoint)
        {
            var dx = (int)(currentPoint.X - startPoint.X);
            var dy = (int)(currentPoint.Y - startPoint.Y);
            if (startPoint.X >= X - 5 && startPoint.X <= X + 5)
            {
                StartPoint = new PointF(currentPoint.X, StartPoint.Y);
                if (ShapeType == ShapeType.Circle || ShapeType == ShapeType.Square)
                {
                    Height = Width;
                }
            }
            else if (startPoint.X >= EndPoint.X - 5 && startPoint.X <= EndPoint.X + 5)
            {
                Width += dx;
                if (ShapeType == ShapeType.Circle || ShapeType == ShapeType.Square)
                {
                    Height = Width;
                }
            }
            else if (startPoint.Y >= Y - 5 && startPoint.Y <= Y + 5)
            {
                Y = currentPoint.Y;
                if (ShapeType == ShapeType.Circle || ShapeType == ShapeType.Square)
                {
                    Width = Height;
                }
            }
            else if (startPoint.Y >= EndPoint.Y - 5 && startPoint.Y <= EndPoint.Y + 5)
            {
```

```
                Height += dy;
                if (ShapeType == ShapeType.Circle || ShapeType == ShapeType.Square)
                {
                    Width = Height;
                }
            }
        }

        /// <summary>
        /// Sets the background brush as hatch.
        /// </summary>
        /// <param name="hatchStyle">The hatch style.</param>
        public virtual void SetBackgroundBrushAsHatch(HatchStyle hatchStyle)
        {
            var brush = new HatchBrush(hatchStyle, BackgroundColor);
            BackgroundBrush = brush;
        }

        /// <summary>
        /// Sets the background brush as linear gradient.
        /// </summary>
        public virtual void SetBackgroundBrushAsLinearGradient()
        {
            var brush = new LinearGradientBrush(StartPoint, EndPoint, ForeColor, BackgroundColor);
            BackgroundBrush = brush;
        }

        /// <summary>
        /// Sets the background brush as solid.
        /// </summary>
        public virtual void SetBackgroundBrushAsSolid()
        {
            var brush = new SolidBrush(BackgroundColor);
            BackgroundBrush = brush;
        }
// Private Methods (1)

        /// <summary>
        /// Calulates the size of the location and.
        /// </summary>
        /// <param name="startPoint">The start point.</param>
        /// <param name="endPoint">The end point.</param>
        private void CalulateLocationAndSize(PointF startPoint, PointF endPoint)
        {
            float x = 0, y = 0;
            float width = Math.Abs(endPoint.X - startPoint.X);
            float height = Math.Abs(endPoint.Y - startPoint.Y);
            if (startPoint.X <= endPoint.X && startPoint.Y <= endPoint.Y)
            {
                x = startPoint.X;
                y = startPoint.Y;
            }
            else if (startPoint.X >= endPoint.X && startPoint.Y >= endPoint.Y)
            {
                x = endPoint.X;
                y = endPoint.Y;
            }
            else if (startPoint.X >= endPoint.X && startPoint.Y <= endPoint.Y)
            {
                x = endPoint.X;
                y = startPoint.Y;
            }
            else if (startPoint.X <= endPoint.X && startPoint.Y >= endPoint.Y)
            {
                x = startPoint.X;
                y = endPoint.Y;
            }
            StartPoint = new PointF(x, y);
            EndPoint = new PointF(X + width, Y + height);
        }

#endregion Methods
    }
}
```

حال به تشریح **سازنده** کلاس می‌پردازیم:

Shape : پارامترهای این سازنده به ترتیب عبارتند از **نقطه شروع** ، **نقطه پایان** ، **عمق شی** ، **رنگ قلم** ، **ضخامت خط** ، **آیا شی توپر باشد** ؟، و **رنگ پر کردن شی** ، در این سازنده ابتدا توسط متدی به نام CalulateLocationAndSize(startPoint, endPoint); b نقاط ابتدا و انتهای شی مورد نظر تنظیم می‌شود، در متد مذکور بررسی می‌شود در صورتی که نقاط شروع و پایان یکی از حالت‌های 1 ، 2، 3، 4 از تصویر ابتدا پست باشد همگی تبدیل به حالت 1 خواهد شد.

سپس به تشریح **متدهای** کلاس Shape می‌پردازیم:
Draw : این متد دارای یک پارامتر ورودی است که **بوم گرافیکی** مورد نظر می‌باشد، در واقع شی مورد نظر خود را بروی این بوم گرافیکی ترسیم می‌کند. در کلاس پایه کار این متد زیاد پیچیده نیست، در صورتی که شی در حالت انتخاب باشد (= IsSelected true) بروی شی مورد نظر 8 مربع کوچک ترسیم می‌شود و اگر شی مورد نظر خط باشد دو مربع کوچک در طرفین خط رسم می‌شود که نشان دهنده انتخاب شدن شی مورد نظر است. این متد به صورت virtual تعریف شده است یعنی کلاس هایی که از Shape ارث میبرند می‌توانند این متد را برای خود از نو بازنویسی کرده (override کنند) و تغییر رفتار دهند.
HasPointInSahpe : این متد نیز به صورت virtual تعریف شده است دارای خروجی بولین می‌باشد. پارامترهای این متد عبارتند از **یک نقطه** و **یک عدد** که نشان دهنده تلرانش نقطه بر حسب پیکسل می‌باشد. کار این متد این است که یک نقطه را گرفته و بررسی می‌کند که آیا نقطه مورد نظر با تلرانس وارد شده آیا در داخل شی واقع شده است یا خیر (مثلا وجود نقطه در مستطیل یا وجود نقطه در دایره فرمول‌های متفاوتی دارند که در اینجا پیش فرض برای تمامی اشیا حالت مستطیل در نظر گرفته شده که می‌توانید آنها را بازنویسی (override) کنید).

Move : این متد به عنوان پارامتر **یک نقطه** را گرفته و شی مورد نظر را به آن نقطه منتقل می‌کند در واقع نقطه شروع و پایان ترسیم شی را تغییر می‌دهد.

Move : این متد نیز برای جابجایی شی به کار می‌رود، این متد دارای پارامترهای **جابجابی در راستای محور Xها** , **جابجایی در راستای محور Yها** ؛ و شی مورد نظر را به آن نقطه منتقل می‌کند در واقع نقطه شروع و پایان ترسیم شی را با توجه به پارامترهای ورودی تغییر می‌دهد.
Resize : این متد نیز برای تغییر اندازه شی به کار می‌رود، این متد دارای پارامترهای **تغییر اندازه در راستای محور Xها** , **تغییر اندازه در راستای محور Yها** می‌باشد و نقطه پایان شی مورد نظر را تغییر می‌دهد اما نقطه شروع تغییری نمی‌کند.

Resize : این متد نیز برای تغییر اندازه شی به کار می‌رود، در زمان تغییر اندازه شی با ماوس ابتدا یک نقطه شروع وجود دارد که ماوس در آن نقطه کلیک شده و شروع به درگ کردن شی جهت تغییر اندازه می‌کند (پارامتر اول این متد نقطه شروع درگ کردن جهت تغییر اندازه را مشخص می‌کند startPoint)، سپس در یک نقطه ای درگ کردن تمام می‌شود در این نقطه باید شی تغییر اندازه پیدا کرده و ترسیم شود (پارامتر دوم این متد نقطه مذکور می‌باشد currentLocation). سپس با توجه با این دو نقطه بررسی می‌شود که تغییر اندازه در کدام جهت صورت گرفته است و اعداد جهت تغییرات نقاط شروع و پایان شی مورد نظر محاسبه می‌شوند. (مثلا تغییر اندازه در مستطیل از ضلع بالا به طرفین، یا از ضلع سمت راست به طرفین و ....). البته برای مربع و دایره باید کاری کنیم که طول و عرض تغییر اندازه یکسان باشد.

CalulateLocationAndSize : این متد که در سازنده کلاس استفاده شده در واقع دو نقطه شروع و پایان را گرفته و با توجه به تصویر ابتدای پست حالت‌های 1 و 2 و3 و 4 را به حالت 1 تبدیل کرده و StartPoint و EndPoint را اصلاح می‌کند.
SetBackgroundBrushAsHatch : این متد یک الگوی Brush گرفته و با توجه به رنگ پس زمینه شی خصوصیت BackgroundBrush را مقداردهی می‌کند.

SetBackgroundBrushAsLinearGradient : این متد با توجه به خصوصیت ForeColor و BackgroundColor یک Gradiant Brush ساخته و آن را به خصوصیت
BackgroundBrush نسبت می‌کند.
SetBackgroundBrushAsSolid : یک الگوی پر کردن توپر برای شی مورد نظر با توجه به خصوصیت BackgroundColor شی ایجاد کرده و آن را به خصوصیت BackgroundBrush شی نسبت می‌دهد.

تذکر : متدهای Move، Resize و HasPointInShape به صورت virtual تعریف شده تا کلاس‌های مشتق شده در صورت نیاز خود کد رفتار مورد نظر خود را override کرده یا از همین رفتار استفاده نمایند.

خوشحال می‌شم در صورتی که در Refactoring کد نوشته شده با من همکاری کنید.

در پست‌های آینده به بررسی و پیاده سازی دیگر کلاس‌ها خواهیم پرداخت.

نظرات خوانندگان

نویسنده: بتیسا
تاریخ: ۱۳۹۱/۱۱/۱۸ ۱۰:۴۱

با سلام

از مطلب مفیدتون ممنونم
در متد **DrawPreview** اصلی که نوشته شده در بخش هایی که اشیاء توپر رسم می‌شوند مانند خط 143، 149 و... بجای استفاده از خصوصیت Brush که در بخش قبل برای پس زمینه در نظر گرفته شده بود هر بار یک براش ایجاد شده که می‌توانیم به صورت زیر اصلاح کنیم.

```
case ShapeType.Ellipse:
                    if (isFill)
                        g.FillEllipse(new SolidBrush(backgroundColor), x, y, width, height);
                    //else
                    g.DrawEllipse(new Pen(foreColor, thickness), x, y, width, height);
                    break;

//--------------------------------[Change to]-------=>

               case ShapeType.Ellipse:
                    if (isFill)
                        g.FillEllipse(Brush, x, y, width, height);
                    //else
                    g.DrawEllipse(new Pen(foreColor, thickness), x, y, width, height);
                    break;
```

نویسنده: صابر فتح الهی
تاریخ: ۱۳۹۱/۱۱/۱۸ ۱۰:۴۶

بله دوست گلم میشد اینکارو انجام داد
اما با توجه به اینکه متد **DrawPreview** به صورت static تعریف شده نمی‌توان از خصوصیات غیر استاتیک کلاس در آن استفاده کرد.
درسته؟

نویسنده: بتیسا
تاریخ: ۱۳۹۱/۱۱/۱۸ ۱۰:۵۸

بله به static بودن متد توجه نکرده بودم

نویسنده: بتیسا
تاریخ: ۱۳۹۱/۱۱/۱۸ ۱۱:۱۹

برای برطرف کردن این مسئله هم می‌توانیم همانطور که در ورودی foreColor را دریافت کردیم brush را نیز دریافت کنیم.

```
public static void DrawPreview(Graphics g, PointF startPoint, PointF endPoint, Color foreColor, byte
thickness, bool isFill, Color backgroundColor, ShapeType shapeType)
//--------------------------------[Change to]-------=>
public static void DrawPreview(Graphics g, PointF startPoint, PointF endPoint, Color foreColor, byte
thickness, bool isFill, Brush backgroundBrush , ShapeType shapeType)


//-------------------------------------------------
    case ShapeType.Ellipse:
                    if (isFill)
                        g.FillEllipse(new SolidBrush(backgroundColor), x, y, width, height);
                    //else
```

```
                    g.DrawEllipse(new Pen(foreColor, thickness), x, y, width, height);
                    break;

//--------------------------------[Change to]-------=>

                case ShapeType.Ellipse:
                    if (isFill)
                        g.FillEllipse(backgroundBrush, x, y, width, height);
                    //else
                    g.DrawEllipse(new Pen(foreColor, thickness), x, y, width, height);
                    break;
```

نویسنده: صابر فتح الهی
تاریخ: ۱۳۹۱/۱۱/۱۸ ۱۲:۷

درسته اما در قسمت اینترفیس کاربر باید Brushهای مورد نظر ساخته شده و به این متد پاس داده شود، در مراحل پایانی فکر می‌کنم بهتر منظورم بتونم برسونم، به دلایلی (که در پست‌های آینده گفته میشه) از این روش‌ها استفاده نکردم.

نویسنده: سعید
تاریخ: ۱۳۹۱/۱۱/۱۸ ۱۴:۱۷

چرا drawpreview به صورت استاتیک تعریف شده؟ چرا دوبار تعریف شده؟ و چرا این کلاس پایه اطلاعات زیادی در مورد رسم زیر مجموعه‌های خودش داره؟ آیا بهتر نیست جرئیات ترسیم هر شیء با override شدن به زیر کلاس‌های مشتق شده واگذار بشن؟ چرا این متدها از خاصیت‌های کلاس تعریف شده استفاده نمی‌کنن و دوباره این خاصیت‌ها رو به صورت پارامتر دریافت کردن؟ (همون بحث اعلام استقلال متد تعریف شده به صورت استاتیک و اینکه چرا؟) و اگر این کلاس پایه تا این اندازه لازم هست در مورد رسم دایره و سایر اشکال اطلاعات داشته باشد، چه ضرورتی به abstarct بودن آن هست؟ اصلا چه ضرورتی به تعریف اشیاء مشتق شده از آن هست؟

نویسنده: صابر فتح الهی
تاریخ: ۱۳۹۱/۱۱/۱۸ ۱۷:۱۱

جز متد **DrawPreview** هیچکدام از متدها پارامتری دریافت نمی‌کنند، اون هم به دلیل اینه که می‌خوام پیش نمایشی از یک شی ترسیم کنم البته می‌تونستیم اون توی یک کلاس جدا بنویسیم که این کلاس زیاد شلوغ نشه فکر می‌کنم با نوشتن کلاس‌های مشتق شده بهتر بشه توی این زمینه‌ها بحث کرد.

پ.ن : متد **DrawPreview** به یک کلاس ثالث منتقل شد.

نویسنده: صابر فتح الهی
تاریخ: ۱۳۹۱/۱۱/۱۸ ۱۷:۵۹

نظر شما اعمال شد

نویسنده: مسعود بهرامی
تاریخ: ۱۳۹۲/۰۹/۰۱ ۱۶:۵۸

با اجازه دوست عزیزم مهندس فتح الهی
من به نظرم Helpers رو اگه به شکل زیر Re factor کنیم بهتر باشه :)
اول یه کلاس تعریف می‌کنیم و اطلاعات لازم برای ترسیم پیش نمایش رو تو اون کلاس میزاریم

```
public class ShapeSpecification
    {
```

```
        public PointF StartPoint{get;set;}
        public PointF EndPoint{get;set;}
        public Color ForeColor{get;set;}
        public byte Thickness{get;set;}
        public bool IsFill{get;set;}
        public Brush BackgroundBrush{get;set;}
    }
```

یه کلاس دیگه هم نقاط ابتدا و انتها و طول و عرض رو تو خودش داره

```
public class StartPoints
    {
        public float XPoint { get; set; }
        public float YPoint { get; set; }
        public float Width { get; set; }
        public float Height { get; set; }
    }
```

حالا یه اینترفیس تعریف می‌کنیم که فقط یه متد داره به نام Draw

```
public interface IPeiview
    {
        void Draw(ShapeSpecification shapeScepification);
    }
```

حالا می‌رسیم به کلاس Helpers اصلیمون که میتونه هم استاتیک باشه و هم معمولی به دو شکل زیر

```
public class Helpers
    {
        private readonly IPeiview peiview;

        public Helpers(IPeiview peiview)
        {
            this.peiview = peiview;
        }

        public void Draw(ShapeSpecification shapeSpecification)
        {
            peiview.Draw(shapeSpecification);
        }
    }
```

```
public static  class Helpers
    {

        public static void Draw(ShapeSpecification shapeSpecification, IPeiview peiview)
        {
            peiview.Draw(shapeSpecification);
        }
    }
```

که فقط یه متد ساده Draw داره و اونم تابع Draw اینترفیسی که بش دادیم رو صدا میزینه

یه کلاس دیگه هم تعریف میکنیم که مسئولیتش تشخیص بوم‌های چهارگانه است برای شروع نقطه‌ی ترسیم

```
public static class AreaParser
    {
        public static StartPoints Parse(PointF startPoint, PointF endPoint)
        {
            var startPoints = new StartPoints();

            startPoints.Width = Math.Abs(endPoint.X - startPoint.X);
            startPoints.Height = Math.Abs(endPoint.Y - startPoint.Y);

            if (startPoint.X <= endPoint.X && startPoint.Y <= endPoint.Y)
            {
                startPoints.XPoint = startPoint.X;
                startPoints.YPoint = startPoint.Y;
            }
```

```
            else if (startPoint.X >= endPoint.X && startPoint.Y >= endPoint.Y)
            {
                startPoints.XPoint = endPoint.X;
                startPoints.YPoint = endPoint.Y;
            }

            else if (startPoint.X >= endPoint.X && startPoint.Y <= endPoint.Y)
            {
                startPoints.XPoint = endPoint.X;
                startPoints.YPoint = startPoint.Y;
            }

            else if (startPoint.X <= endPoint.X && startPoint.Y >= endPoint.Y)
            {
                startPoints.XPoint = startPoint.X;
                startPoints.YPoint = endPoint.Y;
            }
            return startPoints;
        }
    }
```

نکته: این کلاس رو اگه با Func ایجاد کنیم خیلی بهتر و تمیزتر وقشنکتر هم میشد که من می‌گزرم فعلا ازش
حالا ما هر شکل جدید که اضافه کنیم به پروژه Paint خودمون و قصد پیش نمایش اونو داشته باشیم فقط کافیه یه کلاس برا پیش نمایشش ایجاد کنیم که کلاس Ipreview رو Implement کنه و متد Draw مخصوص به خودش را داشته باشه و از شر Swithهای طولانی خلاص میشیم مثلا من برای دایره اینکارو کردم

```
public class CirclePreview:IPeiview
    {
        private readonly Graphics graphics;

        public CirclePreview(Graphics graphics)
        {
            this.graphics = graphics;
        }
        public void Draw(ShapeSpecification shapeScepification)
        {
            var startPoints = AreaParser.Parse(shapeScepification.StartPoint,
shapeScepification.EndPoint);

            float raduis = Math.Max(startPoints.Width, startPoints.Height);

            if (shapeScepification.IsFill)
                this.graphics.FillEllipse(shapeScepification.BackgroundBrush, startPoints.XPoint,
startPoints.YPoint, raduis, raduis);
            else
                this.graphics.DrawEllipse(new Pen(shapeScepification.ForeColor,
shapeScepification.Thickness), startPoints.XPoint, startPoints.YPoint, raduis, raduis);
        }
    }
```

نویسنده: صابر فتح الهی
تاریخ: ۱۳۹۲/۰۹/۰۱ ۲۳:۲۷

ظاهرا همه چیز مرتبه و درست هست.
من بررسی می‌کنم و نتیجش به شما اطلاع میدم
در هر صورت از وقتی که گذاشتین متشکرم

عنوان: **پیاده سازی پروژه نقاشی (Paint) به صورت شی گرا #4**
نویسنده: صابر فتح الهی
تاریخ: ۱۳۹۱/۱۱/۲۲ ۲۳:۳۰
آدرس: www.dotnettips.info
برچسب‌ها: C#, OOP, Project, Paint

در ادامه پست قبل ، در این پست به بررسی کلاس Triangle جهت رسم مثلث و کلاس Diamond جهت رسم لوزی می‌پردازیم.

```
using System.Drawing;

namespace PWS.ObjectOrientedPaint.Models
{
    /// <summary>
    /// Triangle
    /// </summary>
    public class Triangle : Shape
    {
        #region Constructors (2)

        /// <summary>
        /// Initializes a new instance of the <see cref="Triangle" /> class.
        /// </summary>
        /// <param name="startPoint">The start point.</param>
        /// <param name="endPoint">The end point.</param>
        /// <param name="zIndex">Index of the z.</param>
        /// <param name="foreColor">Color of the fore.</param>
        /// <param name="thickness">The thickness.</param>
        /// <param name="isFill">if set to <c>true</c> [is fill].</param>
        /// <param name="backgroundColor">Color of the background.</param>
        public Triangle(PointF startPoint, PointF endPoint, int zIndex, Color foreColor, byte
thickness, bool isFill, Color backgroundColor)
            : base(startPoint, endPoint, zIndex, foreColor, thickness, isFill, backgroundColor)
        {
            ShapeType = ShapeType.Triangle;
        }


        /// <summary>
        /// Initializes a new instance of the <see cref="Triangle" /> class.
        /// </summary>
        public Triangle()
        {
            ShapeType = ShapeType.Triangle;
        }

        #endregion Constructors

        #region Methods (1)

        // Public Methods (1)

        /// <summary>
        /// Draws the specified g.
        /// </summary>
        /// <param name="g">The g.</param>
        public override void Draw(Graphics g)
        {
            var points = new PointF[3];
            points[0] = new PointF(X + Width / 2, Y);
            points[1] = new PointF(X + Width, Y + Height);
            points[2] = new PointF(X, Y + Height);
            if (IsFill)
                g.FillPolygon(BackgroundBrush, points);
            g.DrawPolygon(new Pen(ForeColor, Thickness), points);
            base.Draw(g);
        }

        #endregion Methods
    }
}
```

همانگونه که مشاهده می‌کنید کلاس مثلث از کلاس Shape ارث برده و تشکیل شده از یک سازنده و بازنویسی (override) متد Draw می‌باشد، البته متد HasPointInSahpe در کلاس پایه قاعدتا باید بازنویسی شود، برای تشخیص وجود نقطه در شکل مثلث،

(اگر دوستان فرمولش می‌دونن ممنون می‌شم در اختیار بذارن). در متد Draw سه نقطه مثلث در نظر گرفته شده که بر طبق آن با استفاده از متدهای رسم منحنی اقدام به رسم مثلث توپر یا تو خالی نموده‌ایم.

کلاس لوزی نیز دقیقا مانند کلاس مثلث عمل می‌کند.

```
using System.Drawing;

namespace PWS.ObjectOrientedPaint.Models
{
    /// <summary>
    /// Diamond
    /// </summary>
    public class Diamond : Shape
    {
        #region Constructors (2)

        /// <summary>
        /// Initializes a new instance of the <see cref="Diamond" /> class.
        /// </summary>
        /// <param name="startPoint">The start point.</param>
        /// <param name="endPoint">The end point.</param>
        /// <param name="zIndex">Index of the z.</param>
        /// <param name="foreColor">Color of the fore.</param>
        /// <param name="thickness">The thickness.</param>
        /// <param name="isFill">if set to <c>true</c> [is fill].</param>
        /// <param name="backgroundColor">Color of the background.</param>
        public Diamond(PointF startPoint, PointF endPoint, int zIndex, Color foreColor, byte thickness,
bool isFill, Color backgroundColor)
            : base(startPoint, endPoint, zIndex, foreColor, thickness, isFill, backgroundColor)
        {
            ShapeType = ShapeType.Diamond;
        }

        /// <summary>
        /// Initializes a new instance of the <see cref="Diamond" /> class.
        /// </summary>
        public Diamond()
        {
            ShapeType = ShapeType.Diamond;
        }

        #endregion Constructors

        #region Methods (1)

        // Public Methods (1)

        /// <summary>
        /// Draws the specified g.
        /// </summary>
        /// <param name="g">The g.</param>
        public override void Draw(Graphics g)
        {
            var points = new PointF[4];
            points[0] = new PointF(X + Width / 2, Y);
            points[1] = new PointF(X + Width, Y + Height / 2);
            points[2] = new PointF(X + Width / 2, Y + Height);
            points[3] = new PointF(X, Y + Height / 2);
            if (IsFill)
                g.FillPolygon(BackgroundBrush, points);
            g.DrawPolygon(new Pen(ForeColor, Thickness), points);
            base.Draw(g);
        }

        #endregion Methods
    }
}
```

این کلاس نیز از کلاس Shape ارث برده و دارای یک سازنده بوده و متد Draw را ازنو بازنویسی می‌کند، این متد نیز با استفاده از چهار نقطه و استفاده از متد رسم منحنی در دات نت اقدام به طراحی لوزی توپر یا تو خالی می‌کند، متد HasPointInSahpe در کلاس پایه قاعدتا باید بازنویسی شود، برای تشخیص وجود نقطه در شکل لوزی، برای رسم لوزی توپر نیز خصوصیت BackgroundBrush استفاده کرده و شی توپر را رسم می‌کند.

مباحث رسم مستطیل و مربع، دایره و بیضی در پست‌های بعد بررسی خواهند شد.

پیاده سازی پروژه نقاشی (Paint) به صورت شی گرا #1
پیاده سازی پروژه نقاشی (Paint) به صورت شی گرا #2
پیاده سازی پروژه نقاشی (Paint) به صورت شی گرا #3

موفق وموید باشید.

عنوان: **پیاده سازی پروژه نقاشی (Paint) به صورت شی گرا #5**
نویسنده: صابر فتح الهی
تاریخ: ۱۳۹۱/۱۲/۰۳ ۰:۵۰
آدرس: www.dotnettips.info
برچسب‌ها: C#, OOP, Project, Paint

در ادامه مطلب پیاده سازی پروژه نقاشی (Paint) به صورت شی گرا #4 به تشریح مابقی کلاس‌های برنامه می‌پردازیم.

در این پست به شرح کلاس Rectangle جهت رسم مستطیل و Square جهت رسم مربع می‌پردازیم

```
using System.Drawing;

namespace PWS.ObjectOrientedPaint.Models
{
    /// <summary>
    /// Rectangle
    /// </summary>
    public class Rectangle : Shape
    {
        #region Constructors (2)

        /// <summary>
        /// Initializes a new instance of the <see cref="Rectangle" /> class.
        /// </summary>
        /// <param name="startPoint">The start point.</param>
        /// <param name="endPoint">The end point.</param>
        /// <param name="zIndex">Index of the z.</param>
        /// <param name="foreColor">Color of the fore.</param>
        /// <param name="thickness">The thickness.</param>
        /// <param name="isFill">if set to <c>true</c> [is fill].</param>
        /// <param name="backgroundColor">Color of the background.</param>
        public Rectangle(PointF startPoint, PointF endPoint, int zIndex, Color foreColor, byte
thickness, bool isFill, Color backgroundColor)
            : base(startPoint, endPoint, zIndex, foreColor, thickness, isFill, backgroundColor)
        {
            ShapeType = ShapeType.Rectangle;
        }

        /// <summary>
        /// Initializes a new instance of the <see cref="Rectangle" /> class.
        /// </summary>
        public Rectangle()
        {
            ShapeType = ShapeType.Rectangle;
        }

        #endregion Constructors

        #region Methods (1)

        // Public Methods (1)

        /// <summary>
        /// Draws the specified g.
        /// </summary>
        /// <param name="g">The g.</param>
        public override void Draw(Graphics g)
        {
            if (IsFill)
                g.FillRectangle(BackgroundBrush, StartPoint.X, StartPoint.Y, Width, Height);
            g.DrawRectangle(Pen, StartPoint.X, StartPoint.Y, Width, Height);
            base.Draw(g);
        }

        #endregion Methods
    }
}
```

کلاس Rectangle از کلاس پایه طراحی شده در ^ ارث بری دارد. این کلاس ساده بوده و تنها شامل یک سازنده و متد ترسیم شی مستطیل می‌باشد.

کلاس بعدی کلاس Square می‌باشد، که از کلاس بالا (Rectangle) ارث بری داشته است، کدهای این کلاس را در زیر مشاهده می‌کنید.

```
using System;
using System.Drawing;

namespace PWS.ObjectOrientedPaint.Models
{
    /// <summary>
    /// Square
    /// </summary>
    public class Square : Rectangle
    {
#region Constructors (2)

        /// <summary>
        /// Initializes a new instance of the <see cref="Square" /> class.
        /// </summary>
        /// <param name="startPoint">The start point.</param>
        /// <param name="endPoint">The end point.</param>
        /// <param name="zIndex">Index of the z.</param>
        /// <param name="foreColor">Color of the fore.</param>
        /// <param name="thickness">The thickness.</param>
        /// <param name="isFill">if set to <c>true</c> [is fill].</param>
        /// <param name="backgroundColor">Color of the background.</param>
        public Square(PointF startPoint, PointF endPoint, int zIndex, Color foreColor, byte thickness,
bool isFill, Color backgroundColor)
        {
            float x = 0, y = 0;
            float width = Math.Abs(endPoint.X - startPoint.X);
            float height = Math.Abs(endPoint.Y - startPoint.Y);
            if (startPoint.X <= endPoint.X && startPoint.Y <= endPoint.Y)
            {
                x = startPoint.X;
                y = startPoint.Y;
            }
            else if (startPoint.X >= endPoint.X && startPoint.Y >= endPoint.Y)
            {
                x = endPoint.X;
                y = endPoint.Y;
            }
            else if (startPoint.X >= endPoint.X && startPoint.Y <= endPoint.Y)
            {
                x = endPoint.X;
                y = startPoint.Y;
            }
            else if (startPoint.X <= endPoint.X && startPoint.Y >= endPoint.Y)
            {
                x = startPoint.X;
                y = endPoint.Y;
            }
            StartPoint = new PointF(x, y);
            var side = Math.Max(width, height);
            EndPoint = new PointF(x+side, y+side);
            ShapeType = ShapeType.Square;
            Zindex = zIndex;
            ForeColor = foreColor;
            Thickness = thickness;
            BackgroundColor = backgroundColor;
            IsFill = isFill;
        }

        /// <summary>
        /// Initializes a new instance of the <see cref="Square" /> class.
        /// </summary>
        public Square()
        {
            ShapeType = ShapeType.Square;
        }

#endregion Constructors
    }
}
```

این کلاس شامل دو سازنده می‌باشد که سازنده دوم فقط نوع شی را تعیین می‌کند و بقیه کارهای آن مانند مستطیل است، در واقع می‌توان از یک دیدگاه گفت که مربع یک مستطیل است که اندازه طول و عرض آن یکسان است. در سازنده اول (نحوه ترسیم

شکل) ابتدا نقاط ابتدا و انتهای رسم شکل تعیین شده و سپس با توجه به پارامترهای محاسبه شده نوع شی جهت ترسیم و دیگر خصوصیات کلاس مقدار دهی می‌شود، با این تفاوت که در نقطه EndPoint طول و عرض مربع برابر با بزرگترین مقدار طول و عرض وارد شده در سازنده کلاس تعیین شده و مربع شکل می‌گیرد. مابقی متدهای ترسیم و ... طبق کلاس پایه مستطیل و Shape تعیین می‌شود.

مطالب قبل:

پیاده سازی پروژه نقاشی (Paint) به صورت شی گرا #1

پیاده سازی پروژه نقاشی (Paint) به صورت شی گرا #2

پیاده سازی پروژه نقاشی (Paint) به صورت شی گرا #3

پیاده سازی پروژه نقاشی (Paint) به صورت شی گرا #4

نظرات خوانندگان

نویسنده: کاوه احمدی
تاریخ: ۱۳۹۱/۱۲/۰۳ ۱۰:۵۸

امروز فرصتی دست داد نگاهی اجمالی به این پروژه بیندازم. به نظرم کد نوشته شده تا به اینجا شی گرا محسوب نمی‌شود. یعنی برخی اهدافی که به واسطه آن پارادایم شی گرایی شکل گرفته در آن رعایت نشده است.
به طور مشخص منظورم متد DrawPreview است که در بخش سوم در کلاس Helpers نوشته شده. تکرار کد شدیدی که در دستور switch این متد دیده می‌شود به سادگی قابل حذف است. کد فوق 2 مشکل اساسی دارد: اول آنکه با زیاد شدن تعداد اشیای قابل رسم، این دستور switch بسیار طولانی شده (با تکرار کد) و کد ناخوانا می‌شود و دوم آنکه با اضافه شدن هر شی قابل رسم جدید به پروژه یک case باید به این دستور اضافه شود. یعنی تغییر در یک بخش از نرم‌افزار منجر به تغییر در سایر بخش‌ها (کلاس Helpers) می‌شود. بدیهی است پارادایم شی‌گرا برای جلوگیری از چنین مسائلی شکل گرفته. در غیر این صورت این کد همان کدهای ساخت‌یافته است که در قالب کلاس نوشته شده. به نظر می‌آید بهتر باشد یک اینترفیس drawable در نظر گرفته می‌شد، در این متد از آن استفاده می‌شد و اشیای قابل رسم آنرا پیاده‌سازی می‌کردند. یک راه بسیار ساده و کارامد

نویسنده: محسن
تاریخ: ۱۳۹۱/۱۲/۰۳ ۱۱:۲۵

البته میشه این قسمت کلاس پایه رو که if و else و switch زیاد داره، با توجه به مطلب « کمپین ضد IF ! » بهبود بخشید.

نویسنده: صابر فتح الهی
تاریخ: ۱۳۹۱/۱۲/۰۳ ۲۳:۴

پاسخ شما کاملا صحیح است، توی همون پست گفتم که خیلی خوشحال میشم دوستان ایده ای به من بدهند که این قسمت با استفاده از Action و Func طراحی کنم، خودم راهی به ذهنم نرسید. بله کاملا شما درست می‌فرمایید، راه حل چیست؟

پ. ن: البته این متد کاملا قابل حذف است و می‌تواند در سیستم استفاده نشود. فقط جهت پیش نمایش رسم اشیا بکار می‌رود.

نویسنده: صابر فتح الهی
تاریخ: ۱۳۹۱/۱۲/۰۵ ۱۲:۵

@کاوه احمدی
من خیلی سعی کردم طبق الگوی « کمپین ضد IF ! » عمل کردم، و پیش رفتم درست شد، اما به دلیل اینکه در زمان رسم شی در برنامه کاربری (اینترفیس) در زمان MouseMove پیش نمایش شی رسم می‌شود و در زمان MouseUp خود شی رسم می‌شود این امکان نداشتم تا از شی نمونه سازی کنم و طبق اون الگو پیش برم ، لطفا در صورتی که روشم اشتباست اصلاح بفرمایین.
موفق وموید باشید.

نویسنده: محسن
تاریخ: ۱۳۹۱/۱۲/۰۶ ۱۶:۵۵

سلام

می‌تونید از ابزارهای تزریق وابستگی برای تامین وهله مورد نیاز استفاده کنید.

عنوان: **پیاده سازی پروژه نقاشی (Paint) به صورت شی گرا #6**
نویسنده: صابر فتح الهی
تاریخ: ۱۳۹۱/۱۲/۰۵ ۰:۵
آدرس: www.dotnettips.info
برچسب‌ها: C#, OOP, Project, Paint

در ادامه پست پیاده سازی پروژه نقاشی (Paint) به صورت شی گرا #5 ، در این پست به تشریح کلاس دایره و بیضی می‌پردازیم.

ابتدا به تشریح کلاس ترسیم بیضی (Ellipse) می‌پردازیم.

```
using System.Drawing;

namespace PWS.ObjectOrientedPaint.Models
{
    /// <summary>
    /// Ellipse Draw
    /// </summary>
    public class Ellipse : Shape
    {
        #region Constructors (2)

        /// <summary>
        /// Initializes a new instance of the <see cref="Ellipse" /> class.
        /// </summary>
        /// <param name="startPoint">The start point.</param>
        /// <param name="endPoint">The end point.</param>
        /// <param name="zIndex">Index of the z.</param>
        /// <param name="foreColor">Color of the fore.</param>
        /// <param name="thickness">The thickness.</param>
        /// <param name="isFill">if set to <c>true</c> [is fill].</param>
        /// <param name="backgroundColor">Color of the background.</param>
        public Ellipse(PointF startPoint, PointF endPoint, int zIndex, Color foreColor, byte thickness,
bool isFill, Color backgroundColor)
            : base(startPoint, endPoint, zIndex, foreColor, thickness, isFill, backgroundColor)
        {
            ShapeType = ShapeType.Ellipse;
        }

        /// <summary>
        /// Initializes a new instance of the <see cref="Ellipse" /> class.
        /// </summary>
        public Ellipse()
        {
            ShapeType = ShapeType.Ellipse;
        }

        #endregion Constructors

        #region Methods (1)

        // Public Methods (1)

        /// <summary>
        /// Draws the specified g.
        /// </summary>
        /// <param name="g">The g.</param>
        public override void Draw(Graphics g)
        {
            if (IsFill)
                g.FillEllipse(BackgroundBrush, StartPoint.X, StartPoint.Y, Width, Height);
            g.DrawEllipse(Pen, StartPoint.X, StartPoint.Y, Width, Height);
            base.Draw(g);
        }

        #endregion Methods
    }
}
```

این کلاس از شی Shape ارث برده و دارای دو سازنده ساده می‌باشد که نوع شی ترسیمی را مشخص می‌کنند، در متد Draw نیز با توجه به توپر یا توخالی بودن شی ترسیم آن انجام میشود، در این کلاس باید متد **HasPointInShape** بازنویسی (override) شود، در این متد باید تعیین شود که یک نقطه در داخل بیضی قرار گرفته است یا خیر که متاسفانه فرمول بیضی خاطرم نبود. البته به

صورت پیش فرض نقطه با توجه به چهارگوشی که بیضی را احاطه می‌کند سنجیده می‌شود.

کلاس دایره (Circle) از کلاس بالا (Ellipse) ارث بری دارد که کد آن را در زیر مشاهده می‌نمایید.

```
using System;
using System.Drawing;

namespace PWS.ObjectOrientedPaint.Models
{
    /// <summary>
    /// Circle
    /// </summary>
    public class Circle : Ellipse
    {
#region Constructors (2)

        /// <summary>
        /// Initializes a new instance of the <see cref="Circle" /> class.
        /// </summary>
        /// <param name="startPoint">The start point.</param>
        /// <param name="endPoint">The end point.</param>
        /// <param name="zIndex">Index of the z.</param>
        /// <param name="foreColor">Color of the fore.</param>
        /// <param name="thickness">The thickness.</param>
        /// <param name="isFill">if set to <c>true</c> [is fill].</param>
        /// <param name="backgroundColor">Color of the background.</param>
        public Circle(PointF startPoint, PointF endPoint, int zIndex, Color foreColor, byte thickness,
bool isFill, Color backgroundColor)
        {
            float x = 0, y = 0;
            float width = Math.Abs(endPoint.X - startPoint.X);
            float height = Math.Abs(endPoint.Y - startPoint.Y);
            if (startPoint.X <= endPoint.X && startPoint.Y <= endPoint.Y)
            {
                x = startPoint.X;
                y = startPoint.Y;
            }
            else if (startPoint.X >= endPoint.X && startPoint.Y >= endPoint.Y)
            {
                x = endPoint.X;
                y = endPoint.Y;
            }
            else if (startPoint.X >= endPoint.X && startPoint.Y <= endPoint.Y)
            {
                x = endPoint.X;
                y = startPoint.Y;
            }
            else if (startPoint.X <= endPoint.X && startPoint.Y >= endPoint.Y)
            {
                x = startPoint.X;
                y = endPoint.Y;
            }
            StartPoint = new PointF(x, y);
            var side = Math.Max(width, height);
            EndPoint = new PointF(x + side, y + side);
            ShapeType = ShapeType.Circle;
            Zindex = zIndex;
            ForeColor = foreColor;
            Thickness = thickness;
            BackgroundColor = backgroundColor;
            IsFill = isFill;
        }

        /// <summary>
        /// Initializes a new instance of the <see cref="Circle" /> class.
        /// </summary>
        public Circle()
        {
            ShapeType = ShapeType.Circle;
        }

#endregion Constructors

#region Methods (1)

// Public Methods (1)

        /// <summary>
        /// Points the in sahpe.
```

```
        /// </summary>
        /// <param name="point">The point.</param>
        /// <param name="tolerance">The tolerance.</param>
        /// <returns>
        ///   <c>true</c> if [has point in sahpe] [the specified point]; otherwise, <c>false</c>.
        /// </returns>
        public override bool HasPointInSahpe(PointF point, byte tolerance = 5)
        {
            float width = Math.Abs(EndPoint.X+tolerance - StartPoint.X-tolerance);
            float height = Math.Abs(EndPoint.Y+tolerance - StartPoint.Y-tolerance);
            float diagonal = Math.Max(height, width);
            float raduis = diagonal / 2;
            float dx = Math.Abs(point.X - (X + Width / 2));
            float dy = Math.Abs(point.Y - (Y + height / 2));
            return (dx + dy <= raduis);
        }

#endregion Methods
    }
}
```

این کلاس شامل دو سازنده می‌باشد، که در سازنده اول با توجه به نقاط ایتدا و انتهای ترسیم شکل مقدار طول و عرض مستطیل احاطه کننده دایره محاسبه شده و باتوجه به آنها بزرگترین ضلع به عنوان قطر دایره در نظر گرفته می‌شود و EndPoint شکل مورد نظر تعیین می‌شود.

در متد **HasPointInShape** با استفاده از فرمول دایره تعیین می‌شود که آیا نقطه پارامتر ورودی متد در داخل دایره واقع شده است یا خیر (جهت انتخاب شکل برای جابجایی یا تغییر اندازه).
در پست‌های بعد به پیاده سازی اینترفیس نرم افزار خواهیم پرداخت.

موفق و موید باشید

در ادامه مطالب قبل:
پیاده سازی پروژه نقاشی (Paint) به صورت شی گرا #1
پیاده سازی پروژه نقاشی (Paint) به صورت شی گرا #2
پیاده سازی پروژه نقاشی (Paint) به صورت شی گرا #3
پیاده سازی پروژه نقاشی (Paint) به صورت شی گرا #4
پیاده سازی پروژه نقاشی (Paint) به صورت شی گرا #5

نظرات خوانندگان

نویسنده: بتیسا
تاریخ: ۱۳۹۱/۱۲/۰۷ ۹:۸

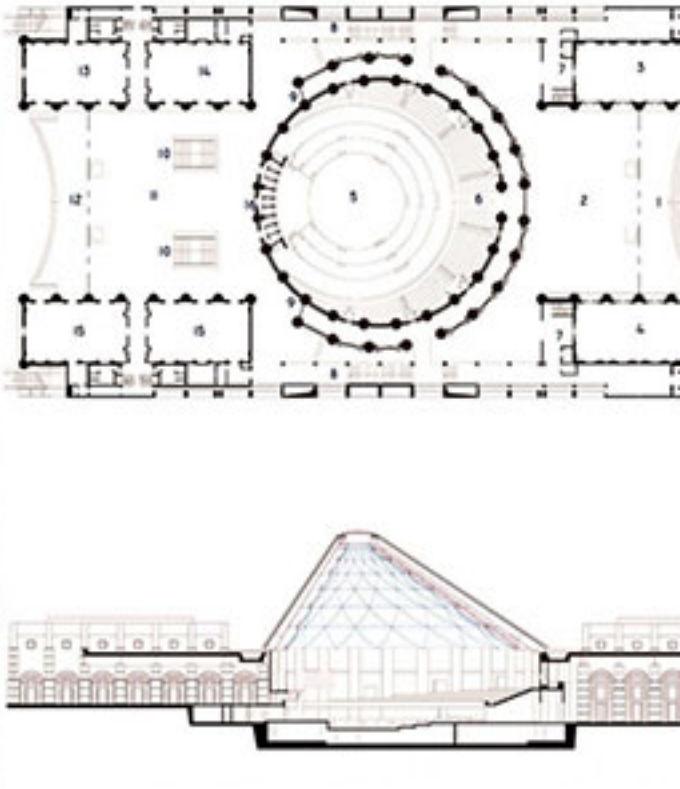
با سلام برای پیدا کردن نقطه در بیضی من چند لینک پیدا کردم امیدوارم که به کارتون بیاد
لینک اول از ویکی پدیا
لینک دوم از stackoverflow
لینک سوم mathforum
لینک چهارم mathopenref

عنوان: **مفاهیم برنامه نویسی ـ مروری بر کلاس و شیء**
نویسنده: آرمان فرقانی
تاریخ: ۱۳۹۲/۰۱/۱۹ ۲۱:۳۵
آدرس: www.dotnettips.info
برچسب‌ها: C#, OOP, Programming, آموزش برنامه نویسی

من قصد دارم در قالب چند مطلب برخی از مفاهیم پایه و مهم برنامه نویسی را که پیش نیازی برای درک اکثر مطالب موجود در وب سایت است به زبان ساده بیان کنم تا دایره افرادی که می‌توانند از مطالب ارزشمند این وب سایت استفاده کنند وسعت بیشتری پیدا کند. لازم به توضیح است از آنجا که علاقه ندارم اینجا تبدیل به نسخه فارسی MSDN یا کتاب آنلاین آموزش برنامه نویسی شود این سری آموزش‌ها بیشتر شامل مفاهیم کلیدی خواهند بود.
این مطلب به عنوان اولین بخش از این سری مطالب منتشر می‌شود.

هدف این نوشته بررسی جزییات برنامه نویسی در رابطه با کلاس و شیء نیست. بلکه دریافتن چگونگی شکل گرفتن ایده شیء گرایی و علت مفید بودن آن است.

مشاهده مفاهیم شیء گرایی در پیرامون خود حتماً در دنیای برنامه نویسی شیء گرا بارها با کلمات کلاس و شیء روبرو شده اید. درک صحیح از این مفاهیم بسیار مهم و البته بسیار ساده است. کار را با یک مثال شروع می‌کنیم. به تصویر زیر نگاه کنید.



در سمت راست بخشی از نقشه یک ساختمان و در سمت چپ ساختمان ساخته شده بر اساس این نقشه را می‌بینید. ساختمان همان شیء است. و نقشه ساختمان کلاس آن است چراکه امکان ایجاد اشیائی که تحت عنوان ساختمان طبقه بندی (کلاس بندی) می‌شوند را فراهم می‌کند. به همین سادگی. کلاس‌ها طرح اولیه، نقشه یا قالبی هستند که جزییات یک شی را توصیف می‌کنند.
حتماً با من موافق هستید اگر بگویم:
در نقشه ساختمان نمی‌توانید زندگی کنید اما در خود ساختمان می‌توانید.
از روی یک نقشه می‌توان به تعداد دلخواه ساختمان ساخت.
هنگامی که در یک ساختمان زندگی می‌کنید نیازی نیست تا دقیقاً بدانید چگونه ساخته شده و مثلاً سیم کشی یا لوله کشی‌های آن

چگونه است! تنها کافیست بدانید برای روشن شدن لامپ باید کلید آن را بزنید.
ساختمان دارای ویژگی هایی مانند متراژ، ضخامت دیوار، تعداد پنجره و ابعاد هر یک و ... است که در هنگام ساخت و بر اساس اطلاعات موجود در نقشه تعیین شده اند.
ساختمان دارای کارکرد هایی است. مانند بالا و پایین رفتن آسانسور و یا باز و بسته شدن درب پارکینگ. هر یک از این کارکردها نیز بر اساس اطلاعات موجود در نقشه پیاده سازی و ساخته شده اند.
ساختمان تمام اجزای لازم برای اینکه از آن بتوانیم استفاده کنیم و به عبارتی در آن بتوانیم زندگی کنیم را در خود دارد.
در محیط پیرامون ما تقریباً هر چیزی را می‌توان در یک دیدگاه شیء تصور کرد. به عبارتی هر چیزی که بتوانید به صورت مستقل در ذهن بیاورید و سپس برخی ویژگی‌ها و رفتارها یا کارکردهای آن‌را برشمارید تا آن چیز را قابل شناسایی کند شیء است. مثلاً من به شما می‌گویم موجودی چهار پا دارد، مو... مو... می‌کند و شیر می‌دهد و ... . شما خواهید گفت گاو! و نمی‌گویید گربه. چرا؟ چون توانستید در ذهن خود موجودیتی را به صورت مستقل تصور کنید و از روی ویژگی‌ها و رفتارش آن‌را دقیقاً شناسایی کنید.
سوال: کلاس یا نقشه ایجاد گاو چیست؟ اگر از من بپرسید خواهم گفت طرح اولیه گاو هم ممکن است وجود داشته باشد البته در اختیار خداوند و با سطح دسترسی ملکوت!
اتومبیل، تلویزیون و ... همگی مثال هایی از اشیاء پیرامون ما در دنیای واقعی هستند که حتماً می‌توانید کلاس یا نقشه ایجاد آن‌ها را نیز بدست آورید و یا ویژگی‌ها و کارکردهای آن‌ها را برشمارید.

مفاهیم شیء گرایی در مهندسی نرم افزار مفاهیمی که تاکنون در مورد دنیای واقعی مرور کردیم همان چیزی است که در دنیای برنامه نویسی ـ به عقیده من دنیای واقعی‌تر از دنیای واقعی ـ با آن سر و کار داریم. علت این امر آن است که اصولاً ایده روش برنامه نویسی شیء گرا با مشاهده محیط پیرامون ما به وجود آمده است.
برای نوشتن برنامه جهت حل یک مسئله بزرگ باید بتوان آن مسئله را به بخش‌های کوچکتری تقسیم نمود. در این رابطه مفهوم شیء و کلاس با همان کیفیتی که در محیط پیرامون ما وجود دارد به صورت مناسبی امکان تقسیم یه مسئله بزرگ به بخش‌های کوچکتر را فراهم می‌کند. و سبب می‌شود هماهنگی و تقارن و تناظر خاصی بین اشیاء برنامه و دنیای واقعی بوجود آید که یکی از مزایای اصلی روش شیء گراست.
از آنجا که در یک برنامه اصولاً همه چیز و همه مفاهیم در قالب کدها و دستورات برنامه معنا دارد، کلاس و شیء نیز چیزی بیش از قطعاتی کد نیستند. قطعه کد هایی که بسته بندی شده اند تا تمام کار مربوط به هدفی که برای آن‌ها در نظر گرفته شده است را انجام دهند.
همان طور که در هر زبان برنامه نویسی دستوراتی برای کارهای مختلف مانند تعریف یک متغیر یا ایجاد یک حلقه و ... در نظر گرفته شده است، در زبان‌های برنامه نویسی شیء گرا نیز دستوراتی وجود دارد تا بتوان قطعه کدی را بر اساس مفهوم کلاس بسته بندی کرد.
به طور مثال قطعه کد زیر را در زبان برنامه نویسی سی شارپ در نظر بگیرید.

```
class Player
{
   public string Name;
   public int Age;
   public void Walk()
   {
      // کدهای مربوط به پیاده سازی راه رفتن
   }
   public void Run()
   {
      // کدهای مربوط به پیاده سازی دویدن
   }
}
```

در این قطعه کد با استفاده از کلمه کلیدی class در زبان سی شارپ کلاسی ایجاد شده است که دارای دو ویژگی نام و سن و دو رفتار راه رفتن و دویدن است.
این کلاس به چه دردی می‌خورد؟ کجا می‌توانیم از این کلاس استفاده کنیم؟
پاسخ این است که این کلاس ممکن است برای ما هیچ سودی نداشته باشد و هیچ کجا نتوانیم از آن استفاده کنیم. اما بیایید فرض کنیم برنامه نویسی هستیم که قصد داریم یک بازی فوتبال بنویسیم. به جای آنکه قطعات کد مجزایی برای هر یک از بازیکنان و کنترل رفتار و ویژگی‌های آنان بنویسیم با اندکی تفکر به این نکته پی می‌بریم که همه بازیکنان مشترکات بسیاری دارند و به عبارتی در یک گروه یا کلاس قابل دسته بندی هستند. پس سعی می‌کنیم نقشه یا قالبی برای بازیکن‌ها ایجاد کنیم که دربردارنده ویژگی‌ها و

رفتارهای آن‌ها باشد.
همان طور که در نقشه ساختمان نمی‌توانیم زندگی کنیم این کلاس هم هنوز آماده انجام کارهای واقعی نیست. چراکه برخی مقادیر هنوز برای آن تنظیم نشده است. مانند نام بازیکن و سن و ....
و همان طور که برای سکونت لازم است ابتدا یک ساختمان از روی نقشه ساختمان بسازیم برای استفاده واقعی از کلاس یاد شده نیز باید از روی آن شیء بسازیم. به این فرآیند وهله سازی یا نمونه سازی نیز می‌گویند. یک زبان برنامه نویسی شیء گرا دستوراتی را برای وهله سازی نیز در نظر گرفته است. در C# کلمه کلیدی new این وظیفه را به عهده دارد.

```
Player objPlayer = new Player();
objPlayer.Name = “Ali Karimi”;
objPlayer.Age = 30;
objPlayer.Run();
```

وقتی فرآیند وهله سازی صورت می‌گیرد یک نمونه یا شیء از آن کلاس در حافظه ساخته می‌شود که در حقیقت می‌توانید آنرا همان کدهای کلاس تصور کنید با این تفاوت که مقداردهی‌های لازم صورت گرفته است. به دلیل تعیین مقادیر لازم، حال شیء تولید شده می‌تواند به خوبی اعمال پیش بینی شده را انجام دهد. توجه نمایید در اینجا پیاده سازی داخلی رفتار دویدن و اینکه مثلاً در هنگام فراخوانی آن چه کدی باید اجرا شود تا تصویر یک بازیکن در حال دویدن در بازی نمایش یابد مد نظر و موضوع بحث ما نیست. بحث ما چگونگی سازماندهی کدها توسط مفهوم کلاس و شیء است. همان طور که مشاهده می‌کنید ما تمام جزییات بازیکن‌ها را یکبار در کلاس پیاده سازی کرده ایم اما به تعداد دلخواه می‌توانیم از روی آن بازیکن‌های مختلف را ایجاد کنیم. همچنین به راحتی رفتار دویدن یک بازیکن را فراخوانی میکنیم بدون آنکه پیاده سازی کامل آن در اختیار و جلوی چشم ما باشد.
تمام آنچه که بازیکن برای انجام امور مربوط به خود نیاز دارد در کلاس بازیکن کپسوله می‌شود. بدیهی است در یک برنامه واقعی ویژگی‌ها و رفتارهای بسیار بیشتری باید برای کلاس بازیکن در نظر گرفته شود. مانند پاس دادن، شوت زدن و غیره.
به این ترتیب ما برای هر برنامه می‌توانیم مسئله اصلی را به تعدادی مسئله کوچکتر تقسیم کنیم و وظیفه حل هر یک از مسائل کوچک را به یک شیء واگذار کنیم. و بر اساس اشیاء تشخیص داده شده کلاس‌های مربوطه را بنویسیم. برنامه نویسی شیء گرا سبب می‌شود تا مسئله توسط تعدادی شیء که دارای نمونه‌های متناظری در دنیای واقعی هستند حل شود که این امر زیبایی و خوانایی و قابلیت نگهداری و توسعه برنامه را بهبود می‌دهد.
احتمالاً تاکنون متوجه شده اید که برای نگهداری ویژگی‌های اشیاء از متغیرها و برای پیاده سازی رفتارها یا کارکردهای اشیاء از توابع استفاده میکنیم.
با توجه به این که هدف این مطلب بررسی مفهوم شیء گرائی بود و نه جزییات برنامه نویسی، بنابراین بیان برخی مفاهیم در این رابطه را که بیشتر در مهندسی نرم افزار معنا دارند تا در دنیای واقعی در مطالب بعدی بررسی می‌کنیم.

نظرات خوانندگان

نویسنده: Hesam
تاریخ: ۱۳۹۲/۰۱/۱۹ ۲۲:۴۵

بسیار عالی :)

نویسنده: من
تاریخ: ۱۳۹۲/۰۱/۲۰ ۱۴:۵۶

شروع خوبی بود، آفرین!
من هنوز وقتی جلسه‌ی اول کلاس میپرسم که پراید یک کلاس هست و یا یک آبجکت. همه میگن آبجکتی از کلاس ماشین!
این در حالیست که خیلی از این افراد سالهاست برنامه نویسی میکنند و هنوز نمیدونند heap یا stack چیه

نویسنده: علي
تاریخ: ۱۳۹۲/۰۱/۲۰ ۱۵:۴

حالا با توجه به توضیحات بالا که گفته شد « *در نقشه ساختمان نمی‌توانید زندگی کنید اما در خود ساختمان می‌توانید.* » با یک پراید می‌شود رانندگی کرد اما با ماشین که بیشتر یک مفهوم است خیر. نه؟

نویسنده: آرمان فرقانی
تاریخ: ۱۳۹۲/۰۱/۲۰ ۱۵:۲۲

تشکر از نظر شما. متوجه صحبت شما هستم ولی این توضیح برای دوستان دیگه می‌تونه مفید باشه.
پراید به عنوان رده ای از اتومبیل‌ها کلاس است. اما اگر به کسی یک اتومبیل پراید در حال عبور را نشان دهید که دارای پلاک و ... است، آن شیء ای است از کلاس پراید. اگر در یک پارکینگ تعدادی اتومبیل باشد و از کسی بخواهیم بر اساس نوع گروه بندی یا کلاس بندی کند، بعد از چند دقیقه خواهیم دید اتومبیل‌های هم نوع را جدا کرده. مثلا همه اتومبیل‌های از نوع پراید را کنار هم قرار داده. این همان مفهوم کلاس بندی است. برای تولید اتومبیلی از نوع پراید کارخانه یک نقشه یا طرحی ایجاد می‌کند که بسیاری مشخصات و چگونگی ساخت را در خود دارد. چگونگی انجام برخی کارکردها مانند حرکت را دارد. این طرح کلاس نامیده می‌شود. اما آیا می‌توان برای آن پلاک در نظر گرفت؟ خیر چون فقط نقشه ایجاد اتومبیل است (یا به عبارتی فقط مفهوم است). همچنین کلاس پراید احتمالاً از کلاس اتومبیل یا ماشین برخی خصوصیات و رفتارها را با ارث برده است.

بیاد داشته باشیم هر فردی در هر سطحی از دانش هم مسلماً بسیاری چیزها را هنوز نمی‌داند و دانسته‌های ما در مقابل نادانسته‌ها قطره ای بیش نیست. تا جایی که می‌توان گفت همه ما از نظر نادانسته‌ها برابریم. تفاوت در دانسته هاست. کسانی که سال‌ها برنامه نویسی می‌کنند هم حتما دانسته هایی دارند که می‌توانند این کار را ادامه دهند. پس کافی است آنچه نمی‌دانند را سعی کنیم به اشتراک بگذاریم تا بدانند و از دانسته هایشان استفاده کنیم.

نویسنده: آرمان فرقانی
تاریخ: ۱۳۹۲/۰۱/۲۰ ۱۵:۲۵

جمله با پراید می‌شود رانندگی کرد ابهام دارد. ابهام آن به این صورت رفع می‌شود که من می‌دانم منظور شما از پراید به عنوان یک اسم عام و یک مفهوم و **نام رده یا کلاسی از اتومبیل ها** نیست. بلکه منظور شما با اتومبیل پرایدی است که دارای یک پلاک مشخص است و مثلاً کسی به تازگی گنجی پیدا کرده و رفته یک پراید خریده! آن اتومبیل پراید مشخص یک شیء است از کلاس پراید. بله با آن شی می‌توان رانندگی کرد. اما با مفهوم یا نقشه یا کلاس یا رده یا گروه یا طرح تولید خودروی پراید یا هر ماشین دیگری نمی‌توان رانندگی کرد.

نویسنده: سید ایوب کوکبی
تاریخ: ۱۳۹۲/۰۱/۲۱ ۱۵:۴۸

ممنون، خیلی خوب بررسی کرده بودید و مثالهایی که هم ارائه کردید به دلم نشست، امیدوارم برای سایر مباحث هم به همین صورت ادامه بدید. البته به نظر بنده نیازی هم نیست خیلی روی این مبحث تناظر بین دنیای واقعی و دنیای شی گرا حساس باشیم، چون اگر به همین نحو بخواهیم این دو را با هم مقایسه کنیم شاید بعضی جاها با مشکل مواجه بشیم. این تناظر فقط برای اینه که دید و تجسمی از این مفهوم داشته باشیم تا بهتر بتوانیم آن را بپذیریم.
یادمون نره که هدف ما یادگیری مفهوم شی گرایی است نه یادگیری تناظر آن با دنیای واقعی، این تناظر فقط یک ابزاره برای دسترسی سریعتر به این هدف!

نویسنده: آرمان فرقانی
تاریخ: ۱۳۹۲/۰۱/۲۱ ۱۷:۳۵

سلام و ممنون از نظر شما.
اتفاق جالبی افتاد و آن این بود که هم اکنون داشتم در OneNote بخشی برای مطلب بعدی می‌نوشتم. دقیقاً داشتم پاراگرافی را می‌نوشتم که جلوی این که ذهن خواننده به سمتی برود که گویی "الزاماً هر مورد در مهندسی نرم افزار را باید پس از یافتن مصداق آن در محیط اطراف یاد گرفت" را بگیرم.
دقیقاً صحیح است. تاکید بر این تناظر در این بخش به دلیل یافتن درک عمیق‌تر از شیء گرایی و علت مفید بودن آن و چگونگی شکل گیری ایده آن است. این درک عمیق‌تر امکان استفاده بهتر و صحیح‌تر این مفاهیم در برنامه را فراهم می‌کند. و سبب می‌شود برنامه نویس شیءگرایی را ابزاری برای حل مسئله بیاد نه راه و روشی که همه میگن خوبه پس باید رعایت کرد. حال آنکه چون درک دقیقی از آن ندارد در حقیقت مسئله را با آن روشی که بهتر بلد است حل می‌کنند و فقط تعدادی کلاس و شیء در برنامه وجود دارد.

نویسنده: آریانا
تاریخ: ۱۳۹۲/۰۲/۰۷ ۱۱:۵۶

بسیار بسیار عالی بود مرسی

نویسنده: سحابی
تاریخ: ۱۳۹۲/۰۷/۱۴ ۱۴:۵۹

سلام
برای یادگیری Desing pattern منابعی وجود دارد ؟ لطفا در صورت امکان اگر منابع و یا لینک کارآمدی معرفی کنید .

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۰۷/۱۴ ۱۷:۱۵

برچسب‌های سایت رو مرور کنید به اندازه کافی در این زمینه مطلب هست. مثلا اینجا

عنوان: **مفاهیم برنامه نویسی ـ مروری بر فیلدها، متدها و ساخت اشیاء**
نویسنده: آرمان فرقانی
تاریخ: ۱۳۹۲/۰۱/۲۳ ۲۰:۱۰
آدرس: www.dotnettips.info
برچسب‌ها: C#, OOP, Programming, آموزش برنامه نویسی

شکستن یک مسئله بزرگ به تعدادی مسئله کوچک‌تر راهکار موثری برای حل آن است. این امر در برنامه نویسی نیز که هدف آن چیزی جز حل یک مسئله نیست همواره مورد توجه بوده است. به همین دلیل روش هایی که به کمک آن‌ها بتوان یک برنامه بزرگ را به قطعات کوچکتری تقسیم کرد تا هر قطعه کد مسئول انجام کار خاصی باشد پیشتر به زبان‌های برنامه نویسی اضافه شده اند. یکی از این ساختارها تابع (Function) نام دارد. برنامه ای که از توابع برای تقسیم کدهای برنامه استفاده می‌کند یک برنامه ساخت‌یافته می‌گوییم.

در مطلب پیشین به پیرامون خود نگاه کردیم و اشیاء گوناگونی را مشاهده کردیم که در حقیقت دنیای ما را تشکیل داده اند و فعالیت‌های روزمره ما با استفاده از آن‌ها صورت می‌گیرد. ایده ای به ذهنمان رسید. اشیاء و مفاهیم مرتبط به آن می‌تواند روش بهتر و موثرتری برای تقسیم کدهای برنامه باشد. مثلاً اگر کل کدهای برنامه که مسئول حل یکی از مسئله‌های کوچک یاد شده است را یکجا بسته بندی کنیم و اصولی که از اشیاء واقعی پیرامون خود آموختیم را در مورد آن رعایت کنیم به برنامه بسیار با کیفیت‌تری از نظر خوانایی، راحتی در توسعه، اشکال زدایی ساده‌تر و بسیاری موارد دیگر خواهیم رسید.

توسعه دهندگان زبان‌های برنامه نویسی که با ما در این مورد هم عقیده بوده اند دست به کار شده و دستورات و ساختارهای لازم برای پیاده کردن این ایده را در زبان برنامه نویسی قرار دادند و آن را زبان برنامه نویسی شیء گرا نامیدند. حتی جهت برخورداری از قابلیت استفاده مجدد از کد و موارد دیگر به جای آنکه کدها را در بسته هایی به عنوان یک شیء خاص قرار دهیم آن‌ها را در بسته هایی به عنوان قالب یا نقشه ساخت اشیاء خاصی که در ذهن داریم قرار می‌دهیم. یعنی مفهوم کلاس یا رده که پیشتر اشاره شد. به این ترتیب یک بار می‌نویسیم و بارها استفاده می‌کنیم. مانند همان مثال بازیکن در بخش نخست . هر زمان که لازم باشد با استفاده از دستورات مربوطه از روی کدهای کلاس که نقشه یا قالب ساخت اشیاء هستند شیء مورد نظر را ساخته و در جهت حل مسئله مورد نظر به کار می‌بریم.

حال برای آنکه به طور عملی بتوانیم از ایده شیء گرایی در برنامه هایمان استفاده کنیم و مسائل بزرگ را حل کنیم لازم است ابتدا مقداری با جزییات و دستورات زبان در این مورد آشنا شویم.

تذکر : دقت کنید برای آنکه از ایده شیء گرایی در برنامه‌ها حداکثر استفاده را ببریم مفاهیمی در مهندسی نرم افزار به آن اضافه شده است که ممکن است در دنیای واقعی نیازی به طرح آن‌ها نباشد. پس لطفاً تلاش نکنید با دیدن هر مفهوم تازه بلافاصله سعی در تطبیق آن با محیط اطراف کنید. هر چند بسیاری از آن‌ها به طور ضمنی در اشیاء پیرامون ما نیز وجود دارند.

زبان برنامه نویسی مورد استفاده برای بیان مفاهیم برنامه نویسی در این سری مقالات زبان سی شارپ است. اما درک برنامه‌های نوشته شده برای علاقه مندان به زبان‌های دیگری مانند وی بی دات نت نیز دشوار نیست. چراکه اکثر دستورات مشابه است و تبدیل Syntax نیز به راحتی با اندکی جستجو میسر می‌باشد. لازم به یادآوری است زبان سی شارپ به بزرگی یا کوچکی حروف حساس است.

تشخیص و تعریف کلاس‌های برنامه کار را با یک مثال شروع می‌کنیم. فرض کنید به عنوان بخشی از راه حل یک مسئله بزرگ، لازم است محیط و مساحت یک سری چهارضلعی را محاسبه کنیم و قصد داریم این وظیفه را به طور کامل بر عهده قطعه کدهای مستقلی در برنامه قرار دهیم. به عبارت دیگر قصد داریم متناظر با هر یک از چهارضلعی‌های موجود در مسئله یک شیء در برنامه داشته باشیم که قادر است محیط و مساحت خود را محاسبه و ارائه نماید. کلاس زیر که با زبان سی شارپ نوشته شده امکان ایجاد اشیاء مورد نظر را فراهم می‌کند.

```
public class Rectangle
{
   public double Width;
   public double Height;

   public double Area()
   {
      return Width*Height;
   }

   public double Perimeter()
   {
      return 2*(Width + Height);
   }
```

```
}
```

در این قطعه برنامه نکات زیر قابل توجه است:
کلاس با کلمه کلیدی class تعریف می‌شود.
همان طور که مشاهده می‌کنید تعریف کلاس با کلمه public آغاز شده است. این کلمه محدوده دسترسی به کلاس را تعیین می‌کند. در اینجا از کلمه public استفاده کردیم تا بخش‌های دیگر برنامه امکان استفاده از این کلاس را داشته باشند.
پس از کلمه کلیدی class نوبت به نام کلاس می‌رسد. اگرچه انتخاب نام مورد نظر امری اختیاری است اما در آینده حتماً اصول و قراردادهای نام‌گذاری در دات‌نت را مطالعه نمایید. در حال حاضر حداقل به خاطر داشته باشید تا انتخاب نامی مناسب که گویای کاربرد کلاس باشد بسیار مهم است.
باقیمانده کد، بدنه کلاس را تشکیل می‌دهد. جاییکه ویژگی ها، رفتارها و ... یا به طور کلی اعضای کلاس تعریف می‌شوند.
نکته : کلماتی مانند public که پیش از تعریف کلاس یا اعضای آن قرار می‌گیرند Modifier یا پیرایندە نام دارند. که البته به نظر من ترجمه این گونه واژه‌ها از کارهای شیطان است. بنابراین از این پس بهتر است همان Modifier را به خاطر داشته باشید. از آنجا که public مدیفایری است که سطح دسترسی را تعیین می‌کند به آن یک Access Modifier می‌گویند. در یک بخش از این سری مقالات تمامی مدیفایرها بررسی خواهند شد.

ایجاد شیء از یک کلاس و نحوه دسترسی به شیء ایجاد شده شیء و کلاس چیزهای متفاوتی هستند. یک کلاس نوع یک شیء را تعریف می‌کند. اما یک شیء یک موجودیت عینی و واقعی بر اساس یک کلاس است. در اصطلاح از شیء به عنوان یک نمونه (Instance) یا وهله ای از کلاس مربوطه یاد می‌کنیم. همچنین به عمل ساخت شیء نمونه سازی یا وهله سازی گوییم.
برای ایجاد شیء از کلمه کلیدی new و به دنبال آن نام کلاسی که قصد داریم بر اساس آن یک شیء بسازیم استفاده می‌کنیم. همان طور که اشاره شد کلاس یک نوع را تعریف می‌کند. پس از آن می‌توان همانند سایر انواع مانند int, string, ... برای تعریف متغیر استفاده نمود. به مثال زیر توجه کنید.

```
Rectangle rectangle = new Rectangle();
```

در این مثال rectangle که با حرف کوچک شروع شده و می‌توانست هر نام دلخواه دیگری باشد ارجاعی به شیء ساخته شده را به دست می‌دهد. وقتی نمونه ای از یک کلاس ایجاد می‌شود یک ارجاع به شیء تازه ساخته شده برای برنامه نویس برگشت داده می‌شود. در این مثال rectangle یک ارجاع به شیء تازه ساخته شده است یعنی به آن اشاره می‌کند اما خودش شامل داده‌های آن شیء نیست. تصور کنید این ارجاع تنها دستگیره ای برای شیء ساخته شده است که دسترسی به آن را برای برنامه نویس میسر می‌کند. درک این مطلب از این جهت دارای اهمیت است که بدانید می‌شود یک دستگیره یا ارجاع دیگر بسازید بدون آنکه شیء جدیدی تولید کنید.

```
Rectangle rectangle;
```

البته توصیه نمی‌کنم چنین ارجاعی را تعریف کنید چرا که به هیچ شیء خاصی اشاره نمی‌کند. و تلاش برای استفاده از آن منجر به بروز خطای معروفی در برنامه خواهد شد. به هر حال یک ارجاع می‌توان ساخت چه با ایجاد یک شیء جدید و یا با نسبت دادن یک شیء موجود به آن.

```
Rectangle rectangle1 = new Rectangle();
Rectangle rectangle2 = rectangle1;
```

در این کد دو ارجاع یا دستگیره ایجاد شده است که هر دو به یک شیء اشاره می‌کنند. بنابراین ما با استفاده از هر دو ارجاع می‌توانیم به همان شیء واحد دسترسی پیدا کنیم و اگر مثلاً با rectangle1 در شیء مورد نظر تغییری بدهیم و سپس با rectangle2 شیء را مورد بررسی قرار دهیم تغییرات داده شده قابل مشاهده خواهد بود چون هر دو ارجاع به یک شیء اشاره می‌کنند. به همین دلیل کلاس‌ها را به عنوان نوع ارجاعی می‌شناسیم در مقایسه با انواع داده دیگری که اصطلاحاً نوع مقداری هستند.
حالا می‌توان شیء ساخته شده را با استفاده از ارجاعی که به آن داریم به کار برد.

```
Rectangle rectangle = new Rectangle();
rectangle.Width = 10.5;
rectangle.Height = 10;
double a = rectangle.Area();
```

ابتدا عرض و ارتفاع شیء چهارضلعی را مقدار دهی کرده و سپس مساحت را دریافت کرده ایم. از نقطه برای دسترسی به اعضای یک شیء استفاده می‌شود.

فیلدها اگر به تعریف کلاس دقت کنید مشخص است که دو متغییر Width و Height را با سطح دسترسی عمومی تعریف کرده ایم. به متغیرهایی از هر نوع که مستقیماً درون کلاس تعریف شوند (و نه مثلاً داخل یک تابع درون کلاس) فیلد می‌گوییم. فیلدها از اعضای کلاس دربردارنده آن‌ها محسوب می‌شوند.
تعریف فیلدها مستقیماً در بدنه کلاس با یک Access Modifier شروع می‌شود و به دنبال آن نوع فیلد و سپس نام دلخواه برای فیلد می‌آید.
تذکر : نامگذاری مناسب یکی از مهمترین اصولی است که یک برنامه نویس باید همواره به آن توجه کافی داشته باشد و به شدت در بالا رفتن کیفیت برنامه موثر است. به خاطر داشته باشید تنها اجرا شدن و کار کردن یک برنامه کافی نیست. رعایت بسیاری از اصول مهندسی نرم افزار که ممکن است نقش مستقیمی در کارکرد برنامه نداشته باشند موجب سهولت در نگهداری و توسعه برنامه شده و به همان اندازه کارکرد صحیح برنامه مهم هستند. بنابراین مجدداً شما را دعوت به خواندن مقاله یاد شده بالا در مورد اصول نامگذاری صحیح می‌کنم. هر مفهوم تازه ای که می‌آموزید می‌توانید به اصول نامگذاری همان مورد در مقاله پیش گفته مراجعه نمایید. همچنین افزونه هایی برای Visual Studio وجود دارد که شما را در زمینه نامگذاری صحیح و بسیاری موارد دیگر هدایت می‌کنند که یکی از مهمترین آن‌ها Resharper نام دارد.
مثال:

```
// public field (Generally not recommended.)
public double Width;
```

همان طور که در این قطعه کد به عنوان توضیح درج شده است استفاده از فیلدهایی با دسترسی عمومی توصیه نمی‌شود. علت آن واضح است. چون هیچ کنترلی برای مقداری که برای آن در نظر گرفته می‌شود نداریم. به عنوان مثال امکان دارد یک مقدار منفی برای عرض یا ارتفاع شیء درج شود حال آنکه می‌دانیم عرض یا ارتفاع منفی معنا ندارد. در قسمت بعدی این سری مقالات این مشکل را بررسی و حل خواهیم نمود.
فیلدها معمولاً با سطح دسترسی خصوصی و برای نگهداری از داده‌هایی که مورد نیاز بیش از یک متد (یا تابع) درون کلاس است و آن داده‌ها باید پس از خاتمه کار یک متد همچنان باقی بمانند استفاده می‌شود. بدیهی است در غیر اینصورت به جای تعریف فیلد می‌توان از متغیرهای محلی (متغیری که درون خود تابع تعریف می‌شود) استفاده نمود.
همان طور که پیشتر اشاره شد برای دسترسی به یک فیلد ابتدا یک نقطه پس از نام شیء درج کرده و سپس نام فیلد مورد نظر را می‌نویسیم.

```
Rectangle rectangle = new Rectangle();
rectangle.Width = 10.5;
```

در هنگام تعریف یک فیلد در صورت نیاز می‌توان برای آن یک مقدار اولیه را در نظر گرفت. مانند مثال زیر:

```
public class Rectangle
{
   public double Width = 5;
   // ...
}
```

متدها متدها قطعه کدهایی شامل یک سری دستور هستند. این مجموعه دستورات با فراخوانی متد و تعیین آرگومان‌های مورد نیاز اجرا می‌شوند. در زبان سی شارپ به نوعی تمام دستورات در داخل متدها اجرا می‌شوند. در این زبان تمامی توابع در داخل کلاس‌ها تعریف می‌شوند و بنابراین همه متد هستند.

متدها نیز مانند فیلدها در داخل کلاس تعریف می‌شوند. ابتدا یک Access Modifier سطح دسترسی را تعیین می‌نماید. سپس به ترتیب نوع خروجی، نام متد و لیست پارامترهای آن در صورت وجود درج می‌شود. به مجموعه بخش‌های یاد شده امضای متد می‌گویند.
پارامترهای یک متد داخل یک جفت پرانتز قرار می‌گیرند و با کاما (,) از هم جدا می‌شوند. یک جفت پرانتز خالی نشان دهنده آن است که متد نیاز به هیچ پارامتری ندارد.
بار دیگر به بخش تعریف متدهای کلاسی که ایجاد کردیم توجه نمایید.

```
public class Rectangle
{
   // ...

   public double Area()
   {
      return Width*Height;
   }

   public double Perimeter()
   {
      return 2*(Width + Height);
   }
}
```

در این کلاس دو متد به نام‌های Area و Perimeter به ترتیب برای محاسبه مساحت و محیط چهارضلعی تعریف شده است.
همانطور که پیشتر اشاره شد متدها برای پیاده سازی رفتار اشیاء یا همان کارکردهای آن‌ها استفاده می‌شوند. در این مثال شیء ما قادر است مساحت و محیط خود را محاسبه نماید. چه شیء خوش رفتاری!
همچنین توجه نمایید این شیء برای محاسبه مساحت و محیط خود نگاهی به ویژگی‌های خود یعنی عرض و ارتفاعش که در فیلدهای آن نگهداری می‌کنیم می‌اندازد.
فراخوانی متد یک شیء همانند دسترسی به فیلد آن است. ابتدا نام شیء سپس یک نقطه و به دنبال آن نام متد مورد نظر به همراه پرانترها. آرگومان‌های مورد نیاز در صورت وجود داخل پرانتزها قرار می‌گیرند و با کاما از هم جدا می‌شوند. که البته در این مثال متد ما نیازی به آرگومان ندارد. به همین دلیل برای فراخوانی آن تنها یک جفت پرانتز خالی قرار می‌دهیم.

در این بخش به دو مفهوم پارامتر و آرگومان اشاره شد. تفاوت آن‌ها چیست؟

در هنگام تعریف یک متد نام و نوع پارامترهای مورد نیاز را تعیین و درج می‌نماییم. حال وقتی قصد فراخوانی متد را داریم باید مقادیر واقعی که آرگومان نامیده می‌شود را برای هر یک از پارامترهای تعریف شده فراهم نماییم. نوع آرگومان باید با نوع پارامتر تعریف شده تطبیق داشته باشد. اما اگر یک متغیر را به عنوان آرگومان در هنگام فراخوانی متد استفاده می‌کنیم نیازی به یکسان بودن نام آن متغیر و نام پارامتر تعریف شده نیست.
متدها می‌توانند یک مقدار را به کدی که آن متد را فراخوانی کرده است بازگشت دهند.

```
Rectangle rectangle = new Rectangle();
rectangle.Width = 10.5;
rectangle.Height = 10;
double p = rectangle.Perimeter();
```

در این مثال مشاهده می‌کنید که پس از فراخوانی متد Perimeter مقدار بازگشتی آن در متغیری به نام p قرار گرفته است. اگر نوع خروجی یک متد که در هنگام تعریف آن پیش از نام متد قرار می‌گیرد void یا پوچ نباشد، متد می‌تواند مقدار مورد نظر را با استفاده از کلمه کلیدی return بازگشت دهد. کلمه return و به دنبال آن مقداری که از نظر نوع باید با نوع خروجی تعیین شده تطبیق داشته باشد، مقدار درج شده را به کد فراخوان متد بازگشت می‌دهد.
نکته : کلمه return علاوه بر بازگشت مقدار مورد نظر سبب پایان اجرای متد نیز می‌شود. حتی در صورتی که نوع خروجی یک متد void تعریف شده باشد استفاده از کلمه return بدون اینکه مقداری به دنبال آن بیاید می‌تواند برای پایان اجرای متد، در صورت نیاز و مثلاً برقراری شرطی خاص مفید باشد. بدون کلمه return متد زمانی پایان می‌یابد که به پایان قطعه کد بدنه خود برسد. توجه نمایید که در صورتی که نوع خروجی متد چیزی به جز void است استفاده از کلمه return به همراه مقدار مربوطه الزامی است.
مقدار خروجی یک متد را می‌توان هر جایی که مقداری از همان نوع مناسب است مستقیماً به کار برد. همچنین می‌توان آن را در یک متغیر قرار داد و سپس از آن استفاده نمود.
به عنوان مثال کلاس ساده زیر را در نظر بگیرید که متدی دارد برای جمع دو عدد.

```
public class SimpleMath
{
   public int AddTwoNumbers(int number1, int number2)
   {
      return number1 + number2;
   }
}
```

و حال دو روش استفاده از این متد:

```
SimpleMath obj = new SimpleMath();

Console.WriteLine(obj.AddTwoNumbers(1, 2));

int result = obj.AddTwoNumbers(1, 2);
Console.WriteLine(result);
```

در روش اول مستقیماً خروجی متد مورد استفاده قرار گرفته است و در روش دوم ابتدا مقدار خروجی در یک متغیر قرار گرفته است و سپس از مقدار درون متغیر استفاده شده است. استفاده از متغیر برای نگهداری مقدار خروجی اجباری نبوده و تنها جهت بالا بردن خوانایی برنامه یا حفظ مقدار خروجی تابع برای استفاده‌های بعدی به کار می‌رود.

در بخش‌های بعدی بحث ما در مورد سایر اعضای کلاس و برخی جزییات پیرامون اعضای پیش گفته خواهد بود.

نظرات خوانندگان

نویسنده: سید ایوب کوکبی
تاریخ: ۱۳۹۲/۰۱/۲۴ ۱۱:۵۶

ممنون بابت مطلب آموزشی تون،
تاکیدتان بر استفاده از قرار دادهای نامگذاری، تاکید مثبتی است و واقعا مهم،
کتاب Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries در این زمینه اطلاعات کاملتر و دقیقتری در بر داره.
یکی از روش هایی که در رعایت استاندارهای کد نویسی تاثیر مفیدی داره این هستش که در قطعه کد هایی که به عنوان مثال در آموزش‌ها ارائه می‌شه تا حد امکان سعی بشه این اصول رعایت بشه تا به صورت تدریجی این روش کد نویسی جزئی از عادات برنامه نویسی ما بشود.

عنوان: **مفاهیم برنامه نویسی ـ مروری بر پروپرتی‌ها**
نویسنده: آرمان فرقانی
تاریخ: ۱۳۹۲/۰۲/۰۴ ۱۷:۴۰
آدرس: www.dotnettips.info
برچسب‌ها: C#, OOP, Programming, آموزش برنامه نویسی

در مطلب پیشین کلاسی را برای حل بخشی از یک مسئله بزرگ تهیه کردیم. اگر فراموش کردید پیشنهاد می‌کنم یک بار دیگر آن مطلب را مطالعه کنید. بد نیست بار دیگر نگاهی به آن بیاندازیم.

```
public class Rectangle
{
    public double Width;
    public double Height;

    public double Area()
    {
        return Width*Height;
    }

    public double Perimeter()
    {
        return 2*(Width + Height);
    }
}
```

کلاس خوبی است اما همان طور که در بخش قبل مطرح شد این کلاس می‌تواند بهتر هم باشد. در این کلاس برای نگهداری حالت اشیائی که از روی آن ایجاد خواهند شد از متغیرهایی با سطح دسترسی عمومی استفاده شده است. این متغیرهای عمومی فیلد نامیده می‌شوند. مشکل این است که با این تعریف، اشیاء نمی‌توانند هیچ اعتراضی به مقادیر غیر معتبری که ممکن است به آن‌ها اختصاص داده شود، داشته باشند. به عبارت دیگر هیچ کنترلی بر روی مقادیر فیلدها وجود ندارد. مثلاً ممکن است یک مقدار منفی به فیلد طول اختصاص یابد. حال آنکه طول منفی معنایی ندارد.

تذکر : ممکن است این سوال پیش بیاید که خوب ما کلاس را نوشته ایم و خودمان می‌دانیم چه مقادیری برای فیلدهای آن مناسب است. اما مسئله اینجاست که اولاً ممکن است کلاس تهیه شده توسط برنامه نویس دیگری مورد استفاده قرار گیرد. یا حتی پس از مدتی فراموش کنیم چه مقادیری برای کلاسی که مدتی قبل تهیه کردیم مناسب است. و از همه مهمتر این است که کلاس‌ها و اشیاء به عنوان ابزاری برای حل مسائل هستند و ممکن است مقادیری که به فیلدها اختصاص می‌یابد در زمان نوشتن برنامه مشخص نباشد و در زمان اجرای برنامه توسط کاربر یا کدهای بخش‌های دیگر برنامه تعیین گردد.
به طور کلی هر چه کنترل و نظارت بیشتری بر روی مقادیر انتسابی به اشیاء داشته باشیم برنامه بهتر کار می‌کند و کمتر دچار خطاهای مهلک و بدتر از آن خطاهای منطقی می‌گردد. بنابراین باید ساز و کار این نظارت را در کلاس تعریف نماییم.
برای کلاس‌ها یک نوع عضو دیگر هم می‌توان تعریف کرد که دارای این ساز و کار نظارتی است. این عضو **Property** نام دارد و یک مکانیسم انعطاف پذیر برای خواندن، نوشتن یا حتی محاسبه مقدار یک فیلد خصوصی فراهم می‌نماید.
تا اینجا باید به این نتیجه رسیده باشید که تعریف یک متغیر با سطح دسترسی عمومی در کلاس روش پسندیده و قابل توصیه ای نیست. بنابراین متغیرها را در سطح کلاس به صورت خصوصی تعریف می‌کنیم و از طریق تعریف Property امکان استفاده از آن‌ها در بیرون کلاس را ایجاد می‌کنیم.
حال به چگونگی تعریف Propertyها دقت کنید.

```
public class Rectangle
{
    private double _width = 0;
    private double _height = 0;

    public double Width
    {
        get { return _width; }
        set { if (value > 0) { _width = value; } }
    }

    public double Height
    {
        get { return _height; }
        set { if (value > 0) { _height = value; } }
    }
```

```
    public double Area()
    {
        return _width * _height;
    }

    public double Perimeter()
    {
        return 2*(_width + _height);
    }
}
```

به تغییرات ایجاد شده در تعریف کلاس دقت کنید. ابتدا سطح دسترسی دو متغیر خصوصی شده است یعنی فقط اعضای داخل کلاس به آن دسترسی دارند و از بیرون قابل استفاده نیست. نام متغیرهای پیش گفته بر اساس اصول صحیح نامگذاری فیلدهای خصوصی تغییر داده شده است. ببینید اگر اصول نامگذاری را رعایت کنید چقدر زیباست. هر جای برنامه که چشمتان به _width بخورد فوراً متوجه می‌شوید یک فیلد خصوصی است و نیازی نیست به محل تعریف آن مراجعه کنید. از طرفی یک مقدار پیش فرض برای این دو فیلد در نظر گرفته شده است که در صورتی که مقدار مناسبی برای آن‌ها تعیین نشد مورد استفاده قرار خواهند گرفت.

دو قسمت اضافه شده دیگر تعریف دو Property مورد نظر است. یکی عرض و دیگری ارتفاع. خط اول تعریف پروپرتی تفاوتی با تعریف فیلد عمومی ندارد. اما همان طور که می‌بینید هر فیلد دارای یک بدنه است که با {} مشخص می‌شود. در این بدنه ساز و کار نظارتی تعریف می‌شود.

نحوه دسترسی به پروپرتی‌ها مشابه فیلدهای عمومی است. اما پروپرتی‌ها در حقیقت متدهای ویژه ای به نام اکسسور (Accessor) هستند که از طرفی سادگی استفاده از متغیرها را به ارمغان می‌آورند و از طرف دیگر دربردارنده امنیت و انعطاف پذیری متدها هستند. یعنی در عین حال که روشی عمومی برای داد و ستد مقادیر ارایه می‌دهند، کد پیاده سازی یا وارسی اطلاعات را مخفی نموده و استفاده کننده کلاس را با آن درگیر نمی‌کنند. قطعه کد زیر چگونگی استفاده از پروپرتی را نشان می‌دهد.

```
Rectangle rectangle = new Rectangle();
rectangle.Width = 10;
Console.WriteLine(rectangle.Width);
```

به خوبی مشخص است برای کد استفاده کننده از شیء که آن‌را کد مشتری می‌نامیم نحوه دسترسی به پروپرتی یا فیلد تفاورتی ندارد. در اینجا خط دوم که مقداری را به یک پروپرتی منتسب کرده سبب فراخوانی اکسسور set می‌گردد. همچنین مقدار منتسب شده یعنی ۱۰ در داخل بدنه اکسسور از طریق کلمه کلیدی value قابل دسترسی و ارزیابی است. در خط سوم که لازم است مقدار پروپرتی برای چاپ بازیابی یا خوانده شود منجر به فراخوانی اکسسور get می‌گردد.

تذکر : به دو اکسسور get و set مانند دو متد معمولی نگاه کنید از این نظر که می‌توانید در بدنه آن‌ها اعمال دلخواه دیگری بجز ذخیره و بازیابی اطلاعات پروپرتی را نیز انجام دهید.

چند نکته :

اکسسور get هنگام بازگشت یا خواندن مقدار پروپرتی اجرا می‌شود و اکسسور set زمان انتساب یک مقدار جدید به پروپرتی فراخوانی می‌شود. جالب آنکه در صورت لزوم این دو اکسسور می‌توانند دارای سطوح دسترسی متفاوتی باشند.

داخل اکسسور set کلمه کلیدی value مقدار منتسب شده را در اختیار قرار می‌دهد تا در صورت لزوم بتوان بر روی آن پردازش لازم را انجام داد.

یک پروپرتی می‌تواند فاقد اکسسور set باشد که در این صورت یک پروپرتی فقط خواندنی ایجاد می‌گردد. همچنین می‌تواند فقط شامل اکسسور set باشد که در این صورت فقط امکان انتساب مقادر به آن وجود دارد و امکان دریافت یا خواندن مقدار آن میسر نیست. چنین پروپرتی ای فقط نوشتنی خواهد بود.

در بدنه اکسسور set الزامی به انتساب مقدار منتسب توسط کد مشتری نیست. در صورت صلاحدید می‌توانید به جای آن هر مقدار دیگری را در نظر بگیرید یا عملیات مورد نظر خود را انجام دهید.

در بدنه اکسسور get هم هر مقداری را می‌توانید بازگشت دهید. یعنی الزامی وجود ندارد حتماً مقدار فیلد خصوصی متناظر با پروپرتی را بازگشت دهید. حتی الزامی به تعریف فیلد خصوصی برای هر پروپرتی ندارید. به طور مثال ممکن است مقدار بازگشتی اکسسور get حاصل محاسبه و ... باشد.

اکنون مثال دیگری را در نظر بگیرید. فرض کنید در یک پروژه فروشگاهی در حال تهیه کلاسی برای مدیریت محصولات هستید. قصد داریم یک پروپرتی ایجاد کنیم تا نام محصول را نگهداری کند و در حال حاضر هیچ محدودیتی برای نام یک محصول در نظر نداریم. کد زیر را ببینید.

```
public class Product
{
    private string _name;
    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }
}
```

همانطور که می‌بینید در بدنه اکسسورهای پروپرتی Name هیچ عملیات نظارتی ای در نظر گرفته نشده است. آیا بهتر نبود بیهوده پروپرتی تعریف نکنیم و خیلی ساده از یک فیلد عمومی که همین کار را انجام می‌دهد استفاده کنیم؟ خیر. بهتر نبود. مهمترین دلیلی که فعلاً کافی است تا شما را قانع کند تعریف پروپرتی روش پسندیده‌تری از فیلد عمومی است را بررسی می‌کنیم.
فرض کنید پس از مدتی متوجه شدید اگر نام بسیار طولانی ای برای محصول درج شود ظاهر برنامه شما دچار مشکل می‌شود. پس باید بر روی این مورد نظارت داشته باشید. دیدیم که برای رسیدن به این هدف باید فیلد عمومی را فراموش و به جای آن پروپرتی تعریف کنیم. اما مسئله اینست که تبدیل یک فیلد عمومی به پروپرتی میتواند سبب بروز ناسازگاری‌هایی در بخش‌های دیگر برنامه که از این کلاس و آن فیلد استفاده می‌کنند شود. پس بهتر آن است که از ابتدا پروپرتی تعریف کنیم هر چند نیازی به عملیات نظارتی خاصی نداریم. در این حالت اگر نیاز به پردازش بیشتر پیدا شد به راحتی می‌توانیم کد مورد نظر را در اکسسورهای موجود اضافه کنیم بدون آنکه نیازی به تغییر بخش‌های دیگر باشد.
و یک خبر خوب! از سی شارپ ۳ به بعد ویژگی جدیدی در اختیار ما قرار گرفته است که می‌توان پروپرتی‌هایی مانند مثال بالا را که نیازی به عملیات نظارتی ندارند، ساده‌تر و خواناتر تعریف نمود. این ویژگی جدید پروپرتی اتوماتیک یا Auto-Implemented Property نام دارد. مانند نمونه زیر.

```
public class Product
{
    public string Name { get; set; }
}
```

این کد مشابه کد پیشین است با این تفاوت که خود کامپایلر یک متغیر خصوصی و بی نام را ایجاد می‌نماید که فقط داخل اکسسورهای پروپرتی قابل دسترسی است.
البته استفاده از پروپرتی برتری دیگری هم دارد. و آن کنترل سطح دسترسی اکسسورها است. مثال زیر را ببینید.

```
public class Student
{
    public DateTime Birthdate { get; set; }

    public double Age { get; private set; }
}
```

کلاس دانشجو یک پروپرتی به نام تاریخ تولد دارد که قابل خواندن و نوشتن توسط کد مشتری (کد استفاده کننده از کلاس یا اشیاء آن) است. و یک پروپرتی دیگر به نام سن دارد که توسط کد مشتری تنها قابل خواندن است. و تنها توسط سایر اعضای داخل همین کلاس قابل نوشتن است. چون اکسسور set آن به صورت خصوصی تعریف شده است. به این ترتیب بخش دیگری از کلاس سن دانشجو را بر اساس تاریخ تولد او محسابه می‌کند و در پروپرتی Age قرار می‌دهد و کد مشتری می‌تواند آن‌را مورد استفاده قرار دهد اما حق دستکاری آن‌را ندارد. به همین ترتیب در صورت نیاز اکسسور get را می‌توان خصوصی کرد تا پروپرتی از دید کد مشتری فقط نوشتنی باشد. اما حتماً می‌توانید حدس بزنید که نمی‌توان هر دو اکسسور را خصوصی کرد. چرا؟
تذکر: در هنگام تعریف یک **فیلد** می‌توان از کلمه کلیدی readonly استفاده کرد تا یک **فیلد فقط خواندنی** ایجاد گردد. اما در اینصورت فیلد تعریف شده حتی داخل کلاس هم فقط خواندنی است و فقط در هنگام تعریف یا در متد سازنده کلاس امکان مقدار دهی به آن وجود دارد. در بخش‌های بعدی مفهوم سازنده کلاس مورد بررسی خواهد گرفت.

نظرات خوانندگان

نویسنده: Kingtak
تاریخ: ۱۳۹۲/۰۲/۰۴ ۱۸:۱۶

مثل همیشه عالی بود....
واقعا با آموزش‌های شما حال میکنم.تا حالاچندین مقاله در مورد پروپرتی‌ها خونده بودم ولی بطور کامل متوجه کاربردش و فوایدش نشده بودم.
منتظر آموزش‌های بعدی هستیم

نویسنده: آرمان فرقانی
تاریخ: ۱۳۹۲/۰۲/۰۴ ۱۹:۱۲

تشکر. شما لطف دارید. امیدوارم مطالب بعدی هم براتون مفید باشه.

نویسنده: سید ایوب کوکبی
تاریخ: ۱۳۹۲/۰۲/۰۴ ۲۱:۲۲

سلام،
خیلی برام لذت بخشه دانسته هام رو به شیوه ای زیباتر مرور کنم و در ضمن با نکاتی ظریفتر آشنا بشم.
چند تا پیشنهاد:
-1 خیلی خوب میشه اگه تا حد امکان معادل انگلیسی کلمات تخصصی مربوطه رو هم بزارید، مثلا از اکسسور استفاده میکنید خوبه ولی داخل پرانتز هم اشاره ای به معادل انگلیسی Accessor بکنید تا بدونیم در سینتکسش چه جوریه، و تا حد امکان هم معادل تخصصی واژه مربوطه ذکر بشه مثلا فیلدهای خصوصی داخل کلاس معمولا با عنوان backing field یا فیدلهای پشتی خطاب میشه که اگه به این موارد هم اشاره ای بشه خوبه.
-2 جاهایی که به استانداردها و کلا هر چیزی در حوزه مهندسی نرم افزار اشاره می‌کنید لطفا تا حد امکان اشاره ای هم به آن بکنید مثلا فرمودید اصول نامگذاری استاندارد فیلد خصوصی، اینجا به نظر من بهتره که اشاره ای هم بکنید مصلا در استاندارد مایکروسافت فیلدهای خصوصی از قاعده نامگذاری Camel Case استفاده میکنند و یا مثلا در متدها و کلاس‌های از روش Pascal Case استفاده میشه.
-3 در مورد اینکه چه مواقعی باید از فلان موضوع، قاعده و مبحث و ... استفاده بشه هم در صورت امکان و تا حد توان اشاره بکنید مثلا فرمودید فیلدهای فقط خواندنی Readonly، مناسبه که اشاره ای هم بکنید معمولا چه زمانی و در چه شرایطی بهتره از این نوع فیلد استفاده کنیم و چرا؟ (البته هنوز که توضیحی ندادید ولی پیشتر اشاره کردم تا انشاء الله در مقاله بعدی تان در صورت صلاحدید لحاظ بفرمایید).

پیشنهاداتی که شد صرفا برای هر چه بهتر شدن مقالات بعدی شماست و اینکه باعث بشه، خواننده وقتی با موضوعاتی در این زمینه در متون تخصصی برخورد داشت احساس بیگانگی نکنه و سریعتر در جریان کار قرار بگیره.
سلسله مباحثی که ارائه کرده اید تا کنون زیبا بود و خواندنی و بنده هم مخاطب همیشگی این سلسله مباحث و البته امیدوارم به کمتر شدن فاصله بین پست‌های ارسالی :).
متشکرم/.

نویسنده: آرمان فرقانی
تاریخ: ۱۳۹۲/۰۲/۰۴ ۲۲:۵۰

ضمن تشکر از پیگیری و پیشنهادهای حضرتعالی و پوزش به جهت طولانی شدن فاصله زمانی ارائه مطالب در مورد پیشنهادهای ارزشمندی که فرمودید باید چند نکته را عرض کنم.
تا حد زیادی معمولاً سعی کردم این موارد محقق بشه. مثلا در مورد همان اکسسور و بیشتر مفاهیم و اصطلاحات مهم، معادل انگلیسی آورده شده است. اصولاً ترجمه برخی مفاهیم را مناسب نمی‌دانم و از طرفی آوردن تعداد زیادی واژه انگلیسی در بین واژگان فارسی سبب کاهش زیبایی متن می‌گردد. بنابراین معمولاً کلمات مهم را یک یا چند بار به صورت انگلیسی بیان می‌کنم و

سپس با حروف فارسی می‌نویسم مانند اکسسور تا به صورت روان‌تری در متن قابل خواندن باشد.
همچنین در امر آموزش ابتدا سعی می‌کنم یک دید کلی و از بالا به دانشجو یا خواننده منتقل کنم. در این مرحله تنها جزییات مهم که برای درک موضوع و شروع کار عملی مانند انجام یک پروژه کاربردی لازم است بیان می‌شود. چراکه اگر از ابتدا ذهن را با تعداد زیادی جزییات درگیر کنیم ممکن است در موقع خواندن هر بخش خواننده مفاهیم را درک کند اما پس از پایان مطالب نمی‌داند از کجا باید شروع کند و قدرت استفاده از آموخته‌ها را ندارد. به همین جهت سعی می‌شود بر روی مفاهیم غیر کلیدی کمتر در مراحل اولیه بحث شود.
از طرفی سعی می‌کنم مطالب دارای حجم مناسب و مفاهیم پیوسته ای باشند تا قابل درک بوده و خسته کننده نباشند. مثلاً از آنجاییکه در بخش‌های پیشین مقاله‌ای که به زحمت یکی از دوستان در سایت قرار گرفته بود برای نامگذاری معرفی شد، از تکرار قوانین یاد شده در این مطالب به جهت جلوگیری از طولانی‌تر شدن خودداری کردم.
با توجه به کارگاه‌های عملی ای که برای تثبیت مطالب در نظر گرفته خواهد شد، تا حد زیادی روش‌های بهینه برای پیاده سازی مفاهیم گوناگون معرفی خواهد شد.

نویسنده: عبداللهی
تاریخ: ۱۳۹۲/۰۲/۰۴ ۲۳:۲۶

سلام.
واقعا عالی بود.مرسی.فقط یه سوالی داشتم. در کلاس student سطح دسترسی بصورت پیش فرض private خواهد بود؟چون اگر درست یادم مونده باشه موقع تعریف متغیر سطح دسترسی بصورت پیشفرض private بود.درسته؟
بازم ممنون.3 تا مقالتون رو خودنم.واقعا مفید بود.باتشکر

نویسنده: آرمان فرقانی
تاریخ: ۱۳۹۲/۰۲/۰۵ ۱۱:۲۲

سلام و تشکر.
سطح دسترسی پیش فرض برای اعضای کلاس (حتی کلاس داخلی‌تر) به صورت پیش فرض private است. اما اکسسورها از سطح دسترسی پروپرتی تبعیت می‌کنند مگر آنکه صراحتاً سطح دسترسی آن‌ها تعیین گردد. بدیهی است در صورتی تعیین صریح سطح دسترسی برای اکسسورها پذیرفته است که نسبت به سطح دسترسی پروپرتی محدودتر باشد. یعنی نمی‌توانید مثلاً پروپرتی ای را private و اکسسور آن را public تعیین کنید.

نویسنده: علی صداقت
تاریخ: ۱۳۹۲/۰۲/۰۷ ۱۰:۲۸

با سپاس فراوان از سلسه مطالب مفید شما. فکر میکنم در کلاس Rectangle و پروپرتی Height در قسمت set، مقدار value باید در شرط مورد ارزیابی قرار گیرد.

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۰۲/۰۷ ۱۱:۲۴

```
set { if (value > 0) { _width = value; } }
```

این نوع طراحی API به نظر من ایراد داره. از این جهت که مصرف کننده نمی‌دونه چرا مقداری که وارد کرده تاثیری نداشته. بهتره در این نوع موارد یک استثناء صادر شود.

نویسنده: آرمان فرقانی
تاریخ: ۱۳۹۲/۰۲/۰۷ ۱۱:۳۱

با تشکر. اصلاح شد.

نویسنده: آرمان فرقانی
تاریخ: ۱۳۹۲/۰۲/۰۷ ۱۱:۴۰

تشکر فراوان از نظر حضرتعالی.
بله صحیح می‌فرمایید. اما کار با این کلاس تمام نشده است و صرفاً مثالی ساده برای بیان مفاهیم پایه ای مورد نظر در مقاله است. در چنین مثالی نباید ذهن خواننده را درگیر مسائلی نمود که مورد هدف بحث نیست. ضمناً هر مقاله دارای یک جامعه هدف است. اگرچه می‌تواند برای افراد دیگر هم مفید واقع شود اما باید دانسته‌های جامعه هدف خود را مد نظر داشته باشد. برای خواننده ای که در حال آشنایی با مفهوم پروپرتی است، صدور یک استثنا و مفاهیم مربوطه نیاز به بحثی جدا دارد.

نویسنده: سید ایوب کوکبی
تاریخ: ۱۳۹۲/۰۲/۱۰ ۲۲:۱۷

البته امیدوارم هستم این موارد را در مقالات بعدی خود شرح دهید.

نویسنده: محمد
تاریخ: ۱۳۹۲/۰۲/۱۷ ۳:۵۲

خیلی ممنون بابت مطلبتون
یه سوالی که مدت‌ها تو ذهنم مونده اینه که فرق این کار (استفاده از property برای مقدار دهی فیلدهای private) با مدل مشابهی که در کتاب‌های جاوا دیده می‌شه (استفاده از دو متد به طور مثال setWidth و getWidth برای مقدار دهی فیلدهای private در اینجا width) در چیه؟

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۰۲/۱۷ ۸:۹

این روش در سی‌شارپ منسوخ شده در نظر گرفته میشه و یکی از مواردی هست که حین تبدیل کدهای جاوا به سی شارپ تبدیل به یک خاصیت خواهد شد بجای دو متد.

نویسنده: صابر فتح الهی
تاریخ: ۱۳۹۲/۰۲/۱۷ ۸:۱۹

با اجازه آقای آرمان فرقانی
دوست گلم دقیقا وقتی برنامه کامپایل میشه خصیصه‌ها به متدهایی مانند جاوا (که با set_ یا get_ شروع شده و به نام خصیصه ختم می‌شود) تبدیل میشوند
مثلا شما توی کلاست اگر خصیصه ای با نام Width داشته باشید و یک متد مانند get_Width تعریف کنی زمان کامپایل خطا زیر دریافت می‌کنی، به منزله اینکه این متد وجود داره:

```
Error 1 Type 'Rectangle' already reserves a member called 'get_Width' with the same parameter types
E:\Test\Rectangle.cs5940
```

نویسنده: آرمان فرقانی
تاریخ: ۱۳۹۲/۰۲/۱۷ ۱۲:۴۱

تشکر از شما و توضیحات ارزشمند دوستان گرامی.
پروپرتی و پروپرتی اتوماتیک امکانی است که در زبان سی شارپ و ... قرار داده شده است. پروپرتی‌ها نیز در حقیقت متدهای مشابهی دارند که همان اکسسورها هستند. تفاوت میزان بیشتر کپسوله سازی و مخفی کردن منطق پیاده سازی، و مهم‌تر سازگاری بیشتر با مفهوم ویژگی است. که البته در هنگام استفاده از پروپرتی سهولت بیشتری را نیز فراهم می‌کند.
همان که دوست عزیزم اشاره فرمودند به دلیل عدم سازگاری ذات زبان‌های مبتنی بر دات فریمورک از اکسسور، به صورت

داخلی به متد تبدیل خواهند شد.
همچنین در مورد جاوا هم پروژه هایی وجود دارند که سعی کرده اند این امکان را به کمک یک سری Annotaion به آن بیافزایند.
در مورد سی شارپ استفاده از پروپرتی روش توصیه شده است.

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۰۲/۱۷ ۱۳:۲۱

دات نت مشکلی با متدهای get و set دار نداره. فقط چون خیلی verbose بوده، جمع و جور شده به auto implemented properties برای زیبایی کار و سهولت تایپ. نکته‌ای هم که آقای فتح الهی عنوان کردند، در مورد ترکیب متد و خاصیت هم نام با هم بود، در یکجا البته اون هم حالتی که بعد از متد get شروع شده با حرف کوچک، یک _ باشد مثلا و نه حالت دیگری.

نویسنده: آرمان فرقانی
تاریخ: ۱۳۹۲/۰۲/۱۷ ۱۴:۲۹

متوجه نکته مورد نظر شما نشدم. بیان شد در زبان سی شارپ و ... ساختار کپسوله‌تر پروپرتی در مقایسه با متدهای صریح تنظیم و بازیابی مقدار فیلدها در جاوا معرفی شده اند ولی پیاده سازی داخلی آن به همان صورت متد است. نکته دوست گرامی آقای فتح الهی هم گمان می‌کنم بیشتر به منظور اشاره به چگونگی پیاده سازی داخلی است و نه اینکه مراقب باشید تداخل نام پیش نیاید.

نویسنده: صابر فتح الهی
تاریخ: ۱۳۹۲/۰۲/۱۷ ۱۷:۱۰

بله من در پاسخ به سوال دوستمون گفتم **پیاده سازی داخلی** خصیصه به این شکل هست که خصیصه به شکل متد پیاده سازی است، و جهت آزمایش گفتم یک متد ... ایجاد کنید.

عنوان: **مفاهیم برنامه نویسی ـ آشنایی با سازنده‌ها**
نویسنده: آرمان فرقانی
تاریخ: ۱۳۹۲/۰۲/۱۴ ۳:۰
آدرس: www.dotnettips.info
گروه‌ها: C#, OOP, Programming, آموزش برنامه نویسی

در مطلب پیشین برای نگهداری حالت شیء یا همان ویژگی‌های آن Propertyها را در کلاس معرفی کردیم و پس از ایجاد شیء مقدار مناسبی را به پروپرتی‌ها اختصاص دادیم.
اگرچه ایجاد شیء و مقداردهی به ویژگی‌های آن ما را به هدفمان می‌رساند، اما بهترین روش نیست چرا که ممکن است مقداردهی به یک ویژگی فراموش شده و سبب شود شیء در وضعیت نادرستی قرار گیرد. این مشکل با استفاده از سازنده‌ها (Constructors) حل می‌شود.
سازنده (Constructor) عضو ویژه ای از کلاس است بسیار شبیه به یک متد که در هنگام وهله سازی (ایجاد یک شیء از کلاس) به صورت خودکار فراخوانی و اجرا می‌شود. وظیفه سازنده مقداردهی به ویژگی‌های عمومی و خصوصی شیء تازه ساخته شده و به طور کلی انجام هر کاری است که برنامه‌نویس در نظر دارد پیش از استفاده از شیء به انجام برساند.
مثالی که در این بخش بررسی می‌کنیم کلاس مثلث است. برای پیاده سازی این کلاس سه ویژگی در نظر گرفته ایم. قاعده، ارتفاع و مساحت. بله مساحت را این بار به جای متد به صورت یک پروپرتی پیاده سازی می‌کنیم. اگرچه در آینده بیشتر راجع به چگونگی انتخاب برای پیاده سازی یک عضو کلاس به صورت پروپرتی یا متد بحث خواهیم کرد اما به عنوان یک قانون کلی در نظر داشته باشید عضوی که به صورت منطقی به عنوان داده مطرح است را به صورت پروپرتی پیاده سازی کنید. مانند نام دانشجو. از طرفی اعضایی که دلالت بر انجام عملی دارند را به صورت متد پیاده سازی می‌کنیم. مانند متد تبدیل به نوع داده دیگر. (مثلاً Object.ToString())

```
public class Triangle
{
    private int _height;
    private int _baseLength;

    public int Height
    {
        get { return _height; }

        set
        {
            if (value < 1 || value > 100)
            {
                // تولید خطا
            }

            _height = value;
        }
    }

    public int BaseLength
    {
        get { return _baseLength; }

        set
        {
            if (value < 1 || value > 100)
            {
                // تولید خطا
            }

            _baseLength = value;
        }
    }

    public double Area
    {
        get { return _height * _baseLength * 0.5; }
    }
}
```

چون در بخشی از یک پروژه نیاز پیدا کردیم با یک سری مثلث کار کنیم، کلاس بالا را طراحی کرده ایم. به نکات زیر توجه نمایید.
• در اکسسور set دو ویژگی قاعده و ارتفاع، محدوده مجاز مقادیر قابل انتساب را بررسی نموده ایم. در صورتی که مقداری خارج از محدوده یاد شده برای این ویژگی‌ها تنظیم شود خطایی را ایجاد خواهیم کرد. شاید برای برنامه نویسانی که تجربه کمتری دارند

زیاد روش مناسبی به نظر نرسد. اما این یک روش قابل توصیه است. مواجه شدن کد مشتری (کد استفاده کننده از کلاس) با یک خطای مهلک که علت رخ دادن خطا را نیز می‌توان به همراه آن ارائه کرد بسیار بهتر از بروز خطاهای منطقی در برنامه است. چون رفع خطاهای منطقی بسیار دشوارتر است. در مطالب آینده راجع به تولید خطا و موارد مرتبط با آن بیشتر صحبت می‌کنیم.
• در مورد ویژگی مساحت، اکسسور set را پیاده سازی نکرده ایم تا این ویژگی را به صورت فقط خواندنی ایجاد کنیم.

وقتی شیء ای از یک کلاس ایجاد می‌شود، بلافاصله سازنده آن فراخوانی می‌گردد. سازنده‌ها هم نام کلاسی هستند که در آن تعریف می‌شوند و معمولاً اعضای داده ای شیء جدید را مقداردهی می‌کند. همانطور که می‌دانید وهله سازی از یک کلاس با عملگر new انجام می‌شود. سازنده کلاس بلافاصله پس از آنکه حافظه برای شیء در حال تولید اختصاص داده شد، توسط عملگر new فراخوانی می‌شود.

سازنده پیش فرض سازنده‌ها مانند متدهای دیگر می‌توانند پارامتر دریافت کنند. سازنده ای که هیچ پارامتری دریافت نمی‌کند سازنده پیش فرض (Default constructor) نامیده می‌شود. سازنده پیش فرض زمانی اجرا می‌شود که با استفاده از عملگر new شیء ای ایجاد می‌کنید اما هیچ آرگومانی را برای این عملگر در نظر نگرفته اید.
اگر برای کلاسی که طراحی می‌کنید سازنده ای تعریف نکرده باشید کامپایلر سی شارپ یک سازنده پیش فرض (بدون پارامتر) خواهد ساخت. این سازنده هنگام ایجاد اشیاء فراخوانی شده و مقدار پیش فرض متغیرها و پروپرتی‌ها را با توجه به نوع آن‌ها تنظیم می‌نماید. مثلاً مقدار صفر برای متغیری از نوع int یا false برای نوع bool و null برای انواع ارجاعی که در آینده در این مورد بیشتر خواهید آموخت.
اگر مقادیر پیش فرض برای متغیرها و پروپرتی‌ها مناسب نباشد، مانند مثال ما، سازنده پیش فرض ساخته شده توسط کامپایلر همواره شیء ای می‌سازد که وضعیت صحیحی ندارد و نمی‌تواند وظیفه خود را انجام دهد. در این گونه موارد باید این سازنده را جایگزین نمود.

جایگزینی سازنده پیش فرض ساخته شده توسط کامپایلر افزودن یک سازنده صریح به کلاس بسیار شبیه به تعریف یک متد در کلاس است. با این تفاوت که:
سازنده هم نام کلاس است.
برای سازنده نوع خروجی در نظر گرفته نمی‌شود.

در مثال ما محدوده مجاز برای قاعده و ارتفاع مثلث بین ۱ تا ۱۰۰ است در حالی که سازنده پیش فرض مقدار صفر را برای آنها تنظیم خواهد نمود. پس برای اینکه مطمئن شویم اشیاء مثلث ساخته شده از این کلاس در همان بدو تولید دارای قاعده و ارتفاع معتبری هستند سازنده زیر را به صورت صریح در کلاس تعریف می‌کنیم تا جایگزین سازنده پیش فرضی شود که کامپایلر خواهد ساخت و به جای آن فراخوانی گردد.

```
public Triangle()
{
    _height = _baseLength = 1;
}
```

در این سازنده مقدار ۱ را برای متغیر خصوصی پشت (backing field یا backing store) هر یک از دو ویژگی قاعده و ارتفاع تنظیم نموده ایم.

اجرای سازنده همانطور که گفته شد سازنده اضافه شده به کلاس جایگزین سازنده پیش فرض کامپایلر شده و در هنگام ایجاد یک شیء جدید از کلاس مثلث توسط عملگر new اجرا می‌شود. برای بررسی اجرا شدن سازنده به سادگی می‌توان کدی مشابه مثال زیر را نوشت.

```
Triangle triangle = new Triangle();

Console.WriteLine(triangle.Height);
Console.WriteLine(triangle.BaseLength);
Console.WriteLine(triangle.Area);
```

کد بالا مقدار ۱ را برای قاعده و ارتفاع و مقدار ۰.۵ را برای مساحت چاپ می‌نماید. بنابراین مشخص است که سازنده اجرا شده و مقادیر مناسب را برای شیء تنظیم نموده به طوری که شیء از بدو تولید در وضعیت مناسبی است.

سازنده‌های پارامتر دار در مثال قبل یک سازنده بدون پارامتر را به کلاس اضافه کردیم. این سازنده تنها مقادیر پیش فرض مناسبی را تنظیم می‌کند. بدیهی است پس از ایجاد شیء در صورت نیاز می‌توان مقادیر مورد نظر دیگر را برای قاعده و ارتفاع تنظیم نمود. اما برای اینکه سازنده بهتر بتواند فرآیند وهله سازی را کنترل نماید می‌توان پارامترهایی را به آن افزود. افزودن پارامتر به سازنده مانند افزودن پارامتر به متدهای دیگر صورت می‌گیرد. در مثال زیر سازنده دیگری تعریف می‌کنیم که دارای دو پارامتر است. یکی قاعده و دیگری ارتفاع. به این ترتیب در حین فرآیند وهله سازی می‌توان مقادیر مورد نظر را منتسب نمود.

```
public Triangle(int height, int baseLength)
{
    Height = height;
    BaseLength = baseLength;
}
```

با توجه به اینکه مقادیر ارسالی به این سازنده توسط کد مشتری در نظر گرفته می‌شود و ممکن است در محدوده مجاز نباشد، به جای انتساب مستقیم این مقادیر به فیلد خصوصی پشت ویژگی قاعده و ارتفاع یعنی _baseLength و _height آنها را به پروپرتی‌ها منتسب کردیم تا قانون اعتبارسنجی موجود در اکسسور set پروپرتی‌ها از انتساب مقادیر غیر مجاز جلوگیری کند.
سازنده اخیر را می‌توان به صورت زیر با استفاده از عملگر new و فراهم کردن آرگومان‌های مورد نظر مورد استفاده قرار داد.

```
Triangle triangle = new Triangle(5, 8);

Console.WriteLine(triangle.Height);
Console.WriteLine(triangle.BaseLength);
Console.WriteLine(triangle.Area);
```

مقادیر چاپ شده برابر ۵ برای ارتفاع، ۸ برای قاعده و ۲۰ برای مساحت خواهد بود که نشان از اجرای صحیح سازنده دارد.
در مطالب بالا چندین بار از سازنده ها صحبت کردیم و گفتیم سازنده دیگری به کلاس اضافه می‌کنیم. این دو نکته را به خاطر داشته باشید:
یک کلاس می‌تواند دارای چندین سازنده باشد که بر اساس آرگومان‌های فراهم شده هنگام وهله سازی، سازنده مورد نظر انتخاب و اجرا می‌شود.
الزامی به تعریف یک سازنده پیش فرض (به معنای بدون پارامتر) نیست. یعنی یک کلاس می‌تواند هیچ سازنده بی پارامتری نداشته باشد.

در بخش‌های بعدی مطالب بیشتری در مورد سازنده‌ها و سایر اعضای کلاس خواهید آموخت.

نظرات خوانندگان

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۰۲/۱۴ ۸:۴۵

ممنون از شما. بنابراین مقدار دهی خواص در سازنده کلاس به معنای نسبت دادن مقدار پیش فرض به آن‌ها است. چون سازنده کلاس پیش از هر کد دیگری در کلاس فراخوانی می‌شود.

نویسنده: آرمان فرقانی
تاریخ: ۱۳۹۲/۰۲/۱۴ ۱۰:۳۱

تشکر. همانطور که گفته شد بلافاصله پس از تخصیص حافظه به شیء سازنده اجرا می‌شود. در صورت استفاده از سازنده پارامتر دار کد مشتری از ابتدا شیء را با مقادیر عملیاتی خود ایجاد می‌کند.

نویسنده: سید ایوب کوکبی
تاریخ: ۱۳۹۲/۰۲/۱۴ ۱۲:۵۹

سلام،
آقای فرقانی بسیار عالی بود، واقعا لذت بردم،
فقط خواهشی از شما دارم این هستش که : دقت و حساسیت مبحث شی گرایی رو احیانا فدای چیزهای دیگر نکنید!، منظورم این هستش که تا حد توان اصول مهندسی نرم افزار رو به صورت کامل رعایت بفرمایید و نگران کاربر مبتدی نباشید، کاربر مبتدی ، ناخودآگاه خودش مطالب غیر قابل فهم رو فیلتر میکنه و در بدترین حالت، در آن مورد، از شما سوال خواهند کرد و شما هم آنها را به موضوع مناسب ارجاع خواهید داد هر چند اگر موضوع ارائه شده بعدا قراره باز بشه، خیلی راحت می‌تونید در متن نوید توضیحات بیشتر رو به خواننده بدهید و در غیر این صورت ارجاع به توضیحات مناسب.
به هر حال باز هم تاکید می‌کنم و سفارش میکنم که دقت بیان و رعایت اصول رو فدای هیچ چیزی نکنید.
ممنونم.

نویسنده: آرمان فرقانی
تاریخ: ۱۳۹۲/۰۲/۱۴ ۱۳:۲۳

تشکر از نظر شما.
اگر مورد خاصی مد نظر دارید بفرمایید تا توضیح بدم یا در صورت لزوم در متن اصلاح کنم. در نظر داشته باشید این سری مطالب با مطالب دیگر موجود در سایت متفاوت است و هدف خاصی را دنبال می‌کند که در بخش اول توضیح داده شد. بنابراین نمی‌توان در این سری مطالب نگران کاربر مبتدی‌تر نبود چراکه جامعه هدف این بحث‌ها هستند. به دلیل همین جامعه هدف مورد نظر، نوشتن این گونه مطالب دشوارتر از بیان مفاهیم پیچیده‌تر برای کاربران حرفه‌ای‌تر است. و همین امر به علاوه وقت محدود بنده سبب تأخیر در ارسال مطالب است. ضمناً صرف بیان انبوهی از اطلاعات تأثیر لازم را در خواننده نمی‌گذارد. در بسیاری مواقع بیان برخی مفاهیم مهندسی نرم افزار یا ویژگی‌های جدید زبان (مانند var) به صورت مقایسه ای با روش پیشین سبب تثبیت بهتر مطالب در ذهن خواننده می‌شود. اما در شیوه کد نویسی تا حد ممکن سعی شده اصول رعایت شود و بیهوده خواننده با روش ناصحیح آشنا نشود. اگر نکته خاصی یافتید بفرمایید اصلاح کنیم.

نویسنده: عبداللهی
تاریخ: ۱۳۹۲/۰۲/۱۶ ۰:۵۱

باسلام.تشکر ازینکه مطالب رو ساده گویا و دقیق بیان میکنید.بسیار عالی بود.سپاسگذارم.
یک کلاس همیشه 1 سازنده پیش فرض بدون پارامتر دارد که هنگام وهله سازی فراخوانی شده و اجرا میشود و در صورت نیاز میتوان 1 سازنده بر اساس نیازهای پروژه تعریف کرد که هنگام وهله سازی از یک کلاس سازنده تعریف شده ما بر سازنده پیش فرض اولویت دارد.درسته؟
بازم متشکرم.مرسی

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۰۲/۱۶ ۱۱:۱۲

محدودیتی برای تعداد متدهای سازنده وجود نداره (مبحث overloading است که نیاز به بحثی جداگانه دارند). در زمان وهله سازی کلاس میشه مشخص کرد کدام متد مورد استفاده قرار بگیره. این متد بر سایرین مقدم خواهد بود. همچنین سازنده استاتیک هم قابل تعریف است که نکته خاص خودش رو داره.

نویسنده: آرمان فرقانی
تاریخ: ۱۳۹۲/۰۲/۱۶ ۱۱:۱۴

سلام و تشکر. به نکات زیر در متن توجه فرمایید.
۱. سازنده پیش فرض منظور همان سازنده بی پارامتر است خواه توسط کامپایلر ایجاد شده باشد خواه توسط برنامه نویس به صورت صریح در کلاس تعریف شده باشد.
۲. اگر توسط برنامه نویس هیچ سازنده ای تعریف نگردد، کامپایلر یک سازنده بدون پارامتر خواهد ساخت که وظیفه تنظیم مقادیر پیش فرض اعضای داده ای کلاس را برعهده بگیرد.
۳. اگر برنامه نویس سازنده ای تعریف کند خواه با پارامتر یا بی پارامتر کامپایلر سازنده ای نخواهد ساخت. پس اگر مثلاً برنامه نویس تنها یک سازنده با پارامتر تعریف کند، کلاس فاقد سازنده بی پارامتر یا پیش فرض خواهد بود.
۴. در یک کلاس می‌توان چندین سازنده تعریف نمود.
موفق باشید.

نویسنده: محسن نجف زاده
تاریخ: ۱۳۹۲/۰۲/۲۸ ۱۴:۴۴

سلام/با تشکر از مطالب پایه ای و مفیدتون
کاربرد دو قطعه کد زیر در چه زمانی بهتر است؟

1.گرفتن مقدار مساحت به صورت یک Property

```
public double Area
{
       get { return _height * _baseLength * 0.5; }
}
```

2.محاسبه مساحت با استفاده از یک Method

```
public double Area()
{
     return _height * _baseLength * 0.5;
}
```

نویسنده: احمد
تاریخ: ۱۳۹۲/۰۲/۲۸ ۱۸:۳۳

Properties vs Methods

نویسنده: محسن نجف زاده
تاریخ: ۱۳۹۲/۰۲/۲۹ ۸:۳۸

احمد عزیز ممنون بابت لینک تون.
سوالم اینه که چرا آقای فرقانی تو این مثال برای Area هم یک Property تعریف کردن (چون صرفا فقط یک مثال | براساس تعریف Method و Property ، متد بهتر نبود؟)

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۰۲/۲۹ ۸:۴۸

خلاصه لینک احمد: اگر محاسبات پیچیده و طولانی است، یا تاثیرات جانبی روی عملکرد سایر قسمت‌های کلاس دارند، بهتره از متد استفاده بشه. اگر کوتاه، سریع و یکی دو سطری است و ترتیب فراخوانی آن اهمیتی ندارد، فرقی نمی‌کنه و بهتره که خاصیت باشه و اگر این شرایط حاصل شد، عموم کاربران تازه کار استفاده از خواص را نسبت به متدها ساده‌تر می‌یابند و به نظر آن‌ها Syntax تمیزتری دارد (هدف این سری مقدماتی).

نویسنده: محسن نجف زاده
تاریخ: ۱۳۹۲/۰۲/۲۹ ۹:۱۳

باز هم به نظر من استفاده از method صحیح‌تر بود
درسته که در اینجا محاسبات پیچیده (computationally complex) نداریم اما چون یک action تلقی می‌شه پس ... .
(البته شاید این حساسیت بیجاست و به قول دوست عزیز 'محسن خان' چون این سری مقدماتی است به این صورت نوشته شده است)

نویسنده: آرمان فرقانی
تاریخ: ۱۳۹۲/۰۲/۳۰ ۱۹:۵

ضمن تشکر از دوستانی که در بحث شرکت کردند و پوزش به دلیل اینکه چند هفته ای در سفر هستم و تهیه مطالب با تأخیر انجام خواهد شد.
برای پاسخ به پرسش دوست گرامی آقای نجف زاده ابتدا بخشی از این مطلب را یادآوری می‌کنم.
"... مساحت را این بار به جای متد به صورت یک پروپرتی پیاده سازی می‌کنیم. اگرچه در آینده بیشتر راجع به چگونگی انتخاب برای پیاده سازی یک عضو کلاس به صورت پروپرتی یا متد بحث خواهیم کرد اما به عنوان یک قانون کلی در نظر داشته باشید عضوی که به صورت منطقی به عنوان داده مطرح است را به صورت پروپرتی پیاده سازی کنید. مانند نام دانشجو. از طرفی اعضایی که دلالت بر انجام عملی دارند را به صورت متد پیاده سازی می‌کنیم. مانند متد تبدیل به نوع داده دیگر. (مثلاً Object.ToString())
..."

بنابراین به نکات زیر توجه فرمایید.
۱. در این مطالب سعی شده است امکان پیاده سازی یک مفهوم به دو صورت متد و پروپرتی نشان داده شود تا در ذهن خواننده زمینه ای برای بررسی بیشتر مفهوم متد و پروپرتی و تفاوت آن‌ها فراهم گردد. این زمینه برای کنجکاوی بیشتر معمولاً با انجام یک جستجوی ساده سبب توسعه و تثبیت علم شخص می‌گردد.
۲. در متن بالاً به صورت کلی اشاره شده است هر یک از دو مفهوم متد و پروپرتی در کجا باید استفاده شوند و نیز خاطرنشان شده است در مطالب بعدی در مورد این موضوع بیشتر صحبت خواهد شد.
۳. نکته مهم در طراحی کلاس، پایگاه داده و ... خرد جهان واقع یا محیط عملیاتی مورد نظر طراح است. به عبارت دیگر گسی نمی‌تواند به یک طراح بگوید به طور مثال مساحت باید متد باشد یا باید پروپرتی باشد. طراح با توجه به مفهوم و کارکردی که برای هر مورد در ذهن دارد بر اساس اصول و قواعد، متد یا پروپرتی را بر می‌گزیند. مثلاً در خرد جهان واقع موجود در ذهن یک طراح مساحت به عنوان یک عمل یا اکشنی که شیء انجام می‌دهد است و بنابراین متد را انتخاب می‌کند. طراح دیگری در خرد جهان واقع دیگری در حال طراحی است و مثلاً متراژ یک شیء خانه را به عنوان یک ویژگی ذاتی و داده ای می‌نگرد و گمان می‌کند خانه نیازی به انجام عملی برای بدست آوردن مساحت خود ندارد بلکه یکی از ویژگی‌های خود را می‌تواند به اطلاع استفاده کننده برساند. پس شما به طراح دیگر نگویید اکشن تلقی میشه پس باید متد استفاده شود. اگر خود در پروژه ای چیزی را اکشن تلقی نمودید بله باید متد به کار ببرید. تلقی‌ها بر اساس خرد جهان واقع معنا دارند.
۴. پروپرتی و متد از نظر شیوه استفاده و ... با هم تفاوت دارند. اما یک تفاوت مهم بین آنها بیان نوع مفاهیم موجود در ذهن طراح به کد مشتری است. فراموش نکنید خود پروپرتی دارای اکسسور است که چیزی مانند متد است. در خیلی از موارد صحیح‌تر بودن پیاده سازی با متد یا با پروپرتی معنا ندارد. انتخاب ما بین متد یا پروپرتی بر اساس نحوه استفاده مطلوب در کد مشتری و نیز اطلاع به مشتری که مثلاً فلان مفهوم از دید ما یک اکشن است و فلان چیز داده صورت می‌گیرد.

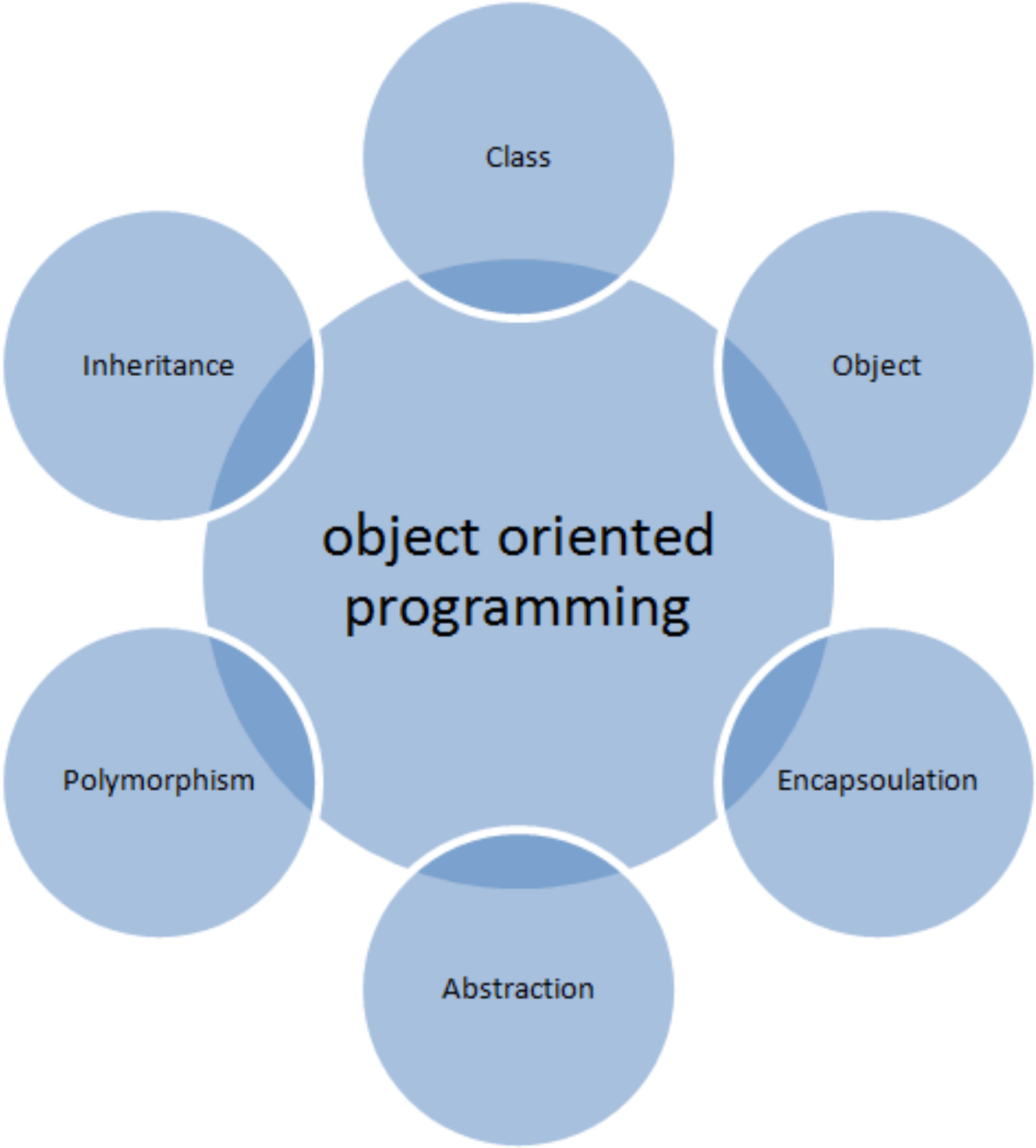
نویسنده: محسن نجف زاده
تاریخ: ۱۳۹۲/۰۳/۲۹ ۸:۹

با تشکر از آرمان فرقانی عزیز / مطلب و بحث مفید بود.

عنوان: **اصول طراحی شی گرا SOLID - #بخش اول اصل SRP**
نویسنده: ناصر طاهری
تاریخ: ۱۳۹۲/۰۷/۰۶ ۲۰:۳۰
آدرس: www.dotnettips.info
گروه‌ها: OOP, SOLID Principals, Object Oriented Design, Single Responsibility

مخاطب چه کسی است؟

این مقاله برای کسانی در نظر گرفته شده است که حداقل پیش زمینه ای در مورد برنامه نویسی شی گرا داشته باشند.کسانی که تفاوت بین کلاس‌ها و اشیاء را میدانند و میتوانند در مورد ارکان پایه ای برنامه نویسی شی گرایی نظیر : کپسوله سازی (Encapsulation) ، کلاس‌های انتزاعی (Abstraction) ، چند ریختی (Polymorphism) ، ارث بری (Inheritance) و... صحبت کنند.



مقدمه :

در جهان شی گرا ما فقط اشیاء را میبینیم که با یکدیگر در ارتباط هستند.کلاس ها، شی ها، ارث بری، کپسوله سازی ، کلاس‌های انتزاعی و ... کلماتی هستند که ما هر روز در حرفه‌ی خودمان بارها آنها را میشنویم.

در دنیای مدرن نرم افزار، بدون شک هر توسعه دهنده‌ی نرم افزار ،یکی از انواع زبان‌های شی گرا را برای استفاده انتخاب میکند. اما آیا او واقعا میداند که برنامه نویسی شی گرا به چه معنی است؟ آیا او واقعا از قدرت شی گرایی استفاده میکند؟

در این مقاله تصمیم گرفته ایم پای خود را فراتر از ارکان پایه ای برنامه نویسی شی گرا قرار دهیم و بیشتر در مورد طراحی شی گرا صحبت کنیم. **طراحی شی گرا :**

طراحی شی گرا یک فرایند از برنامه ریزی یک سیستم نرم افزاری است که در آن اشیاء برای حل مشکلات خاص با یکدیگر در ارتباط هستند. در حقیقت یک طراحی شی گرای مناسب ، کار توسعه دهنده را آسان میکند و یک طراحی نامناسب تبدیل به یک

فاجعه برای او میشود. **هر کسی چگونه شروع میکند؟**

وقتی کسانی شروع به ایجاد معماری نرم افزار میکنند، نشان میدهند که اهداف خوبی در سر دارند.آنها سعی میکنند از تجارب خود برای ساخت یک طراحی زیبا و تمیز استفاده کنند.

اما با گذشت زمان، نرم افزار کارایی خود را از دست میدهد و بلااستفاده میشود. با هر درخواست ایجاد ویژگی جدید در نرم افزار ، به تدریج نرم افزار شکل خود را از دست میدهد و درنهایت ساده‌ترین تغییرات در نرم افزار موجب تلاش و دقت زیاد، زمان طولانی و مهمتر از همه بالا رفتن تعداد باگها در نرم افزار میشود.

چه کسی مقصر است؟

"تغییرات" یک قسمت جدایی ناپذیر از جهان نرم افزار هستند بنابراین ما نمتوانیم "تغییر" را مقصر بدانیم و در حقیقت این طراحی ما است که مشکل دارد.

یکی از بزرگترین دلایل مخرب کننده‌ی نرم افزار، تعریف وابستگی‌های ناخواسته و بیخود در قسمت‌های مختلف سیستم است. در این گونه طراحی‌ها ، هر قسمت از سیستم وابسته به چندین قسمت دیگر است ، بنابراین تغییر یک قسمت، بر روی قسمت‌های دیگر نیز تاثیر میگذارد و باعث این چنین مشکلاتی میشود. ولی در صورتی که ما قادر به مدیریت این وابستگی باشیم در آینده خواهیم توانست از این سیستم نرم افزاری به آسانی نگهداری کنیم.

مثال:

راه حل : اصول ، الگوهای طراحی و معماری نرم افزار معماری نرم افزار به عنوان مثال MVC, MVP, 3-Tire به ما میگویند که پروژه‌ها از از چه ساختاری استفاده میکنند. **الگوهای طراحی** یک سری راه حل‌های قابل استفاده‌ی مجدد را برای مسائلی که به طور معمول اتفاق می‌افتند، فراهم میکند. یا به عبارتی دیگر الگوهای طراحی راه کارهایی را به ما معرفی میکنند که میتوانند برای حل مشکلات کد نویسی بارها مورد استفاده قرار بگیرند. **اصول به ما میگوید** اینها را انجام بده تا به آن دست پیدا کنی واینکه چطور انجامش میدهی به خودت بستگی دارد. هر کس یک سری اصول را در زندگی خود تعریف میکند مانند : "من هرگز دروغ نمیگویم" یا "من هرگز سیگار نمیکشم" و از این قبیل. او با دنبال کردن این اصول زندگی آسانی را برای خودش ایجاد میکند. به همین شکل، طراحی شی گرا هم مملو است از اصولی که به ما اجازه میدهد تا با طراحی مناسب مشکلاتمان را مدیریت کنیم.

آقای رابرت مارتین(Robert Martin) این موارد را به صورت زیر طبقه بندی کرده است :

1- اصول طراحی کلاس‌ها که SOLID نامیده می‌شوند.

2- اصول انسجام بسته بندی

3- اصول اتصال بسته بندی

در این مقاله ما در مورد اصول SOLID به همراه مثال‌های کاربردی صحبت خواهیم کرد.

SOLID مخففی از 5 اصول معرفی شده توسط آقای مارتین است:

```
S -> Single responsibility Principle
```

```
O-> Open Close Principle
```

```
L-> Liskov substitution principle
```

```
I -> Interface Segregation principle
```

```
D-> Dependency Inversion principle
```

اصل 1) S - SRP - Single responsibility Principle

به کد زیر توجه کنید :

```
public class Employee
{
    public string EmployeeName { get; set; }
    public int EmployeeNo { get; set; }

    public void Insert(Employee e)
    {
        //Database Logic written here
    }
    public void GenerateReport(Employee e)
    {
        //Set report formatting
    }
}
```

در کد بالا هر زمان تغییری در یک قسمت از کد ایجاد شود این احتمال وجود دارد که قسمت دیگری از آن مورد تاثیر این تغییر قرار بگیرد و به مشکل برخورد کنید. دلیل نیز مشخص است : هر دو در یک خانه‌ی مشابه و دارای یک والد یکسان هستند.

برای مثال با تغییر یک پراپرتی ممکن است متدهای هم خانه که از آن استفاده میکنند با مشکل مواجه شوند و باید این تغییرات را نیز در آنها انجام داد. در هر صورت خیلی مشکل است که همه چیز را کنترل کنیم. بنابراین تنها تغییر موجب دوبرابر شدن عملیات تست میشود و شاید بیشتر.

اصل **SRP** برای رفع این مشکل میگوید **"هر ماژول نرم افزاری میبایست تنها یک دلیل برای تغییر داشته باشد"** .

(منظور از ماژول نرم افزاری همان کلاس‌ها ، توابع و ... است و عبارت "دلیل برای تغییر" همان مسئولیت است.) به عبارتی هر شی باید یک مسئولیت بیشتر بر عهده نداشته باشد. هدف این قانون جدا سازی مسئولیت-های چسبیده به هم است. به عنوان مثال کلاسی که هم مسئول ذخیره سازی و هم مسئول ارتباط با واسط کاربر است، این اصل را نقض می-کند و باید به دو کلاس مجزا تقسیم شود.

برای رسیدن به این منظور میتوانیم مثال بالا را به صورت 3 کلاس مختلف ایجاد کنیم :

1- Employee : که حاوی خاصیت‌ها است.

2- EmployeeDB : عملیات دیتابیسی نظیر درج رکورد و واکشی رکوردها از دیتابیس را انجام میدهد.

3- EmployeeReport : وظایف مربوط به ایجاد گزارش‌ها را انجام میدهد.

کد حاصل :

```
public class Employee
{
    public string EmployeeName { get; set; }
    public int EmployeeNo { get; set; }
}

public class EmployeeDB
{
    public void Insert(Employee e)
    {
        //Database Logic written here
    }
    public Employee Select()
    {
        //Database Logic written here
    }
}

public class EmployeeReport
{
    public void GenerateReport(Employee e)
```

```
    {
        //Set report formatting
    }
}
```

این روش برای متدها نیز صدق میکند به طوری که هر متد باید مسئولیت واحدی داشته باشد.
برای مثال قطعه کد زیر اصل SRP را نقض میکند :

```
//Method with multiple responsibilities – violating SRP
public void Insert(Employee e)
{
    string StrConnectionString = "";
    SqlConnection objCon = new SqlConnection(StrConnectionString);
    SqlParameter[] SomeParameters=null;//Create Parameter array from values
    SqlCommand objCommand = new SqlCommand("InertQuery", objCon);
    objCommand.Parameters.AddRange(SomeParameters);
    ObjCommand.ExecuteNonQuery();
}
```

این متد وظایف مختلفی را انجام میدهد مانند اتصال به دیتابیس ، ایجاد پارامترها برای مقادیر، ایجاد کوئری و در نهایت اجرای آن بر روی دیتابیس.
اما با توجه به اصل SRP میتوان آن را به صورت زیر بازنویسی کرد :

```
//Method with single responsibility – follow SRP
public void Insert(Employee e)
{
    SqlConnection objCon = GetConnection();
    SqlParameter[] SomeParameters=GetParameters();
    SqlCommand ObjCommand = GetCommand(objCon,"InertQuery",SomeParameters);
    ObjCommand.ExecuteNonQuery();
}

private SqlCommand GetCommand(SqlConnection objCon, string InsertQuery, SqlParameter[] SomeParameters)
{
    SqlCommand objCommand = new SqlCommand(InsertQuery, objCon);
    objCommand.Parameters.AddRange(SomeParameters);
    return objCommand;
}

private SqlParameter[] GetParaeters()
{
    //Create Paramter array from values
}

private SqlConnection GetConnection()
{
    string StrConnectionString = "";
    return new SqlConnection(StrConnectionString);
}
```

نظرات خوانندگان

نویسنده: حسن دهیاری
تاریخ: ۱۳۹۲/۰۷/۲۳ ۲۰:۲۷

با سلام.ضمن تشکر.یک سوال داشتم.آیا با توجه به اصل SRP نباید کلاس EmployeeDB تفکیک شود.چون که دو کار را انجام میدهد.ممنون میشم توضیح دهید.

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۰۷/۲۳ ۲۳:۲۶

تنها دلیل تغییر کلی این کلاس در آینده، تغییر خاصیت‌های شیء کارمند است. بنابراین اصل تک مسئولیتی را نقض نمی‌کند. اگر این کلاس برای مثال دو Select داشت که یکی لیست کارمندان و دیگری لیست نقش‌های سیستم را بازگشت می‌داد، در این حالت تک مسئولیتی نقض می‌شد. ضمنا این نوع طراحی تحت عنوان الگوی مخزن یا لایه سرویس و امثال آن، یک طراحی پذیرفته شده و عمومی است. اگر قصد دارید که کوئری‌های خاص آن‌را طبقه بندی کنید می‌شود مثلا از Specification pattern استفاده کرد.

نویسنده: فراز
تاریخ: ۱۳۹۳/۰۵/۱۹ ۰:۴۹

با سلام ممنون از مقاله ای که گذاشتین من 3 سال هست تقریبا به صورت دست پا شکسته دنبال OOAD می‌گردم که در عمل توضیح داده باشد که شما این کار رو انجام دادین با سپاس فراوان

عنوان: **اصول طراحی شی گرا SOLID - #بخش دوم اصل OCP**
نویسنده: ناصر طاهری
تاریخ: ۱۳۹۲/۰۷/۰۶ ۲۰:۳۵
آدرس: www.dotnettips.info
برچسب‌ها: OOP, SOLID Principals, Object Oriented Design, Open Closed

در قسمت قبل در مورد اصل Single responsibility Principle یا به اختصار SRP صحبت شد. در این قسمت قصد داریم اصل دوم از اصول SOLID را مورد بررسی قرار دهیم.

اصل 2) O - OCP – Open Close Principle

فرض میکنیم که شما میخواهید یک طبقه بین طبقه‌ی اول و دوم خانه‌ی 2 طبقه‌ی خود اضافه کنید. فکرمیکنید امکان پذیر است؟

راه حل هایی که ممکن است به ذهن شما خطور کنند :

-1 زمانی که برای اولین بار در حال ساخت خانه هستید آن را 3 طبقه بسازید و طبقه‌ی وسط را خالی نگه دارید.اینطوری هر زمان که شما بخواهید میتوانید از آن استفاده کنید. به هر حال این هم یک راه حل است.

-2 خراب کردن طبقه دوم و ساخت دو طبقه‌ی جدید که خوب اصلا معقول نیست.

کد زیر را مشاهده کنید :

```
public class EmployeeDB
{
    public void Insert(Employee e)
    {
        //Database Logic written here
    }
    public Employee Select()
    {
        //Database Logic written here
    }
}
```

متد Select در کلاس EmployeeDB توسط کاربران و مدیر مورد استفاده قرار میگیرد. در این بین ممکن است مدیر نیاز داشته باشد تغییراتی را در آن انجام دهد. اگر مدیر این کار را برای برآورده کردن نیاز خود انجام دهد ،روی دیگر قسمت‌ها نیز تاثیر میگذارد، به علاوه ایجاد تغییرات در راه حل‌های تست شده‌ی موجود ممکن است موجب خطاهای غیر منتظره ای شود. چگونه ممکن است که رفتار یک برنامه تغییر کند بدون اینکه کد آن ویرایش شود؟ چگونه می‌توانیم بدون تغییر یک موجودیت نرم افزاری کارکرد آن را تغییر دهیم؟

اصل **OCP** میگوید : **"ماژول‌های نرم افزار باید برای تغییرات بسته و برای توسعه باز باشند."**

راه حل هایی که OCP را نقض نمیکنند : **1- استفاده از وراثت (inheritance):**

ایجاد یک کلاس جدید به نام EmployeeManagerDB که از کلاس EmployeeDB ارث بری کند و متد Select آن را جهت نیاز خود بازنویسی کند.

```
public class EmployeeDB
{
    public virtual Employee Select()
    {
        //Old Select Method
    }
}
public class EmployeeManagerDB : EmployeeDB
{
    public override Employee Select()
    {
        //Select method as per Manager
        //UI requirement
    }
}
```

این انتخاب خیلی خوبی است در صورتی که این تغییرات در زمان طراحی اولیه پیش بینی شده باشد و همکنون قابل استفاده باشند.

کد UI هم به شکل زیر خواهد بود :

```
//Normal Screen
EmployeeDB objEmpDb = new EmployeeDB();
Employee objEmp = objEmpDb.Select();

//Manager Screen
EmployeeDB objEmpDb = new EmployeeManagerDB();
Employee objEmp = objEmpDb.Select();
```

2- متدهای الحاقی (Extension Method):

اگر شما از NET 3.5. یا بالاتر از آن استفاده میکنید، دومین راه استفاده از متدهای الحاقی است که به شما اجازه میدهد بدون هیچ دست زدنی به نوع‌های موجود، متدهای جدیدی را به آنها اضافه کنید.

```
Public static class MyExtensionMethod{
    public static Employee managerSelect(this EmployeeDB employeeDB) {
//Select method as per Manager
    }
}

//Manager Screen
Employee objEmp = EmployeeDB.managerSelect();
```

البته ممکن است راه‌های دیگری هم برای رسیدن به این منظور وجود داشته باشد.

درقسمت‌های بعدی قانون‌های دیگر را بررسی خواهیم کرد.

عنوان: **اصول طراحی شی گرا SOLID - #بخش سوم اصل LSP**
نویسنده: ناصر طاهری
تاریخ: ۱۳۹۲/۰۷/۰۶ ۲۰:۴۰
آدرس: www.dotnettips.info
برچسب‌ها: OOP, SOLID Principals, Object Oriented Design, Liskov substitution

بخش‌های پیشین : اصول طراحی شی گرا SOLID - #بخش اول اصل SRP
اصول طراحی شی گرا SOLID - #بخش دوم اصل OCP

اصل 3 (L – LSP – Liskov substitution principle

اصل **LSP** میگوید : **"زیر کلاس‌ها باید بتوانند جایگزین نوع پایه‌ی خود باشند".**

مقایسه با جهان واقعی :

شغل یک پدر تجارت املاک است درحالی که پسرش دوست دارد فوتبالیست شود.
یک پسر هیچگاه نمیتواند جایگزین پدرش شود، با اینکه که آنها به یک سلسله مراتب خانوادگی تعلق دارند.
در یک مثال عمومی‌تر بررسی میکنیم :
به طور معمول زمانی که ما در مورد اشکال هندسی صحبت میکنیم ، مستطیل را یک کلاس پایه برای مربع میدانیم. به کد زیر توجه کنید :

```
public class Rectangle
{
    public int Width { get; set; }
    public int Height { get; set; }
}

public class Square:Rectangle
{
    //codes specific to
    //square will be added
}
```

و میتوان گفت :

```
Rectangle o = new Rectangle();
o.Width = 5;
o.Height = 6;
```

بسیار خوب، اما با توجه به LSP باید قادر باشیم مستطیل را با مربع جایگزین کنیم. سعی میکنیم این کار را انجام دهیم :

```
Rectangle o = new Square();
o.Width = 5;
o.Height = 6;
```

موضوع چیست؟ مگر مربع می‌تواند طول و عرض نا برابر داشته باشد؟! امکان ندارد.
خوب این به چه معنی است؟ به این معنی که ما نمیتوانیم کلاس پایه را با کلاس مشتق شده جایگزین کنیم و باز هم این معنی را میدهد که ما داریم اصل LSP را نقض میکنیم.
آیا ما میتوانیم طول و عرض را در کلاس Square طبق کد زیر دوباره نویسی کنیم؟

```
public class Square : Rectangle
{
    public override int Width
    {
        get{return base.Width;}
        set
        {
            base.Height = value;
            base.Width = value;
        }
    }
    public override int Height
    {
        get{return base.Height;}
        set
        {
            base.Height = value;
            base.Width = value;
        }
    }
}
```

باز هم اصل LSP نقض میشود چون ما داریم رفتار خاصیت‌های طول و عرض در کلاس مشتق شده را تغییر میدهیم. ولی با توجه به کد بالا یک مستطیل نمیتواند طول و عرض برابر داشته باشد چون در صورت برابری دیگر مستطیل نیست.

اما راه حل چیست؟

یک کلاس انتزاعی (abstract) را به شکل زیر ایجاد و سپس دوکلاس Square و Rectangle را از آن مشتق میکنیم :

```
public abstract class Shape
{
    public virtual int Width { get; set; }
    public virtual int Height { get; set; }
}
```

همکنون ما دو کلاس مستقل از هم داریم. یکی Square و دیگری Rectangle که هر دو از کلاس Shape مشتق شده اند.

حالا میتوانیم بنویسیم :

```
Shape o = new Rectangle();
o.Width = 5;
o.Height = 6;

Shape o = new Square();
o.Width = 5; //both height and width become 5
o.Height = 6; //both height and width become 6
```

زمانی که ما در مورد اشکال هندسی صحبت میکنیم ، هیچ قاعده‌ی خاصی جهت اندازه‌ی طول و عرض نیست. ممکن است برابر باشند یا نباشند.

در قسمت بعدی اصل ISP را مورد بررسی قرار خواهیم داد.

نظرات خوانندگان

نویسنده: فدورا
تاریخ: ۱۳۹۲/۰۷/۰۹ ۱۰:۳۸

سلام.
خیلی ممنون از بابت مقالات آموزشی خوبتون.
فقط سوالی برای من تو این بخش سوم پیش اومد و اون هم اینکه بعد از تعریف کلاس abstract تعریف کلاس‌های rectangle و square به چه شکل شد؟ لطفا کد اون کلاس‌ها رو هم اضافه کنید.
با تشکر

نویسنده: ناصر طاهری
تاریخ: ۱۳۹۲/۰۷/۰۹ ۱۳:۴۸

ممنون.
کلاس های Rectangle و Square هر دو به همون شکل باقی میمونند با این تفاوت که هر دو از کلاس Shape مشتق شده اند و میتوانند خاصیت‌های Width و Height را طبق نیاز خود دوباره نویسی کنند (override). کلاس Restangle:

```
public class Rectangle : Shape
{
    //شما میتوانید خاصیت‌ها طول و عرض در کلاس پایه را در صورت نیاز دوباره نویسی کنید
}
```

کلاس Square :

```
public class Square : Shape
{
    //دوباره نویسی کردن خاصیت‌های طول و عرض در کلاس پایه جهت برابر کردن طول و عرض مربع
    public override int Width
    {
        get{return base.Width;}
        set
        {
            base.Height = value;
            base.Width = value;
        }
    }
    public override int Height
    {
        get{return base.Height;}
        set
        {
            base.Height = value;
            base.Width = value;
        }
    }
}
```

که با توجه به کدهای بالا ، کلاسهای مشتق شده‌ی Square و Restangle میتوانند جایگزین کلاس پایه خود یعنی Shape شوند :

```
Shape o = new Rectangle();
o.Width = 5;
o.Height = 6;

Shape o = new Square();
o.Width = 5; //طول و عرض هر دو برابر 5 میشوند
o.Height = 6; //عرض و طول هر دو برابر 6 میشوند
```

عنوان: **اصول طراحی شی گرا SOLID - #بخش چهارم اصل ISP**
نویسنده: ناصر طاهری
تاریخ: ۱۳۹۲/۰۷/۰۶ ۲۰:۴۵
آدرس: www.dotnettips.info
برچسب‌ها: OOP, SOLID Principals, Object Oriented Design, Interface Segregation

بخش‌های پیشین : اصول طراحی شی گرا SOLID - #بخش اول اصل SRP
اصول طراحی شی گرا SOLID - #بخش دوم اصل OCP اصول طراحی شی گرا SOLID - #بخش سوم اصل LSP

اصل 4) I – ISP- Interface Segregation principle

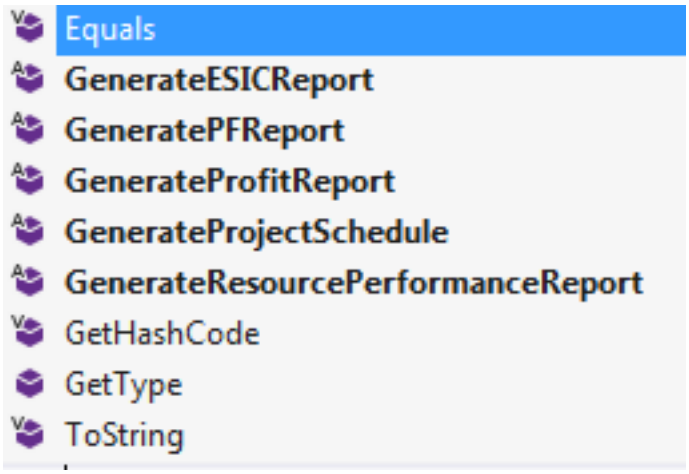
مقایسه با دنیای واقعی:

بیایید فکر کنیم شما یک کامپیوتر دسکتاپ جدید خریداری کرده اید.شما یک زوج پورت USP، چند پورت سریال، یک پورت VGA و ... را پیدا میکنید. اگر شما بدنه‌ی کیس خود را باز کنید، تعدادی اسلات بر روی مادربرد خود که برای اتصال قطعات مختلف با یکدیگر هستند را مشاهده خواهید کرد که عمدتا مهندسان سخت افزار در زمان منتاژ از آنها استفاده میکنند.
این اسلات‌ها تا زمانی که بدنه کیس را باز نکنید قابل مشاهده نخواهند بود. به طور خلاصه تنها رابطه‌های مورد نیازی که برای شما ساخته شده اند، قابل مشاهده خواهند بود.
و فرض کنید شما به یک توپ فوتبال نیاز دارید. به یک فروشگاه وسایل ورزشی میروید و فروشنده شروع به نشان دادن انواع توپ ها، کفش ها،لباس و گرم کن‌های ورزشی،لباس شنا و کاراته ،زانوبند و ... کرده است. در نهایت ممکن است شما چیزی را خریداری کنید که اصلا مورد نیازتان نبوده است یا حتی ممکن است فراموش کنیم که ما چرا اینجا هستیم.! **ارائه مشکل:**
ما میخواهیم یک سیستم مدیریت گزارشات را توسعه بدهیم. اولین کاری که انجام می‌دهیم ایجاد یک لایه منطقی که توسط سه UI مختلف دیگر مورد استفاده قرار میگیرد.
1- EmployeeUI : نمایش گزارش‌های مربوط با کارمند وارد شده‌ی جاری در سایت.
2- ManagerUI : نمایش گزارش‌های مربوط به خود و تیمی که به او تعلق دارد.
3- AdminUI : نمایش گزارش‌های مربوط به کارمندان، مربوط به تیم و مربوط به شرکت مانند گزارش سود و زیان و ...

```
public interface IReportBAL
{
    void GeneratePFReport();
    void GenerateESICReport();

    void GenerateResourcePerformanceReport();
    void GenerateProjectSchedule();

    void GenerateProfitReport();
}
```

```
public class ReportBAL : IReportBAL
{
    public void GeneratePFReport()
    {/*..............*/}

    public void GenerateESICReport()
    {/*..............*/}

    public void GenerateResourcePerformanceReport()
    {/*..............*/}

    public void GenerateProjectSchedule()
    {/*..............*/}

    public void GenerateProfitReport()
    {/*..............*/}
}
public class EmployeeUI
{
    public void DisplayUI()
    {
        IReportBAL objBal = new ReportBAL();
        objBal.GenerateESICReport();
        objBal.GeneratePFReport();
    }
}
public class ManagerUI
{
    public void DisplayUI()
    {
        IReportBAL objBal = new ReportBAL();
        objBal.GenerateESICReport();
        objBal.GeneratePFReport();
        objBal.GenerateResourcePerformanceReport ();
        objBal.GenerateProjectSchedule ();
    }
}
public class AdminUI
{
    public void DisplayUI()
    {
        IReportBAL objBal = new ReportBAL();
        objBal.GenerateESICReport();
        objBal.GeneratePFReport();
        objBal.GenerateResourcePerformanceReport();
        objBal.GenerateProjectSchedule();
        objBal.GenerateProfitReport();
    }
}
```

حال زمانی که توسعه دهنده در هر UI عبارت objBal را تایپ کند، در لیست بازشوی آن (intellisense) به صورت زیر نمایش داده خواهد شد :

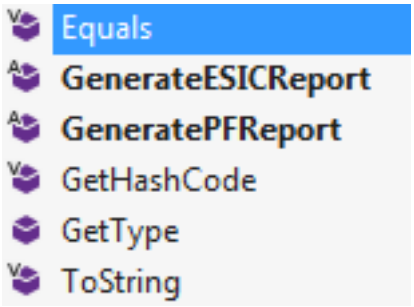


اما مشکل چیست؟ مشکل این است شخصی که روی لایه‌ی EmployeeUI کار میکند به تمام متدهاها به خوبی دسترسی دارد و این متدهای غیرضروری ممکن است باعث سردرگمی او شود.
این مشکل به این دلیل اتفاق می‌افتد که وقتی کلاسی یک واسط را پیاده سازی می‌کند، باید همه متدهای آن را نیز پیاده سازی کند. حالا اگر خیلی از این متدها توسط این کلاس استفاده نشود، می‌گوییم که این کلاس از مشکل واسط چاق رنج می‌برد.
اصل **ISP** میگوید : **" کلاینت‌ها نباید وابسته به متدهایی باشند که آنها را پیاده سازی نمی‌کنند. "**
برای رسیدن به این امر در مثال بالا باید آن واسط را به واسطهای کوچکتر تقسیم کرد. این تقسیم بندی باید بر اساس استفاده کنندگان از واسطها صورت گیرد.
ویرایش مثال بالا با در نظر گرفتن اصل ISP :

```
public interface IEmployeeReportBAL
{
    void GeneratePFReport();
    void GenerateESICReport();
}
public interface IManagerReportBAL : IEmployeeReportBAL
{
    void GenerateResourcePerformanceReport();
    void GenerateProjectSchedule();
}
public interface IAdminReportBAL : IManagerReportBAL
{
    void GenerateProfitReport();
}
public class ReportBAL : IAdminReportBAL
{
    public void GeneratePFReport()
    {/*..............*/}

    public void GenerateESICReport()
    {/*..............*/}

    public void GenerateResourcePerformanceReport()
    {/*..............*/}

    public void GenerateProjectSchedule()
    {/*..............*/}

    public void GenerateProfitReport()
    {/*..............*/}
}
```

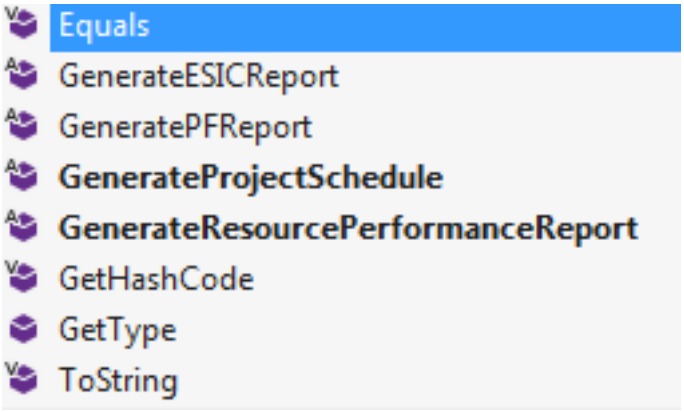
وضعیت نمایش :

```
public class EmployeeUI
{
    public void DisplayUI()
    {
        IEmployeeReportBAL objBal = new ReportBAL();
        objBal.GenerateESICReport();
        objBal.GeneratePFReport();
    }
}
```

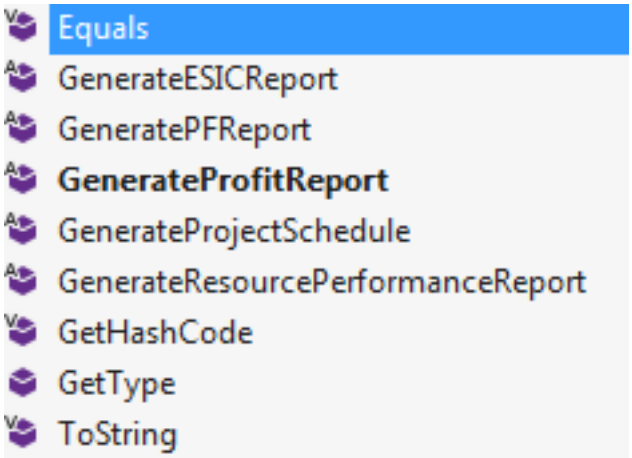



```
public class ManagerUI
{
    public void DisplayUI()
```

```
    {
        IManagerReportBAL  objBal = new ReportBAL();
        objBal.GenerateESICReport();
        objBal.GeneratePFReport();
        objBal.GenerateResourcePerformanceReport ();
        objBal.GenerateProjectSchedule ();
    }
}
```




```
public class AdminUI
{
    public void DisplayUI()
    {
        IAdminReportBAL  objBal = new ReportBAL();
        objBal.GenerateESICReport();
        objBal.GeneratePFReport();
        objBal.GenerateResourcePerformanceReport();
        objBal.GenerateProjectSchedule();
        objBal.GenerateProfitReport();
    }
}
```




```
DIP   :
```

در قسمت بعدی آخرین اصل

را بررسی خواهیم کرد

عنوان: **اصول طراحی شی گرا SOLID - #بخش پنجم اصل DIP**
نویسنده: ناصر طاهری
تاریخ: ۱۳۹۲/۰۷/۰۶ ۲۰:۵۰
آدرس: www.dotnettips.info
گروه‌ها: OOP, SOLID Principals, Dependency Inversion, Object Oriented Design

بخش‌های پیشین : اصول طراحی شی گرا SOLID - #بخش اول اصل SRP اصول طراحی شی گرا SOLID - #بخش دوم اصل OCP
اصول طراحی شی گرا SOLID - #بخش سوم اصل LSP
اصول طراحی شی گرا SOLID - #بخش چهارم اصل ISP

اصل 5) D – DIP– Dependency Inversion principle

مقایسه با دنیای واقعی:

همان مثال کامپیوتر را دوباره در نظر بگیرید.این کامپیوتر دارای قطعات مختلفی مانند RAM ، هارد دیسک، CD ROM و ... است که هر کدام به صورت مستقل به مادربرد متصل شده اند. این به این معنی است که اگر قسمتی از کار بیفتد میتوان آن را با یک قطعه‌ی جدید به آسانی تعویض کرد . حالا فقط تصور کنید که تمامی قطعات شدیداً به یکدیگر متصل شده اند آنوقت دیگر نمیتوانستیم قطعه ای را از مادربرد برداریم و به همین خاطر اگر مثلا RAM از کار بیفتد ، باید یک مادربرد جدید خریداری کنید که برای شما گران تمام می‌شود.

به مثال زیر توجه کنید :

```
public class CustomerBAL
{
    public void Insert(Customer c)
    {
        try
        {
            //Insert logic
        }
        catch (Exception e)
        {
            FileLogger f = new FileLogger();
            f.LogError(e);
        }
    }
}

public class FileLogger
{
    public void LogError(Exception e)
    {
        //Log Error in a physical file
    }
}
```

در کد بالا کلاسCustomerBAL مستقیما به کلاس FileLogger وابسته است که استثناءهای رخ داده را بر روی یک فایل فیزیکی لاگ میکند. حالا فرض کنید که چند روز بعد مدیریت تصمیم میگیرد که از این به بعد استثناءها بر روی یک Event Viewer لاگ شوند.
اکنون چه میکنید؟ با تغییر کدها ممکن است با خطاهای زیادی روبرو شوید(درصورت تعداد بالای کلاسهایی که از کلاس FileLogger استفاده میکنند و فقط تعداد محدودی از آنها نیاز دارند که بر روی Event Viewer لاگ کنند.) **DIP** به ما میگوید : **"ماژول‌های سطح بالا نباید به ماژولهای سطح پایین وابسته باشند، هر دو باید به انتزاعات وابسته باشند. انتزاعات نباید وابسته به جزئیات باشند، بلکه جزئیات باید وابسته به انتزاعات باشند. "**.

در طراحی ساخت یافته، ماژولهای سطح بالا به ماژولهای سطح پایین وابسته بودند. این مسئله دو مشکل ایجاد می‌کرد:
-1 ماژول‌های سطح بالا (سیاست گذار) به ماژول‌های سطح پایین (مجری) وابسته هستند. در نتیجه هر تغییری در ماژول‌های سطح پایین ممکن است باعث اشکال در ماژول‌های سطح بالا گردد.
-2 استفاده مجدد از ماژول‌های سطح بالا در جاهای دیگر مشکل است، زیرا وابستگی مستقیم به ماژول‌های سطح پایین دارند. **راه حل با توجه به اصل DIP :**

```
public interface ILogger
{
    void LogError(Exception e);
}

public class FileLogger:ILogger
```

```
{
    public void LogError(Exception e)
    {
        //Log Error in a physical file
    }
}
public class EventViewerLogger : ILogger
{
    public void LogError(Exception e)
    {
        //Log Error in a Event Viewer
    }
}
public class CustomerBAL
{
    private ILogger _objLogger;
    public CustomerBAL(ILogger objLogger)
    {
        _objLogger = objLogger;
    }

    public void Insert(Customer c)
    {
        try
        {
            //Insert logic
        }
        catch (Exception e)
        {
            _objLogger.LogError(e);
        }
    }
}
```

در اینجا وابستگی‌های کلاس CustomerBAL از طریق سازنده آن در اختیارش قرار گرفته است. یک اینترفیس ILogger تعریف شده‌است به همراه دو پیاده سازی مختلف از آن مانند FileLogger و EventViewerLogger.
یکی از انواع فراخوانی آن نیز می‌تواند به شکل زیر باشد:

```
var customerBAL = new CustomerBAL (new EventViewerLogger());
customerBAL.LogError();
```

اطلاعات بیشتر در دوره آموزشی " بررسی مفاهیم معکوس سازی وابستگی‌ها و ابزارهای مرتبط با آن ".

نظرات خوانندگان

نویسنده: سعید سلیمانی فر
تاریخ: ۱۳۹۲/۰۷/۰۹ ۹:۳۱

خیلی مطلب خوبی بود! لذت بردیم متشکرم :)

نویسنده: بهزاد علی محمدزاده
تاریخ: ۱۳۹۲/۰۷/۱۹ ۱۶:۲۲

اقای طاهری با تشکر . امکان داره منبع رو معرفی کنید . به دنبال یه کتاب یا منبع آموزشی خوب در این زمینه هستم که البته نمونه‌ها رو با C# انجام داده باشه .

نویسنده: ناصر طاهری
تاریخ: ۱۳۹۲/۰۷/۱۹ ۱۷:۴۱

چند مقاله ای که من اونها رو مطالعه کردم : اصول طراحی SOLID SOLIDify your software design concepts through SOLID Articles by Christian Vos Object Oriented Design Principles SOLID by example SOLID Agile Development Articles Of SOLID SOLID Principles in C# - An Overview

عنوان: **نشانه های طراحی ضعیف**
نویسنده: توسعه چابک
تاریخ: ۱۳۹۲/۰۹/۱۱ ۱۲:۲۵
آدرس: www.dotnettips.info
برچسب‌ها: Design, OOP

برای آنکه طراحی قوی و درست را یاد بگیریم، لازم است که نشانه-های طراحی ضعیف را بدانیم. این نشانه‌ها عبارتند از:

۱- Rigidity (انعطاف ناپذیری): یک ماژول انعطاف ناپذیر است، اگر یک تغییر در آن، منجر به تغییرات در سایر ماژولها گردد. هر چه میزان تغییرات آبشاری بیشتر باشد، نرم افزار خشک‌تر و غیر منعطف‌تر است.

۲- Fragility (شکنندگی): وقتی که تغییر در قسمتی از نرم افزار باعث به بروز اشکال در بخش-های دیگر شود.

۳- Immobility (تحرک ناپذیری): وقتی نتوان قسمت هایی از نرم افزار را در جاهای دیگر استفاده نمود و یا به کار گیری آن هزینه و ریسک بالایی داشته باشد.

۴- Viscosity (لزجی): وقتی حفظ طراحی اصولی پروژه مشکل باشد، می-گوییم پروژه لزج شده است. به عنوان مثال وقتی تغییری در پروژه به دو صورت اصولی و غیر اصولی قابل انجام باشد و روش غیر اصولی راحت‌تر باشد، می‌گوییم لزج شده است. البته لزجی محیط هم وجود دارد مثلا انجام کار به صورت اصولی نیاز به Build کل پروژه دارد که زیاد طول می-کشد.

۵- Needless Complexity (پیچیدگی اضافی): زمانی که امکانات بدون استفاده در نرم افزار قرار گیرند.

۶- Needless Repetition (تکرارهای اضافی): وقتی که کدهایی با منطق یکسان در جاهای مختلف برنامه کپی می-شوند، این مشکلات رخ می‌دهند.

۷- Opacity (ابهام): وقتی که فهمیدن یک ماژول سخت شود، رخ می‌دهد و کد برنامه مبهم بوده و قابل فهم نباشد.

چرا نرم افزار تمایل به پوسیدگی دارد؟
در روش‌های غیر چابک یکی از دلایل اصلی پوسیدگی، عدم تطابق نرم افزار با تغییرات درخواستی است. لازم است که این تغییرات به سرعت انجام شوند و ممکن است که توسعه دهندگان از طراحی ابتدایی اطلاعی نداشته باشند. با این حال ممکن است تغییرایی قابل انجام باشد ولی برخی از آنها طراحی اصلی را نقض می‌کنند. ما نباید تغییرات نیازمندیها را مقصر بدانیم. باید طراحی ما قابلیت تطبیق با تغییرات را داشته باشد.

یک تیم چابک از تغییرات استقبال می‌کند. وقت بسیار کمی را روی طراحی اولیه کل کار می‌گذارد و سعی می-کند که طراحی سیستم را تا جایی که ممکن است ساده و تمیز نگه دارد با استفاده از تست‌های واحد و یکپارچه از آن محافظت کند. این طراحی را انعطاف پذیر می‌کند. تیم از قابلیت انعطاف پذیری برای بهبود همیشگی طراحی استفاده میکند. بنابراین در هر تکرار نرم افزاری خواهیم داشت که نیازمندی‌های آن تکرار را برآورده می‌کند.

عنوان: **آشنایی با مفاهیم شیء گرایی در جاوا اسکریپت #1**
نویسنده: مهران ر
تاریخ: ۱۳۹۲/۱۰/۰۴ ۱۴:۳۰
آدرس: www.dotnettips.info
گروه‌ها: JavaScript, OOP

با توجه به فراگیر شدن استفاده از جاوا اسکریپت و بخصوص مبحث شیء گرایی، تصمیم گرفتم طی سلسله مقالاتی با مباحث شیء گرایی در این زبان بیشتر آشنا شویم. جاوا اسکریپت یک زبان مبتنی بر شیء است و نه شیءگرا و خصوصیات زبان‌های شیء گرا، به طور کامل در آن پیاده سازی نمی‌گردد.
لازم به ذکر است که انواع داده‌ای در جاوا اسکریپت شامل 2 نوع می‌باشند:
-1 نوع داده اولیه (Primitive) که شامل Number ، Boolean و Strings می‌باشند.
-2 نوع داده Object که طبق تعریف هر Object مجموعه‌ای از خواص و متدها است.
نوع داده‌ای اولیه، از نوع Value Type و نوع داده ای Object، از نوع Refrence Type می‌باشد.

برای تعریف یک شیء (Object) در جاوا اسکریپت، 3 راه وجود دارد:
1 - تعریف و ایجاد یک نمونه مستقیم از یک شیء (direct instance of an object)
2 – استفاده از function برای تعریف و سپس نمونه سازی از یک شیء (Object Constructor)
3 – استفاده از متد Object.Create

روش اول :
در روش اول دو راه برای ایجاد اشیاء استفاده می‌گردد که با استفاده از دو مثال ذیل، این دو روش توضیح داده شده‌اند:

مثال اول : (استفاده از new)

```
<script type=”text/javascript”>
 var person = new Object();
  person.firstname = “John”;
  person.lastname = “Doe”;
  person.age = 50;
  person.eyecolor = “blue”;
  document.write(person.firstname + “ is “ + person.age + “ years old.”);
</script>
```

```
result : John is 50 years old.
```

در این مورد، ابتدا یک شیء پایه ایجاد می‌گردد و خواص مورد نظر برایش تعریف می‌گردد و با استفاده از اسم شیء به این خواص دسترسی داریم.

مثال دوم (استفاده از literal notation)

```
<script type=”text/javascript”>
var obj = {
   var1: “text1”,
   var2: 5,
   Method: function ()
   {
     alert(this.var1);
   }
  };
  obj.Method();
</script>
```

```
Result : text1
```

در این مورد با استفاده از کلمه کلیدی var یک شیء تعریف می‌شود و در داخل {} کلیه خواص و متدهای این شیء تعریف می‌گردد. این روش برای تعریف اشیاء در جاوا اسکریپت بسیار متداول است.
هر دو مثالهای 1 و 2 در روش اول برای ایجاد اشیاء بکار میروند. امکان گسترش دادن اشیاء در این روش و اضافه کردن خواص و متد در آینده نیز وجود دارد. بعنوان مثال می‌توان نوشت :

```
Obj.var3 = “text3”;
```

در این‌حال، خاصیت سومی به مجموع خواص شی Obj اضافه می‌گردد.
حال در این مثال اگر مقدار شی obj را برابر یک شیء دیگر قرار دهیم به نحو زیر :

```
var newobj = obj;
  newobj.var1 = "other text";
  alert(obj.var1);// other text
  alert(newobj.var1);// other text
```

و برای اینکه بتوان از امکانات زبانهای شیء گرا در این زبان استفاده کرد، بایستی الگویی را تعریف کنیم و سپس از روی این الگو، اشیا مورد نظر را پیاده سازی نمائیم.
می‌بینیم که مقدار هر دو متغیر در خروجی یکسان می‌باشد و این موضوع با ماهیت شیء گرایی که در آن همه‌ی اشیایی که از روی یک الگو نمونه سازی می‌گردند مشخصه‌هایی یکسان، ولی مقادیر متفاوتی دارند، متفاوت است. البته این موضوع از آنجا ناشی می‌گردد که اشیاء ایجاد شده در جاوا اسکریپت ذاتا type refrence هستند و به همین منظور برای پیاده سازی الگویی (کلاسی) که بتوان رفتار شیء گرایی را از آن انتظار داشت از روش زیر استفاده میکنیم. برای درک بهتر اسم این الگو را کلاس مینامیم که در روش دوم به آن اشاره میکنیم.

روش دوم :

```
<script type=”text/javascript”>
  function Person(firstname, lastname, age, eyecolor)
  {
     this.firstname = firstname;
     this.lastname = lastname;
     this.age = age;
     this.eyecolor = eyecolor;
  }

   var myFather = new Person("John", "Doe", 50, "blue");
  document.write(myFather.firstname + " is " + myFather.age + " years old.");
  result : John is 50 years old.

   var myMother=new person("Sally","Rally",48,"green");
   document.write(myMother.firstname + " is " + myFather.age + " years old.");
   result : Sally is 48 years old.
  </script>
```

یا به شکل زیر :

```
var Person = function (firstname, lastname, age, eyecolor)
  {
     this.firstname = firstname;
     this.lastname = lastname;
     this.age = age;
     this.eyecolor = eyecolor;
  }

  var myFather = new Person("John", "Doe", 50, "blue");
  document.write(myFather.firstname + " is " + myFather.age + " years old.");
   result : John is 50 years old.
```

```
   var myMother=new person("Sally","Rally",48,"green");
   document.write(myMother.firstname + " is " + myFather.age + " years old.");
   result : Sally is 48 years old.
```

به این روش Object Constructor یا سازنده اشیاء گفته می‌شود.
در اینجا با استفاده از کلمه کلیدی function و در داخل {} کلیه خواص و متدهای لازم را به شیء مورد نظر اضافه می‌کنیم.
استفاده از کلمه this در داخل function به این معنی است که هر کدام از نمونه‌های object مورد نظر، مقادیر متفاوتی خواهند داشت .

یک مثال دیگر :

```
<script type="text/javascript">
   function cat(name) {
      this.name = name;
      this.talk = function() {
      alert( this.name + " say meeow!" )
   }
 }

cat1 = new cat("felix")
cat1.talk() //alerts "felix says meeow!"
cat2 = new cat("ginger")
cat2.talk() //alerts "ginger says meeow!"
</Script>
```

در اینجا می‌بینیم که به ازای هر نمونه از اشیایی که با function می‌سازیم، خروجی متفاوتی تولید می‌گردد که همان ماهیت شیء گرایی است.

روش سوم :استفاده از متد Object.Create

```
var myObjectLiteral = {
    property1: "one",
    property2: "two",
    method1: function() {
       alert("Hello world!");
}}
var myChild = Object.create(myObjectLiteral);
myChild.method1(); // will alert "Hello world!"
```

در این روش با استفاده از متد Object.Create و استفاده از یک شیء که از قبل ایجاد شده، یک شیء جدید ایجاد می‌شود.
حال برای اضافه کردن متدها و خاصیت‌هایی به کلاس جاوا اسکریپتی مورد نظر، به طوریکه همه‌ی نمونه‌هایی که از این کلاس ایجاد می‌شوند بتوانند به این متدها و خاصیت‌ها دسترسی داشته باشند، از مفهومی به اسم prototype استفاده می‌کنیم. برای مثال کلاس زیر را در نظر بگیرید:
این کلاس یک سیستم ساده امتحانی (quiz) را پیاده می‌کند که در آن اطلاعات شخص که شامل نام و ایمیل می‌باشد گرفته شده و سه تابع، شامل ذخیره نمرات، تغییر ایمیل و نمایش اطلاعات شخص به همراه نمرات نیز به آن اضافه می‌شود.

```
function User (theName, theEmail) {
    this.name = theName;
    this.email = theEmail;
    this.quizScores = [];
    this.currentScore = 0;
}
```

حال برای اضافه نمودن متدهای مختلف به این کلاس داریم :

```
User.prototype = {
  saveScore:function (theScoreToAdd)  {
   this.quizScores.push(theScoreToAdd)
  },
  showNameAndScores:function ()  {
    var scores = this.quizScores.length > 0 ? this.quizScores.join(",") : "No Scores Yet";
    return this.name + " Scores: " + scores;
  },
  changeEmail:function (newEmail)  {
    this.email = newEmail;
    return "New Email Saved: " + this.email;
  }
}
```

و سپس برای استفاده از آن و گرفتن خروجی نمونه داریم :

```
// A User
  firstUser = new User("Richard", "Richard@examnple.com");
  firstUser.changeEmail("RichardB@examnple.com");
  firstUser.saveScore(15);
  firstUser.saveScore(10);
  document.write(firstUser.showNameAndScores()); //Richard Scores: 15,10
  document.write('<br/>');
  // Another User
  secondUser = new User("Peter", "Peter@examnple.com");
  secondUser.saveScore(18);
  document.write(secondUser.showNameAndScores()); //Peter Scores: 18
```

در نتیجه تمام نمونه‌های کلاس User می‌توانند به این متدها دسترسی داشته باشند و به این صورت مفهوم Encapsulation نیز پیاده می‌گردد.

وراثت (Inheritance) در جاوا اسکریپت :

در بسیاری از مواقع لازم است عملکردی (Functionality) که در یک کلاس تعریف می‌گردد، در کلاسهای دیگر نیز در دسترس باشد. بدین منظور از مفهوم وراثت استفاده می‌شود. در نتیجه کلاس‌ها می‌توانند از توابع خود و همچنین توابعی که کلاسهای والد در اختیار آنها می‌گذارند استفاده کنند. برای این منظور چندین راه حل توسط توسعه دهندگان ایجاد شده است که در ادامه به چند نمونه از آنها اشاره می‌کنیم.
ساده‌ترین حالت ممکن از الگویی شبیه زیر است:

```
<script type="text/javascript">
function Base()
  {
     this.color = "blue";
  }
  function Sub()
  {
  }
  Sub.prototype = new Base();
  Sub.prototype.showColor = function ()
  {
     alert(this.color);
  }
  var instance = new Sub();
  instance.showColor(); //"blue"
</Script>
```

در کد بالا ابتدا یک class) function) به نام Base که حاوی یک خصوصیت به نام color می‌باشد، تعریف شده و سپس یک کلاس دیگر بنام sub تعریف می‌کنیم که قرار است خصوصیات و متدهای کلاس Base را به ارث ببرد و سپس از طریق خصوصیت prototype کلاس Sub، که نمونه‌ای از کلاس Base را به آن نسبت می‌دهیم باعث می‌شود خواص و متدهای کلاس Base توسط کلاس Sub قابل دسترسی باشد. در ادامه متد showColor را به کلاس Sub اضافه می‌کنیم و توسط آن به خصوصیت color در این کلاس دسترسی پیدا میکنیم.

راه حل دیگری نیز برای اینکار وجود دارد که الگویی است بنام Parasitic Combination :
در این الگو براحتی و با استفاده از متد Object.create که در بالا توضیح داده شد، هر کلاسی که ایجاد میکنیم، با انتساب آن به یک شیء جدید، کلیه خواص و متدهای آن نیز توسط شیء جدید قابل استفاده میشود.

```
<script language="javascript" type="text/javascript">
if (typeof Object.create !== 'function') {
Object.create = function (o) {
//ایجاد یک کلاس خالی که قرار است خواص کلاس دریافتی توسط آرگومان کلاس پایه را به ارث ببرد
  function F() {
}
با انتساب F  توسط خصوصیت  Prototype باعث میشویم کلیه خواص و متدهای دریافتی توسط  F با ارث برده شود
آرگومان دریافتی که یک شی است به کلاس
  F.prototype = o;
   return new F();
  };
}

var cars = {
type: "sedan",
wheels: 4
  };
  // We want to inherit from the cars object, so we do:
  var toyota = Object.create(cars);
// now toyota inherits the properties from cars
  document.write(toyota.type);
</script>
output :sedan
```

در قسمتهای دیگر به مباحثی همچون Override و CallBaseMethodها خواهیم پرداخت.

برای مطالعه بیشتر :

http://eloquentjavascript.net/chapter8.html

http://phrogz.net/JS/classes/OOPinJS2.html

عنوان: **آشنایی با مفاهیم شیء گرایی در جاوا اسکریپت #2**
نویسنده: مهران ر
تاریخ: ۱۳۹۲/۱۰/۰۸ ۱۸:۵
آدرس: www.dotnettips.info
گروه‌ها: JavaScript, OOP

از آنجا که برای کار با جاوا اسکریپت نیاز به درک کاملی درباره‌ی مفهوم حوزه کارکرد متغیرها (Scope) می‌باشد و نحوه فراخوانی توابع نیز نقش اساسی در این مورد بازی می‌کند، در این قسمت با این موارد آشنا خواهیم شد:
جاوا اسکریپت از مفهومی به نام functional scope برای تعیین حوزه متغیرها استفاده می‌کند و به این معنی است که با تعریف توابع، حوزه عملکرد متغیر مشخص می‌شود. در واقع هر متغیری که در یک تابع تعریف می‌شود در کلیه قسمتهای آن تابع، از قبیل If statement – for loops و حتی nested function نیز در دسترس میباشد.
اجازه دهید با مثالی این موضوع را بررسی نماییم.

```
function testScope() {
var myTest = true;
if (true) {
var myTest = "I am changed!"
}
alert(myTest);
}
testScope(); // will alert "I am changed!"
```

همانگونه که میبینیم با اینکه در داخل بلاک if یک متغیر جدید تعریف شده، ولی در خارج از این بلاک نیز این متغیر قابل دسترسی میباشد. البته در مثال بالا اگر بخواهیم به متغیر myTest در خارج از function دسترسی داشته باشیم، با خطای undefined مواجه خواهیم شد. یعنی برای مثال در کد زیر:

```
function testScope() {
var myTest = true;
if (true) {
var myTest = "I am changed!"
}
alert(myTest);
}
testScope(); // will alert "I am changed!"
alert(myTest); // will throw a reference error, because it doesn't exist outside of the function
```

برای حل این مشکل دو راه وجود دارد:

1 – متغیر myTest را در بیرون بلاک testScope() تعریف کنیم
2 – هنگام تعریف متغیر myTest، کلمه کلیدی var را حذف کنیم که این موضوع باعث میشود این متغیر در کل window قابل دسترس باشد و یا به عبارتی متغیر global میشود.
قبل از پرداختن به ادامه بحث خواندن مقاله مربوط به Closure در جاوااسکریپت توصیه میگردد .
در پایان بحث Scopeها با یک مثال نسبتا جامع اکثر این حالات به همراه خروجی را نشان میدهیم :

```
<script type="text/javascript">
          // a globally-scoped variable
        var a = 1;
        // global scope
        function one()
        {
            alert(a);
        }
        // local scope
        function two(a)
        {
            alert(a);
        }
        // local scope again
        function three()
        {
            var a = 3;
```

```
            alert(a);
        }
        // Intermediate: no such thing as block scope in javascript
        function four()
        {
            if (true)
            {
                var a = 4;
            }
            alert(a); // alerts '4', not the global value of '1'
        }
        // Intermediate: object properties
        function Five()
        {
            this.a = 5;
        }
        // Advanced: closure
        var six = function ()
        {
            var foo = 6;
            return function ()
            {
                // javascript "closure" means I have access to foo in here,
                // because it is defined in the function in which I was defined.
                alert(foo);
            }
        }()
        // Advanced: prototype-based scope resolution
        function Seven()
        {
            this.a = 7;
        }
        // [object].prototype.property loses to [object].property in the lookup chain
        Seven.prototype.a = -1; // won't get reached, because 'a' is set in the constructor above.
        Seven.prototype.b = 8; // Will get reached, even though 'b' is NOT set in the constructor.
        // These will print 1-8
        one();
        two(2);
        three();
        four();
        alert(new Five().a);
        six();
        alert(new Seven().a);
        alert(new Seven().b);
</Script>
```

برای مطالعه بیشتر به اینجا مراجعه نمایید.

Function Invocation Patterns In JavaScript :

از آنجا که توابع در جاوااسکریپت به منظور 1 – ساخت اشیاء و 2 – حوزه دسترسی متغیرها(Scope) نقش اساسی ایفا می‌کنند بهتر است کمی درباره استفاده و نحوه فراخوانی آنها (Function Invocation Patterns) در جاوااسکریپت بحث نماییم.

در جاوااسکریپت 4 مدل فراخوانی تابع داریم که به نامهای زیر مطرح هستند:

1. Method Invocation
2. Function Invocation
3. Constructor Invocation
4. Apply And Call Invocation

در فراخوانی توابع به هر یک از روشهای بالا باید به این نکته توجه داشت که حوزه دسترسی متغیرها در جاوااسکریپت ابتدا و انتهای توابع هستند و اگر به عنوان مثال از توابع تو در تو استفاده کردیم ،حوزه شی this برای توابع داخلی تغییر خواهد کرد .این موضوع را در طی مثالهایی نشان خواهیم داد. **Method Invocation :**

وقتی یک تابع قسمتی از یک شی باشد به آن متد میگوییم به عنوان مثال :

```
var obj = {
    value: 0,
    increment: function() {
        this.value+=1;
    }
};
obj.increment(); //Method invocation
```

در اینحالت this به شی (Object) اشاره میکند که متد در آن فراخوانی شده است و در زمان اجرا نیز به عناصر شی Bind میشود ،در مثال بالا حوزه this شی obj خواهد شد و به همین منظور به متغیر value دسترسی داریم. **Function Invocation:**
در اینحالت که از () برای فراخوانی تابع استفاده میگردد ،This به شی سراسری (global object) اشاره می‌کند؛ منظور اینکه this به اجزای تابعی که فراخوانی آن انجام شده اشاره نمی‌کند. اجازه دهید با مثالی این موضوع را روشن کنیم

```
<script type="text/javascript">
var value = 500; //Global variable
var obj = {
    value: 0,
    increment: function() {
        this.value++;
        var innerFunction = function() {
            alert(this.value);
        }
        innerFunction(); //Function invocation pattern
    }
}
obj.increment(); //Method invocation pattern
<script type="text/javascript">
Result : 500
```

از آنجا که () innerFunction به شکل Function invocation pattern فراخوانی شده است به متغیر value در داخل تابع increment دسترسی نداریم و حوزه دسترسی global میشود و اگر در حوزه global نیز این متغیر تعریف نشده بود به خطای undefined میرسیدیم .
برای حل این گونه مشکلات ساختار کد نویسی ما بایستی به شکل زیر باشد :

```
<script type="text/javascript">
var value = 500; //Global variable
var obj = {
    value: 0,
    increment: function() {
        var that = this;
        that.value++;
        var innerFunction = function() {
            alert(that.value);
        }
        innerFunction(); //Function invocation pattern
    }
}
obj.increment();
<script type="text/javascript">
Result : 1
```

در واقع با تعریف یک متغیر با نام مثلا that و انتساب شی this به آن میتوان در توابع بعدی که به شکل Function invocation pattern فراخوانی میگردند به این متغیر دسترسی داشت . **Constructor Invocation :**
در این روش برای فراخوانی تابع از کلمه new استفاده میکنیم. در این حالت یک شیء مجزا ایجاد شده و به متغیر دلخواه ما اختصاص پیدا می‌کند. به عنوان مثال داریم :

```
var Dog = function(name) {
  //this == brand new object ({});
    this.name = name;
    this.age = (Math.random() * 5) + 1;
};
var myDog = new Dog('Spike');
//myDog.name == 'Spike'
//myDog.age == 2
var yourDog = new Dog('Spot');
//yourDog.name == 'Spot'
//yourDog.age == 4
```

در این مورد با استفاده از New باعث میشویم همه خواص و متدهای تابع function برای هر نمونه از آن که ساخته میشود (از طریق مفهوم Prototype که قبلا درباره آن بحث شد) بطور مجزا اختصاص یابد. در مثال بالا شی mydog چون حاوی یک نمونه از

تابع dog بصورت Constructor Invocation میباشد، در نتیجه به خواص تابع dog از قبیل name و age دسترسی داریم. در اینجا اگر کلمه new استفاده نشود به این خواص دسترسی نداریم؛ در واقع با اینکار، this به mydog اختصاص پیدا میکند.
اگر از new استفاده نشود متغیر undefined، myDog میشود.
یک مثال دیگر :

```
var createCallBack = function(init) { //First function
    return new function() { //Second function by Constructor Invocation
        var that = this;
        this.message = init;
        return function() { //Third function
            alert(that.message);
        }
    }
}
window.addEventListener('load', createCallBack("First Message"));
window.addEventListener('load', createCallBack("Second Message"));
```

در مثال بالا از مفهوم closure نیز در مثالمان استفاده کرده ایم . **Apply And Call Invocation:**

تمامی توابع جاوااسکریپت دارای دو متد توکار apply() و call() هستند که توسط این متدها میتوان این توابع را با context دلخواه فراخوانی کرد.
نحوه فراخوانی به شکل مقابل است :

```
myFunction.apply(thisContext, arrArgs);
myFunction.call(thisContext, arg1, arg2, arg3, ..., argN);
```

که thisContext به حوزه اجرایی (execution context) تابع اشاره میکند. تفاوت دو متد apply() و call() در نحوه فرستادن آرگومانها به تابع میباشد که در اولی توسط آرایه اینکار انجام میشود و در دومی همه آرگومانها را بطور صریح نوشته و با کاما از هم جدا میکنیم .

مثال :

```
var contextObject = {
testContext: 10
}
var otherContextObject = {
testContext: "Hello World!"
}
var testContext = 15; // Global variable
function testFunction() {
alert(this.testContext);
}
testFunction(); // This will alert 15
testFunction.call(contextObject); // Will alert 10
testFunction.apply(otherContextObject); // Will alert "Hello World”
```

در این مثال دو شی متفاوت با خواص همنام تعریف کرده و یک متغیر global نیز تعریف میکنیم. در انتها یک تابع تعریف میکنیم که مقدار this.testContext را نمایش میدهد. در ابتدا حوزه اجرایی تابع (this) کل window جاری میباشد و وقتی testFunction() اجرا شود مقدار متغیر global نمایش داده میشود. در اجرای دوم this به contextObject اشاره کرده و حوزه اجرایی عوض میشود و در نتیجه مقدار testContext مربوطه که در این حالت 10 میباشد نمایش داده میشود و برای فراخوانی سوم نیز به همین شکل .

یک مثال کاملتر :

```
var o = {
  i : 0,
  F : function() {
    var a = function() { this.i = 42; };
    a();
```

```
    document.write(this.i);
  }
};
o.F();
Result :0
```

خط o.f() تابع f را به شکل Method invocation اجرا میکند. در داخل تابع f یک تابع دیگر به شکل function invocation اجرا میشود که در اینحال this به global object اشاره میکند و باعث میشود مقدار i در خروجی 0 چاپ شود .
برای حل این مشکل 2 راه وجود دارد
راه اول :

```
var p = {
  i : 0,
  F : function() {
    var a = function() { this.i = 42; };
    a.apply(this);
    document.write(this.i);
  }
};
 p.F();
Result :42
```

با اینکار this را موقع اجرای تابع درونی برایش فرستاده تا حوزه اجرای تابع عوض شود و به i دسترسی پیدا کنیم .
یا اینکه همانند مثالهای قبلی :

```
var q = {
  i: 0,
  F: function F() {
    var that = this;
    var a = function () {
      that.i = 42;
    }
    a();
    document.write(this.i);
  }
}
 q.F();
```

منابع :

Javascript programmer,s refrence http://coding.abel.nu/2013/03/more-on-this-in-javascript/
http://seanmonstar.com/post/707068021/4-ways-functions-mess-with-this

عنوان: **توسعه سرویس‌های Angular به روش OOP**
نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۳/۰۶/۱۴ ۲۳:۳۰
آدرس: www.dotnettips.info
گروه‌ها: JavaScript, OOP, AngularJS

یک نکته‌ای که در توسعه سیستم‌ها و نرم افزارها تاکید فراوانی به آن می‌شود استفاده مجدد از کدهای نوشته شده قبلی است. یعنی تا جای ممکن باید ساختار پروژه به گونه‌ای نوشته شود که از تکرار کدها در جای جای پروژه جلوگیری شود. این مورد به خوبی در زبان‌های شیءگرا نظیر C# رعایت می‌شود اما در پروژه‌هایی که مبتنی بر Javascript هستند نظیر angular، باید با استفاده از خاصیت prototype جاوا اسکریپ این مورد را رعایت نمود. در مقاله Dr. Axel Rauschmayer، قدم به قدم و به خوبی روش‌های وراثت در Javascript توضیح داده شده است.

در این پست با روش‌های وراثت در کنترلرهای انگولاری آشنا شدید. این وراثت محدود به ارث بری scopeها می‌شود. اما یکی از بخش‌های بسیار مهم پروژه‌های انگولار نوشتن سرویس‌هایی با قابلیت توسعه مجدد در سایر بخش‌های پروژه می‌باشد. معادل آن، مفهوم Overriding در OOP است. با ذکر مثالی این مورد را با هم بررسی خواهیم کرد.
ابتدا یک سرویس به نام BaseService ایجاد کنید:

```
angular.module('myApp').service('BaseService', function() {

    var BaseService = function(title) {
        this.title = title;
    };

    BaseService.prototype.getMessage = function() {
        var self = this;
        return 'Hello ' + self.title;
    };

    return BaseService;
});
```

سرویس بالا دارای سازنده‌ای است که مقدار title باید در اختیار آن قرار گیرد. با استفاده از خاصیت prototype تابعی تعریف می‌کنیم که این تابع خروجی مورد نظر را برای ما تامین خواهد نمود.
حال اگر ماژول و کنترلری جهت نمایش خروجی به صورت زیر ایجاد کنیم:

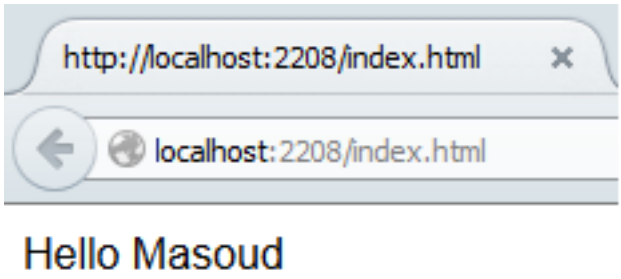
```
var app= angular.module('myApp', []);


app.controller('myCtrl', function ($scope,BaseService) {

    var instance = new BaseService('Masoud');
    $scope.title = instance.getMessage();

});
```

با کدهای Html زیر:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" ng-app="myApp">
<head>
    <title></title>
</head>
    <body ng-controller="myCtrl">

        <div>
            {{title}}
        </div>

    </body>
<script src="Scripts/jquery-2.1.1.min.js"></script>
<script src="Scripts/angular.js"></script>
<script src="App/app.js"></script>
```

```
</html>
```

در نهایت خروجی به صورت زیر قابل مشاهده است:



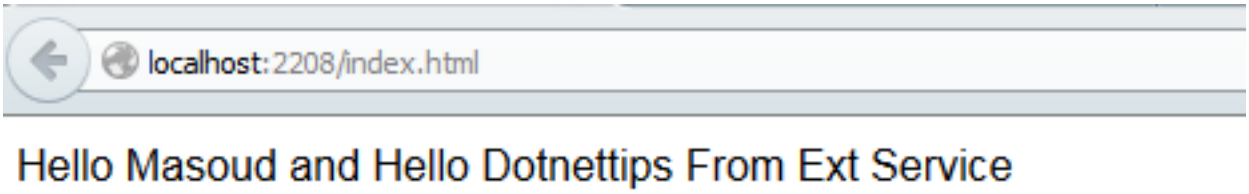
تا اینجای کار روال معمول تعاریف سرویس در انگولار بوده است. اما قصد داریم سرویس جدیدی را ایجاد نمایم تا خروجی سرویس قبلی را اندکی تغییر دهد. به جای اینکه سرویس قبلی را تغییر دهیم یا بدتر از آن سرویس جدیدی بسازیم و کدهای قبلی را در آن کپی کنیم کافیست به صورت زیر عمل نماییم:

```
app.service('ExtService', function(BaseService) {

    var ExtService = function() {
        BaseService.apply(this, arguments);
    };

    ExtService.prototype = new BaseService();

    ExtService.prototype.getMessage = function() {
        var self = this;
        return BaseService.prototype.getMessage.apply(this, arguments) + ' From Ext Service';

    };

    return ExtService;
});
```

حال می‌توان کنترلر را به صورت زیر بازنویسی کرد.

```
app.controller('myCtrl', function ($scope,BaseService , ExtService) {

    var baseInstance = new BaseService('Masoud');
    var extInstance = new ExtService('Dotnettips');
    $scope.title = baseInstance.getMessage() + ' and ' + extInstance.getMessage();

});
```

در کنترلر بالا هر دو سرویس تزریق شده‌اند. خروجی سرویس دوم متن From Ext Service را نیز به همراه خواهد داشت. پس از اجرای برنامه خروجی زیر قابل مشاهده است:


localhost:2208/index.html
Hello Masoud and Hello Dotnettips From Ext Service

عنوان: **آشنایی با الگوی طراحی Builder**
نویسنده: وحید فرهمندیان
تاریخ: ۱۳۹۳/۱۲/۰۲ ۲۰:۵
آدرس: www.dotnettips.info
گروه‌ها: C#, OOP, DesignPatterns

سناریوی زیر را در نظر بگیرید:
از شما خواسته شده است تا نحوه‌ی ساخت تلفن همراه را پیاده سازی نمایید. شما در گام اول 2 نوع تلفن همراه را شناسایی نموده‌اید (Android و Windows Phone). پس از شناسایی، احتمالا هر کدام از این انواع را یک کلاس در نظر می‌گیرید و به کمک یک واسط یا کلاس انتزاعی، شروع به ساخت کلاس می‌نمایید، تا در آینده اگر تلفن همراه جدیدی شناسایی شد، راحت‌تر بتوان آن را در پیاده سازی دخیل نمود.
اگر چنین فکر کرده اید باید گفت که 90% با الگوی طراحی Builder آشنا هستید و از آن نیز استفاده می‌کنید؛ بدون اینکه متوجه باشید از این الگو استفاده کرده‌اید. در کدهای زیر این الگو را قدم به قدم بررسی خواهیم نمود. **قدم 1:** تلفن همراه چه بخش هایی می‌تواند داشته باشد؟ (برای مثال یک OS دارند، یک Name دارند و یک Screen) همچنین برای اینکه تلفن همراهی بتواند ساخته شود ابتدا بایستی نام آن‌را بدانیم. کدهای زیر همین رویه را تصدیق می‌نمایند:

```
public class Product
{
        public Product(string name)
        {
            Name = name;
        }
        public string Name { get; set; }
        public string Screen { get; set; }
        public string OS { get; set; }
        public override string ToString()
        {
            return string.Format(Screen + "/" + OS + "/" + Name);
        }
}
```

یک کلاس ساخته‌ایم و نام آن را Product گذاشتیم. بخش‌های مختلفی را نیز در آن تعریف نموده‌ایم. تابع ToString را برای استفاده‌های بعدی override کرده‌ایم (فعلا نیازی بدان نداریم). **قدم 2:** برای ساخت تلفن همراه چه کارهایی باید انجام شود؟ (برای مثال بایستی OS روی آن نصب شود، Screen آن مشخص شود. همچنین بایستی به طریقی بتوانم تلفن همراه ساخته شده‌ی خود را نیز پیدا کنم). کدهای زیر همین رویه را تصدیق می‌نمایند:

```
public interface IBuilder
    {
        void BuildScreen();
        void BuildOS();
        Product Product { get; }
    }
```

یک واسط تعریف کرده‌ایم تا به کمک آن هر تلفن همراهی را که خواستیم بسازیم. **قدم 3:** از آنجا که فقط دو نوع تلفن همراه را فعلا شناسایی کرده‌ایم (Android و Windows Phone) نیاز داریم تا این دو تا را بسازیم.
ابتدا تلفن همراه Android را می‌سازیم:

```
public class ConcreteBuilder1 : IBuilder
    {
        public Product p;
        public ConcreteBuilder1()
        {
            p = new Product("Android Cell Phone");
        }
        public void BuildScreen()
        {
            p.Screen = "Touch Screen 16 Inch!";
        }

        public void BuildOS()
        {
            p.OS = "Android 4.4";
        }
```

```
        public Product Product
        {
            get { return p; }
        }
    }
```

سپس تلفن همراه Windows Phone را می‌سازیم:

```
public class ConcreteBuilder2 : IBuilder
    {
        public Product p;

        public ConcreteBuilder2()
        {
            p = new Product("Windows Phone");
        }
        public void BuildScreen()
        {
            p.Screen = "Touch Screen 32 Inch!";
        }

        public void BuildOS()
        {
            p.OS = "Windows Phone 2014";
        }
        public Product Product
        {
            get { return p; }
        }
    }
```

قدم 4: اول باید OS نصب شود یا Screen مشخص شود؟ برای اینکه توالی کار را مشخص سازم نیاز به یک کلاس دیگر دارم تا اینکار را انجام دهد:

```
public class Director
    {
        public void Construct(IBuilder builder)
        {
            builder.BuildScreen();
            builder.BuildOS();
        }
    }
```

این کلاس در متد Construct خود یک ورودی از نوع IBuilder می‌گیرد و براساس توالی مورد نظر، شروع به ساخت آن می‌کند.
قدم 5: نهایتا میخواهم به برنامه‌ی خود بگویم که تلفن همراه Android را بسازد:

```
Director d = new Director();
ConcreteBuilder1 cb1 = new ConcreteBuilder1();
d.Construct(cb1);
Console.WriteLine(cb1.p.ToString());
```

و به این صورت تلفن همراه من آماده است!
متد ToString در اینجا، همان ToString ابتدای بحث است که آن را Override کردیم.
به این نکته توجه کنید که اگر یک تلفن همراه جدید شناسایی شود، چه مقدار تغییری در کدها نیاز دارید؟ برای مثال تلفن همراه BlackBerry شناسایی شده‌است. تنها کاری که لازم است این است که یک کلاس بصورت زیر ساخته شود:

```
public class BlackBerry: IBuilder
    {
        public Product p;

        public BlackBerry ()
        {
            p = new Product("BlackBerry");
        }
        public void BuildScreen()
        {
            p.Screen = "Touch Screen 8 Inch!";
        }
```

```
        public void BuildOS()
        {
            p.OS = "BlackBerry XXX";
        }
        public Product Product
        {
            get { return p; }
        }
    }
```

عنوان: **آشنایی با الگوی طراحی Bridge**
نویسنده: وحید فرهمندیان
تاریخ: ۱۳۹۳/۱۲/۰۳ ۱۹:۱۵
آدرس: www.dotnettips.info
گروه‌ها: C#, OOP, DesignPatterns

سناریو زیر را در نظر بگیرید:
قصد دارید تا در برنامه‌ی خود ارسال پیام از طریق پیامک و ایمیل را راه اندازی کنید. هر کدام از این روش‌ها نیز برای خود راه‌های متفاوتی دارند. برای مثال ارسال پیامک از طریق وب سرویس یا یک API خارجی و غیره.
کاری را که می‌توان انجام داد، بشرح زیر نیز می‌توان بیان نمود:
ابتدا یک Interface ایجاد می‌کنیم (IBridge) و در آن متد Send را قرار می‌دهیم. این متد یک پارامتر ورودی از نوع رشته می‌گیرد و به کمک آن میتوان اقدام به ارسال پیامک یا ایمیل یا هر چیز دیگری نمود. کلاس‌هایی این واسط را پیاده سازی می‌کنند که یکی از روش‌های اجرای کار باشند (برای مثال کلاس WebService که یک روش ارسال پیامک یا ایمیل است).

```
public interface IBridge
    {
        string Send(string parameter);
    }
    public class WebService: IBridge
    {
        public string Send(string parameter)
        {
            return parameter + " sent by WebService";
        }
    }
    public class API: IBridge
    {
        public string Send(string parameter)
        {
            return parameter + " sent by API";
        }
    }
```

سپس در ادامه به مکانیزمی نیاز داریم تا بتوانیم از طریق آن پیامک یا ایمیل را ارسال کنیم. خوب می‌خواهیم ایمیل ارسال کنیم؛ اولین سوالی که مطرح می‌شود این است که چگونه ارسال کنیم؟ پس باید در مکانیزم خود زیرساختی برای پاسخ به این سوال آماده باشد.

```
public abstract class Abstraction
    {
        public IBridge Bridge;
        public abstract string SendData();
    }
    public class SendEmail : Abstraction
    {
        public override string SendData ()
        {
            return Bridge.Send("Email");
        }
    }
    public class SendSMS: Abstraction
    {
        public override string SendData ()
        {
            return Bridge.Send("SMS");
        }
    }
```

در کد فوق یک کلاس انتزاعی ایجاد کردیم و در آن یک object از نوع واسط خود قرار دادیم. این object به ما کمک می‌کند تا به طریق آن شیوه‌ی ارسال ایمیل یا پیامک را مشخص سازیم و به سوال خود پاسخ دهیم. سپس در ادامه متد SendData آورده شده است که به کمک آن اعلام می‌کنیم که قصد ارسال ایمیل یا پیامک را داریم و نهایتا هر یک از کلاس‌های ایمیل یا پیامک، این متد را برای خود پیاده سازی کرده‌اند.

قبل از ادامه اجازه دهید کمی در مورد بدنه‌ی یکی از متدهای SendData صحبت کنیم. در این متد با کمک Bridge متد Send موجود

در واسط صدا زده شده است. از آنجا که این object از نظر سطح دسترسی عمومی می‌باشد، لذا از بیرون از کلاس قابل دسترسی است. این باعث می‌شود تا قبل از فراخوانی متد SendData موجود در کلاس ایمیل یا پیامک اعلام کنیم که Bridge از چه نوعی است (به چه روشی می‌خواهیم ارسال رخ دهد).

```
Abstraction ab1 = new Email();
  ab1.Bridge = new WebService();
  Console.WriteLine(ab1.SendData ());
```

```
ab1.Bridge = new API();
  Console.WriteLine(ab1.SendData ());
```

```
Abstraction ab2 = new SMS();
   ab2.Bridge = new WebService();
   Console.WriteLine(ab2.SendData ());
```

```
ab2.Bridge = new API();
    Console.WriteLine(ab2.SendData ());
```

نهایتا در کد فوق ابتدا بیان می‌کنیم که قصد ارسال ایمیل را داریم. سپس اعلام می‌داریم که این ارسال را به کمک WebService می‌خواهیم انجام دهی. و نهایتا ارسال را انجام می‌دهیم.

به کل این الگویی که ایجاد کردیم، الگوی Bridge گفته می‌شود.

حال فکر کنید قصد ارسال MMS دارید. در اینصورت فقط کافیست یک کلاس MMS ایجاد کنید و تمام؛ بدون اینکه کدی اضافی را بنویسید یا برنامه را تغییر دهید. یا فرض کنید روش ارسال جدیدی را می‌خواهید اضافه کنید. برای مثال ارسال به روش XYZ. در اینصورت فقط کافیست یک کلاس XYZ را ایجاد کنید که IBridge را پیاده سازی می‌کند.

عنوان: **آشنایی با الگوی طراحی Decorator**
نویسنده: وحید فرهمندیان
تاریخ: ۱۳۹۳/۱۲/۰۴ ۱۴:۵
آدرس: www.dotnettips.info
گروه‌ها: C#, OOP, DesignPatterns

این بار مثال را با شیرینی و کیک پیش می‌بریم.
فرض کنید شما قصد پخت کیک و نان را دارید. طبیعی است که برای اینکار یک واسط را تعریف کرده و عمل «پختن» را در آن اعلام می‌کنید تا هر کلاسی که قصد پیاده سازی این واسط را داشت، «پختن» را انجام دهد. در ادامه یک کلاس بنام کیک ایجاد خواهید کرد و شروع به پخت آن می‌کنید.
خوب احتمالا الان کیک آماده‌است و می‌توانید آن‌را میل کنید! ولی یک سؤال. تکلیف شخصی که کیک با روکش کاکائو دوست دارد و شمایی که کیک با روکش میوه‌ای دوست دارید چیست؟ این را چطور در پخت اعمال کنیم؟ یا منی که نان کنجدی می‌خواهم و شمایی که نان برشته‌ی غیر کنجدی می‌خواهید چطور؟
احتمالا می‌خواهید سراغ ارث بری رفته و سناریوهای این چنینی را پیاده سازی کنید. ولی در مورد ارث بری، اگر کلاس sealed ((NotInheritable باشه چطور؟
احتمالا همین دو تا سؤال کافی‌است تا در پاسخ بگوئیم، گره‌ی کار، با الگوی Decorator باز می‌شود و همین دو تا سؤال کافی‌است تا اعلام کنیم که این الگو، از جمله الگوهای بسیار مهم و پرکاربرد است.
در ادامه سناریوی خود را با کد ذیل جلو می‌بریم:

```
public interface IBakery
    {
        string Bake();
        double GetPrice();
    }
    public class Cake: IBakery
    {
        public string Bake() { return "Cake baked"; }
        public double GetPrice() { return 2000; }
    }
    public class Bread: IBakery
    {
        public string Bake() { return "Bread baked"; }
        public double GetPrice() { return 100; }
    }
```

در کد فوق فرض کرده‌ام که شما می‌خواهید محصول خودتان را بفروشید و برای آن یک متد GetPrice نیز گذاشته‌ام. خوب در ابتدا واسطی تعریف شده و متدهای Bake و GetPrice اعلام شده‌اند. سپس کلاس‌های Cake و Bread پیاده سازی‌های خودشان را انجام دادند.
در ادامه باید مخلفاتی را که روی کیک و نان می‌توان اضافه کرد، پیاده نمود.

```
public abstract class Decorator : IBakery
    {
        private readonly IBakery _bakery;
        protected string bake = "N/A";
        protected double price = -1;

        protected Decorator(IBakery bakery) { _bakery= bakery; }
        public virtual string Bake() { return _bakery.Bake() + "/" + bake; }
        public double GetPrice() { return _bakery.GetPrice() + price; }
    }
    public class Type1 : Decorator
    {
        public Type1(IBakery bakery) : base(bakery) { bake= "Type 1"; price = 1; }
    }
    public class Type2 : Decorator
    {
        private const string bakeType = "special baked";
        public Type2(IBakery bakery) : base(bakery) { name = "Type 2"; price = 2; }
        public override string Bake() { return base.Bake() + bakeType ; }
    }
```

در کد فوق یک کلاس انتزاعی ایجاد و متدهای پختن و قیمت را پیاده سازی کردیم؛ همچنین کلاس‌های Type1 و Type2 را که من

فرض کردم کلاس‌هایی هستند برای اضافه کردن مخلفات به کیک و نان. در این کلاس‌ها در متد سازنده، یک شیء از نوع IBakery می‌گیریم که در واقع این شیء یا از نوع Cake هست یا از نوع Bread و مشخص می‌کند روی کیک می‌خواهیم مخلفاتی را اضافه کنیم یا بر روی نان. کلاس Type1 روش پخت و قیمت را از کلاس انتزاعی پیروی می‌کند، ولی کلاس Type2 روش پخت خودش را دارد.
با بررسی اجمالی در کدهای فوق مشخص می‌شود که هرگاه بخواهیم، می‌توانیم رفتارها و الحاقات جدیدی را به کلاس‌های Cake و Bread، اضافه کنیم؛ بدون آنکه کلاس اصلی آنها تغییر کند. حال شما شاید در پیاده سازی این الگو از کلاس انتزاعی Decorator هم استفاده نکنید.
با این حال شیوه‌ی استفاده از این کدها هم بصورت زیر خواهد بود:

```
Cake cc1 = new Cake();
 Console.WriteLine(cc1.Bake() + " ," + cc1.GetPrice());

 Type1 cd1 = new Type1 (cc1);
 Console.WriteLine(cd1.Bake() + " ," + cd1.GetPrice());

 Type2 cd2 = new Type2(cc1);
 Console.WriteLine(cd2.Bake() + " ," + cd2.GetPrice());
```

ابتدا یک کیک را پختیم در ادامه Type1 را به آن اضافه کردیم که این باعث می‌شود قیمتش هم زیاد شود و در نهایت Type2 را هم به کیک اضافه کردیم و حالا کیک ما آماده است.

نظرات خوانندگان

نویسنده: محمد اسکندری
تاریخ: ۱۳۹۳/۱۲/۰۵ ۱۰:۵۴

در استفاده از الگوی دکوراتور روش بهتر بهره گیری از آن بصورت سری است و نه ایجاد شیء جدید برای تایپ جدید.
مثلا:

```
Cake c = new Cake();

c = new Type1(c);
c = new SubType(c); //SubType derived from  Cake (e.g. CakeComponent like Cream)

//or: c = new Type1 (new SubType(c));

Console.WriteLine(c.Bake() + ", " + c.GetPrice());
```

نویسنده: محسن خان
تاریخ: ۱۳۹۳/۱۲/۰۵ ۱۱:۴۴

بستگی به هدف نهایی دارد. اگر هدف تولید کیک با روکش کاکائویی و روکش میوه‌ای به صورت همزمان است، نحوه‌ی تزئین آن با کیکی که فقط قرار هست روکش کاکائویی داشته باشه، فرق می‌کنه.

نویسنده: محمد اسکندری
تاریخ: ۱۳۹۳/۱۲/۰۵ ۱۲:۰

فرقی نمیکنه. اگر قرار بود فرق میکرد و نیاز به ایجاد تغییرات در کد بود که این الگوها ارائه نمی‌شدند.

```
// ساخت کیک معمولی با روکش کاکائویی
Cake c = new CommonCake();
c = new Chocolate(c);

// ساخت کیک معمولی با روکش میوه‌ای
Cake c = new CommonCake();
c = new Fruity(c);

// ساخت کیک معمولی مخلوط با روکش کاکائویی و روکش میوه‌ای به صورت همزمان
Cake c = new CommonCake();
c = new Chocolate(c);
c = new Fruity(c);

// ساخت کیک مخصوص مخلوط با روکش کاکائویی و روکش میوه‌ای به صورت همزمان
Cake c = new SpecialCake();
c = new Chocolate(c);
c = new Fruity(c);
```

برای هر c میتوان متدهای اینترفیسش را اجرا کرد.

نویسنده: محسن خان
تاریخ: ۱۳۹۳/۱۲/۰۵ ۱۲:۸

عنوان کردید «در استفاده از الگوی دکوراتور روش بهتر بهره گیری از آن بصورت **سری** است و نه ایجاد شیء **جدید** برای تایپ جدید»، بعد الان برای تهیه روکش فقط میوه‌ای از حالت سری استفاده نکردید و یک وهله جدید ایجاد شده. بحث بر سر سری بودن یا نبودن مراحل بود. بنابراین بسته به هدف، می‌تونه سری باشه یا نباشه و اگر نبود، مشکلی نداره، چون هدفش تولید یک روکش مخصوص بوده و نه ترکیبی.

نویسنده: محمد اسکندری
تاریخ: ۱۳۹۳/۱۲/۰۵ ۱۲:۲۳

فرض من این بود که کاربر نیازی به رفرنس گیری از هر آبجکت ندارد.
مثلا طبق مقاله:

```
// ساخت کیک مخصوص مخلوط با روکش کاکائویی و روکش میوه‌ای به صورت همزمان
Cake c = new SpecialCake();
Chocolate ch = new Chocolate(c);
Fruity f = new Fruity(ch);
```

همانطور که در مقاله گفته شده:

```
Cake cc1 = new Cake();
Type1 cd1 = new Type1 (cc1);
Type2 cd2 = new Type2(cc1);
```

کد فوق را میتوان اینگونه **هم** داشت:

```
// ساخت کیک مخصوص مخلوط با روکش کاکائویی و روکش میوه‌ای به صورت همزمان
Cake c = new SpecialCake();
c = new Chocolate(c);
c = new Fruity(c);
```

بدون اینکه *شیء جدید برای تایپ جدید* بسازیم.

نویسنده: محسن خان
تاریخ: ۱۳۹۳/۱۲/۰۵ ۱۲:۴۳

مهم این نیست که نام تمام متغیرها را c تعریف کردید، مهم این است که به ازای هر new یک شیء کاملا جدید ایجاد می‌شود که ریفرنس آن با ریفرنس قبلی یکی نیست.

عنوان: **آشنایی با الگوی طراحی Prototype**
نویسنده: وحید فرهمندیان
تاریخ: ۱۳۹۳/۱۲/۰۴ ۱۹:۲۰
آدرس: www.dotnettips.info
گروه‌ها: C#, OOP, DesignPatterns

فرض کنید در حال پختن یک کیک هستید. ابتدا کیک را می‌پذید و سپس آن را تزیین می‌کنید. عملیات پختن کیک، فرآیند ثابتی است و تزیین کردن آن متفاوت. گاهی کیک را با کاکائو تزیین می‌کنید و گاهی با میوه و غیره.
پیش از اینکه سناریو را بیش از این جلو ببریم، وارد بحث کد می‌شویم. طبق سناریوی فوق، فرض کنید کلاسی بنام Prototype دارید که این کلاس هم از کلاس انتزاعی APrototype ارث برده است. در ادامه یک شیء از این کلاس می‌سازید و مقادیر مختلف آن را تنظیم کرده و کار را ادامه می‌دهید.

```
public abstract class APrototype : ICloneable     {
        public string Name { get; set; }
        public string Health { get; set; }
    }

    public class Prototype : APrototype
    {
        public override string ToString() { return string.Format("Player name: {0}, Health statuse:
{1}", Name, Health); }
    }
```

در ادامه از این کلاس نمونه‌گیری می‌کنیم:

```
Prototype p1 = new Prototype { Name = "Vahid", Health = "OK" };
Console.WriteLine(p1.ToString());
```

حالا فرض کنید به یک آبجکت دیگر نیاز دارید، ولی این آبجکت عینا مشابه p1 است؛ لذا نمونه‌گیری، از ابتدا کار مناسبی نیست.
برای اینکار کافیست کدها را بصورت زیر تغییر دهیم:

```
public abstract class APrototype : ICloneable
    {
        public string Name { get; set; }
        public string Health { get; set; }
        public abstract object Clone();
    }

    public class Prototype : APrototype
    {
        public override object Clone() { return this.MemberwiseClone() as APrototype; }
        public override string ToString() { return string.Format("Player name: {0}, Health statuse:
{1}", Name, Health); }
    }
```

در متد Clone از MemberwiseClone استفاده کرده‌ایم. خود Clone هم در داخل واسط ICloneable تعریف شده‌است و هدف از آن کپی نمودن آبجکت‌ها است. سپس کد فوق را بصورت زیر مورد استفاده قرار می‌دهیم:

```
Prototype p1 = new Prototype { Name = "Vahid", Health = "OK" };
Prototype p2 = p1.Clone() as Prototype;
Console.WriteLine(p1.ToString());
Console.WriteLine(p2.ToString());
```

با اجرای کد فوق مشاهده میشود p1 و p2 دقیقا عین هم کار می‌کنند. کل این فرآیند بیانگر الگوی Prototype می‌باشد. ولی تا اینجای کار درست است که الگو پیاده سازی شده است، ولی همچنین به نظر نقصی نیز در کد دیده می‌شود:
برای واضح نمودن نقص، یک کلاس بنام AdditionalDetails تعریف می‌کنیم. در واقع کد را بصورت زیر تغییر میدهیم:

```
public abstract class APrototype : ICloneable
    {
        public string Name { get; set; }
        public string Health { get; set; }
```

```
        public AdditionalDetails Detail { get; set; }
        public abstract object Clone();
    }
    public class AdditionalDetails { public string Height { get; set; } }

    public class Prototype : APrototype
    {
        public override object Clone() { return this.MemberwiseClone() as APrototype; }
        public override string ToString() { return string.Format("Player name: {0}, Health statuse:
{1}, Height: {2}", Name, Health, Detail.Height); }
    }
```

و از آن بصورت زیر استفاده می‌کنیم:

```
Prototype p1 = new Prototype { Name = "Vahid", Health = "OK", Detail = new AdditionalDetails { Height =
"100" } };
Prototype p2 = p1.Clone() as Prototype;
p2.Detail.Height = "200";
Console.WriteLine(p1.ToString());
Console.WriteLine(p2.ToString());
```

خروجی که نمایش داده می‌شود در بخش Height هم برای p1 و هم برای p2 عدد 200 را نمایش می‌دهد که می‌تواند اشتباه باشد. چراکه p1 دارای Height برابر با 100 است و p2 دارای Height برابر با 200. به این اتفاق ShallowCopy گفته میشود که ناشی از استفاده از MemberwiseClone است که در مورد ارجاعات با آدرس رخ می‌دهد. در این حالت بجای کپی نمودن مقدار، از کپی نمودن آدرس استفاده میشود (Ref Type چیست؟)

برای حل این مشکل باید DeepCopy انجام داد. لذا کد را بصورت زیر تغییر می‌دهیم:(DeepCopy و ShallowCopy چیست؟)

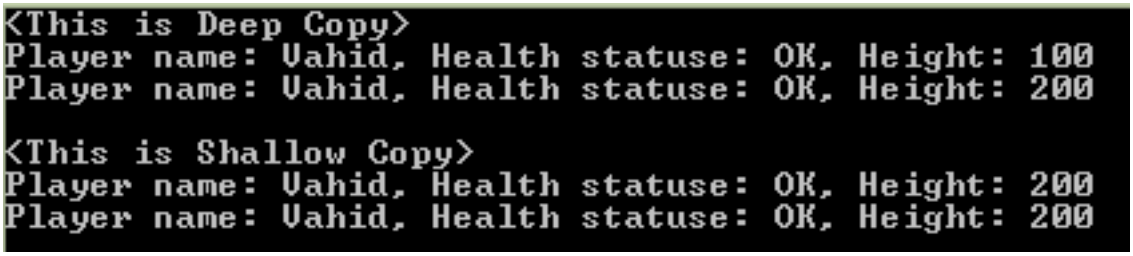
```
public abstract class APrototype : ICloneable
    {
        public string Name { get; set; }
        public string Health { get; set; }
        //This is a ref type
        public AdditionalDetails Detail { get; set; }
        public abstract APrototype ShallowClone();
        public abstract object Clone();
    }

    public class AdditionalDetails { public string Height { get; set; } }

    public class Prototype : APrototype
    {
        public override object Clone()
        {
            Prototype cloned = MemberwiseClone() as Prototype;
            //We need to deep copy each ref types in order to prevent shallow copy
            cloned.Detail = new AdditionalDetails { Height = this.Detail.Height };
            return cloned;
        }
        //Shallow copy will copy ref type's address instead of their value, so any changes in cloned
object or source object will take effect on both objects
        public override APrototype ShallowClone() { return this.MemberwiseClone() as APrototype; }
        public override string ToString() { return string.Format("Player name: {0}, Health statuse:
{1}, Height: {2}", Name, Health, Detail.Height); }
    }
```

و سپس بصورت زیر از آن استفاده نمود:

```
Prototype p1 = new Prototype { Name = "Vahid", Health = "OK", Detail = new AdditionalDetails { Height =
"100" } };
            Prototype p2 = p1.Clone() as Prototype;
            p2.Detail.Height = "200";
            Console.WriteLine("<This is Deep Copy>");
            Console.WriteLine(p1.ToString());
            Console.WriteLine(p2.ToString());

            Prototype p3 = new Prototype { Name = "Vahid", Health = "OK", Detail = new
AdditionalDetails { Height = "100" } };
            Prototype p4 = p3.ShallowClone() as Prototype;
            p4.Detail.Height = "200";
            Console.WriteLine("\n<This is Shallow Copy>");
```

```
            Console.WriteLine(p3.ToString());
            Console.WriteLine(p4.ToString());
```

لذا خروجی بصورت زیر را می‌توان مشاهده نمود:



البته در این سناریو ShallowCopy باعث اشتباه شدن نتایج می‌شود. شاید شما در دامنه‌ی نیازمندیهای خود، اتفاقا به ShallowCopy نیاز داشته باشید و DeepCopy مرتفع کننده‌ی نیاز شما نباشد. لذا کاربرد هر کدام از آنها وابستگی مستقیمی به دامنه‌ی نیازمندی‌های شما دارد.

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());
```

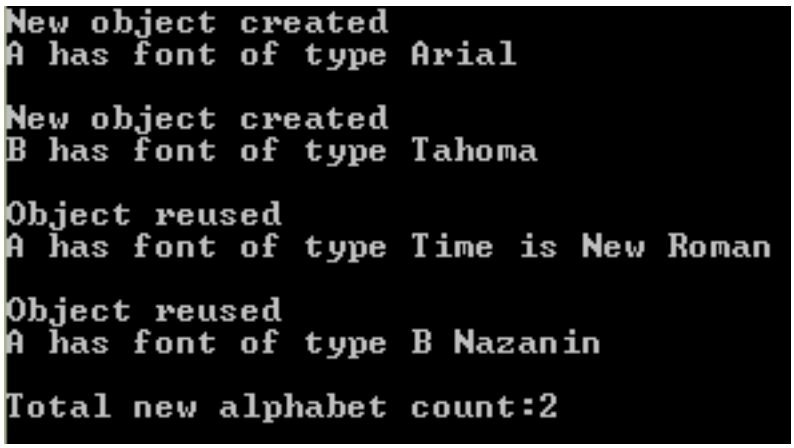
```
                    Console.WriteLine("New object created");
                    break;
                case "B":
                    _dictionary.Add(name, new B());
                    Console.WriteLine("New object created");
                    break;
                default:
                    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.

عنوان: **آشنایی با الگوی طراحی Abstract Factory**
نویسنده: وحید فرهمندیان
تاریخ: ۱۳۹۳/۱۲/۰۵ ۱۹:۱۰
آدرس: www.dotnettips.info
گروه‌ها: C#, OOP, DesignPatterns

قبل از مطالعه‌ی این مطلب، حتما الگوی طراحی Factory Method را مطالعه نمایید.
همانطور که در الگوی طراحی Factory Method مشاهده شد، این الگو یک عیب دارد، آن هم این است که از کدام Creator باید استفاده شود و مستقیما در کد بایستی ذکر شود.

```
class ConcreteCreator : Creator
{
     public override IProduct FactoryMethod(string type)
    {
            switch (type)
           {
                case "A": return  new ConcreteProductA();
                case "B": return  new ConcreteProductB();
                default: throw new ArgumentException("Invalid type", "type");
           }
     }
}
```

برای حل این مشکل می‌توانیم سراغ الگوی طراحی دیگری برویم که Abstract Factory نام دارد. این الگوی طراحی 4 بخش اصلی دارد که هر کدام از این بخش‌ها را طی مثالی توضیح می‌دهم:
1. Abstract Factory: در کشور، صنعت خودروسازی داریم که خودروها را در دو دسته‌ی دیزلی و سواری تولید می‌کنند :

```
public interface IVehicleFactory     {
        IDiesel GetDiesel();
        IMotorCar GetMotorCar();
    }
```

2. Concrete Factory: دو کارخانه‌ی تولید خودرو داریم که در صنعت خودرو سازی فعالیت دارند و عبارتند از ایران خودرو و سایپا که هر کدام خودروهای خود را تولید می‌کنند. ولی هر خودرویی که تولید می‌کنند یا دیزلی است یا سواری. شرکت ایران خودرو، خودروی آرنا را بعنوان دیزلی تولید می‌کند و پژو 206 را بعنوان سواری. همچنین شرکت سایپا خودروی فوتون را بعنوان خودروی دیزلی تولید می‌کند و خودروی پراید را بعنوان خودروی سواری.

```
public class IranKhodro : IVehicleFactory
    {
        public IDiesel GetDiesel() { return new Arena(); }
        public IMotorCar GetMotorCar() { return new Peugeot206(); }
    }
    public class Saipa : IVehicleFactory
    {
        public IDiesel GetDiesel() { return new Foton(); }
        public IMotorCar GetMotorCar() { return new Peride(); }
    }
```

3. Abstract Product: خودروهای تولیدی همانطور که گفته شد یا دیزلی هستند یا سواری که هر کدام از این خودروها ویژگی‌های خاص خود را دارند (در این مثال هر دو دسته خودرو برای خود نام دارند)

```
public interface IDiesel { string GetName();}
    public interface IMotorCar { string GetName();}
```

4. Concrete Product: در بین این خودروها، خودروی پژو 206 و پراید یک خودروی سواری هستند و خودروی فوتون و آرنا، خودروهای دیزلی.

```
public class Foton : IDiesel { public string GetName() { return "This is Foton"; } }
    public class Arena : IDiesel { public string GetName() { return "This is Arena"; } }
    public class Peugeot206 : IMotorCar { public string GetName() { return "This is Peugeot206"; } }
```

```
    public class Peride : IMotorCar { public string GetName() { return "This is Peride"; } }
```

حال که 4 دسته اصلی این الگوی طراحی را آموختیم می‌توان از آن بصورت زیر استفاده نمود:

```
IVehicleFactory factory = new IranKhodro();
 Console.WriteLine("***" + factory.GetType().Name + "***");
 IDiesel diesel = factory.GetDiesel();
 Console.WriteLine(diesel.GetName());
 IMotorCar motorCar = factory.GetMotorCar();
 Console.WriteLine(motorCar.GetName());

 factory = new Saipa();
 Console.WriteLine("***" + factory.GetType().Name + "***");
 diesel = factory.GetDiesel();
 Console.WriteLine(diesel.GetName());
 motorCar = factory.GetMotorCar();
 Console.WriteLine(motorCar.GetName());
```

همانطور که در کد فوق مشاهده میشود، ایراد موجود در الگوی Factory Method اینجا از بین رفته است و برای ساخت آبجکت‌های مختلف از Innterfaceها یا Abstract Classها استفاده می‌کنیم.

کلا Abstract Factory مزایای زیر را دارد:

پیاده سازی و نامگذاری Product در Factory مربوطه متمرکز می‌شود و بدین ترتیب Client به نام و نحوه پیاده سازی Typeهای مختلف Product وابستگی نخواهد داشت.

به راحتی می‌توان Concrete Factory مورد استفاده در برنامه را تغییر داد، بدون اینکه تاثیری در عملکرد سایر بخش‌ها داشته باشد.

در مواردی که بیش از یک محصول برای هر خانواده وجود داشته باشد، استفاده از Abstract Factory تضمین می‌کند که Productهای هر خانواده همه در کنار هم قرار دارند و با هم فعال و غیر فعال می‌شوند. (یا همه، یا هیچکدام)

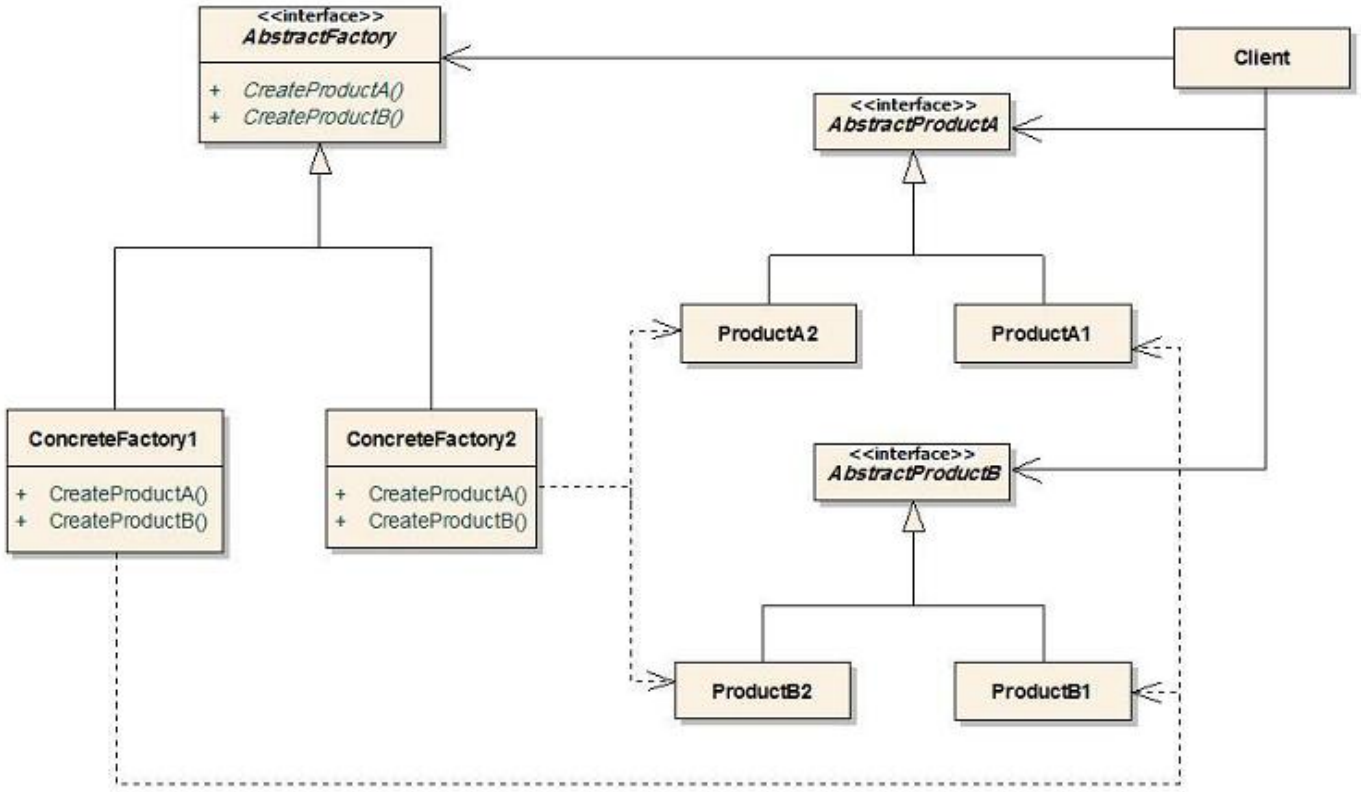
بزرگترین عیبی که این الگوی طراحی دارد این است که با اضافه شدن فقط یک Product تازه، Abstract Factory باید تغییر کند که این مساله منجر به تغییر همه Concrete Factoryها می‌شود.

نهایتا اینکه در استفاده از این الگوی طراحی به این تکنیک‌ها توجه داشته باشید:

Factoryها معمولا Singleton هستند. زیرا هر Application **بطور معمول** فقط به یک instance از هر Concrete Factory نیاز دارد.

انتخاب Concrete Factory مناسب معمولا توسط پارامترهایی انجام می‌شود.

نمودار کلاسی این الگو نیز بصورت زیر میباشد:



و کلام آخر در مورد این الگو:

Abstract Factory یک interface یا کلاس abstract است که signature متدهای ساخت Objectها در آن تعریف شده است و Concrete Factoryها آن‌ها را implement می‌نمایند.

در Abstract Factory Pattern همه Productهای هم خانواده در Concrete Factory مربوط به آن خانواده پیاده سازی و مجتمع می‌گردند.

در کدهای برنامه تنها با Abstract Factory و Abstract Productها سر و کار داریم و به هیچ وجه درگیر این مساله که کدام یک از Concrete Classها در برنامه مورد استفاده قرار می‌گیرند، نمی‌شویم.

عنوان: **آشنایی با الگوی طراحی Template Method**
نویسنده: وحید فرهمندیان
تاریخ: ۱۳۹۳/۱۲/۰۹ ۱۸:۲۵
آدرس: www.dotnettips.info
گروه‌ها: C#, OOP, DesignPatterns

سناریویی وجود دارد که در آن شما می‌خواهید تنها یک کار را انجام دهید، ولی برای انجام آن n روش وجود دارد. برای مثال قصد مرتب سازی دارید و برای اینکار روش‌های مختلفی وجود دارند. برای حل این مساله پیشتر از الگوی طراحی استراتژی استفاده نمودیم.(مطالعه بیشتر در مورد الگوی طراحی استراتژی)
حال به سناریویی برخورد کردیم که بصورت زیر است:
می‌خواهیم یک کار را انجام دهیم ولی برای انجام این کار تنها برخی بخش‌های کار با هم متفاوت هستند. برای مثال قصد تولید گزارش و چاپ آن را داریم. در این سناریو خواندن اطلاعات و پردازش آن‌ها رخدادهایی ثابت هستند. ولی اگر بخواهیم گزارش را چاپ کنیم به مشکل می‌خوریم؛ چرا که چاپ گزارش به فرمت اکسل، فرمت و روش خود را دارد و چاپ به فرمت PDF شرایط خود را دارد.
در این سناریو دیگر الگوی طراحی استراتژی جواب نخواهد داد و نیاز داریم با یک الگوی طراحی جدید آشنا بشویم. این الگوی طراحی Template Method نام دارد.
در این الگو یک کلاس انتزاعی داریم به صورت زیر:

```
public abstract class DataExporter
    {
        public void ReadData()
        {
            Console.WriteLine("Data is reading from SQL Server Database");
        }

        public void ProcessData()
        {
            Console.WriteLine("Data is processing...!");
        }

        public abstract void PrintData();

        public void GetReport()
        {
            ReadData();
            ProcessData();
            PrintData();
        }
    }
```

این کلاس abstract، یک متد بنام GetReport دارد که نحوه‌ی انجام کار را مشخص می‌کند. متدهای ReadData و ProcessData نشان می‌دهند که انجام این دو عمل همیشه ثابت هستند (منظور در این سناریو همیشه ثابت هستند). متد PrintData همانطور که مشاهده می‌شود بصورت انتزاعی تعریف شده است، چرا که چاپ عملی است که در هر فرمت دارای خروجی متفاوتی می‌باشد.
لذا در ادامه داریم:

```
public class ExcelExporter : DataExporter
    {
        public override void PrintData()
        {
            Console.WriteLine("Data exported to Microsoft Excel!");
        }
    }

    public class PDFExporter : DataExporter
    {
        public override void PrintData()
        {
            Console.WriteLine("Data exported to PDF!");
        }
    }
```

کلاس ExcelExporter برای چاپ به فرمت اکسل می‌باشد. همانطور که مشاهده می‌شود این کلاس از کلاس انتزاعی DataExporter ارث بری کرده است. این بدین معنا است که کلاس ExcelExporter کارهای ReadData و ProcessData را از کلاس

پدر خود می‌گیرد و در ادامه نحوه‌ی چاپ مختص به خود را پیاده می‌کند. همین توضیحات در مورد PDFExporter نیز صادق است.
حال برای استفاده‌ی از این کدها داریم:

```
DataExporter dataExporter = new ExcelExporter();
            dataExporter.GetReport();
            Console.WriteLine("**************************");
            dataExporter = new PDFExporter();
            dataExporter.GetReport();
```

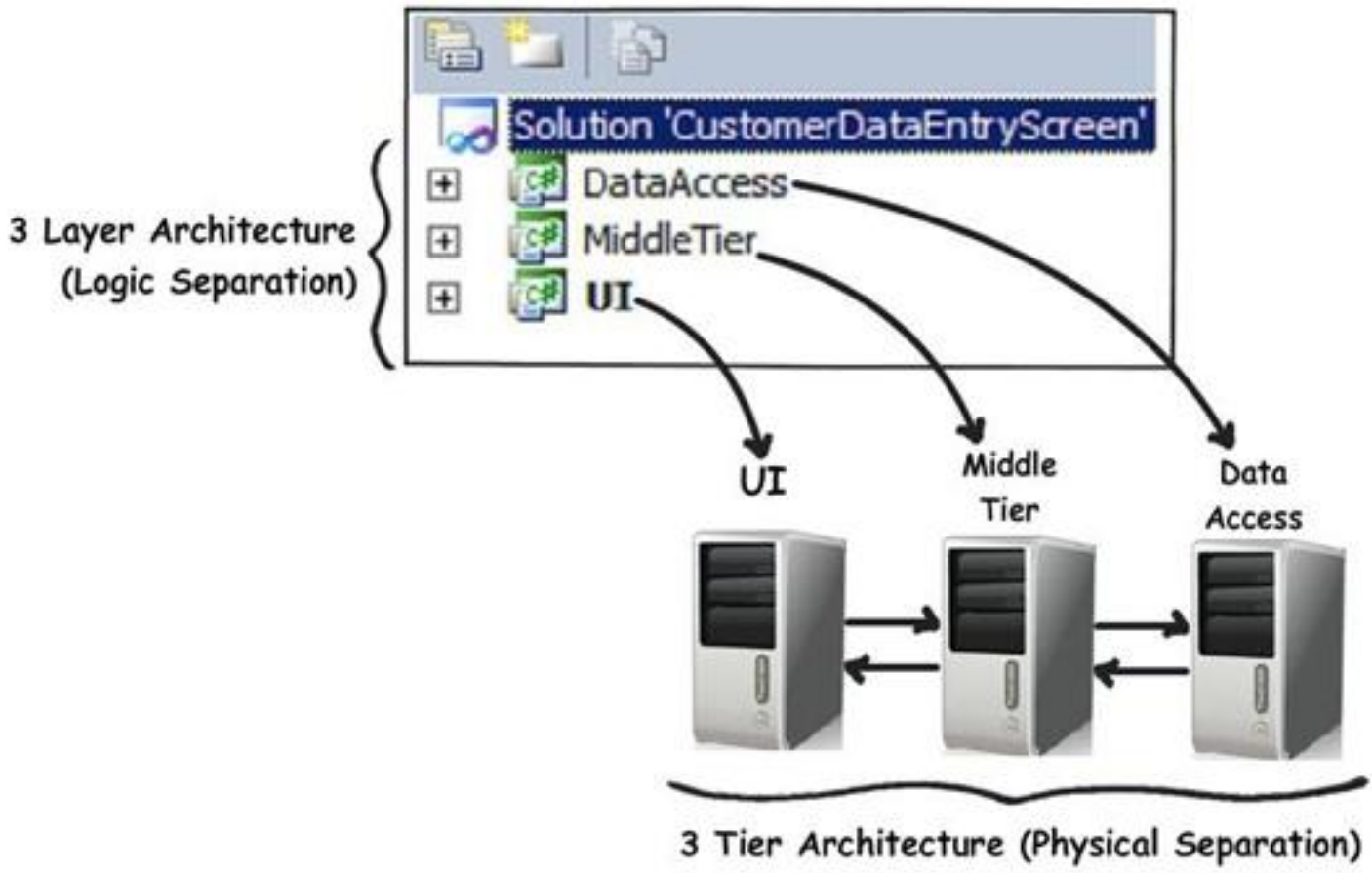
شما شاید بخواهید متدهای ReadData و ExportData و ProcessData را با سطح دسترسی متفاوتی از public تعریف نمایید که در این مقاله به این دلیل که خارج از بحث بود به آنها اشاره نشد و بصورت پیش فرض public در نظر گرفته شد.

عنوان: **تبادل داده ها بین لایه ها- قسمت اول**
نویسنده: ریوف مدرسی
تاریخ: ۱۳۹۴/۰۳/۳۱ ۲۱:۳۰
آدرس: www.dotnettips.info
گروه‌ها: ADO.NET, Design patterns, Architecture, OOP, N-Layer Architecture

معماری لایه بندی شده، یک معماری بسیار همه گیر می‌باشد. به این خاطر که به راحتی decoupling ، SOC و قدرت درک کد را بسیار بالا می‌برد. امروزه کمتر برنامه نویس و فعال حوضه‌ی نرم افزاری است که با لایه‌های کلی و وظایف آنها آشنا نباشد (UI layer آنچه که ما می‌بینیم، middle layer برای مقاصد منطق کاری، data access layer برای هندل کردن دسترسی به داده‌ها). اما مسئله ای که بیشتر برنامه نویسان و توسعه دهندگان نرم افزار با استانداردهای آن آشنا نیستند، راه‌های تبادل داده‌ها مابین layer ها می‌باشد. در این مقاله سعی داریم راه‌های تبادل داده‌ها را مابین لایه‌ها، تشریح کنیم.

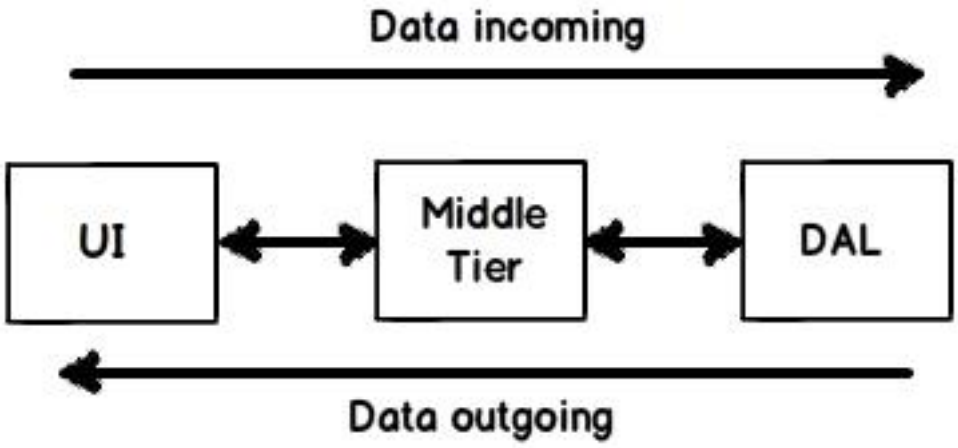
Layers و Tiers با هم متفاوت هستند

Layer با Tier متفاوت است. هنگامیکه در مورد مفهوم layer و Tier دچار شک شدید، دیاگرام ذیل می‌تواند به شما بسیار کمک کند. layer به مجزاسازی منطقی کد و Tier هم به مجزا سازی فیزیکی در ماشین‌های مختلف اطلاق می‌شود. توجه داشته باشید که این نکته یک شفاف سازی کلی در مورد یک مسئله مهم بود.



داده‌های وارد شونده(incoming) و خارج شونده(outgoing)

ما باید تبادل داده‌ها را از دو جنبه مورد بررسی قرار دهیم؛ اول اینکه داده‌ها چگونه به سمت لایه Data Access می‌روند، دوم اینکه داده‌ها چگونه به لایه UI پاس می‌شوند، در ادامه شما دلیل این مجزا سازی را درک خواهید کرد.



روش اول: Non-uniform

این روش اولین روش و احتمالا عمومی‌ترین روش می‌باشد. خوب، اجازه دهید از لایه‌ی UI به لایه DAL شروع کنیم. داده‌ها از لایه UI به Middle با استفاده از getter ها و setter ها ارسال خواهد شد. کد ذیل این مسئله را به روشنی نمایش می‌دهد.

```
Customer objCust = new Customer();
objCust.CustomerCode = "c001";
objCust.CustomerName = "Shivprasad";
```

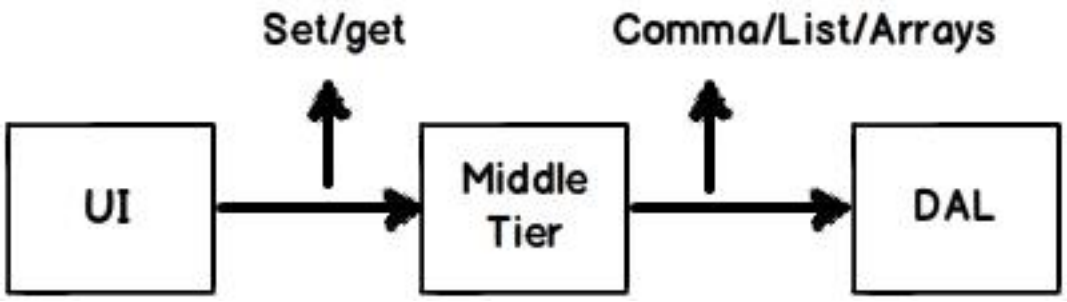
بعد از آن، از لایه Middle به لایه Data Access داده‌ها با استفاده از مجزاسازی به وسیله comma و آرایه‌ها و سایر روش‌های non-uniform پاس داده می‌شوند. در کد ذیل به متد Add دقت کنید که چگونه فراخوانی به لایه Data Access را با استفاده از پارامترهای ورودی انجام می‌دهد.

```
public class Customer
{
    private string _CustomerName = "";
    private string _CustomerCode = "";
    public string CustomerCode
    {
        get { return _CustomerCode; }
        set { _CustomerCode = value; }
    }
    public string CustomerName
    {
        get { return _CustomerName; }
        set { _CustomerName = value; }
    }
    public void Add()
    {
        CustomerDal obj = new CustomerDal();
        obj.Add(_CustomerName,_CustomerCode);
    }
}
```

کد ذیل، متد add در لایه Data Access را با استفاده از دو متد نمایش می‌دهد.

```
public class CustomerDal
{
    public bool Add(string CustomerName,string CustomerCode)
    {
        // Insert data in to DB
    }
}
```

بنابراین اگر بخواهیم به صورت خلاصه نحوه پاس دادن داده‌ها را در روش non-uniform بیان کنیم، شکل ذیل به زیبایی این مسئله را نشان می‌دهد.



- از لایه UI به لایه Middle با استفاده از setter و getter

- از لایه Middle به لایه data access با استفاده از array ، input ، comma

حال نوبت این است بررسی کنیم که چگونه داده‌ها از DAL به UI در روش non-uniform پاس خواهند شد. بنابراین اجازه دهید که اول از UI شروع کنیم. از لایه UI داده‌ها با استفاده از object های لایه Middle واکشی می‌شوند.

```
Customer obj = new Customer();
List<Customer> oCustomers = obj.getCustomers();
```

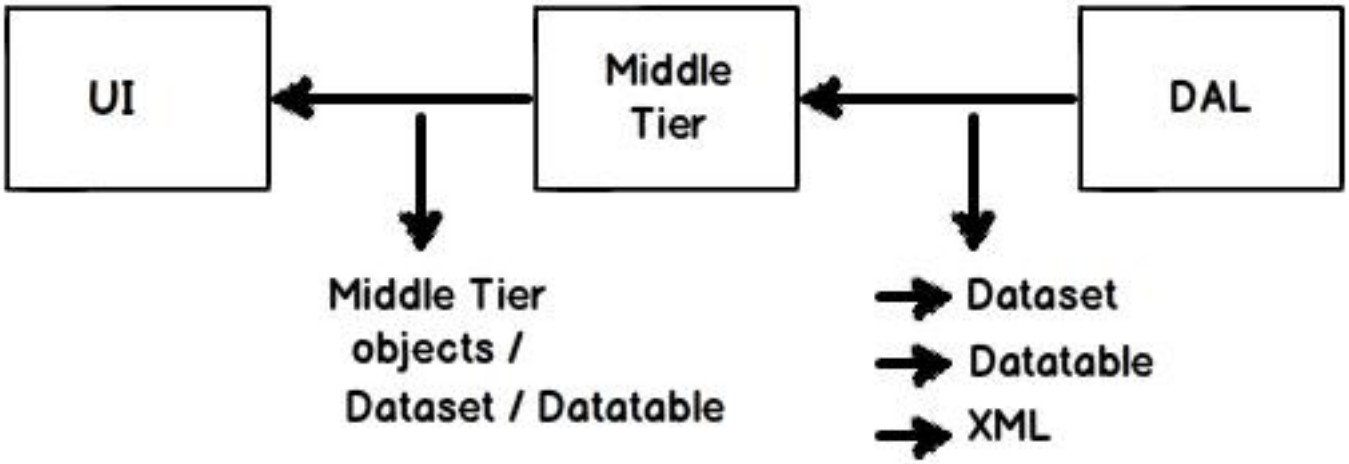
از لایه Middle هم داده‌ها با استفاده از datatable ، dataset و xml پاس خواهند شد. مهمترین مسئله برای لایه middle ، loop بر روی dataset و تبدیل آن به strong type object ها می‌باشد. برای مثال می‌توانید کد تابع getCustomers که بر روی ، dataset loop می‌زند و یک لیست از Customer ها را آماده می‌کند در ذیل مشاهده کنید. این تبدیل باید انجام شود، به این دلیل که UI به کلاس‌های strongly typed دسترسی دارد.

```
public class Customer
{
    private string _CustomerName = "";
    private string _CustomerCode = "";
    public string CustomerCode
    {
        get { return _CustomerCode; }
        set { _CustomerCode = value; }
    }
    public string CustomerName
    {
        get { return _CustomerName; }
        set { _CustomerName = value; }
    }
    public List<Customer> getCustomers()
    {
        CustomerDal obj = new CustomerDal();
        DataSet ds = obj.getCustomers();
        List<Customer> oCustomers = new List<Customer>();
        foreach (DataRow orow in ds.Tables[0].Rows)
        {
            // Fill the list
        }
        return oCustomers;
    }
}
```

با انجام این تبدیل به یکی از بزرگترین اهداف معماری لایه بندی شده می‌رسیم؛ یعنی اینکه « UI نمی‌تواند به طور مستقیم به کامپوننت‌های لایه Data Access مانند OLEDB ، ADO.NET و غیره دستیابی داشته باشد. با این روش اگر ما در ادامه متدولوژی Data Access را تغییر دهیم تاثیری بر روی لایه UI نمی‌گذارد.» آخرین مسئله اینکه کلاس CustomerDal یک Dataset را با استفاده از ADO.NET بر می‌گرداند و Middle از آن استفاده می‌کند.

```
public class CustomerDal
{
    public DataSet getCustomers()
    {
        // fetch customer records
        return new DataSet();
    }
}
```

حال اگر بخواهیم حرکت داده‌ها را به لایه UI ، به صورت خلاصه بیان کنیم، شکل ذیل کامل این مسئله را نشان می‌دهد.



· داده‌ها از لایه DAL به لایه Middle با استفاده از XML ، Datareader ، Dataset ارسال خواهند شد.

· از لایه Middle به UI از strongly typed classes استفاده می‌شود.

مزایا و معایب روش non-uniform

یکی از مزایای non-uniform

· به راحتی قابل پیاده سازی می‌باشد، در مواردی که روش data access تغییر نمی‌کند این روش کارآیی لازم را دارد.

تعدادی از معایب این روش

· به خاطر اینکه یک ساختار uniform نداریم، بنابراین نیاز داریم که همیشه در هنگام عبور از یک لایه به یک لایه‌ی دیگر از یک ساختار به یک ساختار دیگر تبدیل را انجام دهیم.

· برنامه نویسان از روش‌های خودشان برای پاس دیتا استفاده می‌کنند؛ بنابراین این مسئله خود باعث پیچیدگی می‌شود.

· اگر برای مثال شما بخواهید متدولوژی Data Access خود را تغییر دهید، تغییرات بر تمام لایه‌ها تاثیر می‌گذارد.

نظرات خوانندگان

نویسنده: بابک جهانگیری
تاریخ: ۱۳۹۴/۰۴/۰۴ ۱۳:۲۰

آیا در این روش می‌توان به صورت DataView لیست مشتریها رو برگردوند به جای اینکه از List<Customer> استفاده کنیم ؟ باز هم به آن non-uniform می‌گویند ؟

نویسنده: ریوف مدرسی
تاریخ: ۱۳۹۴/۰۴/۰۵ ۱۷:۵۳

در این روش مسئله اصلی این نیست که داده‌ها رو به صورت list یا Dataview برگردونید، بلکه مسئله اصلی این است که شما مجبورید در گذر از هر لایه تبدیل ساختار داده‌ها را انجام دهید، پس نکته این روش این است که تعداد تبدیل ساختار داده‌ها زیاد است.

نویسنده: محسن اسماعیل پور
تاریخ: ۱۳۹۴/۰۴/۰۸ ۸:۲۵

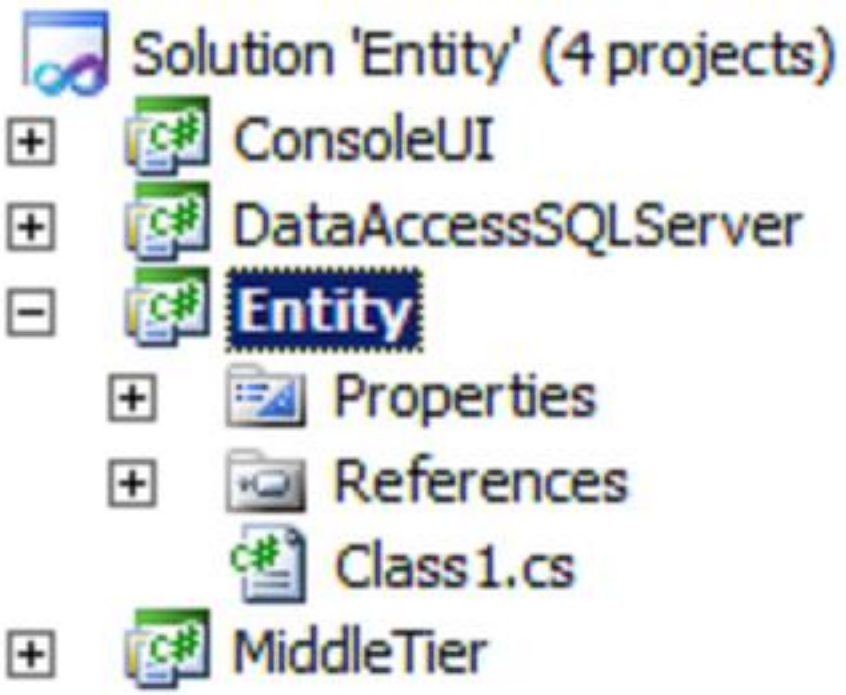
مدل Customer که شما برای مثالهایتان از آن استفاده کرده اید از Active record pattern تبعیت میکند. از آنجا که Entity یا Model با عملیات CRUD لایه دیتا Couple شده و بعضا ممکن است Business Logic داخل این متدها قرار گیرد، این مسئله با Sepration Of Concern منافات دارد.

عنوان: **تبادل داده ها بین لایه ها- قسمت دوم**
نویسنده: ریوف مدرسی
تاریخ: ۱۳۹۴/۰۴/۳۰ ۱:۲۵
آدرس: www.dotnettips.info
گروه‌ها: ADO.NET, Design patterns, Architecture, OOP, N-Layer Architecture

قسمت اول : تبادل داده‌ها بین لایه ها- قسمت اول

روش دوم: Uniform(Entity classes)

روش دیگر پاس دادن داده‌ها، روش uniform است. در این روش کلاس‌های Entity ، یک سری کلاس ساده به همراه یکسری Property های Get و Set می‌باشند. این کلاس‌ها شامل هیچ منطق کاری نمی‌باشند. برای مثال کلاس CustomerEntity که دارای دو Property ، Customer Name و Customer Code می‌باشد. شما می‌توانید تمام Entity ها را به صورت یک پروژه‌ی مجزا ایجاد کرده و به تمام لایه‌ها رفرنس دهید.




```
public class CustomerEntity
{
    protected string _CustomerName = "";
    protected string _CustomerCode = "";
    public string CustomerCode
    {
        get { return _CustomerCode; }
        set { _CustomerCode = value; }
    }
    public string CustomerName
    {
        get { return _CustomerName; }
        set { _CustomerName = value; }
    }
}
```

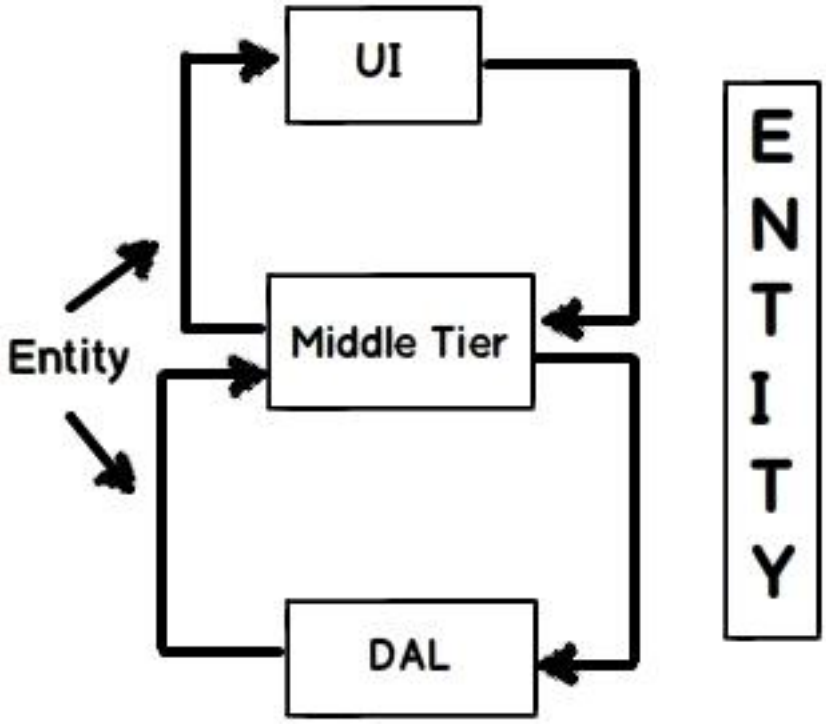
خوب، اجازه دهید تا از CustomerDal شروع کنیم. این کلاس یک Collection از CustomerEntity را بر می‌گرداند و همچنین یک CustomerEntity را برای اضافه کردن به دیتابیس . توجه داشته باشید که لایه Data Access وظیفه دارد تا دیتای دریافتی از دیتابیس را به CustomerEntity تبدیل کند.

```
public class CustomerDal
{
    public List<CustomerEntity> getCustomers()
    {
        // fetch customer records
        return new List<CustomerEntity>();
    }
    public bool Add(CustomerEntity obj)
    {
        // Insert in to DB
        return true;
    }
}
```

لایه Middle از CustomerEntity ارث بری می‌کند و یکسری operation را به entity class اضافه خواهد کرد. داده‌ها در قالب Entity Class به لایه Data Access ارسال می‌شوند و در همین قالب نیز بازگشت داده می‌شوند. این مسئله در کد ذیل به روشنی مشاهده می‌شود.

```
public class Customer : CustomerEntity
{

    public List<CustomerEntity> getCustomers()
    {
        CustomerDal obj = new CustomerDal();

        return obj.getCustomers();
    }
    public void Add()
    {
        CustomerDal obj = new CustomerDal();
        obj.Add(this);
    }
}
```

لایه UI هم با تعریف یک Customer و فراخوانی operation های مربوط به آن، داده‌ی مد نظر خود را در قالب CustomerEntity بازیابی خواهد کرد. اگر بخواهیم عمکرد روش uniform را خلاصه کنیم باید بگوییم، در این روش دیتای رد و بدل شده‌ی مابین کلیه لایه‌ها با یک ساختار استاندارد، یعنی Entity پاس داده می‌شوند.



مزایا و معایب روش uniform

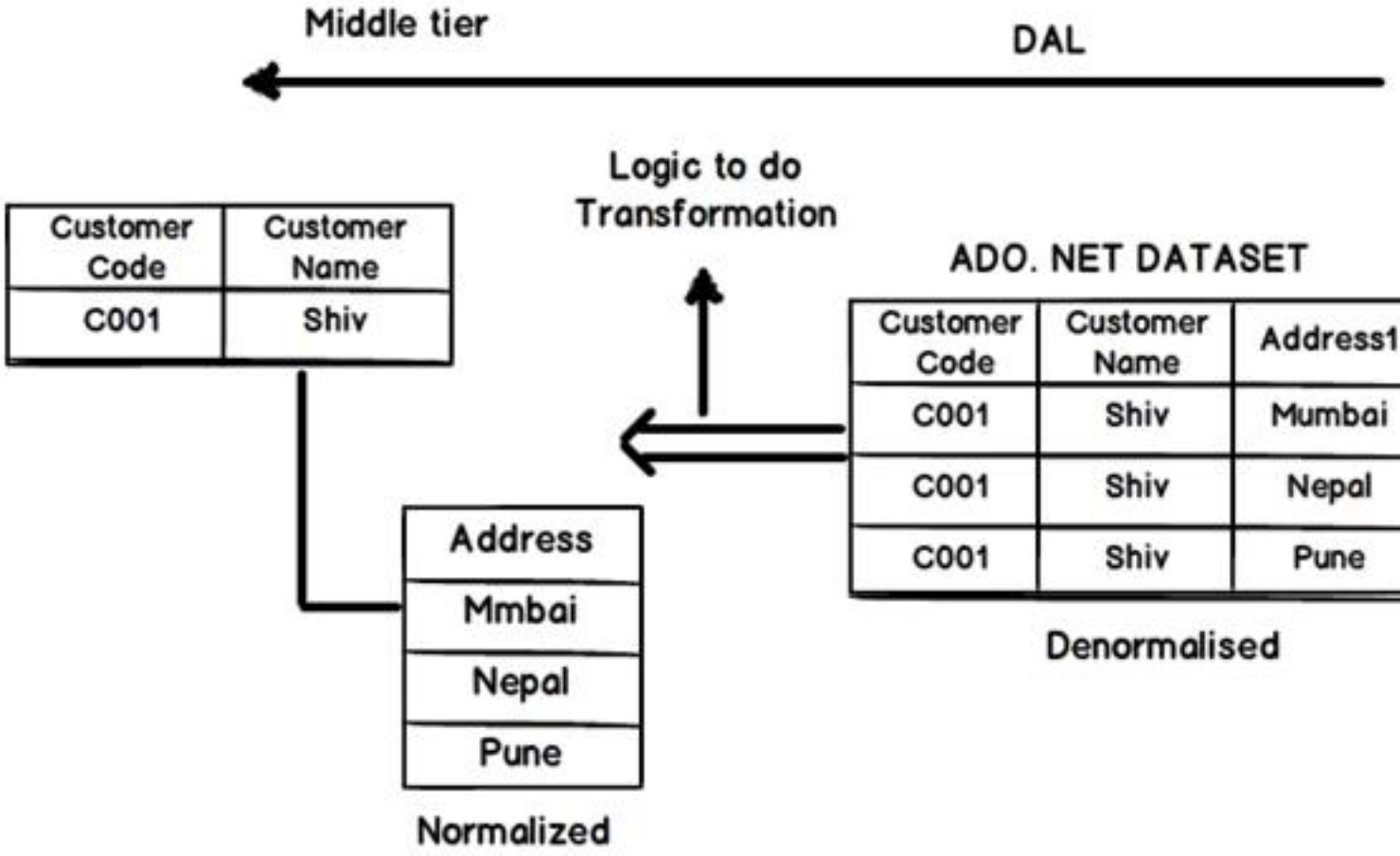
مزایا

- Strongly typed به صورت در تمامی لایه‌ها قابل دسترسی و استفاده می‌باشد.

· به دلیل اینکه از ساختار عمومی Entity استفاده می‌کند، بنابراین فقط یکبار نیاز به تبدیل داده‌ها وجود دارد. به این معنی که کافی است یک بار دیتای واکشی شده از دیتابیس را به یک ساختار Entity تبدیل کنید و در ادامه بدون هیچ تبدیل دیگری از این Entity استفاده کنید.

معایب

· تنها مشکلی که این روش دارد، مشکلی است به نام Double Loop . هنگامیکه شما در مورد کلاس‌های entity بحث می‌کنید، ساختارهای دنیای واقعی را مدل می‌کنید. حال فرض کنید شما به دلیل یکسری مسایل فنی دیتابیس خود را Optimize کرده اید. بنابراین ساختار دنیای واقعی با ساختاری که شما در نرم افزار مدل کرده‌اید متفاوت می‌باشد. بگذارید یک مثال بزنیم؛ فرض کنید که یک customer دارید، به همراه یکسری Address . همان طور که ذکر کردیم، به دلیل برخی مسایل فنی (denormalized) به صورت یک جدول در دیتا بیس ذخیره شده است. بنابراین سرعت واکشی اطلاعات بیشتر است. اما خوب اگر ما بخواهیم این ساختار را در دنیای واقعی بررسی کنیم، ممکن است با یک ساختار یک به چند مانند شکل ذیل برخورد کنیم.



بنابراین مجبوریم یکسری کد جهت این تبدیل همانند کد ذیل بنویسیم.

```
foreach (DataRow o1 in oCustomers.Tables[0].Rows)
{
    obj.Add(new CustomerEntyityAddress()); // Fills customer
    foreach (DataRow o in oAddress.Tables[0].Rows)
    {
        obj[0].Add(new AddressEntity()); // Fills address
    }
}
```

عنوان: **تبادل داده‌ها بین لایه‌ها؛ قسمت آخر**
نویسنده: سید ریوف مدرسی
تاریخ: ۱۳۹۴/۰۶/۰۳ ۲۰:۴۵
آدرس: www.dotnettips.info
گروه‌ها: ADO.NET, Design patterns, Architecture, OOP, N-Layer Architecture, Architectural Patterns

روش سوم: DTO (Data transfer objects)

در قسمت‌های قبلی دو روش از روش‌های موجود جهت تبادل داده‌ها بین لایه‌ها، ذکر گردید و علاوه بر این، مزایا و معایب هر کدام از آنها نیز ذکر شد. در این قسمت دو روش دیگر، به همراه مزایا و معایب آنها برشمرده می‌شود. لازم به ذکر است هر کدام از این روش‌ها می‌تواند با توجه به شرایط موجود و نظر طراح نرم افزار، دارای تغییراتی جهت رسیدن به یکسری اهداف و فاکتورها در نرم افزار باشد.

در این روش ما سعی می‌کنیم طراحی کلاس‌ها را به اصطلاح مسطح (flatten) کنیم تا بر مشکل double loop که در قسمت قبل بحث کردیم غلبه کنیم. در کد ذیل مشاهده می‌کنید که چگونه کلاس CusomerDTO از CustomerEntity ، مشتق می‌شود و کلاس Address را با CustomerEntity ادغام می‌کند؛ تا برای افزایش سرعت لود و نمایش داده‌ها، یک کلاس de-normalized شده ایجاد نماید.

```
public class CustomerDTO : CustomerEntity
{
    public AddressEntity _Address = new AddressEntity();
}
```

در کد ذیل می‌توانید مشاهده کنید که چگونه با استفاده از فقط یک loop یک کلاس de-normalized شده را پر می‌کنیم.

```
foreach (DataRow o1 in oCustomers.Tables[0].Rows)
{
CustomerDTO o = new CustomerDTO();
o.CustomerCode = o1[0].ToString();
o.CustomerName = o1[1].ToString();
o._Address.Address1 =  o1[2].ToString();
o._Address.Address2 = o1[3].ToString();
obj.Add(o);
}
```

UI هم به راحتی می‌تواند DTO را فراخوانی کرده و دیتا را دریافت کند.

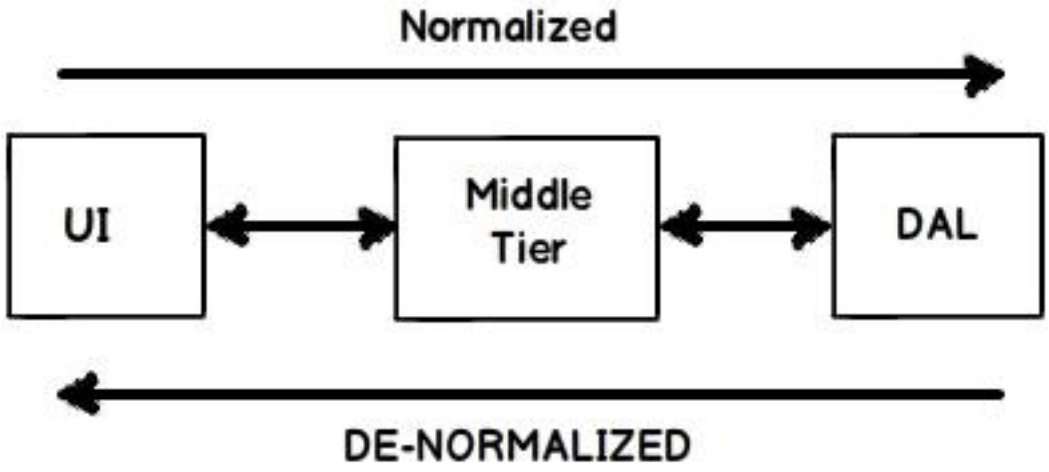
مزایا و معایب روش DTO

یکی از بزرگترین مزایای این روش سرعت زیاد در بارگذاری اطلاعات، به دلیل استفاده کردن از ساختار de-normalized می‌باشد. اما همین مسئله خود یک عیب محسوب می‌شود؛ به این دلیل که اصول شئ گرایی را نقض می‌کند.

روش چهارم: Hybrid approach (Entity + DTO)

از یک طرف کلاس‌های Entity که دنیای واقعی را مدل خواهند کرد و همچنین اصول شئ گرایی را رعایت می‌کنند و از یک طرف دیگر DTO نیز یک ساختار flatten را برای رسیدن به اهداف کارآیی دنبال خواهند کرد. خوب، به نظر می‌رسد که بهترین کار استفاده از هر دو روش و مزایای آن روش‌ها باشد.

زمانیکه سیستم، اهدافی مانند انجام اعمال CRUD را دنبال می‌کند و شما می‌خواهید مطمئن شوید که اطلاعات، دارای integrity می‌باشند و یا اینکه می‌خواهید این ساختار را مستقیما به کاربر نهایی ارائه دهید، استفاده کردن از روش (Entity) به عنوان یک روش normalized می‌تواند بهترین روش باشد. اما اگر می‌خواهید حجم بزرگی از دیتا را نمایش دهید، مانند گزارشات طولانی، بنابراین استفاده از روش DTO با توجه به اینکه یک روش de-normalized به شمار می‌رود بهترین روش می‌باشد.



کدام روش بهتر است؟

Non-uniform : این روش برای حالتی است که متدهای مربوط به data access تغییرات زیادی را تجربه نخواهند کرد. به عبارت دیگر، اگر پروژه‌ی شما در آینده دیتابیس‌های مختلفی را مبتنی بر تکنولوژی‌های متفاوت، لازم نیست پشتیبانی کند، این روش می‌تواند بهترین روش باشد.

Uniform : Entity, DTO, or hybrid : اگر امکان دارد که پروژه‌ی شما با انواع مختلف دیتابیس‌ها مانند Oracle و Postgres ارتباط برقرار کند، استفاده کردن از این روش پیشنهاد می‌شود.