جزوه درس برنامه نویسی شیئ گرا

تعریف شئ گرایی و برنامه نویسی شئ گرا

قبل از هر کاری، بهتر است با مفهوم برنامه نویسی شئ گرا و این که به چه زبانی شئ گرایی گفته می شود آشنا شویم. برنامه نویسی شئ گرا، بر خلاف زبان های Procedural که همه چیز در آن بر اساس روال ها تعریف می شدند، مدل سازی نرم افزار بر اساس اشیاء انجام می شود. بهتر است با یک مثال ادامه دهیم، در دنیایی که ما در آن زندگی می کنیم تمام موجودیت های اطراف ما تحت عنوان شئ شناخته می شوند، خانه هایی که در آن زندگی می کنیم، وسایل داخل خانه مانند یخچال، تلویزیون، مانیتور کامپیوتری که با آن کار می کنیم، ماشینی که سوار می شویم و هر چیزی که در دنیا وجود دارد تحت عنوان یک شئ شناخته می شود .

اما هر شئ که ما به عنوان یک موجودیت به آن نگاه می کنیم شامل یکسری خصوصیات و رفتارها می باشد. در زیر به تعریف خصوصیات و رفتارهای یک شئ می پردازیم :

- خصوصیات یا :Properties خصوصیات مجموعه ای از صفات هستند که یک شئ را توصیف می کنند. برای مثال شئ ای با نام انسان را در نظر بگیرید، این شئ یکسری خصوصیات دارد مانند رنگ مو، قد، وزن، رنگ چشم و غیره. تمامی این پارامترها که به توصیف یک شئ می پردازند تحت عنوان خصوصیت یا Property شناخته می شوند.
- رفتارها یا :Behaviors هر شئ علاوه بر خصوصیات، شامل یکسری رفتارها می باشد، این رفتارها در حقیقت کاریست که یک شئ می تواند انجام دهد. دوباره شئ انسان را در نظر بگیرید، این شئ می تواند نگاه کند، صحبت کند یا بشنود. رفتارها با خصوصیات تفاوت دارند و به کاری گفته می شوند که یک شئ می تواند انجام دهد.

در زبان های برنامه نویسی شئ گرا نیز ما باید به شناسایی موجودیت ها و اشیاء مورد استفاده در برنامه بپردازیم و خصوصیات و رفتارهای آن را تعریف کنیم. فرض کنید تصمیم داریم برنامه ای برای مدیریت یک کتابخانه بنویسیم. برنامه کتابخانه شامل یکسری اشیاء می باشد مانند :

۱. عضو کتابخانه
۲. اپراتور نرم افزار کتابخانه
۳. دسته بندی کتاب (که همان قفسه هایی که کتاب ها در آن دسته بندی می شوند می باشد(
۴. کتاب

پس از شناسایی موجودیت ها باید خصوصیت ها و رفتارهای آن ها را شناسایی کنیم. برای مثال شئ عضو کتابخانه را در نظر بگیرید. این شئ شامل یکسری خصوصیت ها به شرح زیر می باشد :

۱. کد عضویت
۲. نام
۳. نام خانوادگی
۴. شماره ملی
۵. نام پدر
۶. جنسیت

همچنین هر عضو یکسری رفتارهایی دارد که مختص به عملیات های کتابخانه می باشد. برای مثال عضو کتابخانه می تواند رفتارهای زیر را داشته باشد :

۱. دریافت کتاب
۲. پس دادن کتاب
۳. ورود به کتابخانه
۴. خروج از کتابخانه

پس از آنکه رفتارها و خصوصیات اشیاء یک برنامه شناسایی شدند، باید نسبت به پیاده سازی آنها در نرم افزار اقدام کنیم که در قسمت بعدی در مورد پیاده سازی اشیاء و تعریف خصوصیات و رفتارهای آنها توضیح خواهیم داد .

مفاهیم اساسی در برنامه نویسی شئ گرا

برای ادامه مباحث مربوط به برنامه نویسی شئ گرا، لازم است که با چهار مفهوم اساسی در زبان های برنامه شئ گرا آشنا شویم. این چهار مفهوم، ارکان اساسی و ستون های برنامه نویسی شئ گرا می باشند که در زیر به بررسی هر یک از انها خواهیم پرداخت :

Abstraction

زمانی که تصمیم داریم برنامه ای را به صورت شئ گرا بنویسیم، باید شروع به تحلیل سیستم و شناسایی موجودیت های آن کنیم. در بالا مثالی را در مورد برنامه کتابخانه بررسی کردیم. شئ عضو را در نظر بگیرید، شاید این عضو خصوصیت های بسیاری داشته باشد، مانند رنگ چشم، رنگ مو، قد، وزن، رنگ پوست و ... . اما آیا تمامی این خصوصیات در سیستم به کار می آید؟ در مورد رفتارهای یک شئ نیز همین موضوع صدق می کند. مفهوم Abstraction به ما می گوید زمان بررسی یک موجودیت، تنها خصوصیات و رفتارهایی باید در تعریف موجودیت لحاظ شوند که مستقیماً در سیستم کاربرد دارند. در حقیقت Abstractionمانند فیلتری عمل می کنند که تنها خصوصیات و رفتارهای مورد استفاده در برنامه ای که قصد نوشتن آن را داریم از آن عبور می کنند .

Encapsulation

فرض کنید ماشین جدیدی خریداری کرده اید، پشت فرمان ماشین می نشینید و ماشین را استارت می زنید. استارت زدن ماشین خیلی ساده است، قرار دادن سوئیچ و چرخاندن آن و روشن شدن ماشین. اما آیا پروسه ای که داخل ماشین طی شده برای روشن شدن نیز همینقدر ساده است؟ صد در صد، عملیات های بسیار دیگری اتفاق می افتد تا ماشین روشن شود. اما شما تنها سوئیچ را چرخانده و ماشین را روشن میکنید. در حقیقت پیچیدگی عملیات روشن شدن ماشین از راننده ماشین پنهان شده است. به این عملیات Encapsulation یا پنهان سازی پیچیدگی پیاده سازی عملیات های درون یک شئ می گویند .

Inheritance

می توان گفت Inheritance یا وراثت اصلی ترین مفهوم در برنامه نویسی شئ گرا است. زمانی که شما خوب این مفهوم را درک کنید ۷۰ درصد از مفاهیم برنامه نویسی شئ گرا را درک کرده اید. برای درک بهتر این مفهوم مثالی میزنیم. تمامی انسان های متولد شده بر روی کره خاکی از یک پدر و مادر متولد شده اند. در حقیقت این پدر و مادر والدین انسان هستند. زمانی که انسانی متولد می شود یکسری خصوصیات و ویژگی ها را از والدین خود به ارث می برد، مانند رنگ چشم، رنگ پوست یا برخی ویژگی های رفتاری. در برنامه نویسی شئ گرا نیز به همین صورت می باشد. زمانی که شما موجودیت را طراحی می کنید، می توانید برای آن یک کلاس Base یا والد در نظر بگیرید که شئ فرزند تمامی خصوصیات و رفتارهای شئ والد را به ارث خواهد برد. مهمترین ویژگی وراثت، استفاده مجدد از کدهای نوشته شده است که حجم کدهای نوشته شده را به صورت محسوسی کاهش می دهد. در بخش های بعدی در مورد این ویژگی به صورت کامل توضیح خواهیم داد .

Polymorphism

در فرهنگ لغت این واژه به معنای چند ریختی ترجمه شده است. اما در برنامه نویسی شئ گرا چطور؟ خیلی از افراد با این مفهوم مشکل دارند و درک صحیحی از آن پیدا نمی کنند. مفهوم Polymorphism رابطه مستقیمی با Inheritance دارد. یعنی شما ابتدا نیاز دارید مفهوم وراثت را خوب درک کرده و سپس به یادگیری Polymorphism بپردازید. باز هم برای درک مفهوم Polymorphism یک مثال از دنیای واقعی میزنیم. در کره خاکی ما انسان های مختلفی در کشور های مختلف و شهر های مختلف با گویش های مختلف زندگی می کنند. اما تمامی این ها انسان هستند. در اینجا انسان را به عنوان یک شئ والد و انسان چینی، انسان ایرانی و انسان آمریکایی را به عنوان اشیاء فرزند که از شئ انسان مشتق شده اند یا والد آنها کلاس انسان می باشد را در نظر بگیرید. کلاس انسان رفتاری را تعریف می کند به نام صحبت کردن. اما اشیاء فرزند آن، به یک صورت صحبت نمی کنند، انسان ایرانی با زبان ایرانی، چینی با زبان چینی و آمریکایی با زبان آمریکایی صحبت می کند. در حقیقت رفتاری که در شئ والد تعریف شده، در شئ های فرزند مجدد تعریف می شود یا رفتار آن تغییر می کند. این کار

مفهوم مستقیم Polymorphism می باشد. در زبان های برنامه نویسی شئ گرا، Polymorphismبه تغییر رفتار یک شئ در اشیاء فرزند آن گفته می شود. در زبان سی شارپ این کار با کمک تعریف متدها به صورت virtual و override کردن آنها در کلاس های فرزند انجام می شود. همچنین Polymorphism با کمک Interface ها قابل پیاده سازی است که در بخش های بعدی در مورد این ویژگی ها به صورت کامل صحبت خواهیم کرد .

همانطور که در قسمت قبل گفتیم، زمانی که قصد نوشتن برنامه ای به صورت شئ گرا را داریم، باید موجودیت های مورد استفاده را در برنامه مدل سازی کنیم. این موجودیت ها همان اشیاء هستند که در سیستم مورد استفاده قرار میگیرند. اما شیوه مدل سازی و استفاده از اشیاء چگونه خواهد بود؟ در اینجا باید با دو مفهوم آشنا شویم: ۱. کلاس ها و ۲. اشیاء .

۱. کلاس: نمونه ای از یک شئ که داخل برنامه طراحی می شود را کلاس می گویند. برای اینکه با مفهوم کلاس بیشتر آشنا شوید یک مثال از دنیای واقعی می زنیم. فرض کنید تصمیم به ساخت یک خانه دارید. اولین چیزی که به آن نیاز خواهید داشت نقشه خانه ایست که تصمیم دارید بسازید. نقشه یک طرح اولیه و مفهومی از ساختمان به شما می دهد و بعد از روی نقشه اقدام به ساخت خانه می کنید. نقشه شامل تمامی بخش های خانه است، اطاق پذیرایی، آشپزخانه، حمام، سرویس بهداشتی و سایر بخش ها. اما فقط یک نقشه در اختیار دارید. نمی توانید از اطاق پذیرایی داخل نقشه استفاده کنید. کلاس دقیقاً معادل نقشه ای است که شما برای ساختمان خود کشیده اید. کلاس یک نمونه اولیه از موجودیت ایست که باید اشیاء از روی آن ساخته شوند.
۲. شئ: باز هم به سراغ مثال قبلی می رویم. بعد از کشیدن نقشه ساختمان شما باید اقدام به ساخت خانه کنید. بعد از اتمام عملیات ساخت، خانه شما قابل سکونت بوده و شما می توانید از آن استفاده کنید. همچنین از روی یک نقشه ساختمانی می توان چندین ساختمان ساخت. شئ دقیقاً معادل همان مفهوم ساختمانی است که از روی نقشه ساخته شده است. شما بعد از اینکه کلاس را تعریف کردید، باید از روی کلاس شئ بسازید تا بتوانید از آن استفاده کنید. در حقیقت کلاس به صورت مستقیم قابل استفاده نیست، مگر اینکه شامل اعضای static باشد که در بخش های بعدی با آنها آشنا خواهیم شد. همچنین می توان از روی یک کلاس، یک یا چندین شئ تعریف کرد.

حال که با مفاهیم اولیه کلاس و شئ آشنا شدید، بهتر است با نحوه تعریف کلاس و ساخت شئ آشنا شویم. تعریف کلاس انجام می شود. ساختار کلی این دستور به صورت زیر است C# در زبان class بوسیله کلمه کلیدی:

```
{access-modifier} class {name}
{
}
```

قسمت access-modifier سطح دسترسی به کلاس را تعیین می کند. ما زمانی که اقدام به تعریف کلاس یا هر قطعه کدی در زبان c# می کنیم، می توانیم سطح دسترسی به آن کد را تعیین کنیم. اما سطح دسترسی به چه معناست؟ در قسمت های اولیه آموزش گفتیم که زمان ایجاد یک پروژه به زبانC# ، برای شما یک solution ایجاد شده که هر solution می تواند شامل چندین پروژه باشد. برای مثال، کلاس یا اعضای یک کلاس را تعریف می کنیم، می توانیم مشخص کنیم که این کلاس از کدام قسمت های پروژه قابل دسترس باشد. سطوح دسترسی زیر در زبان سی شارپ تعریف شده اند :

۱. private: این سطح دسترسی مشخص می کند که قطعه کد تعریف شده تنها داخل خود پروژه یا Scope مربوطه قابل دسترس باشند. برای مثال کلاسی که به صورت private تعریف شده باشد، تنها داخل همان پروژه قابل

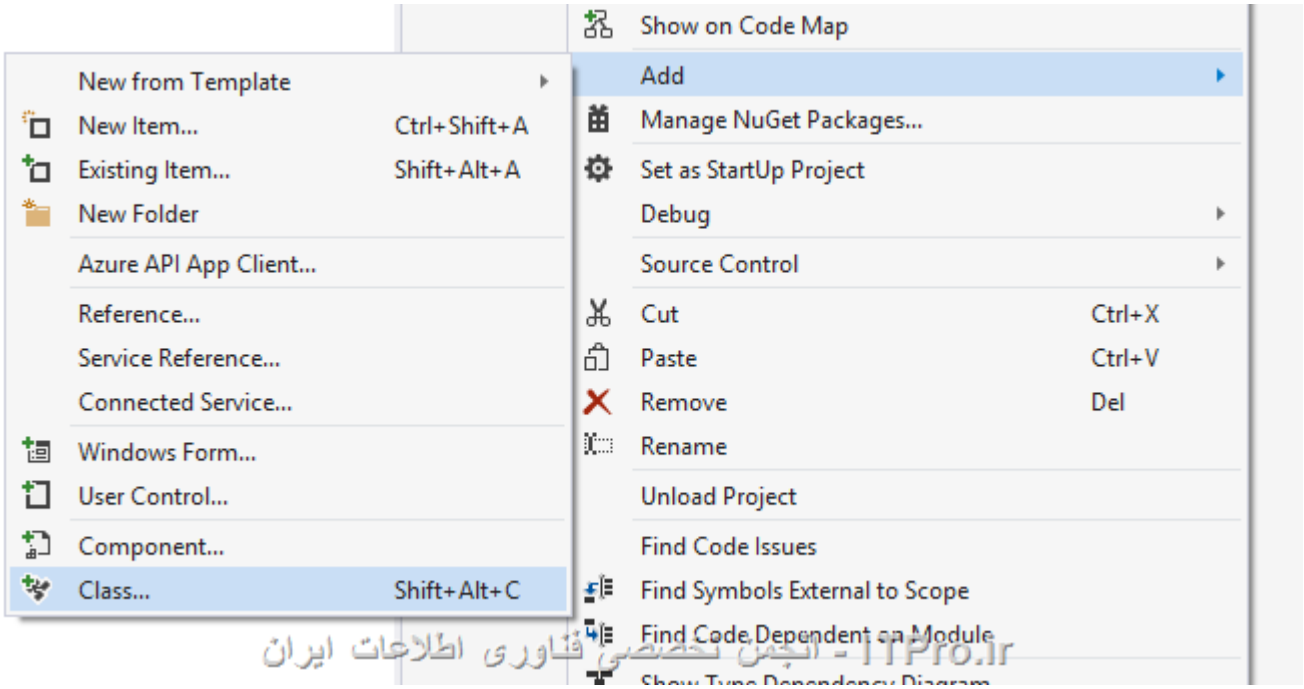
دسترس بوده و از سایر پروژه هایی که در solution تعریف شده قابل دسترس نخواهد بود، یا اعضای کلاسی که به صورت private تعریف شده اند، تنها در Scope همان کلاس که بین علامت های {} می باشد قابل دسترس خواهند بود.

۲. public: کدهایی که با این سطح دسترسی مشخص شده باشند، در تمامی قسمت های پروژه و سایر پروژه ها قابل دسترس خواهند بود.
۳. internal: سطوح دسترسی internal ، تنها داخل همان پروژه قابل دسترس بوده و سایر پروژه ها به آنها دسترسی نخواهند داشت. این سطح دسترسی برای اعضای کلاس ها کاربرد زیادی دارد.
۴. protected: این سطح دسترسی زمانی که از مفهوم inheritance استفاده کنیم کاربرد دارد. در قسمت وراثت این سطح دسترسی را به تفصیل مورد بررسی قرار خواهیم داد.
۵. internal protected: همانند قسمت protected ، این دسترسی نیز در قسمت وراثت توضیح داده خواهد شد که تلفیقی از دسترسی های internal و protected می باشد.

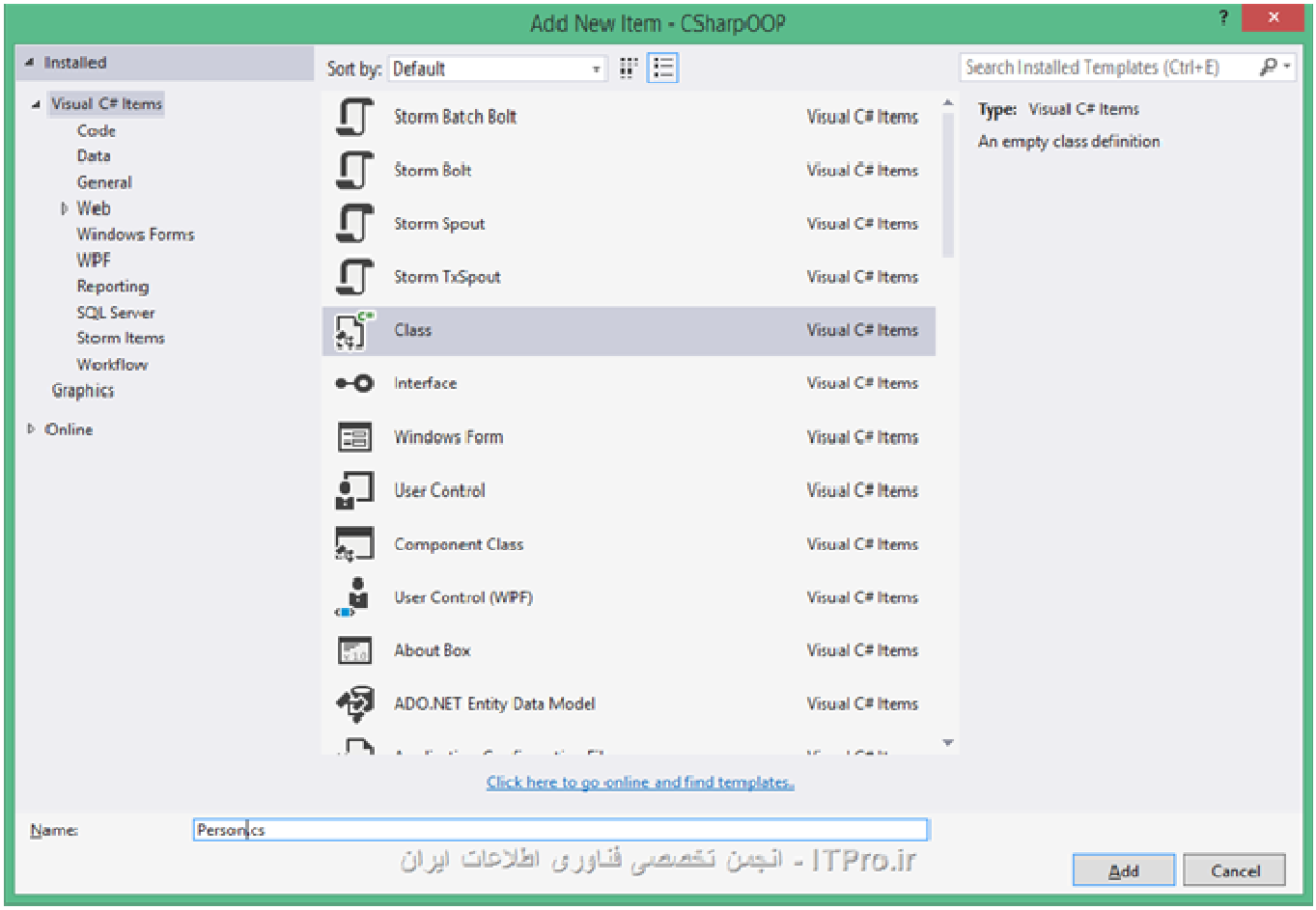
بعد از access-modifier ، با کلمه کلیدی class می گوییم که قصد تعریف یک کلاس را داریم و بعد از کلمه کلیدی class در قسمت name نام کلاس را مشخص می کنیم. نام کلاس باید همیشه بر اساس قاعده PascalCase نام گذاری شود. ما دو شیوه نام گذاری داریم :

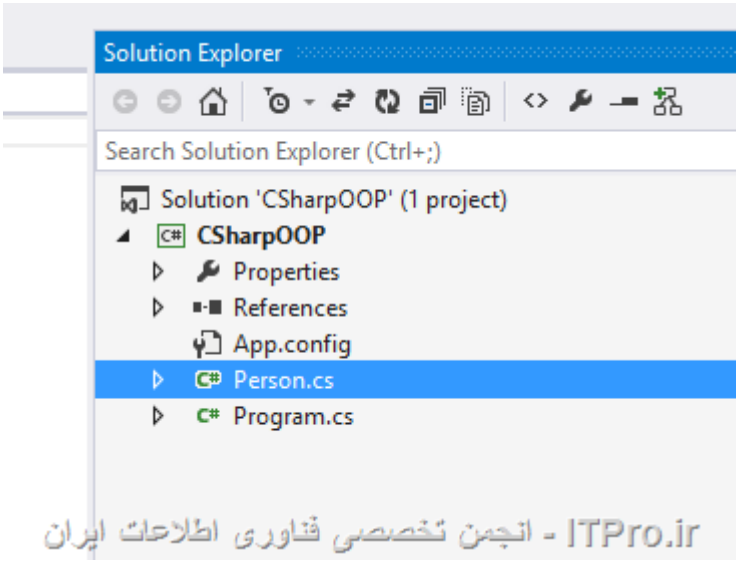
۱. camelCase: در این شیوه نام گذاری، کاراکتر ابتدای هر کلمه باید با حروف بزرگ نوشته شود غیر از کلمه اول. مانند sampleDictionary. .: newEmployee
۲. PascalCase: در این شیوه نام گذاری، کاراکتر ابتدای هر کلمه باید با حروف بزرگ نوشته شود، مانند : SampleDictionary، NewEmployee

حال، تصمیم داریم یک کلاس با نام Person تعریف کنیم. در قسمت های بعدی به این کلاس خصوصیات و رفتارهای مورد نظر را اضافه خواهیم کرد. برای تعریف کلاس، بر روی نام پروژه در پنجره Solution Explorer ، با موس راست کلیک کرده و از منوی ظاهر شده از قسمت گزینه ...Class را انتخاب می کنیم :

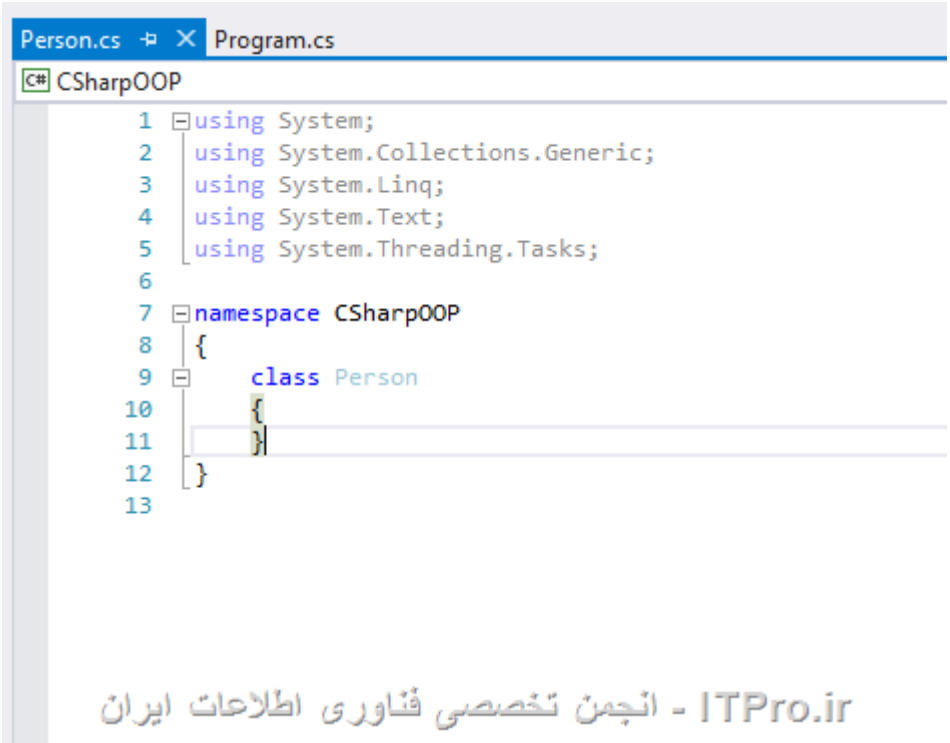


بعد از انتخاب این گزینه، نام کلاس مورد نظر را در پنجره Add New Item وارد کرده و روی دکمه Add کلیک می کنیم.
در اینجا نام Person را وارد می کنیم. بعد از انجام این کار، فایل جدیدی با نام Person.cs به پروژه ما اضافه می شود :

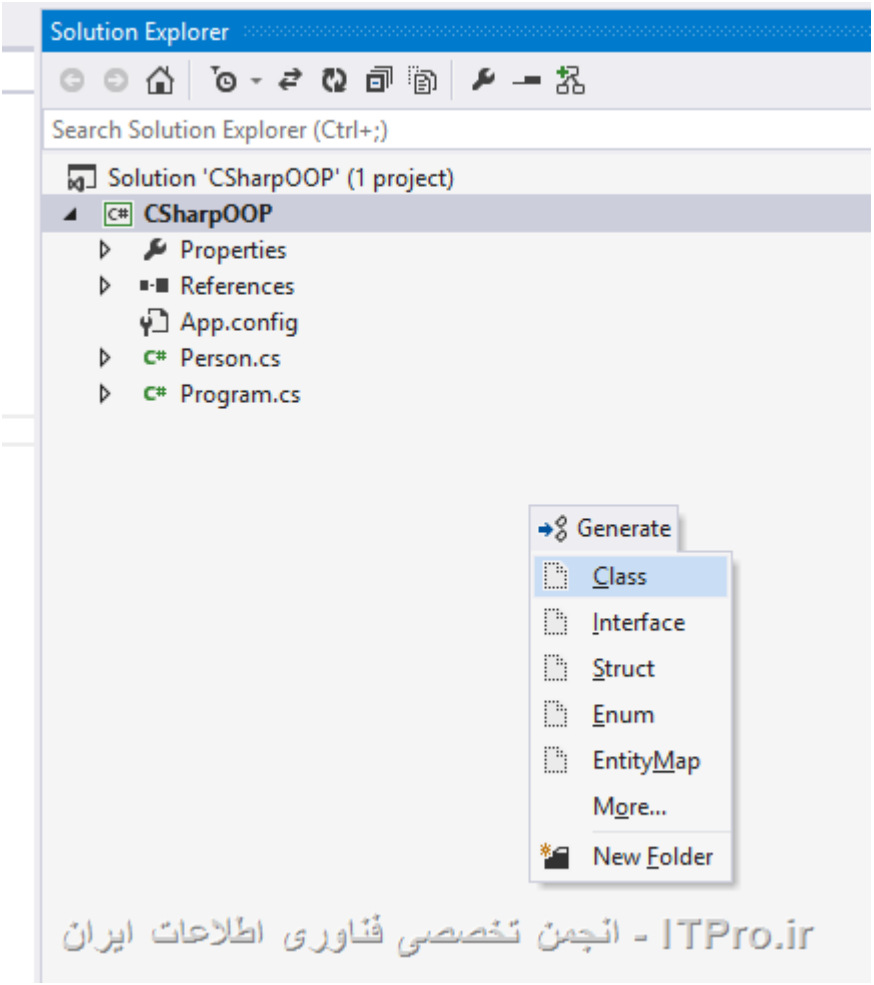




اگر بر روی فایل Person.cs دوبار کلیک کنیم، محتویات فایل مورد نظر به صورت زیر نمایش داده خواهد شد :



نکته: در قسمت معرفی ابزارهای این مجموعه آموزشی، درباره ابزاری به نام Resharper صحبت کردیم. در صورتی که این ابزار را نصب کرده باشید، برای تعریف کلاس جدید، کافیست پروژه ای که قصد تعریف کلاس داخل آن را درید، در پنجره Solution Explorerانتخاب کرده و کلیدهای Alt+Insert را فشار دهید. با اینکار منوی زیر نمایش داده می شود :



بعد از انتخاب گزینه کلاس از منوی ظاهر شده، نام کلاس از شما پرسیده شده و کلاس به پروژه شما اضافه می شود .

به سراغ محتویات فایل اضافه شده برویم :

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CSharpOOP
{
    class Person
    {
```

```
    }
}
```

قسمت using مربوط به استفاده کلاس هایی است که در namespace های دیگر تعریف شده اند .namespace ها برای دسته بندی کدهای پروژه مورد استفاده قرار میگیرند، در حقیقت شما می توانید از لحاظ کاربردی کدهای خود را در زبان سی شارپ بوسیله namespace تقسیم بندی کنید. برای مثال، در کد بالا، کلاسPerson ، در namespace یا فضای نام CSharpOOPتعریف شده است. زمانی که پروژه ای ایجاد می کنید، فضای نام پیش فرض بر اساس نام پروژه ایجاد شده و تمام کدهای شما داخل این فضای نام تعریف خواهند شد. همچنین کلیه کلاس هایی که به صورت پیش فرض در دات نت تعریف شده اند، در فضای نام System قرار دارند. در حقیقت System فضای نام پایه برای کلیه کلاس های موجود در دات نت می باشد. همچنین می توان برای هر فضای نام یک فضای نام زیر مجموعه تعریف کرد که این جداسازی بوسیله کاراکتر . انجام می شود. برای مثال، برای فضای نام CSharpOOP می خواهیم یک فضای نام زیر مجموعه با نام DataTools تعریف کنیم :

```
namespace CSharpOOP.DataTools
{
}
```

به کد کلاس Person برگردیم. در ادامه کد، فضای نام CSharpOOP مشخص شده که داخل آن کلاس Person تعریف شده است. اگر دقت کنید، این کلاس access-modifier ندارد. کدهایی که برای آنها access-modifier مشخص نشده باشد، به صورت پیش فرض private در نظر گرفته می شوند. بعد از تعریف کلاس بوسیله {} محدوده کلاس مشخص شده است که کدهای مربوط به کلاس داخل آن نوشته می شوند .

خوب تا اینجا، ما با شیوه تعریف یک کلاس ساده آشنا شدیم. در مرحله بعد، باید از روی این کلاس یک شئ بسازیم. ساختار کلی تعریف شئ به صورت زیر است :

```
{class-name} {object-name} = new {class-name}();
```

در قسمتclass-name ، نام کلاس را مشخص می کنیم، برای مثال Person و در قسمتobject-name ، نام شئ مورد نظر را مشخص می کنیم. در حقیقت object-name یک متغیر است که به شئ ما اشاره می کند. بعد از علامت انتساب یا = باید عملیات ساخت شئ را انجام دهیم. بوسیله کلمه کلیدی new می گوییم که تصمیم به ساخت یک شئ جدید داریم و در مقال آن نام کلاسی که می خواهیم از روی آن شئ بسازیم را می نویسیم. دقت کنید که بعد از نوشتن نام کلاس در مقال کلمه کلیدی new باید () حتماً نوشته شود، در غیر اینصورت با پیغام خطا مواجه خواهید شد. با توضیحات بالا، می توان گفت عملیات ساخت شئ در دو مرحله انجام می شود :

۱. تعریف متغیری که شئ داخل آن نگهداری می شود (دستورات قبل از عملیات انتساب.(
۲. ساخت شئ و قرار دادن آن داخل متغیر مربوطه (دستورات بعد از عملیات انتساب.(

حال از روی کلاس Person یک شئ ایجاد می کنیم. کد متد Main در فایل Program.cs را به صورت زیر تغییر دهید :

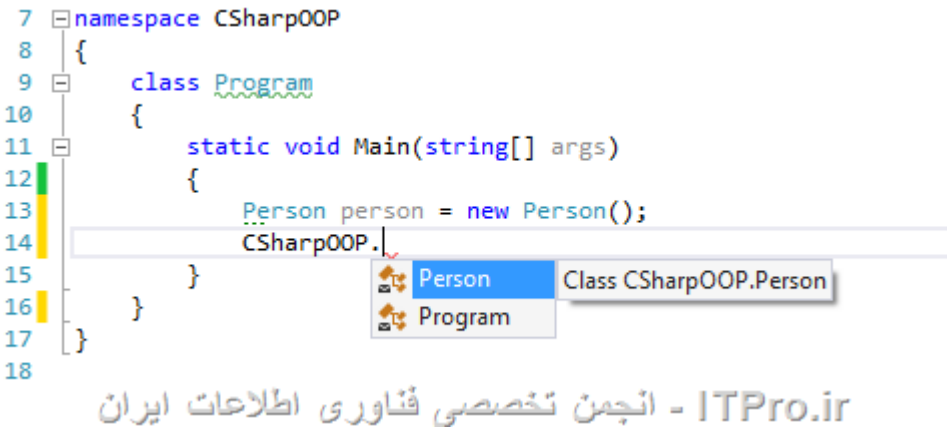
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CSharpOOP
{
    class Program
    {
        static void Main(string[] args)
        {
            Person person = new Person();
        }
    }
}
```

بوسیله کد بالا، عملیات ساخت شئ انجام شد. همانطور که قبلا گفتیم، ما می توانیم چندین شئ با نام های متفاوت از روی یک کلاس ایجاد کنیم :

```
Person person۱ = new Person();
Person person۲ = new Person();
Person person۳ = new Person();
```

دقت کنید، کلاس Program در فایل Program.cs نیز داخل فضای نام CSharpOOP قرار دارد. شما می توانید در فایل های متفاوت فضای نام همنام داشته باشید، بدین معنی که کلیه کدها در همان فضای نام قرار خواهند گرفت. در صورتی که

شما در متد Main نام فضای نام CSharpOOP را تایپ کنید و پس از آن کلید . را بزنید، لیستی که از محتویات آن فضای نام برای شما نمایش داده خواهد شد :



اما فرض کنید، کد ما در فایلPerson.cs ، در فضای نام دیگری با نام CSharpOOP.Entities تعریف شده بود :

```
namespace CSharpOOP.Entities
{
    class Person
    {
    }
}
```

در این حالت، زمانی که شما در فایل Program.cs و متدMain ، تصمیم دارید از روی کلاس Person شئ بسازید، باید آدرس کامل فضای نام را نیز هنگام ساخت شئ مشخص کنید، زیرا فضای نام کلاس های Program و Person دیگر یکسان نیستند :

```
CSharpOOP.Entities.Person person = new CSharpOOP.Entities.Person();
```

اما در اینجا نکته ای وجود دارد، چون ابتدای فضای نام کلاس های Program و کلاس Person یکسان می باشد، یعنی فضای نام Entities زیر مجموعه CSharpOOP قرار دارد و کلاس Program نیز در فضای نام CSharpOOP تعریف شده، می توان از نوشتن قسمت اول فضای نام یعنی CSharpOOP خودداری کرد :

```
Entities.Person person = new Entities.Person();
```

دقت کنید، اگر فضای نام را برای ایجاد شئ ننویسیم، با پیغام خطا مواجه خواهیم شد. اما راهی وجود دارد که آدرس کامل کلاس را ننویسیم، برای اینکار از دستور using استفاده می کنیم که در بالا نیز به آن اشاره شد. دستور using کلیه کدهای داخل یک فضای نام را داخل فضای نام جاری قابل دسترس می کند. برای مثال بالا، کافیست در قسمت using فایل Program.cs، دستور زیر را بنویسیم :

```
using CSharpOOP.Entities;
```

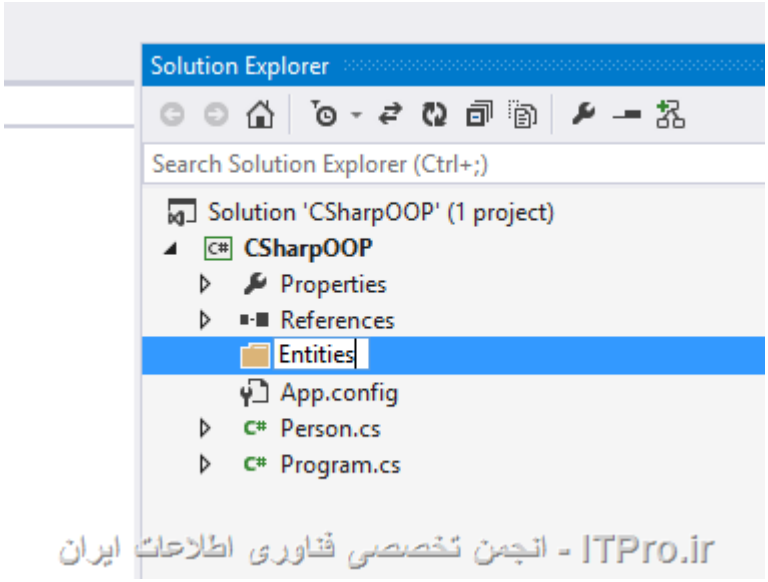
با نوشتن دستور بالا، دیگر نیازی به نوشتن آدرس فضای نام هنگام ساخت شئ نخواهد بود. نمونه دیگر استفاده از دستور using، استفاده از دستورات کلاس Console می باشد که در قسمت های قبل با آن زیاد کار کردیم. کلاس Console داخل فضای نام System که فضای نام پایه کلیه کلاس های دات نت می باشد تعریف شده. اما بدلیل اینکه در ابتدای فایل Program.csدستور using System; نوشته شده است، کافیست تنها نام کلاس Console را بنویسیم و نیازی به نوشتن آدرس کامل آن به صورت System.Console نمی باشد .

شما می توانید داخل یک فایل چندین کلاس را تعریف کنید. برای مثال، در فایل Program.cs می توانید بعد از اتمام کد کلاسProgram.cs ، اقدام به تعریف کلاس Person نمایید :

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using CSharpOOP.Entities;

namespace CSharpOOP
{
    class Program
    {
        static void Main(string[] args)
        {
            Entities.Person person = new Entities.Person();
        }
    }
```

```
    class Person
    {

    }
}
```

اما بهتر است برای هر کلاس، یک فایل جداگانه در نظر بگیرید تا ساختار مناسب برای پروژه ای که تصمیم به انجام آن دارید حفظ شود .
یکی دیگر از قابلیت های موجود در Solution Explorer ، قابلیت پوشه بندی فایل ها داخل پروژه می باشد. برای مثال، می توانید کلاس های مربوط به موجودیت های برنامه را داخل یک پوشه قرار دهید. برای اینکار، بر روی پروژه راست کلیک کرده، از قسمت Add گزینه New Folder را انتخاب کنید. با اینکار پوشه جدیدی به پروژه شما اضافه می شود که می توانید برای آن یک نام دلخواه انتخاب کنید :



در صورتی که ابزار Resharper را نصب کرده باشید، با زدن کلید های Alt+Insert بر روی پروژه داخل Solution Explorerاز منوی ظاهر شده گزینه New Folder را برای افزودن پوشه جدید انتخاب کنید .
پس از تعریف پوشه، با انتخاب آن و تکرار مراحل قبلی برای ایجاد کلاس، می توانید داخل آن پوشه یک فایل جدید ایجاد کنید. برای مثال، پوشه ای با نام Entities داخل پروژه تعریف کرده و کلاسی با نام Car داخل آن تعریف کنید. بعد از اینکار محتویات فایل شما به صورت زیر خواهد بود :

```
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CSharpOOP.Entities
{
    public class Car
    {

    }
}
```

به یک نکته توجه کنید که فضای نام یا namespace کلاس Car به صورت خودکار CSharpOOP.Entities انتخاب شده است، زیرا کلاس داخل پوشه Entities که در پروژه CSharpOOP قرار دارد اضافه شده. در صورتی که شما داخل یک پوشه، پوشه جدیدی اضافه کرده و داخل آن یک کلاس اضافه کنید، آدرس فضای نام مبتنی بر نام آن پوشه انتخاب خواهد شد. پس نکته بعدی که باید مد نظر داشته باشید، زمانی که قصد دارید کدهای خود را بوسیله فضاهای نام دسته بندی کنید، حدالامکان برای آنها پوشه ایجاد کنید، اجباری به اینکار نیست، اما برای حفظ ساختار و نظم پروژه اینکار توصیه می شود .

در این قسمت از سری آموزش سی شارپ، با مفاهیم کلاس، شئ، فضاهای نام، دستور using و پوشه بندی فایل ها داخل پروژه آشنا شدیم. در قسمت بعدی آموزش با نحوه تعریف خصوصیت و رفتار برای کلاس ها و شیوه استفاده از آنها بوسیله اشیاء ساخته شده آشنا خواهیم شد .

مانی که یک فیلد تعریف می کنیم، عملیات انتساب مقدار و گرفتن مقدار به صورت مستقیم از داخل فیلد انجام شده و امکان انجام هیچ گونه نظارتی بر روی این عملیات ها وجود ندارد. برای رفع این مشکل، دو راه وجود دارد :

۱. استفاده از متدها برای ست کردن و گرفتن مقدار از فیلد.
۲. استفاده از Property ها

افرادی که با زبان جاوا آشنا هستند با متدهای get و set در کلاس ها آشنایی دارند. این متدها عملیات خواندن و نوشتن در فیلدها را برای ما انجام می دهند. مثالی از زبان سی شارپ می زنیم. کلاس Person را در نظر بگیرید :

```
public class Person
{
    public string FirstName;
    public string LastName;
}
```

این کلاس، ما زمانی که یک شئ از این کلاس می سازیم به صورت مستقیم فیلدها را مقدار دهی کرده یا مقدار آنها را می خوانیم. اما برای کنترل دسترسی به فیلدها، ابتدا باید سطح دسترسی فیلدها را به private تغییر بدیم. زمانی که یک عضو کلاس که

در اینجا فیلدها هستند را به private تغییر می دهیم، آن عضو تنها داخل همان کلاس قابل دسترس خواهد بود. برای اولین قدم، کلاس Person را به صورت زیر تغییر می دهیم :

```
public class Person
{
    private string firstName;
    private string lastName;
}
```

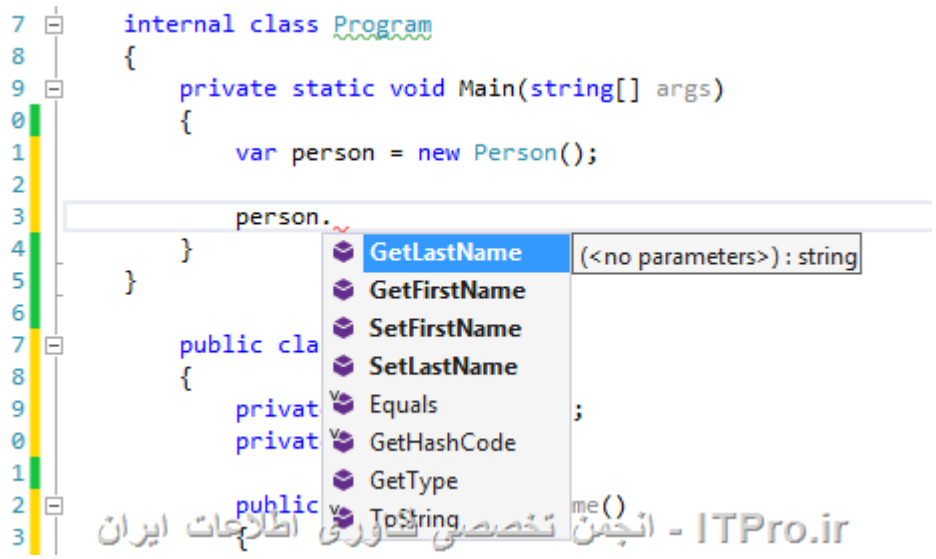
به نام گذاری فیلدها دقت کنید، زمانی که فیلدها به private تغییر کردند، نام گذاری بر اساس قاعده camelCase انجام می شود. این قاعده برای کلیه فیلدهای private کلاس ها حکم می کند. البته الزامی به این کار نیست، اما برای رعایت اصول کد نویسی

بهتر است از این قواعد پیروی کنیم. بعضی از برنامه نویس ها ابتدای نام فیلدهای private از کاراکتر _ استفاده می کنند. این موضوع کاملاً دلخواه می باشد، اما سعی کنید در نام گذاری فیلدها public و private تفاوت قایل شوید .

در قدم بعدی باید بتوانیم در خارج از کلاس، عملیات خواندن و مقدار دهی فیلد را انجام دهیم. در روش اول گفتیم که از متدهای get و set برای اینکار استفاده می کنیم. کلاس Person را به صورت زیر تغییر می دهیم :

```
public class Person
{
    private string firstName;
    private string lastName;

    public string GetFirstName()
    {
        return firstName;
    }

    public void SetFirstName(string value)
    {
        firstName = value;
    }

    public string GetLastName()
    {
        return lastName;
    }

    public void SetLastName(string value)
    {
        lastName = value;
    }
}
```

در تصویر زیر، لیست نمایش داده شده برای شئ ای از کلاس Person را مشاهده می کنید :



حال می توانیم عملیات خواندن و نوشتن فیلدها را در خارج از کلاس بوسیله این دو متد انجام دهیم :

```
var person = new Person();


person.SetFirstName("Hossein");
person.SetLastName("Ahmadi");


Console.WriteLine(person.GetFirstName() + " " +
person.GetLastName());
Console.ReadLine();
```

حال فرض کنید، یک فیلد باید تنها خواندنی باشد، برای اینکار کافیست بخش Set را از کلاس حذف کنید یا فرض کنید عملیات نوشتن باید تنها در داخل خود کلاس انجام شود و از بیرون کلاس دسترسی نوشتن باید بسته شود، برای این کار کافیست که

دسترسی متد Set برای فیلد مورد نظر را به private تغییر دهیم. در مثال زیر عملیات نوشتن برای فیلد firstName به صورت private تعریف شده و فیلد lastName قابلیت نوشتن ندارد :

```
public class Person
```

```
{
    private string firstName;
    private string lastName;

    public string GetFirstName()
    {
        return firstName;
    }

    private void SetFirstName(string value)
    {
        firstName = value;
    }

    public string GetLastName()
    {
        return lastName;
    }
}
```

استفاده از Property ها

اما در زبان سی شارپ به صورت دیگری می توان این کنترل را انجام داد و آن استفاده از Property ها می باشند. ساختار کلی property ها به صورت زیر می باشد :

```
{access-modifier} {data-type} {property-name}
{
    [access-modifier] get
    {
```

```
        // body for get value
    }
    [access-modifier] set
    {
        // body for set value
    }
}
```

اما بررسی هر یک از قسمت های ساختار :Property

۱. access-modifier: سطح دسترسی به Property را تعیین می کند .Property نیز مانند فیلد می تواند سطح دسترسی داشته باشد.
۲. data-type: نوع Property که یکی از Data Type های دات نت یا کلاسی که به صورت دستی نوشته شده باشد.
۳. property-name: نام Property ، برای نام گذاری Property ها همیشه از قاعده PascalCase استفاده کنید، سطح دسترسی تفاوتی ندارد، همیشه PascalCase تعریف کنید.
۴. بدنه :get این بدنه، دقیقاً معادل متد Get ایست که در قسمت قبلی تعریف کردیم. شما داخل بدنه get هر دستوری را می توانید بنویسید، در حقیقت این بدنه مانند یک متد عمل کرده و زمانی که شما مقدار Property را می خوانید (مثلاً برای چاپ با دستور

(WriteLine بدنه get اجرا می شود. دقت کنید بدنه get حتماً باید مقداری را با دستور return بر گرداند. همچنین این بدنه می تواند دارای access-modifier باشد، یعنی سطح دسترسی خواندن مقدار را مشخص می کند. در صورتی که سطح دسترسی را

مشخص نکنید به صورت پیش فرض public در نظر گرفته می شود .

۱. بدنه :set این بدنه، دقیقاً معادل متد Set در مثال قبلی است. زمانی که شما مقداری را داخل Property ست می کنید، بدنه set اجرا می شود. داخل بدنه set پارامتر پیش فرضی وجود دارد به نام value که مقدار ست شده داخل Property داخل آن قرار

گرفته و شما می توانید به آن از داخل بدنه set دسترسی داشته باشید. همچنین می توان برای بدنه set سطح دسترسی را مشخص کرد. در صورتی که سطح دسترسی را مشخص نکنید به صورت پیش فرض public در نظر گرفته می شود .

به مثال کلاس Person بر می گردیم. تصمیم داریم عملیات هایی که در بوسیله متدهای Get و Set انجام دادیم را با

Propertyها پیاده سازی کنیم. کلاس Person را به صورت زیر تغییر دهید :

```
public class Person
{
    private string firstName;
    private string lastName;

    public string FirstName
    {
        get { return firstName; }
        set { firstName = value; }
    }

    public string LastName
    {
        get { return lastName; }
        set { lastName = value; }
    }
}
```

در کد بالا، دو Property با نام های FirstName و LastName تعریف کردیم که عملیات خواندن و نوشتن از فیلدهای مربوطه را انجام می دهند. نتیجه اعضای نمایش داده شده برای کلاس Person هنگام استفاده از نمونه یا Instance آن را در تصویر زیر مشاهده

می کنید :

در ادامه کد متد Main را به صورت زیر تغییر دهید :

```
var person = new Person();

person.FirstName = "Hossein";
person.LastName = "Ahmadi";

Console.WriteLine(person.FirstName + " " + person.LastName);
Console.ReadKey();
```

در کد بالا، زمانی که مقدار Hossein را داخل فیلد FirstName ست می کنیم، بدنه set مربوط به خاصیت FirstName اجرا می شود که در این بدنه ما مقدار value که همان مقدار ست شده در هنگام استفاده از کلاس است را داخل فیلد firstNameقرار می

دهیم. برای LastName هم به همین صورت است. شاید بپرسید پارامتر value کجا تعریف شده است؟ این پارامتر به صورت پیش فرض برای بدنه set وجود دارد که نگه دارنده مقدار ست شده داخل Property است. همچنین زمانی که داخل دستور WriteLine

مقدار FirstName یا LastName را می خوانیم، بدنه get مربوط به همان Property اجرا می شود. در حقیقت Propertyها واسطی میان فیلدها و استفاده کننده از کلاس ها هستند. مانند یک انبار دار که عملیات کنترل ورود و خروج از انبار را کنترل می کند و

عملیات تحویل دادن کالا یا گرفتن کالا و قرار دادن آن در انبار را انجام می دهد .

برای نوشتن Property ها، حتماً نیازی به تعریف Field برای آنها نیست، شما می توانید هر کدی را برای بدنه get یا set

بنویسید. برای مثال، می خواهیم به کلاس Person یک Property تعریف کنیم که نام کامل شخص را بر گرداند. نام این خاصیت را FullName

می گذاریم :

```
public class Person
{
    private string firstName;
    private string lastName;

    public string FirstName
    {
        get { return firstName; }
        set { firstName = value; }
    }

    public string LastName
    {
        get { return lastName; }
        set { lastName = value; }
    }

    public string FullName
    {
        get { return FirstName + " " + LastName; }
    }
}
```

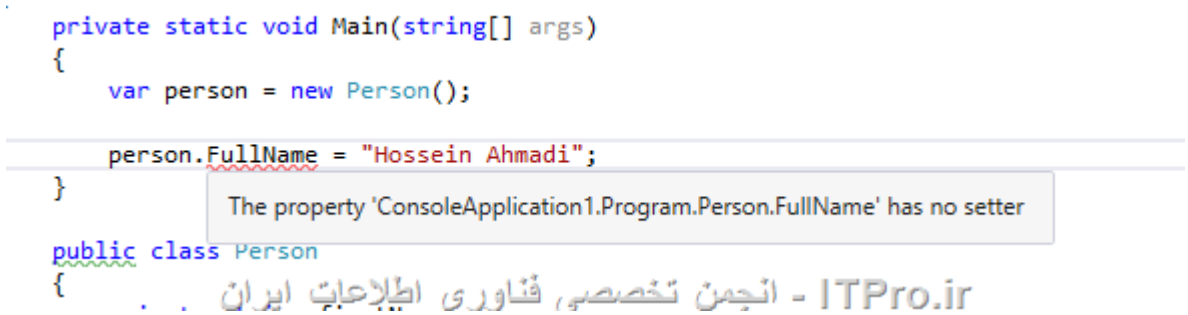
حال کد Main قسمت قبلی را می توان به صورت زیر تغییر داده و از Property جدیدی که تعریف کردیم استفاده کنیم :

```
var person = new Person();
```

```
person.FirstName = "Hossein";
person.LastName = "Ahmadi";

Console.WriteLine(person.FullName);
Console.ReadKey();
```

نگاهی دوباره به کلاس Person و خاصیت FullName می کنیم، اگر دقت کرده باشید این خاصیت تنها بدنه get را دارد و بدنه set را برای آن ننوشتیم. دلیل این امر آن است که FullName تنها برای ترکیبی از firstName و lastName را بر میگرداند. در صورتی که

بخواهیم مقداری داخل FullName بریزیم، با پیغام خطا مواجه می شویم .



Propertyهایی که بدنه get را ندارند Write-Only و آنهایی که بدنه set را ندارند Read-Only می گوییم. همچنین همانطور که قبلاً هم گفتیم می توانیم علاوه بر خود Property برای هر یک از بدنه های get و set نیز سطح دسترسی مشخص کنیم. برای مثال

میخواهیم خاصیت FirstName تنها داخل خود کلاس قابلیت نوشتن داشته باشد، برای اینکار کافیست بدنه set را به صورت private تعریف کنیم :

```
public string FirstName
{
    get { return firstName; }
    private set { firstName = value; }
}
```

Automatic Properties

گاهی اوقات، Propertyکه تعریف می کنیم تنها عملیات خواندن و نوشتن یک فیلد را کنترل می کند. برای مثال، کلاس Personرا در نظر بگیرید :

```
public class Person
{
    private string firstName;

    public string FirstName
    {
        get { return firstName; }
        set { firstName = value; }
    }
}
```

کد مربوط به خصوصیت بالا را می توان به شکل زیر نیز نوشت :

```
public class Person
{
    public string FirstName { get; set; }
}
```

کامپایلر بعد از کامپایل کد بالا، به صورت خودکار یه فیلد برای خاصیت نوشته شده تعریف کرده و بدنه get و set آن را به صورت خودکار می نویسد. از مزیت های Auto-Property ها حجم کد کمتر و البته قابلیت کنترل دسترسی به عملیات های خواندن و نوشتن Property ها می باشد. مثال بالا را جوری تغییر می دهیم که خاصیت FirstName تنها داخل کلاس قابل نوشتن باشد :

```
public class Person
{
    public string FirstName { get; private set; }
```

```
}
```

به این نکته توجه داشته باشید، زمانی که از Auto-Property ها استفاده می کنید، حتماً باید get و set را بنویسید، در غیر اینصورت پیغام خطا دریافت خواهید کرد. البته این مشکل در نسخه ۶ زبان سی شارپ برطرف شده است .

زمانی که شما کلاسی را تعریف می کنید، این کلاس حاوی یکسری خصوصیات و یکسری رفتارها یا همان متدها می باشد. شما بعد از ایجاد شئ، خصوصیات را مقدار دهی کرده و از شئ استفاده می کنید. اما چندین راه دیگر برای مقدار دهی خصوصیات و آماده سازی اولیه کلاس وجود دارد. ما در این قسمت دو روش مختلف را بررسی می کنیم :

۱. مقدار دهی اولیه شئ یاObject Initialization
۲. استفاده از سازنده ها یاConstructors

مقدار دهی اولیه با کمکObject Initialization

در این روش، شما زمانی که اقدام به ایجاد یک شئ می کنید، می توانید مقادیر خصوصیات و فیلدهای آن را مشخص کنید. کلاس Person را در نظر بگیرید :

```
public class Person
{
    public string FirstName;
    public string LastName;
}
```

به صورت پیش فرض، شما یک شئ از کلاس ساخته و خصوصیات آن را مقدار دهی می کنید :

```
var person = new Person();
person.FirstName = "Hossein";
person.LastName = "Ahmadi";
```

مقدار دهی اولیه شئ کار ساده ایست، کافیست پس از نوشتن () بعد از نام کلاس در قسمت new بین علامت های {} مقادیر خصوصیات را مشخص کنیم :

```
var person = new Person()
```

```
{
    FirstName = "Hossein",
    LastName = "Ahmadi"
};
```

با این کار مقادیر FirstName و LastName در کلاس Person مقدار دهی اولیه خواهند شد. می توانید در این حالت، از نوشتن () صرفنظر کنید :

```
var person = new Person
{
    FirstName = "Hossein",
    LastName = "Ahmadi"
};
```

دقت کنید، در هنگام مقدار دهی اولیه قابلیت صدا زدن متدهای کلاس را نخواهید داشت و تنها می توانید فیلدها، خصوصیات و برخی اعضای دیگر که در قسمت های بعدی با آن آشنا خواهیم شد را مقدار دهی کنید .

سازنده ها یاConstructors

اگر دقت کرده باشید، زمانی که شئ ای را تعریف می کنیم، بعد از نوشتن نام کلاس بعد از کلمه new از () استفاده می کنیم، مشابه زمانی که تصمیم به صدا زدن یک متد دارید. دلیل اینکار، پروسه ایست که سی شارپ برای ایجاد کردن کلاس ها انجام می دهد. زمانی که شما شئ ای از یک کلاس ایجاد می کنید، سی شارپ قسمتی با نام سازنده یا Constructor را برای آن کلاس صدا می زند. این سازنده یک متد می باشد که می تواند بدون پارامتر یا با پارامتر باشد و داخل آن کدی نوشته می شود که می خواهیم در هنگام ایجاد شئ اجرا شود. با یک مثال ساده سازنده ها را بررسی می کنیم. کلاس Person را در نظر بگیرید، برای این کلاس یک سازنده تعریف می کنیم که مقادیر FirstName و LastName را به عنوان ورودی گرفته و خصوصیات مربوطه را مقدار دهی می کند :

```
public class Person
{
    public Person(string firstName, string lastName)
    {
```

```
        this.FirstName = firstName;
        this.LastName = lastName;
    }

    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

در کد بالا، قسمت سانده برای کلاس Person با دو پارامتر تعریف شده است. به شیوه تعریف سازنده دقت کنید، ابتدا سطح دسترسی به سازنده مشخص شده، سپس نام کلاس نوشته شده که برای سازنده ها، این نام دقیقاً باید معادل نام کلاس باشد، سپس پارامترهای مورد نظر و بعد از آن ها بدنه سازنده. دقت کنید سازنده ها مقدار بازگشتی ندارند. با توضیحات گفته شده می توان ساختار کلی سازنده را به صورت زیر بیان کرد :

```
{access-modifier} {class-name}([parameters])
{
    // constructor body
}
```

همچنین در بدنه سازنده بالا، به کلمه کلیدی this دقت کنید. کلمه کلیدی this به شئ جاری که روی کلاس ساخته شده است اشاره می کند. فرض کنید شما ده ها شئ از روی یک کلاس ساخته اید، زمانی که یک رفتار را صدا می زنید و داخل آن رفتار از کلمه کلیدی this استفاده می کنید، thisبه همان شئ ای اشاره می کند که رفتار در آن صدا زده شده است. در این سازنده نیز کلمه کلیدی this به شئ ای اشاره می کند که سازنده برای آن صدا زده شده .

پس از تعریف سازنده می توان هنگام ایجاد شئ، مقادیر مورد نظر را به سازنده ارسال کرد :

```
var person = new Person("Hossein", "Ahmadi");
```

با اجرای کد بالا، خصوصیت های FirstName و LastName هنگام ایجاد شئ، مقدار دهی خواهند شد. اما باید به یک نکته در اینجا توجه داشت، زمانی که شما سازنده ای به همراه پارامتر برای یک کلاس تعریف می کنید، دیگر نمی توانید از کلاس بدون ارسال پارامتر در سازنده شئ بسازید. دلیل این موضوع، عدم وجود سازنده ای به نام سازنده پیش فرض یا Default Constructorمی باشد. سازنده پیش فرض، سازنده ایست که هیچ پارامتری را به عنوان ورودی نمی گیرد. زمانی که شما سازنده ای برای یک کلاس تعریف نکرده اید، آن کلاس به صورت پیش Default Constrcutor برایش تعریف شده است. اما زمانی که اقدام به ایجاد یک سازنده برای کلاس کردید، اگر می خواهید از آن کلاس بدون ارسال پارامتر برای سازنده

شئ بسازید، باید سازنده پیش فرض را به صورت دستی برای آن کلاس بنویسید :

```
public class Person
{
    public Person()
    {
    }

    public Person(string firstName, string lastName)
    {
        this.FirstName = firstName;
        this.LastName = lastName;
    }

    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

شما می توانید سطح دسترسی به سازنده ها را مشخص کنید، برای مثال، حالتی پیش می آید که می خواهد یک سازنده فقط داخل همان کلاس در دسترس باشد، یعتی شما از یک کلاس داخل خودش، مثلاً داخل یک رفتار، می خواهید یک شئ بسازید. برای این کار، می توانید سطح دسترسی سازنده مد نظر را private تعریف کنید. برای مثال :

```
public class Person
{
    public Person()
    {
    }

    private Person(string firstName, string lastName)
    {
        this.FirstName = firstName;
```

```
        this.LastName = lastName;
    }

    public Person CreateObject(string firstName)
    {
        return new Person(firstName, null);
    }

    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

در کد بالا، یک متد یا رفتار برای کلاس تعریف کردیم با نام .CreateObject این رفتار یک شئ از روی خود کلاس Personمی سازد و پارامتر ارسالی به متد CreateObject را به سازنده ارسال می کند. اما خارج از شئ، دیگر نمی توانیم از سازنده ای که دو پارامتر را به عنوان ورودی می گیرد استفاده کنیم، زیرا این سازنده با سطح دسترسی private تعریف شده است .

همانند متدها، سازنده ها می توانند overload داشته باشند، یعنی چند سازنده با signature های متفاوت. برای مثال، کلاس Person را به صورت زیر تغییر می دهیم :

```
public class Person
{
    public Person()
    {
    }

    public Person(string firstName)
    {
        this.FirstName = firstName;
    }

    public Person(string firstName, string lastName)
    {
```

```
        this.FirstName = firstName;
        this.LastName = lastName;
    }

    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

در اینجا دو سازنده برای کلاس تعریف کردیم که اولی یک پارامتر گرفته و دومی با دو پارامتر صدا زده می شود .

زنجیره سازنده ها یاConstrcutor Chaining

زمانی که شما چندین سازنده دارید، می توانید از کد های نوشته شده داخل یک سازنده در سازنده دیگر استفاده کنید. برای اینکار از قابلیت constructor chaining استفاده می شود. با یک مثال ادامه می دهیم، در کد قبلی سه سازنده داشتیم، سازنده پیش فرض، سازنده ای که تنها FirstName را می گرفت و سازنده ای که FirstName و LastName را به عنوان پارامتر می گرفت. در سازنده دوم، می توان از سازنده سوم جهت مقدار دهی استفاده کرد. برای این کار، کد بالا را به صورت زیر تغییر می دهیم :

```
public class Person
{
    public Person()
    {
    }

    public Person(string firstName) : this(firstName,null)
    {
    }

    public Person(string firstName, string lastName)
    {
```

```
        this.FirstName = firstName;
        this.LastName = lastName;
    }

    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

سازنده دوم ما به صورت زیر تغییر کرده است :

```
public Person(string firstName) : this(firstName,null)
{
}
```

دقت کنید، بعد از بستن پرانتز پس از علامت : از کلمه کلیدی this مانند یک متد استفاده کرده ایم، در این روش، سازنده کلاس صدا زده شده و به عنوان پارامتر اول، firstNameکه در سازنده تعریف شده را ارسال کرده و عنوان پارامتر دوم مقدار nullرا ارسال کرده ایم. قابلیتconstrcutor chaining ، در کاهش تعداد خطوط نوشته در برنامه کمک زیادی به ما می کند .

به این نکته توجه داشته باشید که نمی تواند سازنده را به صورت متد از داخل کلاس جایی غیر از خود سازنده ها صدا زد.
همچنین سازنده ها تنها برای مقدار دهی خصوصیات استفاده نمی شوند، شما می توانید هر کدی را داخل سازنده بنویسید .

نوع های بدون نام یاAnonymous Types

نوع های بدون نام، به ما این امکان را می دهند تا شئ ای بدون تعریف کلاس ایجاد کنیم. این شئ تنها می تواند شامل خصوصیات باشد و قابلیت تعریف رفتار برای آن را نخواهیم داشت. در مثال زیر یک شئ بدون نام ایجاد کرده ایم که سه خصوصیت با نام FirstName و LastName و Age دارد :

```
var anonymous = new
{
```

```
    FirstName = "Hossein",
    LastName = "Ahmadi",
    Age = ۲۹
};

Console.WriteLine(anonymous.FirstName + " " + anonymous.LastName);
```

همانطور که در کد مشاهده می کنید، کافیست بعد از کلمه کلیدی new بلافاصله به سراغ عملیات Object Initialization برویم و نیازی به نوشتن نام کلاس نیست.

تعریف وراثت یا Inheritance و پیاده سازی آن در زبان#C

همانطور که در مقدمه مبحث برنامه نویسی شئ گرا خدمت دوستان توضیح دادم، وراثت به معنی به ارث بردن یکسری خصوصیات و رفتار بوسیله فرزند از والد است. در برنامه نویسی شئ گرا، زمانی که صحبت از وراثت می کنیم، در حقیقت می خواهیم برای یک کلاسی، یک کلاس والد مشخص کنیم. وراثت در برنامه نویسی شئ گرا کاربردهای بسیاری دارد، به صورتی که اصلی ترین و بنیادی ترین قابلیت در برنامه نویسی شئ گرا نام برده می شود. قبل از شروع به نکته زیر توجه کنید :

زمانی که کلاس A به عنوان والد کلاس B معرفی می شود، یعنی کلاس B فرزند کلاس A می باشد، می گوییم کلاس B از کلاس A مشتق شده است. در طول این دوره از واژه مشتق شده به تکرار استفاده خواهیم کرد .

در ابتدا با شیوه کلی استفاده از وراثت در کلاس ها آشنا می شویم. فرض کنید کلاسی داریم با نام :A

```
public class A
{

}
```

حال تصمیم داریم کلاسی تعریف کنیم با نام B که از کلاس A مشتق شده است، یعنی تمامی خصوصیات و رفتارهای کلاس Aرا به ارث می برد. برای اینکار کلاس B را به صورت زیر تعریف می کنیم :

```
public class B : A
{
}
```

بوسیله دستور بالا، کلاس A به عنوان کلاس والد کلاس B در نظر گرفته خواهد شد. گفتیم یکی از مزایای استفاده از وراثت در برنامه نویسی شئ گرا، استفاده مجدد از کدهایی است که در کلاس والد تعریف شده است. در مثال بالا، کد کلاس A خصوصیتی با نام Item۱ و Item۲ تعریف می کنیم :

```
public class A
{
    public string Item۱ { get; set; }
    public string Item۲ { get; set; }
```

```
}
```

به دلیل اینکه کلاس B از کلاس A مشتق شده است، می توانیم از خصوصیت های Item۱ و Item۲ برای کلاس B استفاده کنیم :

```
B obj = new B();
obj.Item۱ = "Hossein Ahmadi";
obj.Item۲ = "ITPro.ir";
```

مشتق می شود A این کلاس نیز از کلاس .C حال، کلاس سومی تعریف می کنیم با نام:

```
public class C : A
{
}
```

زمانی که شئ ای از کلاس C بسازیم، میبینیم که خصوصیات Item۱ و Item۲ برای این شئ کلاس C نیز وجود دارند، در حقیقت ما این خصوصیات را تنها یکبار در کلاس A تعریف کردیم و با قابلیت وراثت از این کدها برای کلاس های B و C مجدداً استفاده کردیم. زمانی که کلاسی یک یک کلاس والد مشتق می شود، علاوه بر اینکه دارای خصوصیات و رفتارهای کلاس های والد می باشد، می توان برای کلاس فرزند خصوصیات و رفتارهای جدید تعریف کرد. کلاس C را که در بالا تعریف کردیم به صورت زیر تغییر می دهیم :

```
public class C : A
{
    public string Item۳ { get; set; }
}
```

حال زمانی که ما شئ ای از روی کلاس C بسازیم علاوه بر خصوصیت های Item۱ و Item۲ که در کلاس A تعریف شده اند، به خصوصیت دیگری نیز نام Item۳ که در کلاس C تعریف شده دسترسی خواهیم داشت :

```
var instanceOfC = new C();
```

```
instanceOfC.Item۱ = "Hossein Ahmadi";
instanceOfC.Item۲ = "ITPro.ir";
instanceOfC.Item۳ = "C# Course";
```

را D مشتق شد می توان کلاس A از کلاس C وراثت در زبان سی شارپ، به صورت درختی می باشد، یعنی زمانی که کلاس مشتق شده است C نوشت که از کلاس:

```
public class D : C
{
    public string Item۴ { get; set; }
}
```

در مثال بالا، کلاس D علاوه بر خصوصیات کلاس A و کلاس C خصوصیات مربوط به خودش را نیز شامل می شود. در حقیقت زنجیره وراثت را در این مثال مشاهده می کنید. در مثال های بالا، کلاس ها تنها شامل Property بودند، زمانی که شما برای کلاسی یک رفتار تعریف می کنید، کلاس های فرزند آن رفتار را نیز به ارث می برند :

```
public class A
{
    public string Item۱ { get; set; }
    public string Item۲ { get; set; }

    public void PrintItem۱()
    {
        Console.WriteLine(Item۱);
    }
}
```

حال شئ ای از کلاس D می سازیم و رفتار PrintItem۱ را صدا می زنیم :

```
var obj = new D();
d.PrintItem۱();
```

در قسمت های قبلی دیدیم که کلاس D از کلاس C مشتق شده است و خود کلاس C از کلاس .A پس کلیه خصوصیات و رفتارهای کلاس A برای سطوح پایین تر وراثت قابل دسترس هستند .

کلمه کلیدیbase

در قسمت های قبلی، در مورد کلمه کلیدی this توضیح دادیم و گفتیم که این کلمه کلیدی به شئ ای اشاره می کند که از روی کلاس ساخته شده. کلمه کلیدی دیگری وجود دارد با نام base که اشاره به کلاس والد دارد. برای مثال، کلاس B را به صوورت زیر تغییر می دهیم :

```
public class B : A
{
    public void PrintParentItems()
    {
        Console.WriteLine(base.Item۱ + " " + base.Item۲);
    }
}
```

در مثال بالا، کلمه کلیدی base به کلیه اعضای والد اشاره می کند. زمانی که شما از کلمه کلیدی base داخل کلاس استفاده می کنید، تنها اعضای کلاس والد به شما نمایش داده شده و اعضای کلاس فرزند به شما نمایش داده نمی شوند. در قسمت های بعدی با کاربردهای دیگر کلمه base آشنا می شویم .

تبدیل کلاس های مشتق شده به کلاس والد

زمانی که شما از روی یک کلاس، شئ ای می سازید باید نوع آن کلاس را مشخص کنید یا از کلمه کلیدی var استفاده کنید :

```
B obj = new B();
```

زمانی که کلاسی از یک شئ مشتق شده باشد، می توان هنگام تعریف شئ از روی آن کلاس، نوع متغیر را به جای خود کلاس، کلاس وارد قرار داد. برای مثال، در مثال زیر ما یک شئ از روی کلاس C می سازیم :

```
A obj = new C();
```

دقت کنید که نوع متغیر obj را از نوع A در نظر گرفتیم، اما شئ ای از نوع C داخل آن ریختیم. دلیل این امر آن است که کلاس C از کلاس A مشتق شده و به نوعی قابل تبدیل به کلاس A می باشد. اما به این نکته توجه داشته باشید، زمانی که نوع متغیر را از نوع کلاس والد در نظر می گیریم، هنگام استفاده از شئ ساخته شده، تنها خصوصیات و رفتارهایی هایی قابل استفاده هستند که در کلاس والد تعریف شده اند. به عنوان مثال، کد زیر صحیح نمی باشد، به این خاطر که Item۳ داخل کلاس C تعریف شده و ما تنها به Item۱ و Item۲ که داخل A تعریف شده اند دسترسی داریم .

```
A instanceOfC = new C();
instanceOfC.Item۳ = "C# Course";
```

یکی از مهمترین کاربردهای استفاده از نوع داده کلاس والد، مبحث Polymorphism می باشد که در بخش های بعدی با این مفهوم بیشتر آشنا خواهیم شد .

کلاسObject

در کتابخانه دات نت، کلاسی وجود دارد به نام Object یا شئ. در دات نت، کلیه نوع های داده و کلاس ها، چه آنهایی که به صورت دستی می نویسیم و چه آنهایی که در کتابخانه دات نت وجود دارند، از کلاس object مشتق شده اند، به جز کلاس هایی که برای آنها کلاس والد را مشخص کرده ایم. حتی کلاس هایی که برای آنها کلاس والد مشخص شده، باز هم شاخه اصلی زنجیره وراثت به کلاس object ختم می شود. پس به این صورت می گوییم که کلاسobject ، کلاس پایه ای برای کلیه کلاس های دات نت می باشد. یکسری رفتارها برای کلاس Object تعریف شده اند که در تمامی کلاس ها در دسترس هستند، زیرا کلیه کلاس ها از کلاس object مشتق شده اند. این رفتارها به شرح زیر می باشند :

۱. Equals: این رفتار بررسی می کند که دو شئ با یکدیگر برابر هستند یا خیر.

۲. GetHashCode: این رفتار عددی را برمی گرداند که شناسه شئ ایجاد شده می باشد.
۳. GetType: نوع یا Type شئ را بر میگرداند. این متد را در بخش Reflection بیشتر بررسی خواهیم کرد.
۴. ToString: زمانی که این رفتار را برای یک شئ صدا می زنید، رشته ای مرتبط با آن شئ را بر میگرداند که به صورت پیش فرض Type Name یا نام نوع آن کلاس را بر میگرداند. در بخش Polymorphism با این متد بیشتر آشنا می شوید.

گفتیم زمان تعریف کردن یک شئ، نوع داده والد را به جای خود کلاس برای متغیر در نظر گرفت :

```
object number = ۱۲;
object name = "Hossein Ahmadi";
object instance = new A();
```

همینطور که مشاهده می کنید فرقی نمی کند مقدار متغیر چه باشد، هر مقداری را می توان در متغیر از نوع object ذخیره کرد. در این قسمت سعی کردیم مقدماتی از مبحث وراثت را با هم مرور کنیم. در بخش بعدی بحث وراثت را با بررسی مفهوم Polymorphism ادامه خواهیم داد .

همانطور که در قسمت مقدمه گفتیم، Polymorphismبه معنای قابلیت تعریف مجدد رفتار یک موجودیت در کلاس های فرزند می باشد Polymorphism .در زبان سی شارپ به سه روش قابل پیاده سازی است :

۱. استفاده از متد های virtual و override کردن آنها در کلاس های فرزند
۲. استفاده از رفتارهای abstract در کلاس والد
۳. استفاده از قابلیت interface ها

در این قسمت، حالت اول را بررسی می کنیم و حالت دوم و سوم، یعنی استفاده از متدهای abstract و interface ها را در بخش های بعدی توضیح خواهیم داد .

متدهایvirtual

همانطور که گفتیم یکی از روش های پیاده سازی Polymorphism استفاده از متدهای virtual و override کردن آنها در کلاس فرزند است. برای مثال، فرض کنیم کلاس پایه ای داریم با عنوان Shape که در آن رفتاری با نام Draw تعریف کردیم. رفتار Draw وظیفه ترسیم شئ را بر عهده دارد. در این مثال ها، تنها در متدها پیامی را در پنجره کنسول چاپ می کنیم، اما در محیط واقعی هر یک از این متدها وظیفه ترسیم شئ را بر عهده خواهند داشت. همانطور که گفتیم کلاس Shapeرفتار Draw را تعریف می کند. این رفتار در بین تمامی اشیاء ای که از کلاس Shape مشتق می شوند مشترک است. در ابتدا کلاس Shape را به صورت زیر تعریف می کنیم :

```
public class Shape
{
    public void Draw()
    {
        Console.WriteLine("Drawing the shape!");
    }
}
```

حالا باید کلاس های فرزند را تعریف کنیم. ما سه کلاس به نام هایRectangle ، Triangleو Circle که وظیفه ترسیم مستطیل، مثلث و دایره را بر عهده دارند تعریف می کنیم که هر سه از کلاس Shape مشتق شده اند :

```
public class Rectangle : Shape
{

}

public class Triangle : Shape
{

}

public class Circle : Shape
{

}
```

هر سه کلاسی که در بالا تعریف کردیم، شامل متد Draw هستند، زیرا این متد در کلاس پایه یعنی Shape تعریف شده است. حال از هر یک، شئ ای ساخته و متد Draw را صدا می زنیم :

```
var rect = new Rectangle();
var tri = new Triangle();
var circ = new Circle();

rect.Draw();
tri.Draw();
circ.Draw();

Console.ReadKey();
```

خروجی دستورات بالا به صورت زیر می باشد :

```
Drawing the shape!
```

```
Drawing the shape!
Drawing the shape!
```

اما خروجی مدنظر ما تولید نشده است. ما می خواهیم هر کلاس رفتار مربوط به خود را داشته باشد. درست است که رفتار Drawدر کلاس پایه تعریف شده، اما باید بتوانیم این رفتار را برای کلاس های فرزند تغییر دهیم. برای اینکار باید در کلاس پایه مشخص کنیم که کدم رفتار را می خواهیم تغییر دهیم. برای اینکار، کافیست رفتار مورد نظر را از نوع virtual تعریف کنیم. اعضای virtual به ما این اجازه را می دهند تا در کلاس فرزند مجدد آنها را تعریف کنیم. برای اینکار متد Draw در کلاس Shape را به صورت زیر تغییر می دهیم :

```
public virtual void Draw()
{
    Console.WriteLine("Drawing the shape!");
}
```

دقت کنید، کلمه کلیدی virtual قبل از نوع بازگشتی متد نوشته می شود. حالا باید در کلاس فرزند رفتار Draw را مجدداً تعریف کنیم. برای اینکار باید متدی که از نوع virtual تعریف شده است را override کنیم. ابتدا رفتار Draw را برای کلاس Rectangle تغییر می دهیم. کد کلاس Rectangle را به صورت زیر تغییر دهید :

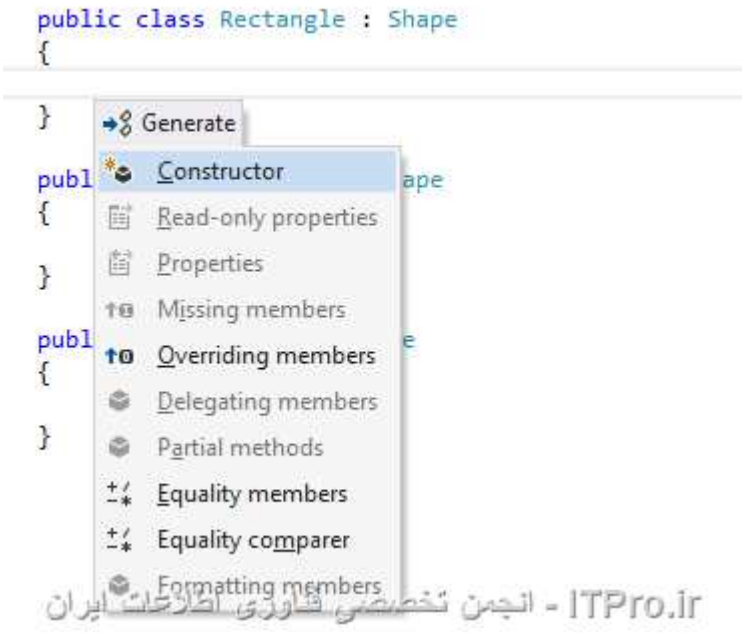
```
public class Rectangle : Shape
{
    public override void Draw()
    {
        Console.WriteLine("Drawing rectangle!");
    }
}
```

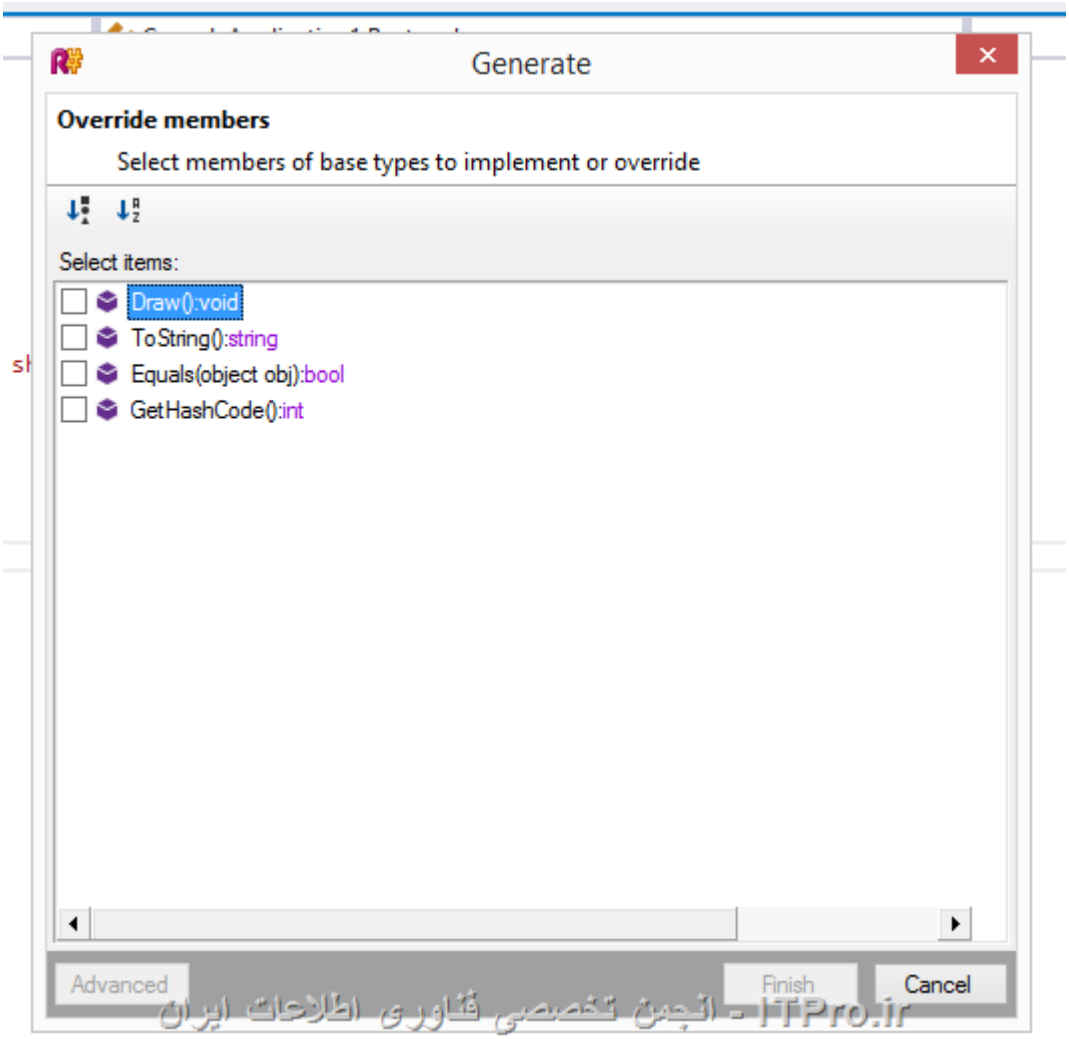
حالا با اجرای مجدد کد خروجی به صورت زیر تغییر می کند :

```
Drawing rectangle!
Drawing the shape!
Drawing the shape!
```

زمانی که شما داخل کلاس Rectangle شروع به تایپ می کنید، بعد از نوشتن کلمه کلیدی override و زدن کلید spaceلیستی از متدهایی که قابل override شدن هستند برای شما نمایش داده می شوند :

همچنین در صورتی که ابزار Resharper را نصب کرده باشید، وارد scope کلاس شده و کلید های Alt+Insert را فشار دهید، با اینکار منوی Generate برای شما نمایش داده می شود، از طریق این منو و انتخاب گزینه Overriding membersلیستی از تمامی اعضای قابل override شدن به شما نمایش داده می شود :





بعد از انتخاب عضو مورد نظر و فشار دادن کلید Finish متد مورد نظر برای شما override می شود. مهمترین کاربرد این ویژگی، زمانی است که شما تصمیم دارید چندین ویژگی را با هم override کنید. به یک نکته توجه داشته باشید، چه با روش اول متد را override کنید، چه با روش دوم، کدی برای شما به صورت خودکار درج می شود که به صورت زیر است :

```
public override void Draw()
{
    base.Draw();
}
```

در قسمت قبل با کلمه کلیدی base آشنا شدید، در کد بالا که به صورت پیش فرض نوشته می شود، با فراخوانی متد Draw از شئ ای که متد داخل آن override شده، نسخه کلاس پایه از رفتار فراخوانی می شود که شما باید بر اساس نیاز خود کد مورد نظر را برای رفتار override شده بنویسید .

کلاس های Triangle و Circle را نیز به صورت زیر تغییر می دهیم :

```
public class Triangle : Shape
{
    public override void Draw()
    {
        Console.WriteLine("Draw triangle!");
    }
}

public class Circle : Shape
{
    public override void Draw()
    {
        Console.WriteLine("Draw circle!");
    }
}
```

بعد از اجرای برنامه، خروجی باید به صورت زیر باشد :

```
Drawing rectangle!
Drawing triangle!
Drawing circle!
```

شاید خیلی از دوستان این سوال برایشان پیش بیاید که دلیل اینکار چیست؟ ما که متدها را در هر کلاس نوشتیم، چرا باید از کلاس پایه و override کردن آنها استفاده می کردیم؟ مهمترین خاصیت استفاده از Polymorphism استفاده از کلاس پدر برای کارهاست. برای درک بهتر، فرض کنید میخواهیم آرایه ای از اشیاء داشته باشیم. همانطور که می دانید، ما سه کلاس مختلف داریم و در صورت استفاده نکردن از قابلیت وراثت باید برای هر یک از کلاس ها یک آرایه تعریف کنیم. اما با قابلیت وراثت می توان یک آرایه از نوع کلاس پایه تعریف کرد و اشیاء فرزند را داخل آن قرار داد :

```
Shape[] shapes = new Shape[۵];
```

```
shapes[۰] = new Circle();
shapes[۱] = new Triangle();
shapes[۲] = new Circle();
shapes[۳] = new Rectangle();
shapes[۴] = new Triangle();

foreach (var shape in shapes)
    shape.Draw();

Console.ReadKey();
```

با اجرای کد بالا خروجی زیر به نمایش داده می شود :

```
Drawing circle!
Drawing triangle!
Drawing circle!
Drawing rectangle!
Drawing triangle!
```

با اینکه ما کلاس پایه یعنی Shape را به عنوان نوع آرایه در نظر گرفتیم، اما در هر خانه از آرایه ای شئ ای از نوع فرزندان کلاس Shape قرار دادیم و به دلیل override کردن رفتار Draw در کلاس های مشتق شده، فراخوانی متد Draw بر اساس تعریفی که در کلاس های فرزند داشتیم انجام می شود .

تعریف مجدد یا override کردن تنها محدود به رفتارها یا همان متدها نمی باشد. شما خصوصیات یا Property ها را نیز می توانید از نوع virtual تعریف کنید. برای مثال کلاسی را با نام Human فرض کنید که دارای سه خصوصیت به نام های FirstNameو LastName و FullSpecification می باشد :

```
public class Human
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
```

```
    public string FullSpecification
    {
        get { return FirstName + " " + LastName; }
    }
}
```

در کلاس بالا، خاصیت FullSpecification نام کامل را بر میگرداند. حالا کلاس فرزندی تعریف می کنیم با نام Employeeیا کارمند که از کلاس Human مشتق شده است و خاصیت جدید با نام JobPosition یا موقعیت شغلی به آن اضافه می کنیم :

```
public class Employee : Human
{
    public string JobPosition { get; set; }
}
```

در صورتی که شئ ای از روی Employee بسازیم و خاصیت FullSpecification آن را چاپ کنیم، نام و نام خانوداگی او در خروجی چاپ می شود. اما می خواهیم این خاصیت علاوه بر نام کامل، موقعیت شغلی او را نیز در خروجی چاپ کند. برای اینکار کافیست که در کلاس Human خاصیت FullSpecification را به صورت virtual تعریف کرده و در کلاس فرزند مجدد بخش get آن را تعریف کنیم :

```
public class Human
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public virtual string FullSpecification
    {
        get { return FirstName + " " + LastName; }
    }
}
```

```
public class Employee : Human
{
    public string JobPosition { get; set; }

    public override string FullSpecification
    {
        get { return base.FullSpecification + " with job position: "
+ JobPosition; }
    }
}
```

در کلاس Employee به بخش get توجه کنید :

```
get { return base.FullSpecification + " with job position: " +
JobPosition; }
```

در این قسمت، از کلمه کلیدی base برای گرفتن مقدار FullSpecification استفاده شده است. این خاصیت نام کامل را برای ما بر میگرداند و ما به انتهای آن موقعیت شغلی شخص را اضافه می کنیم و بر میگردانیم .

قابلیت Polymorphism در برنامه نویسی شئ گرا نقش بسیار پر رنگی دارد و خیلی جاها از نوشتن کدهای تکراری جلوگیری کرده و حجم کد برنامه شما را به صورت محسوسی کاهش می دهد.

در قسمت قبلی آموزش با مفهوم polymorphism آشنا شدیم. در ادامه قصد داریم با کلاس های abstract و sealed آشنا شویم. زمانی که شروع به نوشتن برنامه ای می کنیم، بعد از مشخص کردن موجودیت های برنامه و طراحی کلاس های مربوطه، باید یکسری محدودیت ها برای استفاده از کلاس ها وضع کرد. برای مثال، کلاس پایه ای داریم که این کلاس نباید مورد استفاده قرار بگیرد و تنها باید از کلاس های فرزند قابلیت ایجاد شئ وجود داشته باشد یا کلاسی نوشته ایم و نباید اجازه ایجاد کلاس فرزند از روی آن کلاس داده شود. این قابلیت ها بخصوص در مواقعی که در تیم شما، افرادی از کدهای نوشته شده توسط شما استفاده می کنند یا کدی را برای استفاده از سایر برنامه نویس ها بر روی اینترنت منتشر می کنید کاربرد دارند. راه حل برای این شرایط استفاده از کلاس های abstract و کلاس های sealed می باشد .

کلاس ها و اعضاءabstract

به بخش قبل و کلاس Shape بر میگردیم که سه کلاس فرزند از روی آن ها ساخته بودیم :

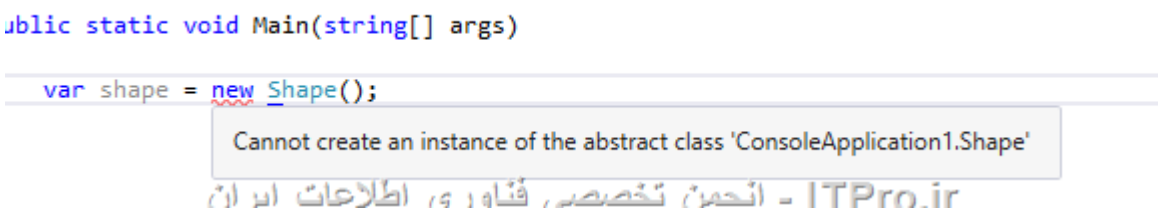
```
public class Shape
{
    public void Draw()
    {
        Console.WriteLine("Drawing the shape!");
    }
}

public class Rectangle : Shape
{

}

public class Triangle : Shape
{

}

public class Circle : Shape
```

```
{


}
```

کلاس Shape به تنهایی برای ما کاربردی نداشته و تنها داخل کد باید از کلاس های فرزند استفاده کرد، یعنی نباید از روی کلاس Shape شئ ایجاد شود. برای اینکار کافیست کلاس Shape را از نوع abstract تعریف کنیم :

```
public abstract class Shape
{
    public virtual void Draw()
    {
        Console.WriteLine("Drawing the shape!");
    }
}
```

حال اگر بخواهیم از روی کلاس Shape یک شئ ایجاد کنیم با پیغام خطا مواجه خواهیم شد :



اما کاربرد کلاس های abstract به همین جا ختم نمی شود، در قسمت قبل متد Draw را در کلاس Shape به صورت virtualتعریف کرده و در کلاس های فرزند آن را override کردیم. یکی از قابلیت های زبان سی شارپ، تعریف متدها به صورت abstract می باشد. متدهای abstract تنها شامل signature که در قسمت های اولیه با آن آشنا شدیم می باشد و بدنه ندارد، علاوه بر آن تمامی کلاس هایی که از یک کلاس abstract مشتق می شوند، در صورتی که کلاس abstract رفتار یا خاصیتی از نوع abstract داشته باشد، باید حتماً آن رفتار یا خاصیت را override کنند. برای مثال کلاس Shape را به صورت زیر تغییر می دهیم :

```
public abstract class Shape
{
    public abstract void Draw();
```

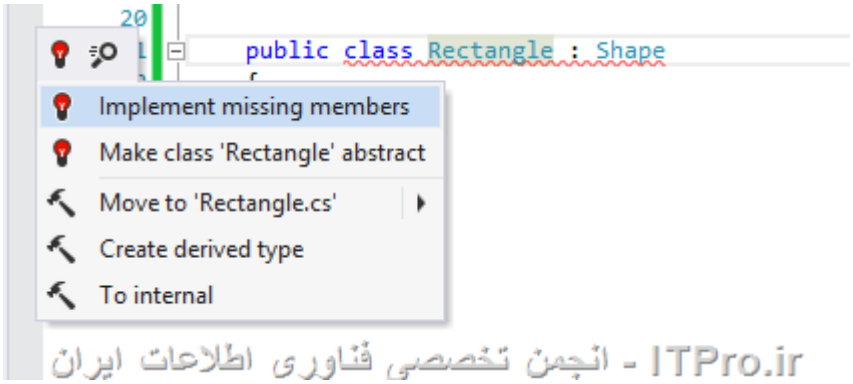
```
}
```

همانطور که مشاهده می کنید، متد Draw تنها تعریف شده و بدنه ای ندارد. حال اگر کلاس فرزندی از کلاس Shape مشتق شود باید متد Draw داخل آن Override شود. در غیر اینصورت با پیغام خطا مواجه خواهیم شد :

```
public abstract class Shape
{
    public abstract void Draw();
}

public class Rectangle : Shape
{
}
```

Abstract inherited member 'void ConsoleApplication1.Shape.Draw()' is not implemented

می توانیم به صورت دستی متد Draw را تعریف کرده یا از قابلیت Resharper برای پیاده سازی اعضای abstract به صورت خودکار استفاده کنیم. برای انکار مکان نما را بر روی نام کلاس قرار داده و کلید های Alt+Enter را فشار می دهیم. با این کار منویی با نام Action Context نمایش داده می شود :



از منوی ظاهر شده، گزینه Implement missing members را انتخاب می کنیم تا اعضای abstract پیاده سازی شوند. بعد از اینکار کد کلاس Rectangle به صورت زیر خواهد بود :

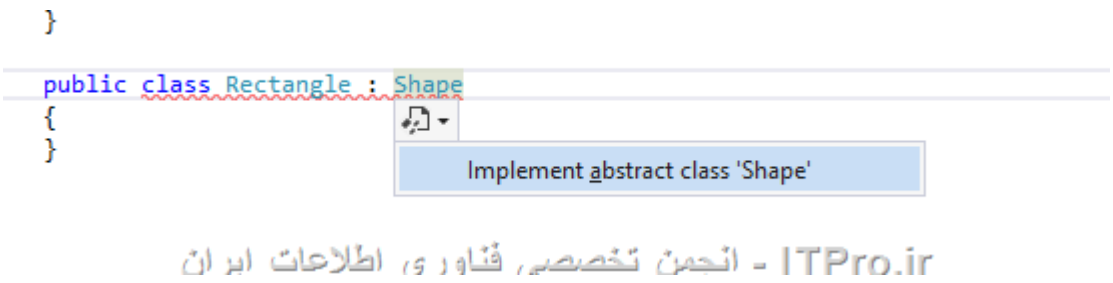
```
public class Rectangle : Shape
{
    public override void Draw()
    {
        throw new NotImplementedException();
    }
```

```
}
```

کدی که به صورت خودکار داخل بدنه متد Draw قرار گرفته مربوط به یکی از ویژگی های زبان سی شارپ با نام استثناها یا Exceptionsمی باشد که در بخش های بعدی با آن آشنا می شویم. حال کد مورد نظر را داخل متد Draw می نویسیم :

```
public class Rectangle : Shape
{
    public override void Draw()
    {
        Console.WriteLine("Drawing rectangle!");
    }
}
```

در صورتی که Resharper را نصب ندارید، برای پیاده سازی خودکار اعضاء abstract کلاس می توانید از قابلیت خود Visual Studioاستفاده کنید. برای اینکار، مکان نما را به انتهای اول تعریف کلاس برده و کلید های .+Alt را فشار دهید. منویی به صورت زیر نمایش داده می شود :



با انتخاب گزینه Implement abstract class Shape اعضاء ای که از نوع abstract تعریف شده اند در کلاس پیاده سازی می شوند .

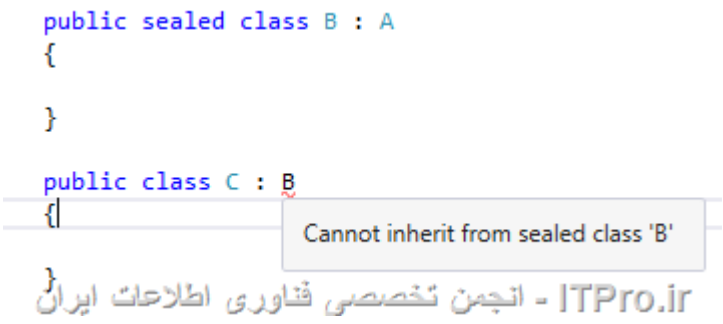
این قابلیت دقیقاً کار مشابهی با متدهای virtual انجام می دهد، با این تفاوت که متدهای abstract بدنه ای ندارند، اما اعضاء virtual می توانند بدنه ای داشته باشند که بوسیله کلمه کلیدی base می توان به آنها در کلاس فرزند دسترسی داشت. همچنین توجه کنید که اعضاء abstract تنها داخل کلاس های abstract قابل تعریف هستند .

کلاس ها و اعضاءsealed

در قسمت وراثت گفتیم که می توان زنجیره وراثت داشت. یعنی کلاس B از کلاس A و کلاس C از کلاس B مشتق شوند. اما فرض کنید بخواهیم زنجیره وراثت را در یک کلاس قطع کنیم. یعنی کلاس دیگری نتواند از کلاس ما ارث بری کند. برای اینکار کافیست کلاس مورد نظر را از نوع sealed تعریف کرد :

```
public class A
{

}

public sealed class B : A
{

}
```

در کد بالا کلاس B از نوع sealed تعریف شده، بدین معنا که هیچ کلاس دیگری نمی تواند از این کلاس مشتق شود. در تصویر زیر، کلاس C از کلاس B مشتق شده و پیغام خطا دریافت کردیم :



یکی دیگر از کاربردهای کلمه کلیدی sealed جلوگیری از override کردن یک متد است. کلاس Shape و Rectangle را مثال میزنیم. میخواهیم اگر کلاسی از کلاس Rectangle مشتق شد قابلیت ovverride ردن متد Draw را نداشته باشد. کافیست متد Draw را از نوع sealed تعریف کنیم :

```
public abstract class Shape
{
    public virtual void Draw()
    {
```

```
        Console.WriteLine("Drawing the shape!");
    }
}

public class Rectangle : Shape
{
    public sealed override void Draw()
    {
        Console.WriteLine("Drawing rectangle!");
    }
}
```

همانطور که در کد بالا مشاهده می کنید، متد Draw در کلاس Rectangle از نوع sealed تعریف شده. حالا اگر کلاسی تعریف کنیم و آن را کلاس Rectangle مشتق کنیم، در کلاس فرزند قابلیت تعریف مجدد متد Draw را نخواهیم داشت .

در قسمت قبل با کلمات کلیدی abstract و sealed آشنا شدیم. در این قسمت و قسمت بعد تصمیم دارم برخی نکات تکمیلی که از بخش برنامه نویسی شئ گرا مانده را خدمت دوستان آموزش بدم. مواردی که در این بخش با آنها آشنا خواهید شد به شرح زیر است :

۱. سازنده ها در کلاس های فرزند و پدر
۲. سطح دسترسیprotected
۳. مخفی کردن متدها با کلمه کلیدیnew
۴. فیلدهایReadonly

سازنده هه در کلاس های فرزند و پدر

در قسمت های قبلی آموزش با مفهوم سازنده و کاربرد آن ها در کلاس ها آشنا شدیم. چند نکته در مورد سازنده ها در کلاس های فرزند وجود دارد که در این بخش آنها را بررسی می کنیم. فرض کنید کلاس پایه ای داریم که به صورت زیر تعریف شده است :

```
public class Human
{
    public Human(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }

    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

در ادامه کلاس Employee را به صورت زیر تعریف می کنیم :

```
public class Employee : Human
{


}
```

اما کد بالا منجر به وقوع خطا خواهد شد، دلیل آن هم عدم وجود سازنده پیش فرض در کلاس پایه می باشد. اگر به خاطر داشته باشید سازنده پیش فرض، سازنده ای است که هیچ پارامتری به عنوان ورودی دریافت نمی کند :

```
public class Human
{
    public Human(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }

    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class Employee : Human
{
}
```

Base class 'ConsoleApplication1.Program.Human' doesn't contain parameterless constructor

برای رفع این مشکل دو راه وجود دارد، یکی تعریف سازنده پیش فرض در کلاس والد و روش دوم پیاده سازی سازنده ای برای کلاس فرزند که پارامتر های مورد نیاز برای کلاس والد را گرفته و به سازنده کلاس والد ارسال می کند. اگر از بخش های قبلی به خاطر داشته باشید، با مبحثی در سازنده آشنا شدیم به نام زنجیره سازنده ها یا .Constructor Chaining در مثال بالا، باید از قابلیت Constructor Chaining استفاده کرد، اما سازنده کلاس والد را صدا زد. کلاس Employee را به صورت زیر تغییر می دهیم :

```
public class Employee : Human
{
    public Employee(string firstName, string lastName) :
base(firstName, lastName)
    {
    }
}
```

همانطور که مشاهده می کنید، سازنده ما دو پارامتر دریافت می کند و هر دوی این پارامترها را به عنوان ورودی به سازنده کلاس والد ارسال می کند. کلمه base در اینجا به سازنده تعریف شده در کلاس والد اشاره می کند. حال فرض کنید خصوصیت جدیدی به نام JobPosition به کلاس Employee اضافه کردیم و میخواهیم این خصوصیت از طریق سازنده مقدار دهی شود. کافیست کد کلاس Employee را به صورت زیر تغییر دهیم :

```
public class Employee : Human
{
    public string JobPosition { get; set; }

    public Employee(string firstName, string lastName, string
jobPosition) : base(firstName, lastName)
    {
        JobPosition = jobPosition;
    }
}
```

همانطور که مشاهده می کنید، پارامتر سومی به نام jobPosition به سازنده اضافه کردیم و داخل سازنده کلاس فرزند خصوصیت JobPosition را مقدار دهی کردیم، اما مقادیر firstName و lastName به سازنده کلاس والد ارسال شد تا برای برای مقدار دهی خصوصیات FirstName و LastName از کد کلاس والد استفاده کنیم .

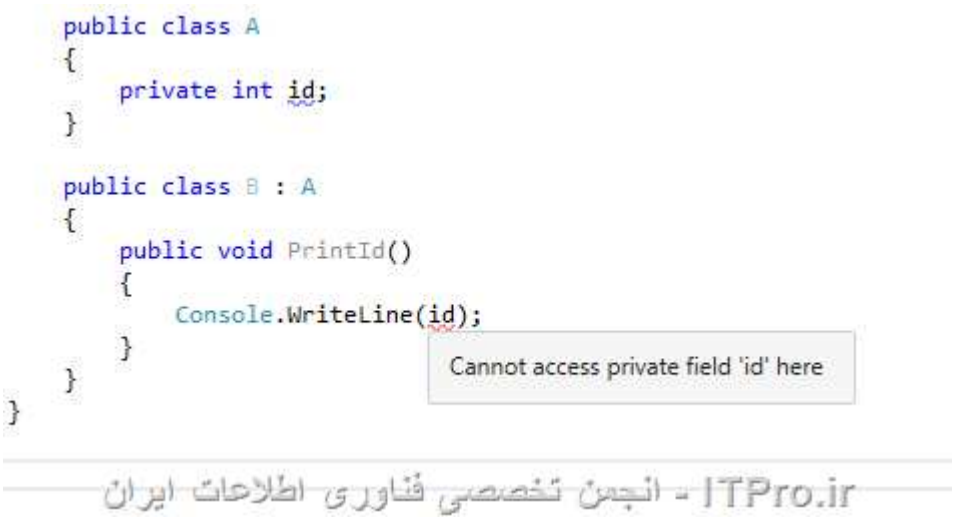
سطح دسترسیprotected

در بخش های قبلی در مورد سطوح دسترسی مختلف صحبت کردیم و گفتیم که هر سطح دسترسی مشخص می کند یک کلاس یا اعضای کلاس تا چه سطحی از برنامه دسترسی داشته باشند. یک سطح دسترسی باقی ماند به نام protected که نیاز به بررسی مفهوم وراثت داشت. کلاس های زیر را در نظر بگیرید :

```
public class A
{
    private int id;
}
```

```
public class B : A
{

}
```

در کد بالا، فیلد id که در کلاس A تعریف شده، خارج از کلاس B قابل دسترس نخواهد بود، زیرا این فیلد به صورت privateتعریف شده است :



حتی در کلاس B که فرزند کلاس A می باشد، این فیلد در دسترس نمی باشد. یک راه حل تعریف فیلد id به صورت publicیا internal است. البته دقت کنید در صورت تعریف کردن فیلد id به صورت internal در صورتی کلاس B به آن دسترسی خواهد داشت که محل تعریف کلاس A و B در یک پروژه باشد. اما در صورتی که بخواهیم فیلد id تنها در کلاس های فرزند و خود کلاس A در دسترس باشد، باید سطح دسترسی فیلد id را protected تعریف کنیم. با این کار قابلیت دسترسی به فیلد id را از کلاس B خواهیم داشت. کلاس A را به صورت زیر تغییر می دهیم :

```
public class A
{
    protected int id;
}
```

یک حالت دیگر نیز وجود دارد و ترکیب استفاده از protected و internal است. در حالت بالا، در صورتی که کلاس A به صورت public تعریف شده باشد و ما خارج از پروژه ای که کلاس A تعریف شده کلاسی بسازیم و از کلاس A مشتق کنیم، به فیلد id دسترسی خواهیم داشت. برای اینکه فیلد id تنها از کلاس هایی که داخل همان پروژه ای که کلاس A وجود دارد

تعریف شده اند و کلاس A مشتق شده اند قابل دسترسی باشد، دسترسی آن را از نوع protected internal تعریف می کنیم :

```
public class A
{
    protected internal int id;
}
```

دقت کنید که می توان protected internal را به صورت internal protected نیز نوشت و هیچ تفاوتی در عملکرد آنها وجود ندارد .

*در توضیحات بالا، منظور از دسترسی خارج از پروژه، پروژه های است که به پروژه ما Reference داده می شوند، منظور از Referenceدادن استفاده از کدهای موجود در یک پروژه می باشد. در بخش های بعدی روش تعریف Solution هایی که بیش از یک پروژه دارند و Reference دادن آنها به هم را بررسی خواهیم کرد .

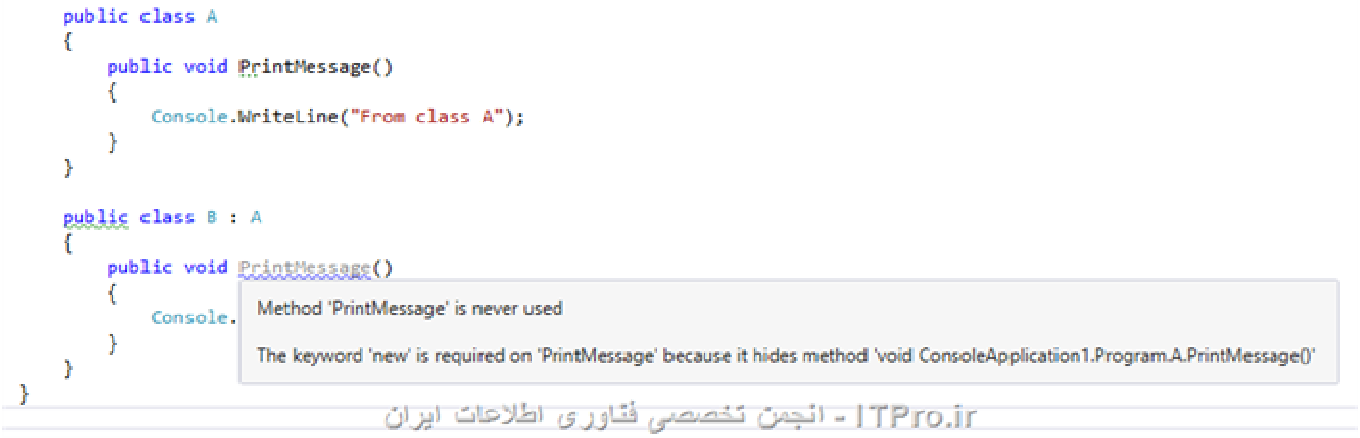
مخفی سازی اعضاء بوسیله کلمه کلیدیnew

در بخش polymorphism روش override کردن متدها و خصوصیات را گفتیم. حالتی وجود دارد که یک عضو کلاس از نوع virtual تعریف نشده، اما در کلاس فرزند عضوی همنام یکی از اعضای کلاس پدر تعریف شده است. به مثال زیر توجه کنید :

```
public class A
{
    public void PrintMessage()
    {
        Console.WriteLine("From class A");
    }
}

public class B : A
{
```

```
    public void PrintMessage()
    {
        Console.WriteLine("From class B");
    }
}
```

به تصویر زیر دقت کنید :



در تصویر بالا مشاهده می کنید که کامپایلر به شما اخطار می دهد که برای مخفی سازی متد در کلاس فرزند بهتر است از کلمه کلیدی new استفاده کنید. متد PrintMessage در کلاس B را به صورت زیر تغییر می دهیم :

```
public class B : A
{
    public new void PrintMessage()
    {
        Console.WriteLine("From class B");
    }
}
```

با کلمه کلیدیnew ، کامپایلر دیگر اخطاری به شما نمی دهد. دقت کنید که کلمه کلیدی new کاملاً با override کردن متدها تفاوت دارد. کد زیر پیغام From Class A را چاپ می کند، اما در صورت override کردن متد در کلاس B پیغام From class Bچاپ می شد :

```
A obj = new B();
obj.PrintMessage();
```

نوشتن و ننوشتن کلمه کلیدی new تغییری در روند اجرا و عملیات کلاس ایجاد نمی کند، این اخطار تنها برای این است که شما اشتباهاً در کلاس های فرزند عضوی همنام با اعضای کلاس والد تعریف نکنید و این کار با آگاهی شما انجام شود .

فیلدهایreadonly

فیلدهای readonly فیلدهایی هستند که تنها در سازنده می توان آنها را مقدار دهی کرد و خارج از سازنده شما امکان مقدار دهی آن ها را نخواهید داشت و تنها می توان از مقدار آنها استفاده کرد :

```
public class A
{
    private readonly int value;

    public A(int value)
    {
        this.value = value;
    }
}
```

در کلاس بالا فیلد value از نوع readonly تعریف شده است، در صورتی که جایی خارج از سازنده شما value را مقدار دهی کنید با پیغام خطا مواجه خواهید شد :

```
public class A
{
    private readonly int value;

    public A(int value)
    {
        this.value = value;
    }

    public void ChangeValue()
    {
        value = 12;
    }
}
}
```

Readonly field cannot be used as an assignment target

تعریف interface ها

در حقیقت interface ها، به ما امکان تعریف مجموعه ای از خصوصیات و متدهای مرتبط را می دهند و قابلیت پیاده سازی در کلاس ها یا struct ها را دارند. با استفاده از interface ها، شما قابلیت پیاده سازی چندین ویژگی از چندین interface مختلف را در یک کلاس یا struct خواهید داشت. کلاس ها به صورت پیش فرض قابلیت ارث بری از چند کلاس را پشتیبانی نمی کنند و برای شبیه سازی این کار باید از interface ها استفاده کرد. در ابتدا با ساختار کلی تعریف interface آشنا می شویم :

```
{access-modifier} interface {name}
{
    {members}
}
```

۱. در قسمت access-modifier که سطح دسترسی به interface را مشخص می کنیم.
۲. در قسمتname ، نام interface مشخص می شود. به این نکته توجه داشته باشید، استانداردی برای نام گذاری interfaceها وجود دارد، به این صورت که در ابتدای نام interface بهتر است کاراکتر I قرار بگیرد تا کلاس ها و interfaceها از یکدیگر مجزا شوند. برای مثال IPerson :یا.IDatabaseManager
۳. در قسمتmember ، باید اعضای interface را تعریف کنیم، دقت کنید که در این بخش برای اعضا نه می توان access-modifierمشخص کرد و نه بدنه ای برای متدها، تنها باید تعریف کلی از اعضاء نوشته شود و همچنین اعضای تعریف شده برای یک interface همگی به صورت پیش فرض سطح دسترسی public خواهند داشت.

با یک مثال ساده ادامه می دهیم. در نمونه کد زیر یک interface با نام INamed تعریف کردیم که یک خصوصیات به نام Nameو یک متد با نام PrintName در آن تعریف شده است :

```
public interface INamed
{
    string Name { get; set; }

    void PrintName();
}
```

همانطور که در کد بالا مشاهده می کنید، سطح دسترسی نه برای خصوصیت مشخص شده و نه برای متد، همچنین متد ما بدنه نداشته و فقط حاوی signature می باشد. در ادامه قصد داریم تا از این interface استفاده کنیم. استفاده از interfaceدقیقاً مانند حالتی است که می خواهیم کلاس پدر را برای یک کلاس فرزند در وراثت مشخص کنیم :

```
public class Car : INamed
{


}
```

اما با نوشتن کد بالا پیغام خطا دریافت خواهید کرد. زیرا interface تنها حاوی تعریف کلی بوده و باید پیاده سازی در کلاسی که interface را به ارث برده انجام شود :

```
public class Car : INamed
{
    public string Name { get; set; }
    public void PrintName()
    {
        Console.WriteLine(Name);
    }
                                                                    }
```

برای پیاده سازی خودکارinterface ، در صورتی که Resharper را نصب کرده باشید با بردن مکان نما بر روی نام کلاس، فشردن کلیدهای Alt+Enter و انتخاب گزینه Implement missing members عملیات پیاده سازی کلاس به صورت خودکار برای شما انجام می شود و در صورتی که Resharper نصب نباشد، با رفتن بر روی نام interface در مقابل کلاس و فشردن کلیدهای .+Ctrl عملیات پیاده سازی را می توانید انجام دهید.تا اینجا عملیات پیاده سازی کلاس را انجام دادیم، اما این پیاده سازی چه ویژگی هایی برای ما دارد و چگونه باید از interface استفاده کنیم؟ اگر توضیحات قسمت وراثت را به خاطر داشته باشید، گفتیم زمانی که یک کلاس، از کلاس دیگری ارث بری می کنید، می توان برای ساختن یک نمونه از کلاس، از نوع داده پدر استفاده کرد. این مورد، در باره interface ها هم صدق می کند، در حقیقت ما می توانیم از interfaceبرای ایجاد شئ مورد نظر استفاده کنیم :

```
INamed namedInstance = new Car();
```

اما باید به یک نکته توجه داشته باشید، زمانی که از interface برای ساخت شئ استفاده می کنید تنها می توانید به اعضایی از کلاس دسترسی داشته باشید که در interface تعریف شده اند.یکی دیگر از قابلیت هایinterface ، همانطور که در ابتدای این بخش گفته شد، امکان Multiple-Inheritance می باشد. به طور پیش فرض، شما تنها می توانید از یک کلاس در دات نت ارث بری کنید و امکان ارث بری از چند کلاس وجود ندارد. برای رفع این مشکل می توان از interface ها استفاده کرد. یعنی شما می توانید نام چند interface را در مقابل نام کلاس بنویسید. برای مثال، یک interface جدید با نام INotify تعریف می کنیم :

```
public interface INotify
{
    void Notify();
}
```

حال می توانیم در کلاسCar ، علاوه بر INamed از INotify نیز استفاده کنیم :

```
public class Car : INamed, INotify
{
    public string Name { get; set; }
    public void PrintName()
    {
        Console.WriteLine(Name);
    }

    public void Notify()
    {
        Console.WriteLine("Notify me via Email!");
    }
}
```

پیاده سازی interface ها به صورت implicite وexplicit

پیاده سازی interface ها در کلاس ها به دو صورت انجام می شود، implicite و explicit. تفاوت این دو روش در این است که در ابتدای نام عضو interface در کلاس، نام interface به همراه کاراکتر . قرار میگیرد. در قسمت های قبل، پیاده سازی ها بر اساس روش implicit انجام شد و در این قسمت با روش explicit آشنا می شویم. برای مثال فرض کنید می خواهیم INotify را به صورت explicit پیاده سازی کنیم، کد زیر پیاده سازی با این روش را نشان می دهد :

```
public class Car : INamed, INotify
{
    public string Name { get; set; }
    public void PrintName()
    {
        Console.WriteLine(Name);
    }

    void INotify.Notify()
    {
        Console.WriteLine("Notify me via Email!");
    }
}
```

اگر در کد بالا دقت کنید، برای متد Notify هیچ سطح دسترسی مشخص نشده است، زمانی که شما یک عضو را به صورت explicitپیاده سازی می کنید، هیچ سطح دسترسی نباید برای آن مشخص کنید. همچنین اعضایی که به صورت Explicit پیاده سازی می شوند تنها در صورتی قابل دسترس هستند که با نام interface از روی آنها شئ ساخته شود. یعنی در کد زیر شما به متد Notify دسترسی نخواهید داشت :

```
Car carInstance = new Car();
carInstance.Notify();
```

کد بالا منجر به پیغام خطا خواهد شد. اما کد زیر بدون مشکل اجرا می شود :

```
INotify notifyInstance = new Car();
notifyInstance.Notify();
```

یکی از مهمترین کاربردهای پیاده سازیexplicit ، امکان پیاده سازی چند interface با اعضای هم نام در یک کلاس است! برای روشتر شدن موضوع فرض کنید ما Interface ای داریم با نام IEmailNotify که عملیات اطلاع رسانی را به بوسیله ایمیل و interface دیگری داریم با نام ISMSNotify که عملیات اطلاع رسانی را بوسیله پیامک انجام می دهد. هر دوی این interface ها متدی دارند با نام :Notify

```
public interface IEmailNotify
{
    void Notify();
}

public interface ISMSNotify
{
    void Notify();
}
```

حال کلاس Car را به صورت زیر تغییر می دهیم :

```
public class Car : INamed, IEmailNotify, ISMSNotify
{
    public string Name { get; set; }
    public void PrintName()
    {
        Console.WriteLine(Name);
    }

    void IEmailNotify.Notify()
    {
        Console.WriteLine("Notify via Email!");
    }

    void ISMSNotify.Notify()
```

```
    {
        Console.WriteLine("Notify via SMS!");
    }
}
```

کد بالا دو پیاده سازی برای متد Notify دارد. یکی برای IEmailNotify و یکی برای ISMSNotify که بر اساس نوع داده مورد استفاده برای شئ، متد مربوطه فراخوانی خواهد شد. در ادامه کد زیر را در متد Main می نویسیم :

```
var car = new Car();

ISMSNotify smsNotify = car;
smsNotify.Notify();
IEmailNotify emailNotify = car;
emailNotify.Notify();
```

با اجرای کد بالا، ابتدا خروجی Notify via SMS و سپس Notify via Email در خروجی چاپ خواهد شد. ما در حقیقت یک شئ از نوع Car ایجاد کردیم و یکبار آن را داخل متغیری از نوع ISMSNotify قرار دادیم و بار دوم در متغیری با نام .IEmailNotify عملیات فراخوانی متد Notify به صورت خودکار بر اساس نوع متغیر انجام خواهد شد.مباحث مربوط به interface بسیار گسترده می باشد. یکی از مهمترین کاربردهای interface پیاده سازی IoC یا Inversion of Controlو DI یا Dependency Injection در برنامه ها می باشد

بروز خطا در برنامه امری اجتناب نا پذیر است و یک برنامه نویس موظف است که خطاها را به درستی در برنامه ها مدیریت کرده و زمان بروز خطا، پیغامی مناسب به کاربر نمایش دهد. در زبان سی شارپ، به خطاها Exception یا استثنا می گویند. در برنامه های کامپیوتری خطاها بر دو دسته اند :

۱. خطاهای نحوی یا :Syntax Errors این خطاها به دلیل نوشتن اشتباه دستورات ایجاد شده و معمولاً زمان کامپایل برنامه قابل رفع هستند.
۲. خطاهای منطقی یا :Logical Errors این خطاها به دلیل انجام اشتباه یک عملیات یا ورود اشتباه یک دستور در زمان اجرا اتفاق می افتند.

بیشترین تمرکز ما برای مدیریت خطاها، روی دسته دوم خطاهاست، برای شروع کد زیر را در نظر بگیرید :

```
var firstNumber = int.Parse(Console.ReadLine());
var secondNumber = int.Parse(Console.ReadLine());

Console.WriteLine(firstNumber/secondNumber);
```

کد بالا، دو عدد را از ورودی خوانده و حاصل تقسیم این دو عدد را در خروجی چاپ می کند، اما فرض کنید مقدار عدد دوم صفر وارد شود، امکان تقسیم اعداد بر عدد صفر وجود ندارد و در صورت ورود عدد صفر به عنوان ورودی دوم، با پیغام خطای DivideByZeroمواجه می شویم. برای رفع ایین مشکل، می بایست از مکانیزم کنترل Exception ها استفاده کنیم. در زبان سی شارپ این مکانیزم، با ساختار try..catch انجام می شود :

```
try
{
    // place your code here
}
catch([ExceptionType])
{
}
catch([ExceptionType])
{
}
finally
```

```
{
}
```

قسمتی از کد که احتمال وقوع خطا در آن وجود دارد را باید داخل بدنه try بنویسید، با این کار، در صورت وقوع خطا در کدی که داخل بدنه try نوشته شده، قسمت catch اجرا می شود. اما نحوه اجرای قسمت catch به چه صورت است؟ در کتابخانه، برای هر نوع خطا، یک کلاس تعریف شده، برای مثال، برای خطای تقسیم بر صفر کلاسی با نام DivideByZeroExceptionوجود دارد، کلاً تمامی کلاس های مرتبط با خطا های مختلف با کلمه Exception تمام می شوند، کلاس هایی مانند InvalidOperationException یاStackOverlowException ، تمامی این کلاس از کلاس پایه ای با نام SystemException مشتق شده اند که خود کلاس SysteException از کلاس Exception مشتق شده است، در حقیقت کلاس Exception کلاس پایه ای برای کلیه خطاهای سیستم می باشد. حال شما بر اساس نوع خطایی که قصد مدیریت آن را دارید، نام Data Type آن را در مقابل catch می نویسید، برای مثال، اگر تصمیم دارید خطای تقسیم بر صفر را مدیریت کنید، ساختار try..catch به صورت زیر نوشته می شود :

```
try
{

}
catch (DivideByZeroException)
{
}
```

در صورتی که خطای تقسیم بر صفر در سیستم رخ دهد، بدنه catch اجرا خواهد شد، کد ابتدای آموزش را به صورت زیر تغییر می دهیم :

```
try
{
    var firstNumber = int.Parse(Console.ReadLine());
    var secondNumber = int.Parse(Console.ReadLine());

    Console.WriteLine(firstNumber / secondNumber);
}
catch (DivideByZeroException)
{
```

```
    Console.WriteLine("Second number must be greater than zero!");
}

Console.ReadKey();
```

با اجرای کد بالا، در صورتی که عدد دوم را صفر وارد کنیم، به جای متوقف شدن برنامه و بروز خطا، پیغام مناسب برای کاربر نمایش داده می شود. اما نکته ای که وجود دارد، شما می توانید بیشتر از یک بدنه catch داشته باشید. برای مثال، در کد بالا در صورتی که شما به جای عدد کاراکتر a را وارد کنید، با خطای FormatException مواجه می شوید، برای مدیریت این خطا کافیست کد بالا را به صورت زیر تغییر دهید :

```
try
{
    var firstNumber = int.Parse(Console.ReadLine());
    var secondNumber = int.Parse(Console.ReadLine());

    Console.WriteLine(firstNumber/secondNumber);
}
catch (DivideByZeroException)
{
    Console.WriteLine("Second number must be greater than zero!");
}
catch (FormatException)
{
    Console.WriteLine("Invalid input format!");
}
```

با کد بالا، در صورت اشتباه در ورودی، خطا مدیریت شده و پیغام مناسب نمایش داده می شود. قسمت دیگر ساختار try..catch، بدنه finally می باشد، این قسمت از ساختار، در هر صورت اجرا خواهد شد، چه خطا رخ بدهد، چه خطا رخ ندهد، کد بالا را به صورت زیر تغییر می دهیم :

```
try
{
```

```
    var firstNumber = int.Parse(Console.ReadLine());
    var secondNumber = int.Parse(Console.ReadLine());

    Console.WriteLine(firstNumber/secondNumber);
}
catch (DivideByZeroException)
{
    Console.WriteLine("Second number must be greater than zero!");
}
catch (FormatException)
{
    Console.WriteLine("Invalid input format!");
}
finally
{
    Console.WriteLine("Thank you for choosing ITPRO.IR!");
}
```

با کد بالا، پیغام داخل بدنه finally در هر صورت در خروجی چاپ خواهد شد، استفاده از بدنه finally بیشتر زمانی کاربرد دارد که شما می خواهید بعد از اتمام عملیات، اقدام به پاک سازی حافظه و آزاد سازی منابع کنید .

امکان مدیریت خطاها به صورت عمومی نیز وجود دارد، گفتیم کلیه کلاس های مربوط به خطاها از کلاس Exception مشتق شده اند، برای مدیریت عمومی خطاها، کافیست در بدنه catch به جای یک نوع مشخص از خطا، نام Exception را بنویسیم :

```
try
{
    var firstNumber = int.Parse(Console.ReadLine());
    var secondNumber = int.Parse(Console.ReadLine());

    Console.WriteLine(firstNumber/secondNumber);
}
```

```
catch (Exception)
{
    Console.WriteLine("Oops! Your input stopped me!");
}
```

با کد بالا دیگر نوع خطا تفاوتی نمی کند، با وقوع هر خطایی، پیغام داخل بدنه catch در خروجی چاپ می شود .

همانطور که گفتیم، کلاسException ، کلاس پایه ای برای کلیه خطاهای سیستم می باشند، این کلاس حاوی یک سری خصوصیات است که اطلاعات دقیق تری به ما می دهند. برای دسترسی به اطلاعات خطا، به جای نوشتن تنها نام Exceptionدر مقابل بدنهcatch ، داخل پرانتز Exception را به صورت یک پارامتر تعریف می کنیم تا بتوانیم به اطلاعات آن داخل بدنه catch دسترسی داشته باشیم :

```
try
{
    var firstNumber = int.Parse(Console.ReadLine());
    var secondNumber = int.Parse(Console.ReadLine());

    Console.WriteLine(firstNumber/secondNumber);
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
    Console.WriteLine(ex.StackTrace);
    Console.WriteLine(ex.InnerException.ToString());
}
```

همانطور که مشاهده می کنید کلاس Exception شامل یکسری خصوصیات است، در زیر به بررسی مهمترین خصوصیات کلاس Exception می پردازیم :

۱. خصوصیت :Message این خصوصیت حاوی پیغام خطای تولید شده توسط برنامه است

۲. خصوصیت StackTrace: این خصوصیت شامل جزئیاتی از خطای رخ داده شده است، ممکن است خطاها در هر قسمت از برنامه رخ دهند، بوسیله stack trace قابلیت ردیابی خطا و اینکه وقوع این خطا بعد از فراخوانی کدام متدها و در کدام فایل و خط از برنامه اتفاق افتاده را خواهیم داشت.
۳. خصوصیت InnerException: گاهی اوقات، یک خطا می تواند شامل یک خطای درونی باشد، برای مثال، شما زمانی که با بانک SQL Server کار می کنید، ممکن از زمان کار با بانک، خطایی دریافت کنید، خصوصیت Inner Exceptionاطلاعات جزئی تری از خطاهای اتفاق افتاده به شما می دهد. این خصوصیت از نوع Exceptionبوده و اطلاعات خطاهای درونی یک خطا را به ما می دهد.

خطاهای دلخواه و دستورthrow

در زبان سی شارپ امکان تعریف خطاهای دلخواه وجود دارد، همانطور که در قسمت قبلی گفتیم، هر خطا از کلاس SystemExceptionمشتق شده که خود SystemException از کلاس Exception مشتق می شود، در دات نت کلاس دیگری وجود دارد به نام ApplicationException که از کلاس Exception مشتق شده و ما می توانیم با ایجاد کلاس هایی که از ApplicationException مشتق شده اند، خطاهای دلخواه خود را تعریف کنیم. برای مثال، کد زیر را در نظر بگیرید :

```
public class StudentManager
{
    public void RegisterStudent(string firstName, string lastName,
byte age)
    {
        // add student to database
    }
}
```

فرض کنید، می خواهیم از ثبت نام افرادی که سنشان کمتر از ۱۸ سال است جلوگیری کنیم. برای اینکار می توانیم به این صورت عمل کنیم، ابتدا یک کلاس برای خطای سن کمتر از ۱۸ سال تعریف می کنیم :

```
public class InvalidAgeException : ApplicationException
{
    public InvalidAgeException(string message) : base(message)
```

```
    {
    }
}
```

دقت کنید به عنوان پارامتر ورودی سازنده، پیغام خطا را دریافت و به سازنده کلاس پدر ارسال می کنیم. حال باید از این خطا در کلاس StudentManager استفاده کنیم، یعنی با ورود سن کمتر از ۱۸ سال، خطای InvalidAgeException صادر شود، در زبان سی شارپ، برای صدور خطا از دستور throw استفاده می کنیم :

```
public class StudentManager
{
public void RegisterStudent(string firstName, string lastName, byte age)
{
if (age < ۱۸)
throw new InvalidAgeException("Age must be greater that ۱۸!");
/ add student to database
}
}
```

در مرحله بعد، با اجرای دستور زیر خطا دریافت خواهیم کرد :

```
new StudentManager().RegisterStudent("Hossein", "Ahmadi", ۱۷);
```

حال می توانیم با ساختار try..catch این خطا را مدیریت کنیم :

```
try
{

}
catch (InvalidAgeException ex)
{
    Console.WriteLine(ex.ToString());
    throw;
}
```

همانطور که مشاهده می کنید در ساختار بالا، خطای InvalidAgeException مدیریت شده و پیغام خطا در خروجی چاپ می شود .

منبع : وبسایت programming.itpro.ir